SQLite Database System
Design and Implementation

Sibsankar Haldar
Motorola Mobility, Inc
809 11th Avenue
Sunnyvale, CA 94089
United States of America

October 21, 2011

This page is intentionally kept blank.

Preface

This book aims to setting up a mental framework for readers to study, design, develop, maintain, and enhance database management system softwares. At the very beginning of the book, I review many fundamental database concepts that are formulated over the past several decades and that prevail in many modern database systems. I am mostly brief in discussing these concepts. I put more emphasize on the presentation of a particular SQL-based relational database management system, namely SQLite. In a major part of this book, I discuss design principles, engineering trade offs, implementation issues, and operations of SQLite. The book presents a comprehensive description of all important components of SQLite. That is, it presents an end-to-end picture on SQLite.

Many different varieties of database management systems have been developed over the past several decades. DB2, Informix, Ingres, MySQL, Oracle, PostgreSQL, SQL Server, Sybase are a few to mention here. These systems are commercially very successful. There were some more database systems that were commercially unsuccessful or became obsolete over time. These database systems (successful or not) profess many theoretical and technological challenges. All the theories and technologies are primarily geared toward two goals: user convenience in the management of data, and efficient ways of storing, retrieving, and manipulating data.

Because the database field is so large, it is impossible to accommodate all known theories and technologies in a single book like this one, especially when I am primarily interested in presenting a particular database management system. I do not attempt to cover everything from the field. As far as theories are concerned, I discuss core concepts that prevail in many modern relational database management systems. As far as technologies are concerned, I present internals of SQLite database management system. SQLite database system is the only focus of this book. It is a small, embeddable, SQL-based relational database management system. It has been widely used in low to medium tier database applications, especially in embedded devices.[1] One can find some well known users of SQLite at the webpage http://www.sqlite.org/famous.html. It is an open source software, and is available in the public domain free of cost. There is no copyright claim on any part of the core

[1] SQLite received the 2005 OSCON Google and O'Reilly Integrator category award.

SQLite source code.[2] One may download the source code from its homepage http://www.sqlite.org, and experiment with it by modifying different parts of the code. SQLite runs on Linux, Windows, MAC OS X, and a few other operating systems. In this book I restrict myself to the Linux version of SQLite 3.7.8 release that is the latest version as of September 19, 2011.

This book would be a valuable asset to database course students in academic institutions as well as to professional developers of database management systems and SQLite applications in the computer industry. It can be used as a companion text book for advance database courses in Computer Science and Information Technology disciplines. It will be highly useful for those students who want to quickly learn about internals of a typical database system or take short term projects in database management systems. Project students will get access to some basic infrastructure to experiment with a real database system; they will get acquainted with core implementation concepts in a short time. There is no license issue in using SQLite in what so ever way possible. It would certainly be a valuable book for those who want to develop SQLite applications. This book however does not teach how to diagnose or debug defects in database management system softwares or applications. In essence, after reading this book, students and professional software developers would be aware of principal challenges in designing and developing an embedded database system, and solutions adopted by the SQLite development team in particular. And, they (readers) may get emotional and start developing their own database management systems.

I would assume that readers are familiar with structured computer programming practices, and they have some work experience in developing and testing softwares using the **C** programming language. SQLite itself is written in **ANSI C**; the source distribution also comes with numerous test cases in C. All example programs in this book are presented in the **C** language syntax, pseudo codes, and sometimes, in plain English sentences. Readers are also expected to have some basic knowledge of computer operating system and database theory. I would also assume that they are familiar with SQL that is widely used as the application programming language for relational databases. All database applications in this book are written in SQL.

Designing a new database management system depends on what services (aka, features) the database system would have for users. I will identify many fundamental features, and discuss them in-depth in this book, geared toward how they are handled in SQLite. In the design process, a database management system is split into many component subsystems. Different subsystem offers different services to database users and/or to other subsystems. Designers need to clearly specify interfaces to these subsystems. Finally, the subsystem softwares are developed, integrated, and tested to form a complete database management system that can be used in a production environment. A database management system should be simple, efficient, and reliable. In addition,

[2]Interesting copyright statement can be seen at http://www.sqlite.org/copyright.html.

the system should also be easy to maintain and enhance, and easy to port across different computer platforms. SQLite has these properties.

Contents

This book consists of 11 chapters. Chapter 1 discusses general concepts from the domains of operating systems and databases, and Chapters 2–10 SQLite internals. Chapter 11 provides some references to the literature.

Chapter 1 first presents a very general introduction to computer hardware and operating system. It reviews some essential concepts from the operating system domain, which are required for the development of database management systems. It briefly explains why we need operating systems in computers, and what they really do for computer users. It presents a bird's-eye view of typical operating systems, and provides readers an intuitive but overall understanding of working principles of the systems. It then explains why we need databases and database management systems. It explains what the systems really do for database users. It presents a bird's-eye view of typical database management systems, and provides readers an intuitive understanding of working principles of the systems. It touches upon almost all major component subsystems of a typical database management system. It introduces several key concepts and terminologies related to databases, especially those from the domain of relational database management. It talks about data model, relation, schema, integrity constraint, index, transaction, ACID transactional properties, and so forth. It explains various ways of manipulating data in databases. It also presents a few simple examples of SQL programming. These concepts are explored in later chapters, in the context of the SQLite database system. Overall, it is a very short tour of operating systems and database management systems.

Chapter 2 presents an overview of SQLite that is an SQL-based relational database management system. It introduces all components of SQLite in top-down fashion. These components are elaborately discussed in bottom-up fashion in later chapters. The chapter also talks about a few SQLite APIs, and presents some very simple SQLite applications. Overall, the chapter is a short tour of SQLite, and it sets up the ground plan for the rest of the book.

Chapter 3 presents the lowest level storage organization in SQLite. It defines naming conventions for database and journal files, and their formats. It talks about how a database file is partitioned into fixed size pages, who uses those pages, and organization of the pages. It also presents formats of journal files that are used to store log records produced by applications.

Chapter 4 discusses how SQLite manages user queries and updates via transactions. SQLite executes each and every SQL statement in the abstraction of a transaction, and ensures ACID

properties to transactions. The chapter discusses the file locking scheme that is used to control transaction concurrency, and the page journaling scheme that is used for recovery from transaction aborts and system failures.

Chapter 5 presents pager module. The module implements higher-level page oriented files on the top of ordinary byte oriented native files. It helps its clients to read and write pages from database files with relative ease. It is also the transaction manager and the log manager in SQLite. It decides when to acquire and release locks on database files, and when and what log records to write in journal files.

Chapter 6 presents tree module that resides on the top of the pager module. The tree module organizes raw database pages into B- and B^+-trees. SQLite organizes the content of a database into several trees. A B^+-tree stores in entirety a data table, and a B-tree an index on a data table. A tree stores keys and data in uninterpreted byte images. The chapter discusses various data structures that are used to store, retrieve, and manipulate data in trees.

Chapter 7 presents virtual machine (VM) module. The VM is an interpreter that executes programs written in the SQLite's internal bytecode programming language. It creates the abstractions of tables and indexes on the top of B- and B^+-trees. The chapter discusses various data structures that are used to interpret information stored in those trees. It presents five primitive datatypes, and discusses the formats of data records stored in table and index trees. (The pager, the tree, and the VM modules collectively implement the backend or the engine of SQLite.)

Chapter 8 discusses the frontend module. The frontend pre-processes all SQL queries, and produces bytecode programs that the engine can execute. The chapter presents four subsystems of the frontend, namely tokenizer, parser, optimizer, and code generator.

Chapter 9 presents the `sqlite3` user interface data structure. It presents an integration of all internal data structures presented in previous chapters. It gives readers a global picture of how different data structures are interconnected to one another and how they work in unison.

Chapter 10 discusses some advance features of SQLite. The notable ones are subquery, view, trigger, collation, pragma, autovacuum, wal journaling, and so forth.

Chapter 11 presents some references from the literature for further studies in databases and related systems.

Acknowledgments

I would like to convey my sincere thanks to Dr. Richard Hipp, the founder architect and lead developer of SQLite. He read a previous edition of this book and helped me correct many mistakes. He gave me many valuable suggestions to improve the presentation of that book. More importantly, he answered numerous questions very fast, often in minutes or hours. He permitted me to include any part of SQLite documentations in this book. This book, though not an exact copy, is in fact a synthesis of various SQLite documents and comments from source code, with due permission from Hwaci, Inc.

This book first appeared as an electronic form from O'Reilly Media, Inc (title: "Inside SQLite") in 2007. I would like to thank Brian Jepson of O'Reilly who helped me in preparing that book and taught me many valuable things about writing books. This book is an extension of the "Inside SQLite" book.

I started working on this print version while I was visiting the Applied Statistics Unit, Indian Statistical Institute, Kolkata, India for two short visits in February–March 2010 and January–March 2011, on vacation from Motorola Mobility, Inc (formerly a part of Motorola, Inc). Friends such as Palash Sarkar, Bimal Roy, Kishan Gupta, Subhamoy Maitra at the Unit were very helpful during my stay there.

I would also like to thank managers and colleagues at Motorola Mobility, Sunnyvale, California facility. They have been very cooperative and supportive to complete writing this book. At Motorola Mobility, I have been working on-and-off on SQLite for past eight years to develop database solutions for cellphones. In that process I interacted with many people and some of them such as Harish Sarma, Mani Bharadrajan (now at Google, Inc), Xuezhang Dong (now at Lab126), and Jun Lu (now at Paypal) highly encouraged me write this book.

Sibsankar Haldar
Sunnyvale, California, USA

Contents

List of Figures

Chapter 1

Database Overview

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- a database system, what it does, and why it is important
- some key concepts and terminologies pertaining to database systems
- SQL language and how SQL statements are executed by database systems
- transactions and their roles in SQL statement executions
- the roles computer platforms (operating systems) play on database systems

Chapter synopsis

This chapter presents a very general introduction to database theory, with an intuitive but overall understanding of the core subject matter of this book (i.e., of database management). It explains the purpose of databases and database management systems, and discuses what they actually do for database users. It introduces several key concepts and terminologies from the domain of relational databases and their management. It touches upon almost all major subsystems of a hypothetical database management system. It is very brief in presenting these concepts. You may consult text books on database theory to learn more about these and other related concepts.

A database management system operates either as a user-level standalone application program, or as a user-level (shared) library that is directly linked into standalone application programs. In either case, a database system depends on the various services offered by the underlying native operating system. Thus, before discussing database systems, this chapter provides you an introduction to operating system first. I assume that you are familiar with operating system concepts.

I review here, to brush up your memory, those operating system concepts that are essential for designing and developing database systems. I provide you a bird's-eye view of a typical computer platform, consisting of hardware resources, an operating system, and other utilities, and provide you an intuitive understanding of working principles of typical operating systems. If you are familiar with operating system concepts, you may skip or skim the following section. The section is a review from [12].

1.1 Introduction to Computer System

A *computer* is a powerful data manipulating engine that is energized by an electric power supply unit. Further, it is composed of many independent physical devices that we call *hardware resources.* Hardware resources in a computer minimally include a processor, a bank of main memory, and some input–output (I/O, for short) devices. Keyboard, mouse, display monitor, disk, printer, network interface card, etc., are examples of I/O devices.

Each hardware resource does some specific work such as storing, transmitting, or manipulating information. Hardware resources exchange information among themselves for accomplishing their individual work, and for effective and smooth functioning of the computer as a whole. They are connected to one another by electrically conducting wires via which they communicate among themselves. The most important hardware resource is the processor. It is the *data manipulating engine*, and is the overall in-charge governing activities of all other resources. The next important resource is the main memory, and it stores information that is needed by the processor and I/O devices. I/O devices are used by the processor to interact with the external environment as well as to store information for future retrieval. For example, a display monitor is used to display information in readable form, a disk to store information that would survive power disconnections, a network interface card to exchange information with other computers, and so forth.

The processor, once energized, after a very short initialization time perpetually executes machine instructions one after another until its power supply is disconnected. The processor cyclically fetches, decodes, and executes instructions from the main memory. Each instruction execution identifies a memory location from where the processor would execute the next instruction. In short, the processor goes on executing one instruction to the next, and the *flow* of instruction executions is determined by the outcome of the instruction executions themselves. However, in some occasions, some events in the computer may alter the normal execution flow abruptly. These events are called *interrupts.* When an interrupt occurs, the processor completes the on-going instruction execution, breaks the current flow of instruction executions, and starts a new execution flow by executing instructions from some other memory locations. In that case, we say the processor is handling the interrupt. On the termination of the interrupt processing the processor resumes the original

interrupted execution flow.

Users[1] intend to use computers to solve their various (computational) problems. They develop and run applications on computers to get their work done. Modern day hardware resources mostly provide complicated 'difficult to use' intricate interfaces, and maneuvering them by application developers would be a time consuming, daunting task. To alleviate their hardship, we need sophisticated special-purpose programs to maneuver hardware resources. Such special-purpose programs can be written by experienced developers who have extensive knowledge on hardware resources. These special programs would shield application developers from complexities of hardware resources, and simplify the use of hardware resources by applications. These individual hardware manipulating special-purpose program pieces are put together to form a monolithic piece of software to control all hardware resources in a computer, and the monolithic piece forms a part of a larger software called operating system.

In essence, an *operating system* is a software that controls all hardware resources, and that effectively provides users with a new software machine that is more convenient to use than the base hardware resources. It provides an execution environment in which users can develop and run their applications with relative ease. Two primary goals of an operating system are (1) user convenience in using computers and (2) effective and efficient utilization of hardware resources. Each operating system strikes a balance between the two.

An operating system sits in between user applications and hardware resources, and provides many services that the applications can avail at runtime. It relies on the underlying hardware resources to implement the services. It defines a set of *service access points*, and applications connect to appropriate access points to obtain desired services from the operating system. A connection call to a service access point is known as a *system call*. System calls are akin to making ordinary function invocations in the operating system space. They are the *programming interface* to the operating system. To make application development even more easier and to make applications portable across different operating systems, IEEE POSIX[2] standards defines a set of *application programming interfaces* (APIs, in short).[3] Applications, in general, do not directly make system calls. Instead, they invoke these APIs like ordinary function calls. These APIs, in turn, make system calls to obtain appropriate operating system services for applications. These

[1] A *user* is any person or device that uses a computer to carry out some data processing work.

[2] POSIX stands for Portable Operating System Interface for Unix. See http://standards.ieee.org/develop/wg/POSIX.html

[3] An *interface* is a contract between a system and its environment, aka, the system users. It specifies how the system interacts with the users. In software, it is a named collection of functions and constant declarations. It defines a protocol of communications between the users and the system, and defines behaviors for these functions. It describes the input assumptions the system makes on the environment and the output guarantees it provides to its users. It is a mechanism that unrelated entities use to interact with one other. An implementation of an interface constructs all the functions declared in the interface specification. The purpose of an interface is to minimize dependencies between applications and service providers.

APIs are available in the form of *shared libraries*. For example, in Linux systems, `libc` shared library contains implementations of read, write, sleep, etc., API functions for **C** language based applications. The libraries get embedded in applications at runtime.

By a *computer system* we mean a computer that is fitted with an operating system. That is, it consists of bare hardware resources and an interface operating system software. It, as a whole, provides a set of services for computer users to develop and run their applications. An operating system distribution normally comes with a set of utilities. *Utilities* are ready-made special-purpose standalone application programs, different from the operating system programs, that help users to maneuver the computer as conveniently as possible.[4] Compiler, assembler, loader, linker, debugger, database management software (which is the focus of this book), shell, common I/O operations, etc., are examples of utilities.

In the following two subsections, I briefly discuss some concepts and features of hardware and operating system, that developers of database management systems may need to be familiar with. I am very brief in discussing the concepts. Interested readers may consult any operating systems text book to learn more about these and other related concepts.

1.1.1 Hardware platform

Ordinary application developers normally write POSIX compliant applications that run on the top of the native operating system. They may know a bit of the operating system, but they do not need to know much about hardware platforms. But, some application (especially, system program or utility) developers may need to know about some hardware devices. A database management system is a utility that utilizes persistent storage devices. It is better that database system designers and developers have some basic knowledge about some hardware resources, especially about the working principles of external storage devices such as disks (that store data persistently). I briefly talked about the functions of processor and main memory before. In this subsection, I first talk about I/O controller that connects I/O devices to the host computer system and then talk about disk and its access overheads.

1.1.1.1 I/O controller

I/O devices are connected to the host computer system via I/O controllers. Some devices come with built-in I/O controllers. They carry out I/O operations on I/O devices on behalf of the main processor. Modern I/O controllers are mostly autonomous devices in the sense that they can carry out I/O operations independently without interventions from the main processor, even though their work assignments are controlled by the main processor.

[4]Users always interact with the computer by executing utilities and applications they themselves develop.

Each I/O controller cyclically accepts commands and input data from the main processor, executes the commands, and returns back output data and status information to the processor. The processor directs an I/O controller what to do by sending the controller an I/O command and input data. After receiving the command from the processor, the controller starts executing the command in its own way and own speed, and it may take a while to complete the command execution. When the command execution is indeed over, the controller sends the processor the status of the command execution and output of the command.

Almost all modern I/O controllers are fitted with interrupt circuits. When a controller wants the attention of the processor, it triggers an interrupt to the processor by raising signal on its (controller's) outgoing interrupt request line that is connected to the processor's interrupt pin. On receiving the interrupt, the processor suspends its on-going program execution, and starts a new program execution (in the operating system) that handles the interrupt. In the interrupt service handler, the processor collects status information and output data from the controller. Normally an I/O controller waits until the interrupt is serviced by the main processor. As soon as the interrupt is cleared, it can start new work.

1.1.1.2 Disk

Disks are widely used as online persistent storage devices. Logically, a disk is an array of fixed size *blocks*. A block is a sequence of bytes, and typically stores 1024, 2048, or 4096 bytes of data. Blocks are units of data transfer between the host computer system and the disk system. Disk is a *direct addressed device*, and the blocks can be accessed in arbitrary order at reasonable speed.

To understand how disks work, we need to understand their physical structure. A disk consists of many flat circular metallic plates called *platters*. Platters are made of highly polished glass or ceramic materials coated with a fine layer of iron oxide magnetic materials that can store and retain information in magnetically coded form. All platters are stacked on a single spindle. When a disk is in operation, the platters rotate at a high, but constant speed. For each platter surface, there is a read-write head that is responsible for reading and writing of data on the surface. All the read-write heads are attached to a single arm, and they are all aligned in a vertical line. The arm moves horizontally back and forth to position the heads on platter surfaces, and thereby, all read-write heads move together in unison.

Before we can use a disk to store information, we need to organize the space on platter surfaces. Each platter surface is partitioned into a number of concentric circular tracks. A *track* is a thin circular stripe of area on a platter surface. Tracks are concentric rings centered on the disk spindle. All tracks that are at one disk arm position are collectively called a *cylinder*. A cylinder contains one track per platter surface. Each track is divided into a number of *sectors*, and the track may

contain hundreds of sectors. Sectors are fixed size space. Sector size is a characteristic of the disk, and the size value is set when the disk is manufactured, and the value cannot be changed. Sector is the smallest unit of data that can be written to or read from the disk.

As mentioned previously, a disk space is logically viewed as an array of *disk blocks*. The operating system stores information on the disk in units of blocks. Each block is a contiguous sequence of sectors that reside on the same track. When a disk is formatted, the block size may be set to a multiple of sector size. Different blocks occupy different sectors. Each block is addressed by a tuple, consisting of a cylinder number, a track number, and a beginning sector number.

1.1.1.3 Disk access overheads

The main processor cannot directly access individual bytes of a disk block unless the block is copied into the main memory. Even if the processor needs a few bytes from a block, the entire block is read into the main memory. Similarly, it writes an entire block to the disk. The processor provides the block's tuple address to the disk's I/O controller, which in turn, makes the whole block (all its sectors) available in the main memory. There are three time components involved in a disk access. First, there is a time required to position the disk arm to the correct cylinder, and it is called *seek time*. Second, there is a time required for the read-write head to be at the beginning of the first sector of the requested block, and it is called *rotational latency time*. Third, the time required to transfer the data between the disk and the main memory, and it is called *transfer time*. To improve performance sector interleaving and sector skewing are normally done.

1.1.1.4 Disk operations

We assume that a disk controller supports at least two operations, namely read and write. A read takes a block address and returns the content of the block without altering the original content. A write takes a block address and a value, and overwrites the block content with the new value; it is an in-place update and the previous value is destroyed. Some disk drives cache blocks in their built-in memory. In such disks, the value of a write may not reach the disk surface immediately. For such disks, we need an additional operation to flush/sync the drive's cache on to the disk surface. Without the sync operation it may be risky to store databases there.

Nonatomic Write: A write on the disk surface is by no means an atomic, i.e., indivisible operation, and the write may fail (e.g., due to power failure) in the middle of overwriting the block's original content by the new one, thereby partially modifying the block. Such block content is considered corrupted, and hence is risky to use. Unfortunately, there is a possibility that a damaged block may not be detected even by the parity checking logic of the native operating system. Database system designers must take a note of this fact and take some preventive measures in the database management software to avoid or circumvent data

corruptions in databases as far as possible. ◁

1.1.2 Operating system

An operating system, along with an underlying hardware platform, essentially provides a software platform (i.e., a software machine) that appears to be independent of the hardware platform. It governs the usage of all hardware resources, optimizes their performance, and resolves access conflicts on them. For ease in development and maintenance, operating system software is partitioned into many component subsystems. Each subsystem may manage a specific hardware resource, and/or provide some services to the subsystem users. Notable subsystems are process system, virtual memory, and file system. We also have external interfaces consisting of system call, interrupt, and exception handling routines. (Exceptions are internal interrupts generated by the processor itself.) I assume that you are familiar with operating system concepts. You can refer to any operating system text book such as [12] to learn more about working principles of operating systems. In the following two sub-subsections, I review two concepts that database system designers and developers may need to know, namely process and file system.

1.1.2.1 Process and thread

Users run application programs (and utilities) such as database management systems to accomplish their tasks. The processor executes these programs. The operating system keeps track of activities of program executions in the abstraction of processes. A *process* is an entity that represents one program execution. The entity models the state of the program execution. At the conceptual level, the term process is commonly used to denote an entity that *as if* carries out some task (solved by a program), and that requests resources from and releases the acquired resources back to the operating system as it carries out the task. I refer to the term process to actually identify a program execution. A process is normally identified by a number called *process identifier* (pid), and different processes have distinct process identifiers. You may inquire about a process by using the pid. (In Linux systems, you may look at /proc/pid directory for more information about a process.)

The operating system starts a (new) execution of an application program by creating a new process. The pid of the new process is guaranteed to be different from those of other processes in the system. The process is allocated some main memory to hold the program and data. The operating system would allocate various resources to the process as the program execution evolves, and reclaim those resources when the program execution is complete or the process does not need them.

Each process is associated with an address space, called *logical address space*, from where the process executes instructions and accesses data. The logical address space of a process contains

programs and data from many sources. The address space includes an executable image of one application program and those of various libraries. They collectively form the code, aka, the application logic. These sources also supply program data, often called *static data*. In addition, a process will have different data, called *dynamic data*, when it executes the program. An address space is typically partitioned into a number of variable size blocks. In Linux systems, a process address space is composed of four blocks: code, data, stack, and heap. The latter two are dynamic data. The address space is mapped into the main memory by the memory management component of the operating system.

In traditional operating systems, two program executions (even if, they involve executing the same program) are represented by two different processes. Resources acquired by one process cannot be used by the other process. In some modern operating systems, address spaces can be shared by many strands of executions. That is, they (operating systems) allow concurrency within a single address space, and many strands share the same address space at the same time. Consequently, a process can execute multiple sequences of instructions from its address space. Each strand of execution sequence is popularly called a *thread*, and it executes the same program concurrently with other sibling threads in the same process. This one is called a *multithreaded process*. Each thread has a distinct identifier called *thread identifier* that is different from its process identifier. Thread is the unit of work allocations, though process is the unit of resource allocations in multithreaded systems. That is, threads execute the same application program on their own, but they acquire resources via the container process, and they can share the resources irrespective of which particular sibling thread has acquired the resources.

1.1.2.2 File system

Users create and store information in a computer system for later retrieval and manipulation. The system stores information in persistent external storage devices such as disks that can retain the information across power disconnections. Users see their stored information in the abstraction of files. An operating system implements a software layer, called *file management system* (*FMS*, in short), to bridge the gap between the user view of the storage system and the physical storage devices. The FMS helps users to manipulate information, stored in physical devices, through device-independent file abstraction by hiding physical properties of storage devices from them. For example, disk and flash devices work differently, but file system users are not normally interested to know the differences. They access files on either device in the same way via the same set of APIs.

Loosely, a file stores a collection of related information in the form of a sequence of bytes. Applications can read and write bytes from the file; they may specify starting points from which

certain number of bytes to be read or written. The FMS helps applications in storing and retrieving bytes from files. Users are not normally concerned about how and where the file contents are stored in storage devices. They identify a particular piece of information by a filename and a relative position within the file. The filename and the relative position are translated to appropriate physical locations in devices by the FMS. As far as the file content is concerned, the FMS is a passive entity. It does not see any structures in the content. It treats the content as a byte string. It is the user application that sees structures in the content. The application considers the file as a collection of records of different types. For example, a database stores structured information in files and only the database management system understands the structure.

The management of lower-level device space is taken care of by the FMS. As I discussed in Section 1.1.1.2 on page 5, the FMS sees a disk as an array of fixed size blocks. It keeps track of which blocks are in use and which ones are free. Initially, all blocks are free. When needs arise, it allocates free blocks for various purposes. When a file is created or expanded due to an append of new data to the file, new blocks are allocated to the file. When a file is deleted or truncated, blocks from the file are released to the device. Space allocation and deallocation are costly operations as the FMS needs to maintain various data structures on the storage device. You may note that accessing a disk is a very slow operation compared to that of the main memory.

There are a large number of file management systems that have been developed for different storage devices. Here I briefly discuss the disk based Ext2 system of Linux. A file is represented by file control information that is stored in an *inode* object on the disk. The Ext2 system partitions each file into fixed size logical blocks, and the logical blocks are stored in any physical data blocks (*d-blocks*)—one logical block in one d-block and vice versa. An inode object stores about 12 direct pointers to d-blocks that store the first 12 logical blocks. Other d-blocks are accessed via various index blocks (*i-blocks*). One entry in the inode uses a single indirect index, one entry double indirect index, and another entry triple indirect index. For very large files, we need more indirect i-blocks. The inode also keeps other information such as file owner, access permission, size, various timestamps, and many more. The d- and i-blocks are normally scattered across the disk.

Before using a file, a process needs to tell the operating system to open the file for it. Linux systems, on a successful open, return a *file descriptor* to the process, which is a small integer value. The process uses the descriptor as a handle in future operations on the file until it closes the file. Initially, the file (the inode and the content proper) is in disk. When the process reads and writes the file, the FMS reads and writes disk blocks. The FMS maintains an in-memory cache of disk blocks to speedup accesses to the blocks.

There are several things that may become unpredictable while a process manipulates a file. A read operation execution does not change the file content, but a write operation execution does.

(1) A write on a file reaches the in-memory cache (of the FMS), but may not reach the disk surface immediately. The data and index blocks are normally asynchronously written to the disk by the FMS. The blocks may not be written to disk in the order expected by applications. (2) The file content proper may not be in agreement with the file inode. For example, the actual size of the file is not yet updated in the file inode; or the size is updated but the data blocks are not yet linked to the file. (3) In the worst case, some blocks may be corrupted due to power failure during transferring of data to the disk surface. Database system designers and developers must be aware of these complications, and they must have a plan to handle these aspects in database management software. Linux systems implement `fsync` system call that forces both inode and file content proper on the disk surface; but, unfortunately, the operation is not atomic, i.e., not indivisible.

There is another problem. Many processes can concurrently open the same file and work on the file. Concurrent conflicting operations (e.g., writes) on the same file may create some additional problems. The Ext2 file system provides a single view of the file to all processes: a write to the file is immediately visible to all processes that have open the file. Concurrent writes may be arbitrarily interleaved; the Ext2 does not ensure atomicity of reads and writes. In case finer synchronization is necessary between reads and writes, the processes need to do so explicitly by themselves. Linux systems support file locking primitives that may be used for synchronization purposes. A process can set read/write locks on an entire file or on a region of the file to prohibit others from applying conflicting operations on the (region of the) file.

1.2 Introduction to Database System

This section introduces some basic database concepts, their implied meanings, and terminologies, especially those from the domain of relational database management. More on relational databases is presented in Section 1.3 on page 28.

1.2.1 Data item

A *data item* is something that carries or holds a piece of information, and the information represents the state of some physical or logical entity. The information is encoded in the form of a *value* of the data item. That is, the meaning we assign to the value is the information. However, we often use the three terminologies, i.e., data, value, and information, interchangeably to mean the same. A data item can be an integer, a name of a person, an address of a house, a blob, a table, or anything like that. The size of a data item is called its *granularity*. For example, an integer data item can be a one, two, four, eight, etc., byte number. The granularity specifies how much information a data item can carry. We often refer to data items (or simply, data) in this book in an abstract sense without specifying their meanings or granularities.

In essence, each data item resides in some space and has a name or address by which we reference the item. What values can be stored in a data item are determined by its type. The type also defines various operations that we can apply on a data item of that type to manipulate its value. At the least, we should be able to read the current value and overwrite it with a new value.

1.2.2 Database

A *database* is a single repository that contains many individual (*persistent*) data items. It is however a lot more than a mere repository. A data item normally does not exist in isolation in a database. Data items are related to one another. Thus, a database contains a collection of related data items. The relationships on data items collectively satisfy some *integrity constraints* that are assertional conditions on the values of the data items. A *database state* is determined by the values of all data items in the database at any one time. A database state is said to be *consistent* if the values of all data items in that state satisfy all the integrity constraints defined on the database.

The reason behind constructing a database is to help people to keep track of various important information there. A database normally stores information about some real world system, and a consistent database state represents a particular real world scenario. For example, a university database may store information about all students, courses, faculties, employees, and so forth. Users apply various operations, called *database operations*, on a database to retrieve or delete information from the database, and to store new information or modify existing information in the database.

In computer systems, a database normally resides in one or multiple ordinary native files, and these are often called *database files*. When operations are applied on databases, the contents of these files are manipulated to reflect the new database states. That is, database (aka, higher level) operations are translated into file (aka, lower level) operations by a database management system.

1.2.3 Database applications

Users create, store, retrieve, maintain, and manipulate data in databases. They may store data on durable storage devices such as disks (in the abstraction of one or more files), and they may manage the data on their own manually by using various system utilities such as shell script, file editor, etc., in an ad hoc manner. This approach, in old days, is used to be called a *file processing system*. But, the approach soon becomes cumbersome, and a risky way of manipulating databases, especially by naive database users. The alternative is to develop and run specially designed applications, called *database applications*, against databases. These applications can be developed by database experts and can be used by ordinary users with relative ease without much botheration about how data items are stored in the databases. Modern day databases are accessed by executing database applications. Anyone may execute these applications with relative ease. The applications

manipulate a database only by applying well defined operations on the database. Users run these applications independently, and often concurrently. They apply queries and updates on databases to read and manipulate, respectively, data stored there.

1.2.4 Data model

If data items in a database are stored haphazardly without a correlation among them, manipulation of data may become a complex, tedious, error prune task. Applications may become tuned to unstructured/unorganized data, and application development becomes cumbersome. For ease in their management, data items are structured in some standard form before they are stored in a database. In addition, related data items are clustered together so that they can be found easily, and can be iterated over efficiently. The structuring and organizing of data is even more desirable when the same database is accessed by many independent applications written by different developers. All applications should see the same data in the same (or similar) structure.

We need a tool to represent all data items and their relationships. The very first thing we need to do in data processing is to create a model of user data. A *model* creates a representation of the 'user view' of data. It identifies what data items will be stored in the database, defines their structures, and sets up relationships among them. A *data modeling scheme* consists of a collection of concepts that helps in describing data items, and a collection of rules that helps in combining data items into different classes. All data items in a given class have a common set of properties, and the properties collectively define the *schema* of the class. The schema is a template that specifies what kind of data can be in the class and it is a gross description of structures of data items in the class.

Different data models have been invented in the past decades. Different database systems use different data models to describe, structure, and organize data items. Hierarchical, network, relational, and object data models have been used successfully for such purposes. The most widely used model is the relational data model. In this book I deal with the relational data model only.

1.2.4.1 Relational data model

In the relational data model, the most fundamental concepts are entity and relationship. An *entity* is an abstract object of some sort describing some real world (physical or logical) thing distinguishable from other entities, and a *relationship* is an association between two or more entities. An entity is something about which users want to store information in the database. Relationships setup connections between different entities. The relationship information is as valuable as the entity information is. For example, in a university database, a particular student or a particular course is an entity, and an enrollment sets up a relationship between the two entities. In the

relational data model, a data item can be an entity data or a relationship data or a collection of them. I discuss it in the next sub-subsection.

1.2.4.2 Entity-relationship model

As mentioned above, data items in a database do not exist in isolation. They form cohesive collections of data items. Entity-relationship (ER, in short) model is one way of representing the cohesiveness among data items, and it is the most common model used in practice for organizing data at the conceptual level. The ER model defines the connections between entities and relationships graphically at the conceptual level of database design. The graphs are called *ER diagrams.* (The model does not however define operations on any data item.) The model uses three principal elements: entity set, attribute, and relationship set.

In the ER model, an entity is characterized by a set of *attributes.* Each attribute has a distinct name in the attribute set. The attributes describe the characteristics or properties of the entity that users are interested in. A collection of similar entities (having the same attributes) form an *entity set* or *entity class.* In the ER model, an entity set is represented by an abstract name with all associated attributes. The name is a template that describes what kind of entities can be in the entity set. For example, in a university system, a template name 'Students' defines common properties of all students in the university, and another template 'Departments' defines common characteristics of all departments. All students in the university database are in the same entity class. Figure 1.1 depicts a typical template for the Students entity class; all attributes are shown in ovals and the class is in rectangle. The sketches in the figure are ER diagrams. The figure also depicts the Departments entity class.

Figure 1.1: ER diagrams of two entity classes.

By definition, elements in a set are distinct. Often only a few attribute values can be used to test the distinctness. For any given entity set, the attributes whose values distinguish one entity from another in the set are collectively called a *key* for the entity set. In Fig. 1.1, the key is shown by bold attributes. For the Students ER diagram, **SID** (student identifier) is the key, and for the Departments **DID** is the key.

A *relationship* is an association among two or more entities. (The entities need not come from the same entity class.) A collection of similar relationships form a *relationship set* or *relationship*

class. That is, a relationship class describes a particular kind of associations among elements from some entity classes. In the ER model, a relationship class is shown by a diamond box (see Fig. 1.2). An n-ary relationship class has n associated entity classes, and may have its own attributes. Shown in the figure, `Admitted-to` is a binary relationship class (between Students and Departments entity classes) that has `doj` an attribute indicating a student's "date of joining" the department. You may note that the same entity class can participate in many different relationship classes.

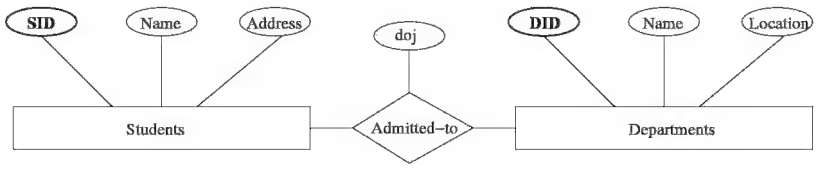


Figure 1.2: ER diagram of a relationship class.

1.2.5 Relational database

In the previous subsection, I have presented ER model to represent a "user view" of a data world. I need to translate the elements of the model (namely entity sets and relationship sets) into database objects, namely relations. In a relational database, all database objects are some kind of relations. In mathematics, a *relation* is a set of ordered pairs. Here, for our purpose, a relation at the conceptual level is a *table* (aka, a two dimensional matrix) with rows and columns.[5] Relations are units of basic elements in relational databases. Both entity sets and relationship sets are represented as separate relations. Users see their data in a relational database as a finite collection of relations, and nothing else. In essence, relations are the only (higher level) data structure that is used to represent all information in a relational database.

There are known techniques to derive definitions of relational schemas from ER diagrams. They also help you in reducing or eliminating redundant information from databases by decomposing relations into smaller relations in standard normal forms. I do not discuss these techniques in this book. Any standard text book on database theory has one or more chapters on this topic.

Let us study the table concept a little deeper. Columns are often called *attributes* or *fields*, and rows *tuples* or *records*. Attributes serve as the names of the columns, and each attribute has a distinct name. Each column of a table identifies a single property for all rows in the table. The number of columns determines the *degree* or *arity* of the table. Every table is characterized by a *schema* that describes the column attributes of the table plus other properties. Each cell in the table stores a *value*. A row can hold a single value for each attribute from a predefined set of values called *attribute domain*. Each row represents an entity or a relationship by the values of

[5] Although technically table and relation are two different terminologies, I use the terms interchangeably to mean the same.

all attributes. You may note that a table can be empty, that is, it does not have any row, but it must have a schema. This is because without the schema we do not know what kind of rows can be stored in the table. The number of rows in a table is called its *cardinality.*

All rows in a relation have the same format, i.e., the same sequence of typed values. Figure 1.3 presents a typical instance of the aforementioned Students relation. The top line in the figure identifies the attribute names of the relation and is not a part of the actual relation. The arity of the relation is 3. The **SID** is the key to the relation that uniquely identifies all rows. You may note that the values in a particular column in all rows need not be different. There are four distinct rows in the relation, that is, the cardinality of the relation is 4.

SID	Name	Address
1001	Sibsankar	Sunnyvale, California
1002	Richard	Charlotte, North Carolina
1003	Richard	Sunnyvale, California
1004	Sibsankar	Charlotte, North Carolina

Figure 1.3: A typical instance of Students relation.

A major strength of relational database is that it supports many simple but powerful higher-level operations on relations. The operations are often called *queries* in the database world. They are expressed in *query languages.* Unlike procedural languages, queries are normally declarative. They specify what is to be done. You formulate them in the query language, and may not need to worry about 'bits and bytes' of data items. The query processing system is responsible in executing queries to obtain the desired results from the database. The *structured query language* (SQL, in short) is the most widely used query language in modern relational databases. The single language is used both to define database objects such as table and to manipulate the objects. More on SQL is presented in Section 1.3.3 on page 32.

1.2.6 Integrity constraints

As mentioned previously, a *database state* consists of values of all data items the database has at any one time. The values of data items may be related to one another, and the relationships satisfy some properties expressed in terms of the values. The properties are called *invariants* or *constraints.* They enforce relationships between information stored in various data items. In the relational databases, an *integrity constraint* is expressed as a condition on (partial) values of rows from one or more relations. Integrity constraints are normally formulated based on the semantics of real world data the database encompasses. A database state is said to be *valid* or *consistent* if the state satisfies all (i.e., does not invalidate any) integrity constraints; otherwise, the state is *invalid* or *inconsistent.* Integrity constraints are specified when schema for relations are defined or altered,

and are checked and enforced when relations are modified. Insert, update, and delete operations on relations should be rejected if they violate any integrity constraint. A constraint can enforce a domain integrity (what values are permitted for a column), row integrity (defining permitted combinations of column-values to construct a row), referential integrity (relating row values of two relations). In the following sub-subsections, I talk about a few widely used integrity constraints in relational databases.

1.2.6.1 Domain constraint

Given a relation, each of its attributes has a datatype that defines the set of values the attribute can have. This is called *domain constraint.* In addition to a domain constraint we can specify what legal ranges of values from the domain the attribute can have; this constraint is often called *check constraint.* For example, we can declare an attribute as integer first and then specify that the values are greater than zero. The former is a domain constraint, and the latter a check constraint. When a row is inserted in the relation, the system needs to make sure that the values are valid, i.e., they do not violate the domain and check constraints.

1.2.6.2 Not NULL constraint

NULL is a special SQL value that could be substituted for real values of any attribute. The SQL NULL value is distinct from all valid values for a given datatype. It represents an unknown or a "don't care" value. You may note that the NULL is different from zero or blank space as used in some programming languages. If a column is specified 'NOT NULL', the database system must prohibit storing the NULL value in it.

1.2.6.3 Primary key constraint

Given a relation, a set of columns for the relation is said to be a *key* if no two distinct rows will ever have the same values in all the key columns and this is not true for any strict subset of the columns. (It is permissible to have the same values for some key columns, but not for all.) Thereby, a key uniquely identifies a row in the relation. If the second part of the key definition is not true, we call it a *superkey.* If there are multiple keys for a given relation, they are called *candidate keys*, and one of them is designated as the *primary key* by the schema designer. The SID column in Fig. 1.3 on page 15 is a key for the Students relation. There is no other key, so SID is the only candidate key, and hence the primary key. The set {SID, Name} is a superkey. Columns in the primary key cannot have NULL values.

1.2.6.4 Unique key constraint

Primary key is unique by definition: no two rows can have the same primary key value. There cannot be two primary keys for a relation. There is an alternative way to specify multiple uniqueness. The schema designer may specify any set of columns of a relation unique: no two rows in the relation can have the same value for the column(s). Unique columns can though have NULL values; two NULL values in the same column are considered distinct in such case.

1.2.6.5 Foreign key constraint

A set of columns in one relation may be used to refer to tuples in another relation. The set must correspond to the primary key of (or all the columns are declared unique in) the second relation. The set is treated as a symbolic pointer from the first relation to the second one. The columns in the set are collectively called a *foreign key* with respect to the first relation. If for every value of the foreign key in the first relation, there is a tuple in the second relation that has the same value in the foreign key column(s), we say that the database satisfies the foreign key constraint between the two relations. If all foreign key constraints in a database are enforced, we say the database achieves *referential integrity.* These constraints effectively enforce 'existence relationships' between relations.

1.2.7 Indexing

An *index* on a set of data items is a data structure that helps faster lookup of data items in the set by their values. The index structure stores control information that directs the data lookup via the most efficient route. In relational databases, we define indexes on columns of relations. An index is always associated with a single relation, but a relation can have none, one, or many indexes. Although indexes are not a part of the relational data model, they are widely used in databases to speed up searching of rows in relations (for existence of particular values). Indexes can make a huge difference in speed when a query does not need to scan the entire relation. (A full *relation scan* retrieves all tuples from the relation, and it is detrimental to performance and can slowdown query processing speed considerably.)

An index is defined on a set of attributes from a base relation. The attributes are collectively called the *index search key*, or simply the index key. Any subset of schema attributes (of the base relation) can be used as the index search key, and the search key need not be a key for the base relation. (Practical database systems though may restrict the number and the types of attributes from occurring in index keys.) An index on a relation speeds up selection of rows on the basis of the values for the search key fields. Each index structure stores the corresponding search key values and references/pointers to tuples in the base relation. An index is said to be a *primary index* if

the index key is the primary key of the relation. Otherwise, it is a *secondary index*. Most database systems allow many secondary indexes on the same relation.

Indexes are created and deleted by users, but are maintained by the database system. There are no read/update operations that database users can directly apply on indexes. The database system automatically maintains required searching information in indexes as rows are inserted, deleted, and updated in the base relation. The index information is stored in the database itself, either as separate relations or as some other data structures. You may note that the presence of many indexes on a relation may substantially slowdown executions of insert, update, and delete operations on the relation.

Two types of indexes that are widely used in relational database systems are hash and B-tree, and their variants. A *hash index* is a fast point look up index. A hash index is normally implemented as a collection of buckets holding search routing information (and pointers to tuples in the base relation). The index implements a hash function that is used to map any search value into a bucket where the actual search is performed for the given value. Hash indexing is not good for searching the base relation for rows in the range of two given search values. A *B-tree index* is an ordered tree. It maintains a predefined sorting order on search key values. It allows point lookup searches as well as selecting a range of values (in the sorting order) from the base relation without having to scan the entire relation. Each node in the tree contains data and child pointers. The search key value in the data is used for routing the traversal of the tree. A variant of B-tree, called B^+*-tree*, is one in which only leaf nodes contain data items (or pointers to base tuples), and internal nodes only contain search routing information and children pointers. Given a search value, the non leaf nodes direct the search to an appropriate leaf node where the actual search is performed.

1.2.8 Database management system

There are two ways to maintain data items in databases. One (old way) is to custom-tailor applications so that they 'directly' manipulate data in databases. The applications directly read database files, manipulate information that is just read, and write the new information back to the files. Applications become tuned to the internal structures and organizations of data in databases. That is, they become dependent on file formats. It might be difficult for application developers to maintain large databases on their own. It will be very hard to ensure database integrity constraints, especially when applications or the native operating system may fail in the middle of database updates. Recovering a database into an acceptable consistent state becomes a phenomenal challenge to application developers. The task becomes even more cumbersome when the database size grows and many users access the database concurrently. There is a high possibility that data items lose their integrity constraints and their cohesiveness due to concurrency even when applications are

absolutely correct. Although some application developers may be able to write super efficient data manipulation applications, in general, they make inefficient accesses to the database. In addition, application development time could become significantly long, because developers need to worry about 'bits and bytes' of every data item in the database. Applications become unnecessarily large. If we ever decide to change internal structuring of data, all applications may need to be rewritten, and this may not be cost effective.

The other alternative (modern way) is to use a separate, dedicated software system via which applications access their databases. The system defines many higher-level operations (in contrast to read, write operations supported by the native file systems) that applications can use as database APIs. Expert database system developers can write efficient data access and manipulation programs to implement these APIs. They can take care of issues related to concurrency control and recovery from failures. Database applications are decoupled from lower-level management of data. They become independent of file formats and file management, and they do not have file handling functionalities built-in them. Thereby, application development time could be significantly reduced, and so is their size and maintenance cost.

A *database management system* (DBMS, in short) is a software that primarily helps users to store, retrieve, and manipulate data items in databases as conveniently as possible. The concept of DBMS has been conceived and developed to overcome many problems we envisage in file processing systems. The DBMS provides a platform (a new software environment residing on the top of the native operating system) in which users can develop and run their database applications with relative ease (see Fig. 1.4). Users see a database as a cohesive collection of higher-level data items such as relations instead of bits and byte strings in ordinary native files. They interact with database applications that, in turn, interact with the DBMS that, in turn, accesses data items from database files. The applications do not do any file processing, i.e., they do not have file processing logic embedded in them. When needs arise, they query databases via the DBMS to read data out of databases, and when they are finished with the data, they may write the data back into the databases, again via the DBMS.

In a nutshell, applications see a database at the conceptual level as a collection of abstract data items (e.g., tables), and manipulate data by applying higher-level operations (APIs) on the database via the DBMS. Applications are not involved in the management of underlying resources such as files. They are only concerned with conceptual data, and they apply higher-level operations on the database to manipulate the data values. The DBMS acts as a black box in translating those operations and transferring data between databases and applications. It actually does all file processing work to access data items in database files. A typical DBMS allows users to access data in an organized, efficient way. The DBMS maintains structures of data, their relationships,

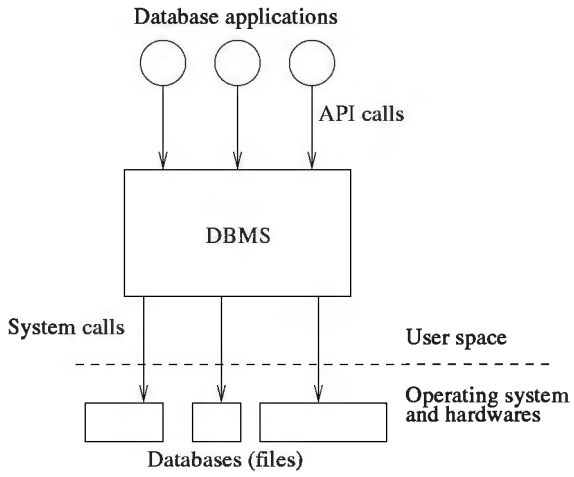


Figure 1.4: A database management environment.

and various constraints. It also takes care of concurrency control and failure recovery and data protection the applications envisage at runtime. In the rest of this subsection I briefly discuss how a DBMS carries out its various works.

1.2.8.1 Transaction

Users see data items and their relationships at the conceptual level. DBMS stores the data in database files, and maintains the logical structures of data items and their relationships as defined by users. A database stores information about some real world system. The database must accurately reflect the state of the real world system. Users change the state of the database by executing applications on it. The applications read and modify related data items in the database to mimic the evolution of the real world system. We may need to be very careful when modifying data items. Data items must be mutually consistent. Failures of applications and interleaving of operations form multiple applications must not produce an inconsistent or unexpected database state or unexpected application behavior. The DBMS must not malfunction correct applications.

A database that is not currently accessed by any user is assumed to be in a consistent state. Read operations do not change database states, but write operations do. When a user applies write operations on a database, the database transits from one consistent state to another. Often, the state transition moves through temporary inconsistent states, though only for a short period of time. There are two concerns. (1) If the (temporary) inconsistent database state is seen by other applications, they may malfunction though because of no defects of theirs. (2) If the application fails or system crashes or power supply fails, the database unwittingly remains in the same inconstant state. But, the DBMS does not know this inconsistency unless it is explicitly informed. To circumvent temporary inconsistencies in a transparent manner, a sequence of database operations

from one application are grouped together that collectively preserve database consistency. What it means is that when the group of database operations is performed in isolation (of other such groups), the operations collectively preserve database consistency. The group is referred to as a *transaction.* You may think a transaction as a higher-level operation that is composed of a collection of (lower-level) database operations, and when it is executed in isolation (from other transactions), the lower-level operations collectively transform a database from one consistent state into another. As long as a transaction is active, i.e., not complete, the DBMS suspects that the database may be inconsistent and takes necessary precautionary measures. Transactions are logical units of work in the DBMS. The database must be consistent across transactions.

Transaction: A *transaction* is a sequence of actions on data items. The duty of the DBMS is to make the sequence of actions indivisible and instantaneous (that is, all actions happening together at the same time instant) to an outside observer. The heart of modern day database programming model is transaction. It is the single most important concept in database management as of today. ◁

Transactions are initiated by applications. What database operations can be in a transaction is determined by application developers, and not by the DBMS. The DBMS assumes that transactions are units of consistency. What I mean by this is that a transaction that is started with a consistent database, when terminates, it produces a new consistent database if it is not interfered by other transactions. But a partially executed transaction may produce an inconsistent database. Transactions are normally representations of real world events. The events either take place or they do not happen; there is no in-between state. If a transaction fails (aka, aborts), the DBMS must remove the effects of the transaction from the database and restore the database state to the one as of the start of the transaction.

Each DBMS provides an interface using which an application can start a transaction, and execute database operations as a part of the transaction. Later, the application either commits or aborts the transaction. The commit operation advises the DBMS to make all changes, if any, done by the transaction permanent in the database. The abort operation advises the DBMS to remove the transaction's effects from the database. In either case, we say that the transaction is complete. The application would start another transaction for new work. Applications can apply operations on databases only via transactions. In the next sub-subsection, I discuss key properties of transactions.

1.2.8.2 ACID properties

As mentioned earlier, a partially executed transaction may produce an inconsistent or unexpected database state. Thus, during a transaction execution, if the application or system crashes, the DBMS must recover the database into an acceptable consistent state first upon system restart

before starting its normal business with the database. The application however is not involved in the recovery operation. The DBMS alone must assure transactions this feature and other such ones.

The feature a DBMS normally ensures to transactions is called ACID (a short acronym form for combined atomicity, consistency, isolation, and durability) properties to avoid producing inconsistent and unexpected database states and transaction behavior. When these properties are assured, manipulations of databases become easier by application developers. These four properties are discussed in the following bullets.

- Atomicity: Given a transaction, all its operations either all happen or none happens. There is no in-between situation when the transaction is complete. The database must have the effects of all operations from the transaction, or 'as if' the transaction is not executed at all. That is, failed or aborted transactions have no effects on the database. (Some authors refer this property as failure atomicity.)

- Consistency: A transaction, when executed to the completion, always transforms a consistent database state into a new consistent state, and the DBMS does not alter or malfunction the transaction behavior. The behavior of the transaction must not become unpredictable in the presence of multiple transactions and various failures. You may note that consistency is a property of applications and databases, and the DBMS does not alter it.

- Isolation: Even though many transactions may execute concurrently and they each may perform many database operations, the net result is that the transactions are *as if* executed in a serial order, i.e., one after another without interleaving of their operations. Said differently, each transaction is *as if* executed instantaneously relative to the others. In other words, all operations of each transaction happen 'together' instantaneously.

- Durability: Effects of successful (i.e., committed) transactions must become a part of the database. Once the DBMS has committed a transaction, it must be able to derive the effect of the transaction after any subsequent system failures.

Ensuring ACID properties to transactions is the main objective of a DBMS. It, in addition, can enforce that database integrity constraints are respected by transactions. If they violate any integrity constraint, they are forcefully aborted, i.e., rolled back (the nothing part of the atomicity property). Apart from this the DBMS does not know the semantics of data items, and does not have control over any transaction composition and what the transaction does with data. It is the application developers' responsibility to maintain database consistency across transactions. The

ordering of operation executions in a transaction is determined by the application logic, and the DBMS respects the ordering of operations.

In essence, an application creates a transaction on a database, applies operations on the database through the transaction, and finally commits or aborts the transaction. If the application commits the transaction, all its changes become effective in the database. If it aborts the transaction, the changes are annulled from the database. An abort can be initiated by the application or force initiated by the DBMS. A commit decision however is always initiated by the application.

1.2.8.3 Transaction management

You may note that applications run as single- and/or multithreaded processes. Consequently, transactions are executed by native application processes (or their threads). Though processes are units of work assignments in operating systems, transactions are units of work assignments in DBMS. The operating system tracks down activities of a process by using a process identifier, a process descriptor, and many other related resources in the operating system space. The operating system does not understand transactions that are executed by processes, and cannot ensure ACID properties to them. The DBMS alone needs to ensure the ACID properties. A single process or thread can have multiple concurrent active transactions that may access the same or different databases. The DBMS needs to track down activities of all transactions, and it does so by tracking down operations of each transaction by using a transaction identifier, a transaction descriptor, and other transaction control data structures. The operating system does not know this and transaction management is a user-space task carried by the DBMS.

Except the consistency property, applications are not involved in ensuring the other three properties, that is, they do not contain program fragments to ensure the other properties. The DBMS alone ensures these properties. Ensuring these properties is a considerable amount of work because no database operation is truly atomic. (Even an execution of an instruction by the processor is not truly atomic.) Atomicity of a transaction is an illusion that database users see. Designers of DBMS software envisage two fundamental questions in implementing the ACID properties. (1) When can the results of a transaction be made accessible to other transactions? (2) How is partial execution of a transaction nullified (i.e., rolled back)? There are two things the DBMS is responsible as far as transaction management is concerned: (1) concurrency control and (2) recovery from failures. Concurrency control coordinates or regulates accesses from transactions to databases. Recovery deals with restoring a database to an acceptable consistent state upon a failure. In the next two sub-subsections, I briefly discuss two schemes that are widely used to achieve the ACID properties.

1.2.8.4 Concurrency control

Concurrency is the notion of performing two or more actions simultaneously or in some interleaved manner. Concurrent executions of database applications are essential to enhance resource utilization and to reduce query response times. But, concurrent executions of applications that make conflicting accesses to the database may lead to an inconsistent or unexpected database state and/or may exhibit unwanted behaviors of applications. When applications are executed in isolation, i.e., one after another, each application sees a consistent database state produced by its predecessor and hence they all behave expectedly. When many applications are executed concurrently, they may interfere one another in unexpected ways. There is a possibility that some applications see intermediate transient inconsistent database states produced by others. If an application sees inconsistent database states, it may behave unexpectedly even though it is absolutely correct. The DBMS must not cause correct applications to malfunction. It is the duty of the DBMS alone to ensure database consistency by controlling application concurrency. *Concurrency control* is the activity of coordinating accesses from applications to data items in databases. It is a prevention mechanism that ensures one application's work is not interfered with others'. It must not have high overhead such that it would cause a substantial slowdown of normal application execution speed.

A DBMS makes all accesses (reads and writes) from applications to a database in the abstraction of transactions. It makes sure that the execution of many concurrent transactions is equivalent to some *serial* execution of the same transactions. This is called the serializability theory. The DBMS controls scheduling of database operations from transactions to achieve a serialized execution (of transactions). The concurrency control subsystem of the DBMS is responsible for ensuring transaction isolation and atomicity properties. Most DBMSs use some kind of locking schemes for this purpose to isolate conflicting operations affecting one another. There is a *lock manager* (a subsystem of the DBMS) that maintains locks on data items. A read of a data item does not change the value of the item, but a write does. Many transactions can read the same data item concurrently without affecting one another. But, writes on a particular data item are excluded from one another and from reads on the data item. The simplest lock based concurrency control protocol is the following. Before reading (or writing) a data item, a transaction obtains a read (or write) lock on the data item. A write lock is incompatible with any other lock, but many read locks can coexist. (Sophisticated transaction management systems such as [26] employ many other lock types.) A transaction that fails to obtain a lock must wait until the lock is granted, or it aborts itself to rollback its changes. The transaction releases all its locks at the time of its commit/abort. This is known as strict two phase locking protocol. The protocol produces serializable executions of transactions.

Serializability: A history H of a set of transactions is *serializable* if all committed transactions in H issue

the same database operations and receive the same responses as in some sequential history that consists only of all committed transactions in H. $\triangleleft$

1.2.8.5 Failure recovery

Concurrency control ensures correct behavior of (concurrent) transactions when no failure occurs in the system. This is a perfect world scenario. In practice, several things would go wrong at runtime. For example, a transaction may voluntarily abort its execution without any advance notice. In this case, the DBMS must annul all changes made by the transaction and recover the database to the state as of the start of the transaction. The DBMS must also ensure transaction atomicity even when the computer system or processes (executing transactions) crash in the middle of transaction executions. The DBMS has a bigger role to play. It ensures that the database is not corrupted by software, hardware, or power failures. It uses recovery techniques to repair the database after failures. In short, at any given moment of time, the DBMS must be able to recover a database without advance notice. The recovered database must have updates of all committed transactions, and must not have updates from any non-committed transaction.

The recovery subsystem of the DBMS is responsible for ensuring transaction atomicity and durability properties. For the recovery purpose, the DBMS maintains additional information that is separate from the database content proper. It maintains a 'journal' (often called a *log*) to store this information. A *journal* is merely a single sequence of records, called *log records,* that are stored in one or many sequential files. The journal keeps an account of all write operations carried out by transactions on the database. There is a *log manager* (another subsystem of the DBMS) that maintains log records in journal files. Each log record has a header that is a descriptor for the record, and a payload containing redo-undo information for the operation that has produced this log record. The log manager prepares the descriptor, and the log client the payload.

For efficiency reason, normally a part of the journal is kept in stable storage and the remaining part in the volatile main memory. This may create some problems (e.g., lost log records) in case of system failures. A DBMS normally follows two rules to overcome these problems. (1) Before a change is made to the database file, the DBMS creates a stable (undo) log record to reflect what has been changed. This rule is called *write ahead log* (or *WAL*, in short) protocol. (2) When a transaction commits, the DBMS makes sure that all (redo) log records of the transaction are in stable journal storage. This rule is called *flush-log-at-commit* protocol.

Log records are mostly write-only information. They are not read under normal circumstances. Only in the event of transaction aborts, process crashes, or system failures, log records are read and processed to restore the database to an appropriate acceptable consistent state. After a crash, when the DBMS restarts the database, it executes a recovery operation on the database first before it

starts the normal business with the database. During the recovery, the effects of partially executed transactions are undone and the effects of committed transactions are redone using the log records from the journal. The DBMS reads log records one after another and plays them against the database to carry out redo and/or undo operations accordingly. One important requirement is that the recovery operation must be *idempotent*: even if recovery is carried out many times in a row, it would produce the same database state at the end. Depending on logging and recovery strategies, a DBMS may do physical (or value) logging or logical (or operation level) logging or physiological logging containing undo or redo or both undo-redo information.

1.2.8.6 Checkpoint

When the DBMS restarts a database upon a failure, the very first thing it needs to do is to recover the database. The only recovery information available is the journal content that has survived the failure. The DBMS may need to read all log records from the journal and process the records against the database. Consequently, the recovery becomes a time consuming operation, especially when the journal contains a large number of log records. To reduce recovery time at restarts, the DBMS checkpoints the database periodically. A *checkpoint* is a kind of synchronization point between the database and the recovery subsystem. A checkpoint establishes a relatively recent database state and journal content and these can be used as a basis for recovery at future restarts. Checkpoints are taken solely to cut down restart processing time. In essence, a checkpoint helps eliminating some old log records from the journal, and thereby, it helps speeding up restart processing upon system failures.

1.2.9 Database interactions

At modern times, database applications interact with a DBMS to work with databases. There are two aspects in interacting with a DBMS: (1) how applications get connected and exchange information with the DBMS, and (2) how they define and manipulate data in a database via the DBMS. I address these two aspects in the next two sub-subsections.

1.2.9.1 Model of interaction

A DBMS normally functions in one of the two models: embedded or client-server. In the embedded model, the DBMS is implemented as a user-level shared library that becomes a part of database applications at runtime. Application processes or threads directly execute the DBMS program/code to obtain services from the DBMS. This is a self-service system. Applications communicate with the DBMS through a set of APIs (see Fig. 1.5). These are ordinary function calls in the same application process address space. Some embedded DBMSs, in addition, allow application developers

to implement custom-tailored "callback" functions in the applications. They register the callbacks with the DBMS at the beginning of their executions. The DBMS executes these callback functions when needs arise to manipulate data in the application space.

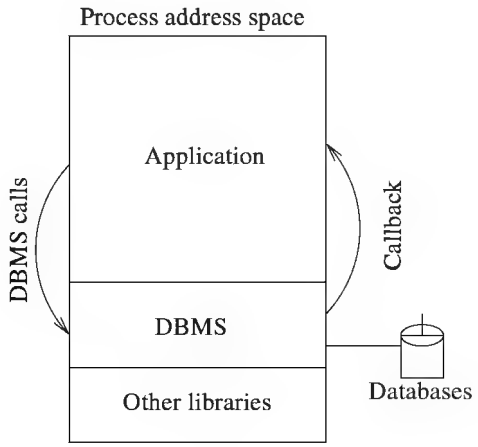


Figure 1.5: Typical embedded database management system.

In the client-server model, application processes act as clients to the DBMS that run as a separate server process (see Fig. 1.6). (The server and clients can, in fact, run on different computer systems in a network of computers.) The DBMS program proper is not accessible to application processes, but a small DBMS driver is embedded in them. The application processes and the server process communicate among themselves by using some interprocess communications primitives of the native operating system, such as TCP/IP socket or shared memory. Clients send operation requests to the server that performs the operations and replies back output to the clients. There is no callback mechanism in the client-server model.

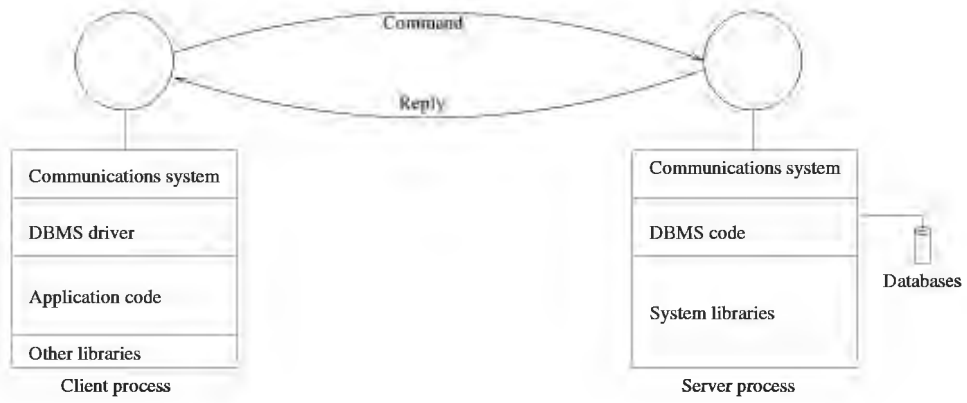


Figure 1.6: Typical client-server database management system.

Note: Most commercial DBMSs are large, highly concurrent, transaction processing systems, and they are implemented as servers, popularly called *database servers.* This operational model helps us to keep client

address spaces to the minimum size. As the DBMS runs in a separate server process, it is not affected by application bugs. There are small devices such as cell phones, PDAs that are constrained by a limited availability of the main memory space, lower batter power, etc. These devices normally use embedded DBMSs with a minimal set of features. Embedded systems do not need to maintain a separate server process. The application processes carry out database operations on their own by executing API functions in the DBMS library. ◁

1.2.9.2 Language of interaction

A database client sets up a connection with a DBMS to do some useful work with databases. There are various ways to make a DBMS work. The means are different in different systems. There are two broad alternatives: (1) procedural or prescriptive, and (2) non procedural or descriptive. In procedure oriented systems, users tell the DBMS, at each small step at a lower level, what it should do on a database. In description oriented systems, users tell the DBMS what is to be done at a higher-level and the DBMS carries out the task on its own in the best possible way.

Most modern DBMSs implement a specific descriptive programming language (also known as query language) so that users can express, with relative ease, what actions to be performed by the DBMS on databases. Users send commands (that are programs written in the query language) to the DBMS that, in turn, carries out the commands against databases and reports the results back to the users. Structured Query Language (SQL, in short) is the most commonly used database language in modern relational databases. It is a non procedural language. Each SQL statement is a separate program or subprogram in the language, which describes what to compute, but not how to compute it. The beauty of SQL is that it lets the application developers to focus on the 'what' part and lets the DBMS implementers to focus on the 'how' part.

Like any other programming language, SQL has its own syntax to construct SQL programs. It defines a standard set of SQL specific primitive types (integer, real, varchar, blob, date, time, timestamp, and so forth). Users may also define their own datatypes, called *domains*, that are composed of primitive types and other domains. SQL statements are divided into two categories: data definition language (DDL, in short) and data manipulation language (DML, in short). The DDL is used to create tables, indexes, constraints, domains, triggers, and views, and the DML to query, insert, delete, and update rows in tables. More on SQL is discussed in Section 1.3.3 on page 32.

1.3 Relational Database Management System, RDBMS

A *relational database management system* (*RDBMS*, in short) is a DBMS that implements the data structuring and data organization as specified by the relational data model (see Section 1.2.4

on page 12.). The RDBMS supports the user view of relational database, i.e., a collection of (conceptual) relations aka tables, and stores them along with their schema in one or more database files.

Relation is a two dimensional concept, whereas file is a one dimensional concept. Actually, in most operating systems, a file is merely a sequence of bytes, and they (or their FMSs) do not maintain any structures on file contents. Thereby, we need a mapping from relations into the file space. That is, there is a need to appropriately structure and organize a file content to host relations. Users apply higher-level relational operations on databases to store, remove, and manipulate relations. The RDBMS translates these database operations into lower-level file operations. For this purpose, we need a different schema, called *physical schema*, to represent data in files. The (conceptual or logical) schema defines structures of relations, and the physical schema defines structures on files. Database users never see physical schema, and hence database applications are insulated from physical structures and organizations of relations.

In essence, a *relational database* is a collection of relations, and these store entities and relationships information as tuples. Each relation has its own schema that describes what kind of tuple values can be stored in the relation. The schema information is also stored in the database in separate relations, called *catalogs* or *system tables*. (A catalog is also variously called by other names such as data directory, data dictionary.) A catalog stores meta information about other relations and catalogs. For example, a catalog may store information about a user relation: name of the relation, number of columns of the relation. For another example, a catalog can store column name, relation name of the column, column datatype, default value, and so forth. All schema (of all relations) collectively define the *database schema*. (You may note that schema design and refinement is a user activity, and not a database system activity.) The database schema describes all properties of the database. It is the foundation for database application development work, as it clearly identifies what kinds of data can be stored in the database.

Most RDBMSs permit applications to read catalogs, but do not permit them to update catalogs. Catalogs are updated by the DBMS as a side effect of applications' creating and updating user relations. For example, creating a new relation causes to insert new tuples in catalogs describing the new relation and its columns. In addition to having user relations and system catalogs, a database may also have indexes that help speeding up of searching the database. Indexes help faster lookup of specific data values in the database. In the next two subsections I discuss various relational operations and algebra of those operations. Following that I present the SQL query language and discuss components of a typical RDBMS in two subsections.

1.3.1 Relational operations

Given a set of relations, we need a mechanism to manipulate the relations. We should at least be able to retrieve, store, delete, and modify tuples in relations. Manipulating a relation by applying lower-level operating system supported read/write primitives on database files can be a daunting and error prune task for application developers. Moreover, they would need to know the details of physical structures (aka, physical schema) of database files. Certainly, that is not a goal of the DBMS. To alleviate these difficulties, we define a set of higher-level operations that application developers can apply on relations with relative ease to accomplish their tasks. These are called *relational operations*. The DBMS will carry out these operations in the best possible way, of course, by using lower-level primitives on files. There are two well known mathematical disciplines for specifying relational operations.

1. *Relational algebra.* It is an algebra of relational operations. It defines many operations that can be applied on relations. The results of these operations are relations themselves. Thus, the algebra is closed. You can compose, i.e., functionally connect these relational operations to define complex operations on relations.

2. *Relational calculus.* It is a descriptive language that allows you to describe what you want to do with relations, rather than how to compute the final results.

Notations in relational calculus are a little more involved. In this book I restrict myself only to relational algebra.

1.3.2 Relational algebra

As all tuples in a relation are distinct and have the same format, we may treat the relation as a mathematical set (of tuples) and each tuple as an ordered list of columns, and define some well known set-theoretic operations for the relation. Primitive relational operations are (1) projection, (2) selection, (3) Cartesian product or cross-product, (4) set-difference, and (5) union. The other widely used operations are intersection, join, division. For all these operations, the schema of output relations depend on those of input relations. These operations are defined in the next sub-subsections to brush up your memory.

1.3.2.1 Projection

Projection is a unary operation on a relation, and it returns an output relation that is obtained by deleting unwanted columns from the given relation with no duplication of tuples in the output relation. The schema of the output relation contains those columns of the given relation that are not deleted.

1.3.2.2 Selection

Selection is a unary operation on a relation, and it returns an output relation that is obtained by deleting unwanted tuples from the given relation based on a selection condition. Only those tuples that satisfy the selection condition are in the output relation. The schema of the output relation is the same as that of the given relation.

1.3.2.3 Set-difference

Set-difference is a binary operation on two relations that have the same schema, and it returns an output relation with those tuples from the first relation that are not in the second relation. The schema of the output relation is the same as that of the given relations.

1.3.2.4 Union

Union is a binary operation on two relations that have the same schema, and it returns an output relation with tuples that are in the first or the second or both the relations. The schema of the output relation is the same as that of the given relations.

1.3.2.5 Intersection

Intersection is a binary operation on two relations that have the same schema, and it returns an output relation with tuples that are in both the relations. The schema of the output relation is the same as that of the given relations.

1.3.2.6 Cross-product

Cross-product is a binary operation on two relations (that may have different schema), and it returns an output relation each tuple of which is constructed by combining a tuple from each input relation. Basically, each tuple of the first input relation is paired with each tuple of the second input relation to produce a tuple in the output relation. The schema of the output relation is the composition of the two schema of the given relations. If the two schema have some names in common, they are renamed to make all names in the resulting schema distinct.

1.3.2.7 Join

Join is a binary operation on two relations (that may have different schema), and it returns an output relation that depends on various join-criteria. There are a variety of join operations that are derived from primitive relational operations. A conditional (or theta) join is a selection operation composed with the cross-product operation: the selection operation is applied on the output relation

returned by the cross-product of the two given relations. A equi-join is a special case of conditional join where the selection condition contains only equality of some given columns. A natural join is a equi-join on all common columns; in the result relation only one of every common columns is retained.

1.3.3 Structured Query Language, SQL

I talked about various relational operations in the previous subsection. We need programming languages so that application developers can compose and apply relational operations on databases. The languages are called *query languages*. As mentioned before, Structured Query Language (SQL, in short) is the most widely used query language to write applications for relational databases. It is the language of choice today for manipulating relational databases.[6] SQL offers features for set-oriented, non procedural data definition, manipulation, and control operations. Unlike programming languages such as C, SQL is a declarative language that is used to specify what is to be done. Each database query is formulated in SQL statements, where each statement is a separate program or subprogram. The DBMS finds out the best possible way to execute the statements.

SQL includes capabilities close to that of relational algebra. In addition, it includes features for inserting, deleting, and updating tuples in relations, and creating, altering, and deleting relations in databases. In spite of that SQL is not a Turing-complete language.[7] For example, it does not have looping capabilities. It is not intended to be used in general-purpose complex computations. It is developed to help users accessing and managing a large data set efficiently. Consequently, SQL programs are normally embedded in other Turing-complete programming languages such as C and COBOL. These are called native or host languages relative to SQL. Complex application logic is written in the native language and database accesses are done only using embedded SQL statements. This though causes some problems due to datatype mismatch between the SQL and the native language. SQL is set-oriented, whereas the native languages are record-oriented. Different DBMSs solve the datatype mismatch problem differently; they employ different mechanisms to transfer data between the SQL space and the native language space.

1.3.3.1 SQL features

As mentioned previously, SQL statements are divided into two partitions: data definition language (DDL) that is used to define schema, and data manipulation language (DML) that is used to formulate queries. DDL is used to create, alter, and delete new database objects such as table, index,

[6] SQL development is originally started by IBM in the mid 1970s under the name SEQUEL. It is renamed as SQL in 1980. The SQL standards is written in 1986 (SQL-86), and is extended in 1989 (SQL-89), 1992 (SQL-92), and 1999 (SQL-99).

[7] SQL is though equivalent to first order predicate calculus.

view, trigger, constraint, domain. DML is used to insert, update, and delete tuples in relations, and to select tuples from relations. The SQL select construction does the work of projection, selection, cross-product, and join relational operations. Each SQL select specifies one or more relational operations and a list of relations on which the operations are applied to produce the output relation. There are no DMLs for manipulating indexes. Indexes are automatically manipulated by the DBMS when you apply operations on the corresponding base relations.

You may note that most commercial RDBMSs use a slightly different definition of relation. It is treated as a bag or multiset, and it may not be a pure set. (This treatment is solely for computational efficiency and for reduced DBMS implementation complexity.) Thus, a relation may have duplicate tuples. SQL users should not be surprised if they see duplicates in the output of an SQL statement. SQL also supports many non relational operations: duplicate elimination, aggregation, grouping, sorting, outer join. Let us study some simple SQL examples in the next sub-subsections.

1.3.3.2 SQL examples

This is the only place in this book where I present a few simple formal SQL constructions to brush up your mind. SQL statement `create table Students(sid integer primary key, name varchar, address varchar)` creates a new table named Students in the database. (If there is already a table with the same name in the database, the statement execution would fail.) The table has three columns, namely `sid`, `name`, and `address`. The type of the `sid` column is integer, and the other two are variable length character strings. The keyword "`primary key`" is a constraint and it says that all values of the `sid` column in the table at one time must be unique and not NULL; the other two columns may not be unique. You can insert a new row in the Students table by executing SQL statement `insert into Students values (1001, 'Sibsankar', 'Sunnyvale, California')`; this statement inserts a new row with `sid` value equal to 1001, `name` equal to 'Sibsankar', and `address` equal to 'Sunnyvale, California' in the Students table. If you re-execute the same statement again, it results in a constraint violation for duplicate `sid` value. (You may note that SQL is case insensitive, but anything put within quotes is treated as it is.)

You can retrieve all rows from the Students table by executing SQL statement `select * from Students`. It is an unconditional selection operation on the Students table, and it returns all rows from the table. You can obtain information about a particular student by executing a simple SQL statement such as `select * from Students where sid = 1001`. It is a conditional selection operation on the Students table, and it returns those rows from the table that have value 1001 in the `sid` column. As `sid` is the primary key, the statement returns at most one row. The `where` clause is used to limit the number of rows from the table over which the query operates; the clause

specifies a selection condition. In the previous two SQL statements, the '*' means all columns from the given table in the schema-defined order; it is a shorthand notation. You can as well spell out all column names and their order separated by commas. You can selectively return some (and may not be all) column values. For example, you can obtain student names by executing `select name from Students`. It is a projection operation on the Students table. It may return duplicate names because the `name` is not a unique column in the Students table. You can eliminate duplicates by executing `select distinct(name) from Students`. You can obtain the name of a particular student by executing `select name from Students where sid = 1001`. It is a composition of projection with conditional selection operation. Suppose there is another table, `Admitted-to` with three columns `sid`, `did`, and `doj`. You can obtain student names and their joining dates by executing `select name, doj from Students, Admitted-to where Students.sid = Admitted-to.sid`. This is a join operation. The SQL select does the work of projection, selection, cross-product, and join relational operations. You can delete a row from the Students table by executing `delete from Students where sid = 1001`. You can delete all rows from the table by executing `delete from Students`. This statement does not destroy the schema for the Students table. You can destroy the Students table along with its schema by executing `drop table Students`.

SQL has five built-in functions, namely `count`, `sum`, `avg`, `max`, and `min`. The `count` function computes the total number of rows in a given table. (For example, SQL statement `select count(*) from Students` returns the total number of rows in the Students table.) The `sum` function computes the summation of numeric columns, the `avg` computes the average value, the `max` the maximum value, and the `min` the minimum value of numeric columns. The `max` and `min` functions are also applicable for varchar columns with collations. The built-in functions (except in `group by` statements) cannot be mixed with column names. For example, `select sid, count(*) from Students` is an illegal SQL statement. The built-in functions cannot be used as a part of `where` clauses. The functions can however be used with column names in `group by` statements; `select name, count(*) from Students group by name` returns an output table with two columns, where each row has a distinct name and a repetition count for the name in the Students table. (You may recollect from your previous SQL experience that a GROUP BY clause causes one or more rows of the result to be combined into a single row of output. This is especially useful when the result contains aggregate functions.) In the next subsection I discuss how a RDBMS handles SQL queries.

1.3.4 Components of a typical RDBMS

Figure 1.7 shows a typical architecture of a typical SQL-based RDBMS. There are two major component subsystems: (1) SQL translator or query preprocessor and (2) query execution engine and its supporting subsystems. The former is often called the *frontend*, and the latter the *backend*. The

frontend translates each and every SQL statement into an internal (RDBMS specific) program or data structure that the engine can execute or manipulate to carry out the SQL statement execution. Some RDBMSs generate native machine code instead of engine code; many generate parse trees that the engine walks through; some (such as SQLite) generate internal bytecode programs that resemble assembly language programs. Whatever be the mechanism, the engine is a virtual machine that interprets the frontend's output and produces outputs of the SQL.

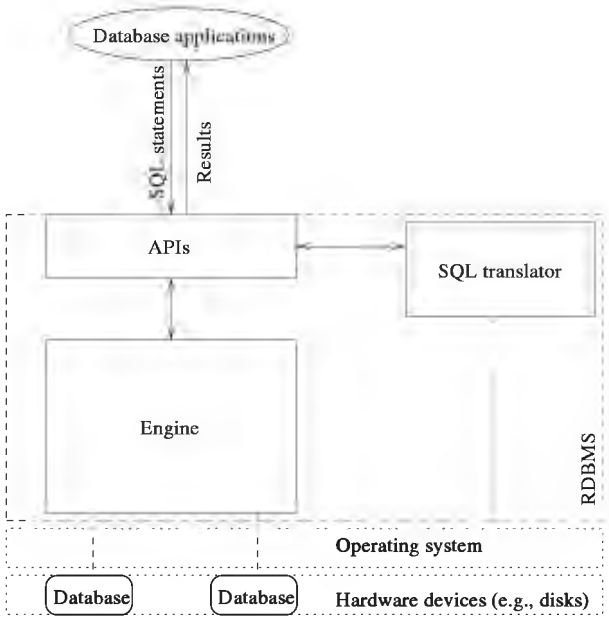


Figure 1.7: Two typical components of a typical RDBMS.

The frontend has three subsystems: parser, optimizer, and code generator. The parser splits the input SQL statement into tokens and forms a parse tree of the statement. The tree is transformed into another tree by the optimizer. The code generator processes a parse tree, and produces the code or data structures that the engine understands. The backend consists of many subsystems: (1) storage management, (2) in-memory buffer or cache management, (3) concurrency control, (4) recovery management, (5) transaction management, (6) interpreter. The storage management subsystem manages space on physical devices (i.e., as files). When data items from files are copied in the main memory, the in-memory copies of data items are organized in a buffer. The buffer management subsystem manages this buffer as an in-memory cache of data from files. The concurrency control subsystem coordinates concurrent (simultaneous or interleaved) accesses from transactions to databases. This subsystem allows as much concurrency as possible without violating data consistency semantics. The recovery management subsystem recovers databases in case of system/process failures or transaction aborts.

Let us study a very simple use case example here. Suppose you are connected to a particular database. If you execute an SQL statement `select * from Students`, the DBMS returns all rows

from the Students table one by one. To execute the given `select` statement, the engine may perform the following actions.

1. Open the database file containing the Students table;
2. Rewind the file to position it at the beginning of the Students table;
3. While not at the end-of-table, perform the following substeps in the listed order:
   - read all column values out of the current record,
   - return the values to the caller,
   - advance file position to the next record;
4. Close the file.

All these file processing actions are transparent to application programs. Actual file processing magic happens in the engine.

Summary

Although the main focus of this chapter is introducing concepts related to database systems, it starts with introducing concepts related to computer systems. A computer system is composed of many hardware resources (such as processor, main memory, and I/O devices such as disks, keyboards, display monitors, network interface cards, etc.), an operating system, many shared libraries, and utilities. The processor executes software programs, the main memory stores programs and data, and I/O devices help the processor to communicate with the external world and store information on persistent devices such as disks. The operating system is the overall in-charge of hardware resources, and it creates a user friendly environment in which users can develop and execute applications with relative ease. Each application execution is represented by a (single- or multi-threaded) process that carries out the application logic. A process can read and write information from and/or to I/O devices. Disk is a persistent I/O device that can retain stored information when its power supply is disconnected. Users see information stored in a disk in terms of files. A file management system (a component of the operating system) manages these files on disks, and it helps users to access the stored information with relative ease. It implements open, close, read, write, sync, etc., operations on files that applications can invoke to manipulate information stored in files. One point to note that many file operations are not atomic (i.e., not indivisible), which may become problematic for some applications such as database systems.

The rest of the chapter introduces various concepts from database systems, especially from the relational ones. It defines concepts such as data item, data type, value, information. It then defines

what a database is and describes how a database is stored in native files. A database is accessed by applying various database operations that are building blocks of database applications. Naive users prefer executing database applications to manipulate their data in databases. A database should always be in consistent states satisfying some user defined integrity constraints.

The very first step in setting up a database to storing data items is to create a model to represent the data items. Various data models such as hierarchical, network, relational, and object are successfully used. This book deals with the relational data model called the entity-relationship model. All data items in a database are categorized into different entities and relationships between those entities. They are respectively called entity set and relationship set. Each entity is described by a set of attributes out of which one or a few form a key whose value distinguishes one entity from others.

A relational database, in theory, is a set of relations. These are also called tables. The tables store entity sets and relationship sets in the form of rows. A table can be created by users or by the database system itself. Each table is characterized by its schema that describes properties of its columns and various constraints. Relational databases are normally accessed by queries written in special query languages. SQL (structured query language) is the most popular of them, and is used for both data definition and data manipulation. Although relational theory does not support the concept of index, most database designers create indexes on tables for faster data lookups. Predominantly used indexes are hash and B-tree. Hash indexes are used when fast point look up is required, and B-trees when a range search (based on some sorting order) is required.

Two predominant issues in accessing a database are failure recovery and coordinating concurrent accesses to the same data by multiple application processes. A DBMS (database management system) takes care of these issues to relieve application developers to concentrate on their application processing logic. The DBMS employs a concept of transaction to ensure that the two issues do not cause a database corruption or malfunction of database applications. Each transaction is composed of one or more database operations, and the DBMS ensures that either all of them are performed together indivisibly or none happened. The DBMS actually ensure the ACID (atomicity, consistency, isolation, and durability) properties to transactions.

1.4 Book layout

Many database concepts and terminologies have been introduced in this chapter. These concepts and new ones are explored in-depth in the coming chapters, in the context of SQLite embedded database system. The rest of the book is organized as follows.

Chapter 2 presents an overview of SQLite that is an SQL-based relational database manage-

ment system. It introduces all components of SQLite in top-down fashion. These components are elaborately discussed in bottom-up fashion in later chapters. The chapter also talks about a few SQLite APIs, and presents some very simple SQLite applications. Overall, the chapter is a short tour of SQLite, and it sets up the ground plan for the rest of the book.

Chapter 3 presents the lowest level storage organization in SQLite. It defines naming conventions for database and journal files, and their formats. It talks about how a database file is partitioned into fixed size pages, who uses those pages, and organization of the pages. It also presents formats of journal files that are used to store log records produced by applications.

Chapter 4 discusses how SQLite manages user queries and updates via transactions. SQLite executes each and every SQL statement in the abstraction of a transaction, and ensures ACID properties to transactions. The chapter discusses the file locking scheme that is used to control transaction concurrency, and the page journaling scheme that is used for recovery from transaction aborts and system failures.

Chapter 5 presents pager module. The module implements higher-level page oriented files on the top of ordinary byte oriented native files. It helps its clients to read and write pages from database files with relative ease. It is also the transaction manager and the log manager in SQLite. It decides when to acquire and release locks on database files, and when and what log records to write in journal files.

Chapter 6 presents tree module that resides on the top of the pager module. The tree module organizes raw database pages into B- and B^+-trees. SQLite organizes the content of a database into several trees. A B^+-tree stores in entirety a data table, and a B-tree an index on a data table. A tree stores keys and data in uninterpreted byte images. The chapter discusses various data structures that are used to store, retrieve, and manipulate data in trees.

Chapter 7 presents virtual machine (VM) module. The VM is an interpreter that executes programs written in the SQLite's internal bytecode programming language. It creates the abstractions of tables and indexes on the top of B- and B^+-trees. The chapter discusses various data structures that are used to interpret information stored in those trees. It presents five primitive datatypes, and discusses the formats of data records stored in table and index trees. (The pager, the tree, and the VM modules collectively implement the backend or the engine of SQLite.)

Chapter 8 discusses the frontend module. The frontend pre-processes all SQL queries, and produces bytecode programs that the engine can execute. The chapter presents four subsystems of the frontend, namely tokenizer, parser, optimizer, and code generator.

Chapter 9 presents the `sqlite3` user interface data structure. It presents an integration of all internal data structures presented in previous chapters. It gives readers a global picture of how different data structures are interconnected to one another and how they work in unison.

Chapter 10 discusses some advance features of SQLite. The notable ones are subquery, view, trigger, collation, pragma, autovacuum, wal journaling, and so forth.

Chapter 11 presents some references from the literature for further studies in databases and related systems.

Chapter 2

SQLite Overview

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- the SQLite database system and what it does
- what salient features SQLite supports
- how SQLite stores tables in database files
- how to write, compile, and execute SQLite applications
- some most frequently used SQLite APIs
- the modular SQLite architecture
- SQLite limitations

Chapter synopsis

SQLite is a small, zero-configuration, custom-tailored, embeddable, thread-safe, easily maintainable, transaction-oriented, SQL-based, relational database management system. It stores an entire database in a single file that holds all tables and indexes. It organizes all tables into separate B^+-trees, and indexes into B-trees. It supports the core transactional properties, namely, atomicity, consistency, isolation, and durability. It uses a lock based concurrency control scheme and a journal based recovery scheme.

SQLite supports a large subset of ANSI SQL-92 features and many SQLite specific commands. In addition, it provides a good framework in which you can define custom tailored SQL functions, aggregators, and collating sequences. It also supports both UTF-8 and UTF-16 standards based encoding for Unicode texts.

This chapter touches upon almost all features of SQLite. It presents a high-level overview of how SQLite works with SQL applications. It familiarizes you with some SQLite APIs, that are used for normal interactions between SQLite and database applications, by presenting some simple applications. It also presents structures and organizations of SQLite source code. Overall, this chapter is a short tour of SQLite.

2.1 Introduction to SQLite

Many database management systems (DBMSs) have been developed over the past several decades. DB2, Informix, Ingres, MySQL, Oracle, PostgreSQL, SQL Server, Sybase are a few to mention here that are commercially successful for enterprise database applications. Examples of successful embedded database systems include Sybase iAnywhere, InterSystems Cache, Microsoft Jet. SQLite [22] is a recent addition to the relational DBMS (RDBMS) family and it is also a very successful embedded database system.[1] SQLite started its debut on May 29, 2000 as initial public release of alpha code with a very limited set of features. SQLite 1.0 was released on August 17, 2000. It has made a long journey since then. SQLite 2.0.0 was released on September 20, 2001, and SQLite 3.0.0 on June 18, 2004. The latest release is SQLite 3.7.8 as of September 19, 2011 as of finalizing the writing of this book. This book is based on this particular release. The SQLite development team continues releasing new versions. You may visit the SQLite homepage http://www.sqlite.org to get to the latest release. You can find the chronological events of feature development at the http://www.sqlite.org/changes.html webpage. (By the time this book is at your hand, SQLite will be definitely at a different newer release. But, the core features and the way of handling data are not expected to change much. The knowledge gained here will be useful in deciphering new features. And, by reading this book, you will definitely get a feeling for what it requires to design and develop an embedded database system. I encourage you to develop one by yourself.)

2.1.1 Salient SQLite characteristics

SQLite is entirely developed using the **ANSI C** programming language. It is an easily maintainable, reasonably fast, SQL-based RDBMS. It has the following nice, differentiable, and commendable characteristics.

- *Zero configuration*: You do not need to carry out any separate install or setup steps to initialize SQLite database management software before using it. There are no specific steps to start running SQLite. There is no configuration file to control differentiated behaviors. Databases do not need any administration. You can download the SQLite source code from its

[1] SQLite received the 2005 OSCON Google and O'Reilly Integrator category award.

homepage http://www.sqlite.org/download.html, compile it into an executable library using your favorite **C** compiler, and start using the library as a part of your database applications. For a limited number of platforms, you can get libraries from there.

- *Embeddable*: You do not need a separate server process dedicated to SQLite. The SQLite library is embeddable in your own applications. The applications do not need to include any interprocess communications schemes to interact with SQLite.

- *Application interface*: SQLite provides an SQL environment for **C** applications to manipulate databases. It provides a set of call-level application programming interface (API) functions for dynamic SQL; you can assemble SQL statements on the fly and pass them down on to the interface for execution. In addition, you can use many callback features. There are no special preprocessing and compilation requirements for applications; a normal C compiler will do the work.

- *Transactional support*: SQLite supports the core transactional properties, namely atomicity, consistency, isolation, and durability (ACID). No actions are required from database users or administrators upon a system crash or power failure to recover databases. When SQLite reads a database, it automatically performs necessary recovery actions on the database in a user-transparent manner.

- *Thread-safe*: SQLite is a thread-safe library, and many threads in an application process can access the same or different databases concurrently. SQLite takes care of database-level thread concurrency.

- *Lightweight*: SQLite library has a small footprint of about 324KB (331835 bytes with gcc -Os on Linux) when all SQLite features are enabled. The footprint can be reduced down to about 190KB by disabling all advanced features when you build the library from the source code.

- *Customizablity*: SQLite provides a good framework in which you can define and use custom-tailored SQL functions, aggregate functions, and collating sequences.

- *Unicode:* SQLite supports UTF-8 and UTF-16 standards-based encoding for Unicode text. UTF16 supports both little- and big-endian forms.

- *Memory leak proof*: If applications strictly follow the recommended protocols of interactions with the SQLite library, the library is claimed to never leak memory.

- *Memory requirement*: Though SQLite can use unlimited amount of stack and heap spaces, it can be made to run with minimal stack space of 4KB and about 100KB of heap. This feature is very effective for small devices (such a cell phones) that are constrained with a

small amount of the main memory. But, the more memory is available, the better is the SQLite performance.

- *Multi-platform*: SQLite runs on Linux, Windows, Mac OS X, OS/2, OpenBSD, and a few other operating systems. It also runs on embedded operating systems such as Android, Symbian, Palm, VxWroks.

- *Single database file*: Each database is stored entirely in a single native file; user data and metadata are stored in the same file. The single file approach eases moving/copying a database from one place to another. (SQLite though uses many temporary files while manipulating a database.)

- *Cross-platform*: SQLite allows you to move database files between platforms. For example, you can create a database on a Linux x86 machine, and use the same database (by making a copy) on an ARM or Windows or MAC platform without any alterations. The database behaves identically on all supported platforms. You can use the same database without any problems on both 32-bit and 64-bit machines or between big- and little-endian systems.

- *Backward compatibility*: SQLite 3 is backward compatible. This means any later version of the library can work with databases created by earlier library versions. The SQLite development team strives to keep the library backward compatible. But, a version 3 library cannot work with version 2 databases.

2.1.2 Usage simplicity

SQLite is different from most other modern day SQL database management systems in the sense that its primary design goal is to be simple. The SQLite development team believes the KISS philosophy: keep it simple and splendid. They strive to keep SQLite simple, even if it leads to occasional inefficient implementations of some features. In essence, SQLite is

- simple to administer,

- simple to operate,

- simple to embed in C applications,

- simple to maintain,

- simple to customize, and it employs

- simple means to implement ACID properties.

Simplicity: A simple software is one that is easier to implement, test, maintain, enhance, integrate, document, etc. SQLite meets these criteria. ◁

To achieve simplicity, the SQLite development team has chosen to sacrifice many DBMS characteristics that some database users find useful such as high transactional concurrency, fine-grained access control, many built-in functions, stored procedures, some SQL language features (such as object-relational), tera- or peta-byte scalability, and so forth.

Reliability: SQLite is very reliable. The reliability seems to be a consequence of its simplicity. ◁

2.1.3 SQL features and SQLite commands

SQLite supports a large subset of ANSI SQL-92 data definition and data manipulation features, and some SQLite specific commands. (These commands are akin to SQL statements, but they do not manipulate user data per se.) You can create tables, indexes, triggers, and views using standard data definition SQL constructs. You can manipulate stored information using INSERT, DELETE, UPDATE, and SELECT SQL constructs. The following is a list of SQL features supported as of the SQLite 3.7.8 release. (Every new future release may have additional features. The latest set of supported features can be obtained from the SQLite webpage http://www.sqlite.org/lang.html.)

1. Data definition language, DDL:
    - Creation of table, index, view, and trigger;
    - Deletion of table, index, view, and trigger;
    - Partial support for ALTER TABLE (rename table and add column);
    - UNIQUE, NOT NULL, and CHECK constraints;
    - Foreign key constraint;
    - Autoincrement, collation column;
    - Conflict resolution.
2. Data manipulation language, DML:
    - INSERT, DELETE, UPDATE, and SELECT;
    - Subqueries including correlated subqueries;
    - group by, order by, offset–limit, collation;
    - INNER JOIN, LEFT OUTER JOIN, NATURAL JOIN;
    - UNION, UNION ALL, INTERSECT, EXCEPT;
    - Named parameter and parameter binding;

- For each row trigger.

3. Transactional commands:
   - BEGIN;
   - COMMIT;
   - ROLLBACK;
   - SAVEPOINT;
   - ROLLBACK TO;
   - RELEASE.
4. SQLite commands:
   - reindex;
   - attach, detach;
   - explain;
   - pragma.

The SQL standard specifies a huge number of keywords which may not be used as the names of tables, views, indexes, columns, constraints, or databases. SQLite relaxes this restriction and allows you to use keywords as identifiers by surrounding them with back quote or single quote or double quote or '[' and ']' pair. In addition, SQLite provides a good framework in which you can define and use custom made SQL functions, aggregate functions, and collating sequences. Pragmas are special SQLite commands that are used to alter the behavior of the SQLite library or to query the library for internal (non-table) metadata. The SQLite *attach* command helps a transaction to work on multiple databases simultaneously. Such transactions are also ACID-compliant.

The following ANSI SQL-92 features are not yet supported as of the SQLite 3.7.8 release. (For the current list, see the http://www.sqlite.org/omitted.html webpage.)

1. Many ALTER TABLE features such as renaming or dropping a column, adding or dropping a constraint;
2. For each statement trigger;
3. RIGHT- and FULL OUTER JOIN;
4. Updating a VIEW;
5. GRANT and REVOKE.

2.1.4 Database storage

SQLite stores an entire database in a single, ordinary native file and the file can reside anywhere in a directory of the native file system. We often say the file is synonymous to the database as no other file stores information about the database proper. A user who has permission to read the file can read anything from the database. A user who has write permission on the file and the container directory can change anything in the database. A database can grow as long as the native operating system/file system allows the file to grow. SQLite supports the very large file (> 2 GBytes) option on Linux systems if those systems have the option. It organizes all tables and indexes into separate B^+-trees and B-trees, respectively. It uses a separate journal file to save transaction recovery information that is used in the event of transaction aborts or system failures.

2.1.5 Limited concurrency

SQLite allows multiple applications to access the same database concurrently. However, it supports a limited form of concurrency among transactions. It allows any number of concurrent read-transactions on a database, but only one exclusive write-transaction. It does not have capability to support concurrency at a finer data granule such as table, page, row, column, or cell.

2.1.6 SQLite usage

SQLite is a very successful embedded RDBMS. It has been widely used in low-to-medium tier database applications such as web services, cell phones, PDAs, set-top boxes, standalone appliances. You could even use it in teaching relational databases and the SQL language to students in a beginner database course. You can also use it in advanced database management courses, or in database projects as a reference technology. It is available freely, and has no licensing complications because it is in the public domain. (There are though various optional proprietary parts such as SQLite for smartcards, encryption solutions that you need to order from Hwaci, the owner of SQLite. Course students may not worry about these proprietary components.)

Web Server: A SQLite based web server works fine with up to 100,000 evenly distributed hits a day; the SQLite development team has demonstrated that SQLite may even withstand 1,000,000 hits a day. ◁

SQLite is open source, and is available in the public domain (for more information on open source, visit http://opensource.org). You can download SQLite source code from the webpage http://www.sqlite.org/download.html, compile it into an executable library using your favorite C compiler, and start using the library with your database applications. SQLite runs on Linux, Windows, MAC OS X, OS/2, Solaris, OpenBSD, and a few other operating systems. In this book I restrict myself to the Linux version of SQLite 3.7.8 release that is the latest version as of September

19, 2011.

2.2 Sample SQLite Applications

In this section I am going to present you some simple database applications illustrating various core features of SQLite. You will get familiar with some most important and frequently used SQLite API functions and API constants. The applications are presented in the following subsections except Section 2.2.2 where I discuss some APIs. You may recall that SQLite is an embeddable library and it gets embedded in application process address space. Figure 2.1 depicts a generic schemata for SQLite applications. As shown in the figure, the SQLite library is embedded in the application process, and a part of the process heap space is used to store SQLite's runtime data. Of course, SQLite uses the stack when its API functions are called.

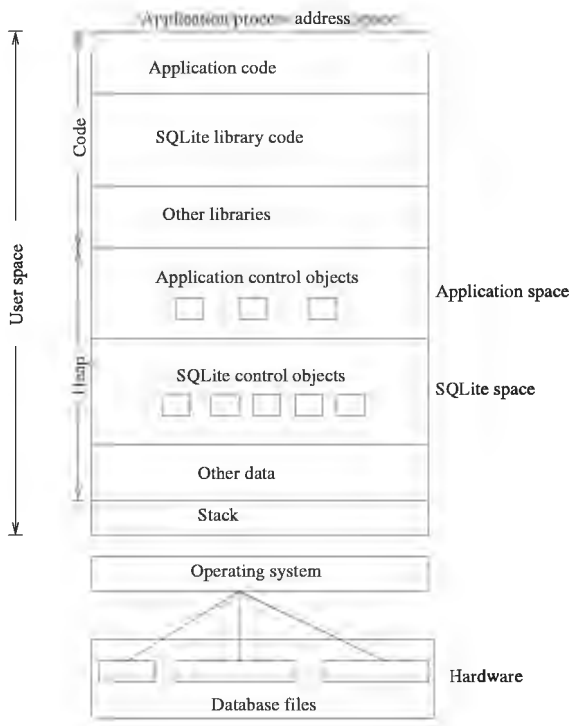


Figure 2.1: A generic database application using SQL library.

2.2.1 A simple application

Let us begin our exploration of the SQLite land by studying a very simple application. Figure 2.2 presents a typical SQLite application. It is a typical C program that invokes SQLite API functions to work with a single SQLite database. It demonstrates the simple way of using SQLite in accessing a database by executing SQL queries.

```
#include <stdio.h>
#include "sqlite3.h"

int main(void)
{
    sqlite3* db = 0; /* connection handle */
    sqlite3_stmt* stmt = 0; /* statement handle */
    int retcode;

    retcode = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (retcode != SQLITE_OK) {
      sqlite3_close(db);
      fprintf(stderr, "Could not open the MyDB database\n");
      return retcode;
    }
    retcode = sqlite3_prepare(db, "select SID from Students order by SID", -1, &stmt, 0);
    if (retcode != SQLITE_OK) {
      sqlite3_close(db);
      fprintf(stderr, "Could not compile a select statement\n");
      return retcode;
    }
    while (sqlite3_step(stmt) == SQLITE_ROW) {
      int i = sqlite3_column_int(stmt, 0);
      printf("SID = %d\n", i);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return SQLITE_OK;
}
```

Figure 2.2: A typical SQLite application.

You may compile the above example application and execute it. The sample outputs shown throughout this document were generated on a Linux machine, but these examples will work on other platforms that SQLite runs on.

Note: These examples assume that you have already prepared the *sqlite3* executable, the *libsqlite3.so* (*sqlite3.dll* on Windows and *libsqlite3.dylib* for Mac OS X) shared library, and the *sqlite3.h* interface definition file. You can obtain these in source or binary form from http://www.sqlite.org. (Binaries are available for Linux, MAC OS X, and Windows only.) You may find it easier to work with these examples if you put all the three (*sqlite3*, the shared library, and *sqlite3.h*) in the same directory as the example applications. ◁

For example, suppose you are on a Linux system and that you have saved the sample program as *app1.c* in the same directory as *libsqlite3.so*, *sqlite3*, and *sqlite3.h*. You can compile the file by executing this command using the GNU C compiler gcc:

gcc app1.c -o ./app1 -lsqlite3 -L.

It will produce a binary named app1 in the current working directory. You may execute the application to see the output. To pull in the SQLite library, you may need to include your working directory name in the LD_LIBRARY_PATH environment variable on Linux systems. Yes, you run app1, but it does not produce any output; this is because you do not have the "MyDB" database required by the application in the current working directory.

Note: Both the SQLite source code and the application must be compiled with the same compiler. If you have installed SQLite as a package, or if your operating system distribution came with it preinstalled, you may need to use a different set of compiler arguments. For example, on Ubuntu, you can install SQLite with the command `sudo aptitude install sqlite3 libsqlite3-dev`, and you can compile the example application with the command `cc app1.c -o ./app1 -lsqlite3`. Because SQLite is included with recent versions of Mac OS X, this same compilation command works there as well. ◁

The app1 application opens the MyDB database file in the current working directory. The database needs at least one table, named Students; this table must have at least one integer column named SID. In the next example application, you will learn how to create new tables in databases, and how to insert rows in tables, but for the moment, you can create and populate the table with these commands using the sqlite3 utility:

./sqlite3 MyDB "create table students (SID integer)"

./sqlite3 MyDB "insert into students values (200)"

./sqlite3 MyDB "insert into students values (100)"

./sqlite3 MyDB "insert into students values (300)"

If you run the `app1` now, you will see the following output:

SID = 100

SID = 200

SID = 300

Note: On Linux, Unix, and Mac OS X, you may need to prefix `app1` with ./ when you type its name at the command prompt, as in: `./app1` ◁

After opening the database, the app1 application first prepares the SQL statement: `select SID from Students order by SID`. It then steps through the resulting rowset produced by the statement, fetches SID values one by one, and prints the values. Finally, it closes the prepared statement and the database.

SQLite is a *call-level interface library* that is embedded in application process address space at runtime. The library implements all SQLite APIs as C functions. All API function names are prefixed with `sqlite3_` (and API constants are prefixed with SQLITE_) and their signatures are declared in sqlite3.h. A few of them are used in the app1 application, namely `sqlite3_open`, `sqlite3_prepare`, `sqlite3_step`, `sqlite3_column_int`, `sqlite3_finalize`, and `sqlite3_close`.

The application also uses some mnemonic API constants, namely SQLITE_OK and SQLITE_ROW, for comparing values returned by the API functions. Some key SQLite APIs are discussed in the next subsection, before I present other SQLite applications.

2.2.2 SQLite APIs

SQLite interface defines a set of APIs (that are a set of C functions and a set of named constants). The API functions are the sole means of normal communications between applications and the SQLite library. (SQLite also uses callback C functions that reside in the application space.) I outlined a few API functions in the previous subsection. Here I present a basic set of API functions that are used most frequently in SQLite applications. Detailed discussions for these and other API functions can be found at the SQLite webpage http://www.sqlite.org/capi3ref.html. There are about 185 API functions. A list of all API functions and constants is available at the webpage http://www.sqlite.org/c3ref/funclist.html.

1. `sqlite3_open:` This function has two parameters, one input and the other output. The input is a database file name. By executing the open function, an application opens a new connection or session with the SQLite library to access the given database file. In this book I refer it as a *library connection.* (The application may have other open library connections to access the same or different databases. SQLite treats these library connections distinctly, and they are independent of one another as far as SQLite is concerned.) Inside the library connection, the function opens the database file. The function automatically creates the database file if the file does not already exist; the default file permission is 0644. If the database is opened (or created) successfully, then this function returns SQLITE_OK to the application. Otherwise, the application gets an error code.

   Lazy File Opening: When opening or creating a database file, SQLite follows a lazy approach—the actual opening or creation is deferred until the file is accessed for reading. If the database file does exist, SQLite automatically recovers the database into a consistent state if there is a need. The lazy file creation gives you an opportunity to (re)define values for various database setting parameters using pragma commands. (These setting parameters will be discussed in Chapter 3.) ◁

   The open function returns a connection handle (a pointer to an object of type `sqlite3`) via the output parameter (`db`, in the preceding example), and the handle is used to apply further operations on the library connection (for this open SQLite session). The handle represents the complete state of this library connection.

   Figure 2.3 shows a typical scenario where an application has opened two connections to the SQLite library to access the same database file. The library connections are independent of one another, and they each are represented by separate `sqlite3` objects. A single `sqlite3`

object in the SQLite library represents and manages a single library connection. As shown in the figure, one connection has three prepared statements, but the other has none. I discuss prepared statements next.

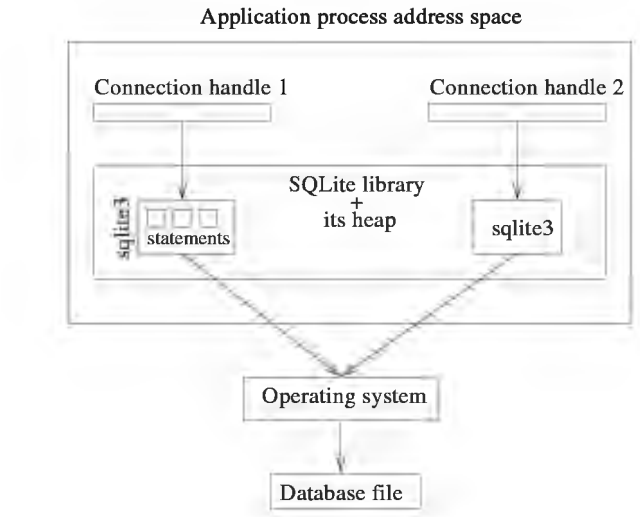


Figure 2.3: Application's connections to the SQLite library.

Newer APIs: Of late, the SQLite development team discourages using this open function; they recommend to use the `sqlite3_open_v2` function instead. There are many other `*_v2` API functions. In order to maintain the simplicity in the presentation of applications, I avoid using these newer API functions in this book. ◁

2. `sqlite3_prepare:` This function compiles an SQL statement, and produces an equivalent internal object (of type `sqlite3_stmt`). This object is commonly referred to as a *prepared statement* in database literature, and is implemented as a *bytecode program* in SQLite. A bytecode program is an abstract representation of an SQL statement that is executed by a database engine. I will discuss the bytecode programming language in Section 7.2 on page 175. I use the terms bytecode program and prepared statement interchangeably in this book to mean the same. This function returns SQLITE_OK on success, and an appropriate error code on failure.

   The prepare function returns a statement handle (a pointer to an object of type `sqlite3_stmt`) via a formal parameter (`stmt`, in the preceding example), and the handle is used to apply further operations to manipulate the prepared statement. In the previous example program, I prepared the `select SID from Students order by SID` statement as the statement handle. This handle acts like an open cursor and it is used to obtain the resulting rowset that the `select` statement produces, one row at a time. The cursor is moved forward by executing the `sqlite3_step` API function, which I discuss next.

3. `sqlite3_step`: After an SQL statement has been prepared with a call to the `sqlite3_prepare` function, the `sqlite3_step` function must be called one or more times to execute the prepared statement. Each call to the step function executes the bytecode program until it hits a break point (because it has produced a new output row), or until it halts (because there are no more rows). The function returns the caller SQLITE_ROW in the former case, and SQLITE_DONE in the latter case. In the former case, the application can read the column values of the row using the appropriate `sqlite3_column_*` API functions. (See the next item on the list.) The step function is called again to retrieve the next row. The step function moves the position of the cursor for the results of a SELECT statement. Initially, the cursor points before the first row of the output rowset. Every execution of the step function moves the cursor pointer to the next row of the rowset. The cursor moves only in the forward direction. For SQL statements that do not return rows (such as UPDATE, INSERT, DELETE, CREATE, and DROP), the step function always returns SQLITE_DONE because there is no row to process. Eventually the step function returns SQLITE_DONE. (The step function should not be called again on this statement handle without first calling the `sqlite3_reset` function to reset the program execution back to its initial state. I discuss the reset function shortly.)

   In case there is an error during an execution of the step function, the return code is SQLITE_BUSY, SQLITE_ERROR, or SQLITE_MISUSE. SQLITE_BUSY means that the engine has attempted to access a busy (aka, locked) database and there is no callback function registered to resolve this situation or the callback function has decided to break the execution. The application may call the step function again later to retry the prepared statement execution. SQLITE_ERROR means that a run-time error (such as a constraint violation) has occurred; the step function should not be called again on the statement handle. SQLITE_MISUSE means that the step function has been called inappropriately. Perhaps it was called on a prepared statement that had already been finalized (aka, closed) or on one that had previously returned SQLITE_ERROR or SQLITE_DONE.

4. `sqlite3_column_*`: If the `sqlite_step` function returns SQLITE_ROW, you can retrieve the value of each column of the row by executing one of the `sqlite3_column_*` API functions. The datatype mismatch between SQL/SQLite and the C language is handled automatically by the engine: the column functions convert data between the two languages and from storage types to requested types. (For example, if the internal representation of a value is FLOAT and the application requests a text output value, SQLite uses `sprintf()` internally to do the value conversion.)

   The following five column API functions are available: `sqlite3_column_int`, `sqlite3_column_int64`,

`sqlite3_column_double`, `sqlite3_column_text`, and `sqlite3_column_blob` to read data out of a column. The last part of each function name indicates what kind of values the application can expect out of the SQLite library. In the previous example application, each output row is an integer value, and we read the value of the SID column by executing the `sqlite3_column_int` function that returns integer values. (If the statement handle is not currently pointing to a valid row, or if the the column index is out of range, the output these functions produce is undefined. The left most column has index 0, the next one 1, the next one 2, and so forth. You can get the total number of columns using the `sqlite3_column_count` API function. For non select statements, it returns 0.) Blob and text values require applications to know their sizes. SQLite has `sqlite3_column_bytes` function that returns the size of the column value in the number of bytes.

5. `sqlite3_finalize:` This function closes and destroys a statement handle and the associated prepared statement. That is, it erases the bytecode program, and frees all resources allocated to the statement handle. The statement handle becomes invalid and must not be reused.

   If the statement was executed successfully or not executed at all, then the finalize function returns SQLITE_OK. If the previous execution of the statement failed, then the function returns an error code. The finalize function can be called at any point during the execution of the prepared statement. If the engine has not completed the statement execution when this routine is called, it is like encountering an error or an interrupt in the execution. Incomplete updates are rolled back and the execution is aborted, and the result code returned will be SQLITE_ABORT.

6. `sqlite3_close:` This function closes a library connection, and frees all resources allocated to the connection. The connection handle becomes invalid. This function returns SQLITE_OK on success, and other error codes on failure. If there are prepared statements that have not been finalized, then SQLITE_BUSY is returned and the connection remains open.

7. `Other useful functions:` The above discussed six (category of) API functions are the core of the SQLite library that deals with the two main data structures, namely `sqlite3` and `sqlite3_stmt`. The other widely used API functions are `sqlite3_bind_*` and `sqlite3_reset`.

   In an SQL statement string (input to the `sqlite3_prepare` function), you can replace one or more literal values by the SQL positional parameter marker '?' (or numbered or named parameter ?NNN, :AAA, @AAA, or $AAA, where NNN is an integral number and AAA is an alphanumeric identifier). They become input parameters to the prepared statement. For unnumbered/unnamed parameters, the left most one has index 1. For a numbered parameter, the index is the number. For a named parameter, the index can be obtained by

invoking the `sqlite3_bind_parameter_index` API function. The values of these parameters can be set using the bind functions. (If a named or numbered parameter is used at multiple places, the same bound value is used for all the places.) If no value is bound to a parameter, the SQL NULL value is taken. The following seven bind API functions are available: `sqlite3_bind_null`, `sqlite3_bind_int`, `sqlite3_bind_int64`, `sqlite3_bind_double`, `sqlite3_bind_text`, `sqlite3_bind_blob`, and `sqlite3_bind_value`. The last part of each function name indicates what kind of values can be bound to parameters using the function. (The `sqlite3_bind_value` function helps binding a generic value.)

The reset API function resets a statement handle (i.e., the prepared statement) back to its initial state with one exception: all parameters that had values bound to them retain their values. The statement becomes ready for reexecution by the application, and reuses these bound values in the reexecution. However, the application may replace some or all these values by new ones by executing the bind functions once again before it starts the reexecution. Or, the all bound values can be removed by executing the `sqlite3_clear_bindings` API function. The reset function is very useful for repetitive queries.

8. `Return values:` All API functions return zero or positive integer values. The SQLite development team strongly recommends using mnemonics for return value checks instead of hard coded integer values. The return value SQLITE_OK indicates a success; SQLITE_ROW indicates that the `sqlite3_step` function has found a new row in the rowset that the SELECT statement returned; SQLITE_DONE indicates that the statement execution is complete. There are twenty-eight main and some more extended success and error codes as of SQLite 3.7.8 release. As the return codes are a part of SQLite interface, their values do not change from one minor release to another.

In summary, an application prepares an SQL statement, if there is a need it binds values to the prepared statement, steps through the prepared statement one or more times, resets the prepared statement for another execution of the statement with the same or different bound values. The application finally finalizes the statement to destroy the prepared statement.

Unicode APIs: The above API functions handle UTF-8 encoded input texts. There are separate API functions that handle only UTF-16 encoded texts. ◁

2.2.3 Direct SQL execution

Figure 2.4[2] presents another SQLite application that can be run from a command line to manipulate databases interactively. The command takes two arguments: the first one is a database file name,

[2]This example is taken almost verbatim from the SQLite webpage www.sqlite.org/quickstart.html.

and the second one an SQL statement. It first opens the database file, then applies the statement to the database by executing the `sqlite3_exec` API function, and finally closes the database file. The exec function executes the SQL statement directly, without the need by the application manually to go through the prepare, step, and finalize API functions, as was done in the previous example application. In case the statement produces output, the exec function executes the callback function for each output row and lets the application to process the row further. You must have read permission on the given database file, and depending on the query type, you may need to have a write permission on the file and the directory it is contained in.

```
#include <stdio.h>
#include "sqlite3.h"

static int callback(void *unused, int argc, char **argv, char **colName)
{
   int i;
   for(i = 0; i < argc; i++){ // Loop over each column in the current row
      printf("%s = %s\n", colName[i], argv[i] ? argv[i] : "NULL");
   }
   printf("\n");
   return 0;
}

int main(int argc, char **argv){
   sqlite3* db = 0;
   char* errMsg = 0;
   int rc;

   if (argc != 3){
      fprintf(stderr, "Usage: %s DATABASE-NAME SQL-STATEMENT\n", argv[0]);
      return -1;
   }
   rc = sqlite3_open(argv[1], &db);
   if (rc != SQLITE_OK){
      fprintf(stderr, "Can't open database %s: %s\n", argv[1], sqlite3_errmsg(db));
      sqlite3_close(db);
      return -2;
   }
   rc = sqlite3_exec(db, argv[2], callback, 0, &errMsg);
   if (rc != SQLITE_OK){
      fprintf(stderr, "SQL execution error: %s\n", errMsg);
   }
   sqlite3_close(db);
   return rc;
}
```

Figure 2.4: A command-line based application.

`sqlite3_exec:` This function executes one or more SQL statements directly. (Two consecutive SQL statements are separated by a semicolon.) Internally, it compiles and executes the statements one after another, in the left to the right order of input. If any statement execution results in an error, the remaining statements are not executed. If a statement has SQL parameter markers, the SQL NULL values are taken. If a statement produces output, the exec function calls a user specified callback function for each output row. The signature of the callback function can be found in Fig. 2.4. The exec function is a convenient wrapper for prepare, step, (column), and finalize functions. Nonetheless, the SQLite development team discourages in using the function because they may remove it in future releases.

`sqlite3_errmsg:` In case an error occurs during an API function execution, more information about the error can be obtained by invoking this function. The function returns the last error occurred on the library connection. The message is basically an English language description of the error.

You can compile the application code into an executable, such as app2. You can now issue SQL statements that operate on a database. Suppose you are working on the same MyDB database in the current working directory. By executing the following command lines, you can insert new rows in the Students table:
./app2 MyDB "insert into Students values(100)"
./app2 MyDB "insert into Students values(10)"
./app2 MyDB "insert into Students values(1000)"
If you run the previous application (app1) now, you will see the following output:
SID = 10
SID = 100
SID = 100
SID = 200
SID = 300
SID = 1000
You can also create new tables in databases; for example, ./app2 MyDBExtn "create table Courses(name varchar, SID integer)" creates the Courses table in a new MyDBExtn database in the current working directory.

Note: SQLite has an interactive command-line utility program (*sqlite3*), mentioned earlier, which you can use to issue SQL commands. You can download a precompiled binary version of it from the SQLite download webpage, or compile it from the source. This app2 example is essentially a bare-bones implementation of *sqlite3*. ◁

2.2.4 Multithreaded applications

SQLite can be used in single- or multi-threading mode. For the latter, many threads in a process can access the same or different databases concurrently via the same library connection. But, to make it a thread-safe library, it has to be built a little differently.

Threading Modes: Threading mode is controlled by the SQLITE_THREADSAFE preprocessor macro. In order to be thread-safe, the SQLite source code must be compiled with the macro set to 1 for serialized and 2 for normal multithread. If the macro is set to 0, the library is in the single-threading mode. What this means is that multiple threads in a single process can use the same SQLite library, but SQLite (connection and statement) handles created by one thread cannot be safely used by another thread; also it is unsafe to use SQLite by multiple threads simultaneously. In the former two cases, this restriction is relaxed and the library is said to be 'thread-safe'. In the normal multithreading mode (safeness value 2), although multiple threads can use the same library connection, they cannot use it simultaneously; they can use the connection mutually exclusively; they can though use different connections simultaneously. In the serialized multithreading mode, there is no such restriction. The default is the serialized mode. You can invoke the `sqlite3_threadsafe` API function to find out whether the SQLite library you are using is thread-safe or not. If the compile time option is multithread or serialized, you can change this option at library start time or runtime using `sqlite3_open_v2` or `sqlite3_config` API function. ◁

Figure 2.5 presents a very simple multithreaded application. The application creates 10 threads, and each of them tries to insert one row in the Students table in the same MyDB database. SQLite implements a lock-based concurrency scheme, so some of the INSERT statements may fail due to lock conflict. The application does not need to worry about concurrency control and database consistency issues; it cannot corrupt the database. SQLite takes care of concurrency control and consistency issues. However, you will need to check for failures, and handle them appropriately within the code (for example, you might retry the statement that failed, or inform the user that it failed and let her[3] decide what to do next).

[3]In this book, *she* stands for he or she; *her* for his or her; *herself* for himself or herself.

```
#include <stdio.h>
#include <pthread.h>
#include "sqlite3.h"

void* myInsert(void* arg)
{
    sqlite3* db = 0;
    sqlite3_stmt* stmt = 0;
    int val = (int)arg;
    char SQL[100];
    int rc;

    rc = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (rc != SQLITE_OK) {
      fprintf(stderr, "Thread[%d] fails to open the MyDB database\n", val);
      goto errorRet;
    }
    sprintf(SQL, "insert into Students values(%d)", val); /* Dynamically compose a SQL*/
    rc = sqlite3_prepare(db, SQL, -1, &stmt, 0); /* Prepare the insert statement */
    if (rc != SQLITE_OK) {
      fprintf(stderr, "Thread[%d] fails to prepare SQL: %s; return code %d\n", val, SQL, rc);
      goto errorRet;
    }
    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
      fprintf(stderr, "Thread[%d] fails to execute SQL: %s; return code %d\n", val, SQL, rc);
    } else {
      printf("Thread[%d] successfully executes SQL: %s\n", val, SQL);
    }
    sqlite3_finalize(stmt);
    errorRet:
      sqlite3_close(db);
      return (void*)rc;
}

int main(void)
{
    pthread_t t[10];
    int i;

    for (i = 0; i < 10; i++)
      pthread_create(&t[i], 0, myInsert, (void*)i); /* pass the value of i */
    for (i = 0; i < 10; i++) pthread_join(&t[i], 0); /* wait for all threads to finish */
    return 0;
}
```

Figure 2.5: A typical multithread application.

Warning!: This application may not work "out of the box" on Windows and Mac OS X. You may need to recompile SQLite with threading support, and/or obtain the pthread libraries to make the application work on those platforms. Mac OS X includes pthreads, and you can obtain pthread libraries for Windows at http://sourceware.org/pthreads-win32/. ◁

In the example application, each thread opens its own connection to the same database and works on the connection handle. This used to be the working model in earlier versions of SQLite. For those versions, the SQLite development team does not recommend the use of any SQLite handles across threads. SQLite APIs though may work in case handles are used across threads, but their correctness is not guaranteed. In fact, SQLite library may break and produce coredump in some versions of Linux.

The above restriction against sharing a library connection among threads is somewhat relaxed in SQLite 3.3.1 and subsequent versions. Threads can use a library connection safely in mutual exclusion (in the normal multithreading mode). What this means is that you can switch a connection from one thread to another as long as the former thread does not hold any native file locks on the connection. You can safely assume that no locks are held if the thread has no pending transaction, and if it has reset or finalized all statements on the connection. In the serialized mode, there is no such restriction.

Fork Warning!: Under Unix/Linux systems, you must not carry an open SQLite database across a `fork` system call into the child process. Problems such as database corruption or an application crash may arise if you do this. ◁

2.2.5 Working with multiple databases

Figure 2.6 shows a typical SQLite application that works on two databases. (I have simplified the code by not including error checks for function calls.) The application first opens the MyDB database, and then attaches the MyDBExtn database to the current library connection. After completion of the attach command execution, the single library connection has two database connections and the application can now access all tables in the two databases via the same library connection. I assume that the MyDB database has a Students(SID) table, and the MyDBExtn database a Courses(name, SID) table. The application executes an SQL select statement that accesses the two tables from the two databases.

Library Connection vs. Database Connection Confusion: A connection to the SQLite library can have more than one database associated with it. See Fig. 2.7. In the figure, the single library connection has three database connections, each to a different database file. The application can access all the databases via the same library connection. Although the application sees only one library connection, internally SQLite opens up multiple *database connections*, one for each database. If a library connection has only one database,

```
#include <stdio.h>
#include "sqlite3.h"

int main(void){
   sqlite3* db = 0;
   sqlite3_open("MyDB", &db); /* Open a database named MyDB */
   sqlite3_exec(db, "attach database MyDBExtn as DB1", 0, 0, 0);
   sqlite3_exec(db, "select * from Students S, Courses C where S.sid = C.sid", callback, 0, 0);
   sqlite3_close(db);
   return 0;
}
```

Figure 2.6: Working with multiple databases.

we naively call it also a database connection. You have been warned about the connection confusion. ◁

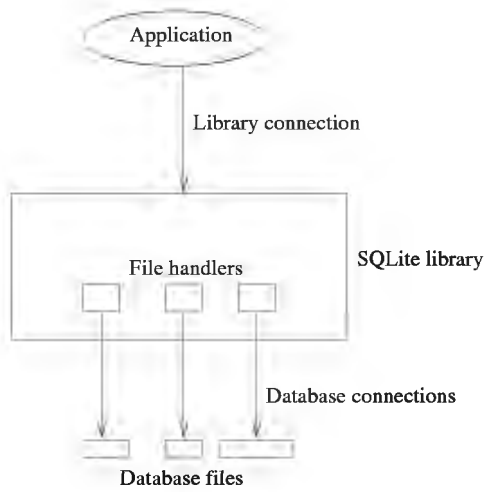


Figure 2.7: Library connection vs. database connection.

2.2.6 Working with transactions

Figure 2.8 shows a typical SQLite application that works with a transaction. The application opens a transaction by executing the `begin` command, inserts one row in the Students table and another one in the Courses table inside the transaction, and finally commits the transaction by executing the `commit` command. INSERT statements do not need a callback function, and hence, I pass 0 as the callback argument in the `sqlite3_exec` calls in the example application. If the second insert fails, you can execute the `rollback` command instead of the commit command, and the first insert will be undone. I will talk more on transactions in Section 2.3 on page 63.

Note: SQLite permits multiple SQL statements in a single exec API call; the same batch of commands in Fig. 2.8 can be executed by passing this sequence of statements in a single exec call: "begin; insert into

```
#include <stdio.h>
#include "sqlite3.h"

int main(void){
   sqlite3* db = 0;
   sqlite3_open("MyDB", &db); /* Open a database named MyDB */
   sqlite3_exec(db, "attach database MyDBExtn as DB1", 0, 0, 0);
   sqlite3_exec(db, "begin", 0, 0, 0);
   sqlite3_exec(db, "insert into Students values(2000)", 0, 0, 0);
   sqlite3_exec(db, "insert into Courses values('SQLite Database', 2000)", 0, 0, 0);
   sqlite3_exec(db, "commit", 0, 0, 0);
   sqlite3_close(db);
   return 0;
}
```

Figure 2.8: Working with transaction.

Students values(2000); insert into Courses values('SQLite Database', 2000); commit". This is better—if the second insert fails, the transaction is aborted by the system. If the batch contains SELECT statements, the same callback function is used to process the resulting rowsets. ◁

2.2.7 Working with a catalog

A database system also stores (meta) information about user information. The meta information is also represented as tables called *catalogs* or *system tables* to differentiate them from user tables. In essence, a catalog is a table that is created and maintained by SQLite itself, and it stores some meta information about the database. SQLite maintains one master catalog, named `sqlite_master`, in every database. The master catalog stores schema information about tables, indexes, triggers, and views. You can query the master catalog (e.g., select * from sqlite_master), but you cannot manually drop or directly modify the catalog. There are other optional catalog tables. All catalog table names start with the sqlite_ prefix, and the names are reserved by the SQLite development team for internal use. (You cannot create database objects such as tables, views, indexes, triggers with such names in upper, lower, or mixed case.) I will talk more about catalogs in Section 2.4 on page 67.

2.2.8 Using the sqlite3 executable

The above example applications use SQLite in the library form. You can build SQLite as an standalone utility application; it is popularly named `sqlite3`. (This is different from the sqlite3 objects used as connection handles.) This utility allows you to manually execute SQL statements. It additionally supports SQLite specific dot commands: these commands are prefixed with a dot '.'. For example, `sqlite3 .help` gives you a list of all dot commands the utility supports. The dot

commands are handy utility functions to obtain information about schema, import/export data, set various display options, etc. For example, `sqlite3 MyDB .dump` dumps the entire database on stdout. In this book I do not discuss the sqlite3 utility. You may visit the SQLite webpage http://www.sqlite.org/sqlite.html to know about dot commands.

2.3 Transactional Support

SQLite provides an environment for database users to develop and run database applications (C programs) with relative ease. It handles dynamic SQL statements that can be assembled on the fly, and it ensures ACID properties to the statement executions. By default, SQLite operates in the *autocommit* mode. In this mode, it executes each and every SQL statement in a separate transaction: read-transaction for SQL select statements and write-transaction for others. It creates a new transaction for each SQL statement, and closes (i.e., commits or aborts) the transaction at the end of the statement execution. That is, for each SQL statement all changes to the database are committed or aborted as soon as the statement execution is successfully complete or failed, respectively. These transactions are transparent to applications. That is, applications do not have to include code to handle those transactions, and application logic does not depend on the management of those transactions.

Warning!: In SQLite documentations, by a transaction they often mean a write-transaction. In this book, when needed I differentiate read- and write-transactions. Read-transactions are implicit in SQLite. And, hence, there is a scope of some confusion. You have been warned! ◁

The default autocommit mode might be very expensive and performance detrimental for some applications, especially for those that are highly write intensive. This is due to the fact that SQLite requires reopening, writing to, and closing the journal file for each of SQL insert, delete, and update statements. In addition, there is also concurrency control overhead as applications need to reacquire and release locks on database files for each SQL statement execution. These overheads can be a significant performance penalty (especially for large applications). The overhead can only be curtailed by opening a 'user-level' transaction surrounding many SQL statements. The application can encompass a sequence of SQL statements within 'BEGIN TRANSACTION' command and 'COMMIT TRANSACTION' or 'ROLLBACK TRANSACTION' command. (The keyword TRANSACTION is optional.) I presented one such application in Fig. 2.8 on page 62. With a few exceptions you can put any SQL statements in a user transaction.

The BEGIN command takes SQLite out of the autocommit mode, and we say that the system is in the manual commit mode. Effects of successive SQL statements become a part of the user transaction. An execution of the COMMIT/ROLLBACK command closes the user transaction and SQLite returns back to the autocommit mode. The COMMIT command actually works a little

differently. It may not then-and-there actually commit the entire transaction. If there are pending update operations, the commit fails and the transaction remains open. Otherwise, it commits all the changes of the transaction to the database and then it turns on the default autocommit mode. (You are assured that at the end of the commit, all changes of all SQL statements that are performed in the transaction become effective and permanent.) This commits the write-transaction, but if there are in-progress select statement executions, the transaction is converted into a read-transaction in the autocommit mode. Then, at the conclusion of all pending select statement executions (if there are any) within the read-transaction, the regular autocommit logic takes over and causes the actual commit of the read-transaction. The ROLLBACK command also operates by turning on the autocommit back, but it also sets a flag that tells the autocommit logic to rollback rather than commit the user transaction. The ROLLBACK command however can terminate some or all pending select-executions. (The select executions that have read from tables modified by the write-transaction will be canceled. Their respective next call to the `sqlite3_step` API function will get the SQLITE_ABORT error code.)

In summary, a typical application starts a user transaction by executing the 'BEGIN TRANSACTION' command. All the successive SQL statements are executed within the transaction. At some time later, the application executes the 'ROLLBACK TRANSACTION' command to abort the transaction or the 'COMMIT TRANSACTION' command to make the updates durable. In either case, the transaction ends and SQLite reverts back to the autocommit mode. To start a new user transaction, the application needs to execute the 'BEGIN TRANSACTION' command once again. If the application does not explicitly execute the 'COMMIT TRANSACTION' or 'ROLLBACK TRANSACTION' command in the user transaction, SQLite rolls back the transaction when the application closes the database connection.

Figure 2.9 presents another application that uses the user transaction capability. The application opens a user-level transaction, then inserts four rows inside the transaction, and finally commits the transaction. If any error occurs before reaching the commit statement, it closes the database connection. Consequently, the transaction is automatically rolled back by SQLite during the next database close API call.

SQLite supports the standard flat transaction model: all operations within a transaction either survive together or they are all rolled back. There is no way of committing or aborting parts of a transaction. SQLite does not support nested transactions. Thus, an execution of the 'BEGIN TRANSACTION' command within a transaction has no effect at all; in fact, SQLite does not process the statement, and instead returns an error code to the application. As an application cannot open more than one user transaction on an open library connection at a time, transaction management is simplified considerably in SQLite. If a statement execution fails, SQLite does not

```
int main(void)
{
    sqlite3* db = NULL;
    int retcode;
    retcode = sqlite3_open("MyDB", &db); // Open a database named MyDB
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "begin transaction", NULL, NULL, NULL);
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "insert into Students values(1001, 'Sibsankar',
                                'Sunnyvale, California')", NULL, NULL, NULL);
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "insert into Students values(1002, 'Richard',
                                'Charlotte, North Carolina')", NULL, NULL, NULL);
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "insert into Students values(1003, 'Richard',
                                'Sunnyvale, California')", NULL, NULL, NULL);
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "insert into Students values(1004, 'Sibsankar',
                                'Charlotte, North Carolina')", NULL, NULL, NULL);
    if (retcode != SQLITE_OK) goto errorRet;
    retcode = sqlite3_exec(db, "commit transaction", NULL, NULL, NULL);
    errorRet:
       sqlite3_close(db);
       return retcode;
}
```

Figure 2.9: A typical transactional application.

forcefully abort the container transaction (unless the conflict resolver indicates a rollback). If you split SQL statements in a transaction over several API calls, your application must handle failures in each of these calls, because otherwise the application may behave oddly. If a failure indeed occurs, SQLite automatically aborts the corresponding SQL statement, but not the entire user transaction. The transaction can carry on executing other new SQL statements before a final commit or abort. SQLite also supports setting up manual savepoints to which the application can fall back later by rolling back the effects of some recent SQL statement executions. (Savepoints allow a transaction to revert back to previously established database states.)

Two pillars of transaction management, namely concurrency control and failure recovery are discussed in the next two subsections.

2.3.1 Concurrency control

SQLite stores an entire database in a single native file. It implements a simple database level (and not table nor row nor column nor cell level) locking framework, on the top of the native operating system supported file locking primitives, to coordinate accesses from transactions to the database. It permits many concurrent read-transactions, but only one write-transaction on a database. This implies that if any transaction is reading from any part of the database, all other transactions (in this and other processes) are prevented from writing any other part of the database. Similarly, if any one transaction is writing to any part of the database, all other transactions are prevented from reading or writing any other part of the database. SQLite follows strict two phase locking (that is, it releases locks at the transaction termination), and hence ensures serializable executions of concurrent transactions.

2.3.2 Database recovery

SQLite uses a single separate journal file per database file to provide the ability to rolling back a write-transaction in case the application decides to abort the transaction. (There is no rollback journal for read-transactions.) The rollback journal is always created in the same directory where the database file resides and has the same name but with '-journal' appended. (For example, MyDB database will have the 'MyDB-journal' file for storing recovery information.) The journal file stores information related to all changes made to the database file during a transaction execution in the form of log records. The journal is an entry-sequenced file, and stores log records in the same sequence as they are produced by the current transaction. SQLite uses physical or value logging for undo purposes. SQLite logging is inefficient: every log record contains the image of the entire database page even when the transaction modifies a single byte in the page. It is made inefficient so that recovery logic becomes as simple as possible, and to keep the SQLite library size in check.

Journal File Retention: You may note that in SQLite the journal is a transient file in the default operating mode. SQLite creates the journal file for every write-transaction and deletes the file when the transaction is complete. In fact, in case of commit, the journal deletion is the transaction commit point. There are options where the journal file is not deleted; it is truncated to zero, or invalidated at transaction commit/abort. (The invalidation option may also truncate the file to a priori defined size.) These options will be quite beneficial in those platforms where file creation and deletion are costly operations. In the sequel, I use the term *'journal finalization'* to mean either of the four options. In the SQLite 3.7.0 release, they have introduced a new journaling scheme called WAL in which '-wal' journal files are used instead of '-journal' files and are retained after transactions commit/abort. I will talk about WAL journaling in Section 10.17 on page 249. ◁

When in a user transaction, SQLite executes each (non select) SQL statement in a separate

statement subtransaction. You may note that there can be at most one subtransaction open in a user transaction. That is, there cannot be concurrent updates in the transaction. (There though can be any number of select statement executions concurrent with a subtransaction execution. I discuss it elaborately in a latter chapter.) Failure of the current subtransaction does not automatically abort the container transaction (or concurrent select statement executions). Each subtransaction uses a separate, temporary file as statement journal that is used to store information only for statement level recovery.

SQLite permits applications to manipulate multiple databases in a single user transaction. In this case, a transaction on a library connection manifests into a separate transaction on each database connection. SQLite uses their respective rollback journal files and, in addition, uses a separate master journal file. The master journal records only the names of individual rollback journal files. I talk more about journals in Chapters 4 and 5.

2.4 SQLite Catalog

Most DBMSs keep meta information about all user tables and indexes in different tables called *catalogs*. In RDBMSs, catalogs themselves are tables (often called *system tables*). RDBMSs usually store them in the same way user tables are stored in the database. For example, the schema descriptions (SQL create statements) are stored as rows in catalogs. There are normally different catalogs for different purposes. We may have one catalog for storing the name of all tables; another one for table attribute names, their types, and default values if there are any; another one for integrity constraints. We may also have a catalog for view names and their definitions; a different one for index information. And, so on and so forth.

Different RDBMSs maintain different number of catalogs. SQLite simplifies the use of catalogs, and maintains just *one* catalog. It stores schema information about tables, indexes, triggers, and views in a single catalog, named `sqlite_master`.[4] The master table is stored in the database file itself at a particular location. The structure of the master table is given below in term of an equivalent SQL create table statement.

```
create table sqlite_master (
    type      text,
    name      text,
    tbl_name  text,
```

[4]There are though other optional catalogs. The master catalog is the only one that is always present in the database. All catalog names start with `sqlite_` prefix, and the names are reserved by the SQLite development team for internal use. You cannot create database objects (tables, views, indexes, and triggers) with such names (in upper, lower, or mixed case).

```
    rootpage integer,
    sql      text
);
```

The master table is created and initialized to empty when the database is initialized. A new row is added to the master table when you execute a new schema definition (a create SQL statement). The row describes the new object that is just created. The `type` column specifies whether the object is a table, view, index, or trigger with respective values "table", "index", "view", or "trigger". The `name` column specifies the name of the object. (For an automatically created index, the name is 'sqlite_autoindex_TABLE_N', where TABLE is the name of the table on which the index is and N is an integer starting with 1.) The `tbl_name` column specifies the name of the table or view the object is associated with. (For table and view, its value is the same as that of the `name` column.) The `rootpage` column specifies the database-page number where the root of the object (a B- or B^+-tree) is available.[5] The `sql` column specifies the SQL statement that creates such an object. (In the `sql` column, SQLite transforms keywords in upper case, removes redundant spaces, etc. The sql column is NULL for those indexes that SQLite creates for unique constraints.) Every object (except the `sqlite_master` table) in an SQLite database has an entry in the `sqlite_master` table. When SQLite opens and reads a database file, it first scans the entire master table, and preprocesses the `sql` column in each row and produces many in-memory catalog objects that are effectively equivalent to different persistent catalog tables used in many DBMSs. These in-memory catalog objects collectively define a *schema cache*.

There is another catalog table, named `sqlite_temp_master`, that is available at runtime and that stores schema information about all temporary objects (table, index, trigger, and view). You may note that, for every open library connection, SQLite maintains a parallel, user transparent, temporary database that stores all temporary objects created on the library connection. (This means that each library connection establishes at least two database connections, one of them is for the temporary database.) For example, you can create a temporary table temp1 in the temporary database by executing `create temp table temp1(a integer primary key, b varchar)` statement on the library connection. The temporary database is stored in a temporary file in one of the default directories for temporary files in the native file system. The file is not visible to other open connections to the SQLite library by the same or different process. The temporary file is deleted by SQLite when the library connection is closed by the application. The logical structure of the temp catalog is equivalent to a table created by the following SQL statement: `create temp table sqlite_temp_master(type text, name text, tbl_name text, rootpage integer, sql text)`. The columns are as in the `sqlite_master` schema.

[5] If the `rootpage` value is 0, the object does not physically exist. It is non zero for table and index, and zero for view and trigger.

You can query the two master tables by executing SELECT statements, just like they are any other user tables. But you are not allowed to directly change the two tables using INSERT, DELETE, or UPDATE on the tables. Neither you can create indexes on the tables. SQLite does not create indexes on them either. Changes to the master tables have to occur using the CREATE, ALTER, and DROP statements for user objects (such as table, index) because SQLite also has to update some of its internal in-memory catalog objects when tables and indexes are added or destroyed. The SQLite engine automatically carries out these actions. You may note that through the master table, SQLite tracks down all other table- and index trees in the database, and hence, it is the most precious object in an SQLite database.

There are no other create statement related catalogs. There are optional catalogs. For example, SQLite, depending on the need, creates another catalog, named `sqlite_sequence`. If any user table has an 'integer primary key autoincrement' column, SQLite maintains a row in the sequence catalog for the user table. The catalog structure is equivalent to a table created by the following SQL statement: `create table sqlite_sequence(name text, seq integer)`. The `name` column specifies the name of the table. The `seq` is the largest value so far issued for the autoincrement column. (You may note that a table can have at most one autoincrement column, and that is why the column name does not appear in the sequence catalog.) The sequence catalog is created on the very first attempt of your inserting a row in any user table with an integer primary key autoincrement column. Once created, the table is never deleted. SQLite also uses other optional catalogs, and I will discuss some later.

2.5 SQLite Limitations

In the preceding sections, you have seen the strength of SQLite, nonetheless it has some shortcomings too. SQLite is different from most other modern SQL databases in that its primary design goal is to be simple. The SQLite development team keeps this goal in mind to add new features to the DBMS, even if it leads to occasional inefficient implementations of some features. The following is a list of shortcomings of SQLite:

- *SQL-92 feature*: As mentioned previously, SQLite does not support some ANSI SQL-92 features that are available in many enterprise database systems. You can obtain the latest information from the http://www.sqlite.org/omitted.html webpage.

- *No nesting*: SQLite supports only flat transactions; it does not have general nesting capability. (Nesting means the capability of having full fledged subtransactions in a transaction. The latter provides an execution environment for the former.)

- *Low concurrency*: SQLite is not capable of ensuring a high degree of transaction concurrency. It uses file-level locks for the purpose of concurrency control, that is, it detects access conflicts at the granule of database file. It permits many concurrent read-transactions, but only one exclusive write-transaction on a single database file. This limitation means that if any transaction is reading from any part of a database file, all other transactions are prevented from writing any part of the file. Similarly, if any one transaction is writing to any part of a database file, all other transactions are prevented from reading or writing any part of the file.

- *Application restriction*: Because of its limited transactional concurrency, SQLite is only good for small-size transactions where each one does its database work quickly and completes, and hence no database is held up by a transaction for more than a few milliseconds. But there are some applications, especially write-intensive ones, that require finer granule concurrency (table or row level locks instead of database level ones), and you would rather use different DBMS solutions for such applications. SQLite is not intended to replace an enterprise DBMS. It is a good choice in those situations where simplicity of database implementation, maintenance, and administration is more important than the complex features that enterprise DBMSs provide.

- *NFS problems*: SQLite uses file locking primitives supported by the native operating system for concurrency control. This may cause some problems when database files reside on network partitions. Many NFS implementations are known to contain bugs (on Unix and Windows) in their file locking logic. If file locking does not work the way it is expected by SQLite, it is possible to modify the same database at the same time by two or more transactions; this might result in a database corruption. Because this problem arises due to bugs in the underlying file system implementation, the SQLite development team is unable to find a workaround solution to prevent it.

  Another naughty thing is that because of the high latency associated with most NFSs, database performance may not be good. In such environments, where the database files must be accessed across a network, a DBMS that implements a client-server model might be more effective than SQLite.

- *Number and type of database objects*: A table or index is limited by at most $2^{64} - 1$ entries. (Of course, you cannot have so many entries because of the 2^{47} bytes database size limit; I discuss this limit in Section 3.2.1 on page 84.) A single entry can hold up to $2^{31} - 1$ (=2,147,483,647) bytes of data in SQLite's current implementation. (The underlying file format though supports entry sizes up to about 2^{62} bytes of data. Similarly, the maximum size of a string or blob data is $2^{31} - 1$; the default value is one billion.) Upon opening a database file, SQLite reads and preprocesses all entries from the master catalog table and

creates many in-memory catalog objects. So, for best performance, it is better to keep down the number of tables, indexes, views, and triggers. Likewise, there is a limit on the number of columns in a table, index, view, select's result-set, update's set list, number of terms in groupby/orderby, number of values in insert—the default is 2000, but it can be as high as 32,767. But, only the first 63 columns of an index are candidates for certain optimizations. There are other limits on the length of an SQL statement (max is one GB = 2^{30}, default is one million), number of tables in a join (max is 64), etc.

You can change limit values of various parameters (for a library connection) using the `sqlite3_limit` API function per limit category. The limit categories are (1) length of string, blob, or row (of a table), (2) length of SQL statement, (3) number of columns in a table definition, a select's resulting columns, an index, orderby, groupby, (4) depth of a parse tree on any expression, (5) number of terms in a compound select statement, (6) number of arguments in an SQL function, (7) number of attached databases, (8) length of pattern argument in LIKE or GLOB, (9) number of parameters in an SQL statement that have bounded values, (10) recursion depth of a trigger, (11) number of tables in a join, (12) number of pages in a database file. Each category has hard upper bound that cannot be crossed by your applications. See http://www.sqlite.org/limits.html.

- *Host Variable Reference*: In some embedded DBMSs, SQL statements can reference host variables (i.e., those from the application space) directly. This is not possible in SQLite. SQLite instead permits binding of host variables to SQL statements using the `sqlite3_bind_*` API functions for input parameters, and not for output values. This approach is generally better than the direct access approach because the latter requires a special preprocessor that converts SQL statements into special API calls.

- *Stored Procedure*: Many DBMSs have capability to create and store what are called *stored procedure*—a stored procedure is a group of SQL statements that form a logical unit of work and perform a particular task. SQL queries can use these procedures. SQLite does not have this capability.

2.6 SQLite Architecture

The SQLite development team professes a very modular architecture. The architecture consists of seven major component subsystems (also known as modules) partitioned into two divisions: frontend parsing system and backend engine. The frontend compiles each SQL statement and the backend executes the compiled statement. Two block diagrams, showing the component subsystems and how they interrelate, are given in Fig. 2.10. Each is a stack of modules.

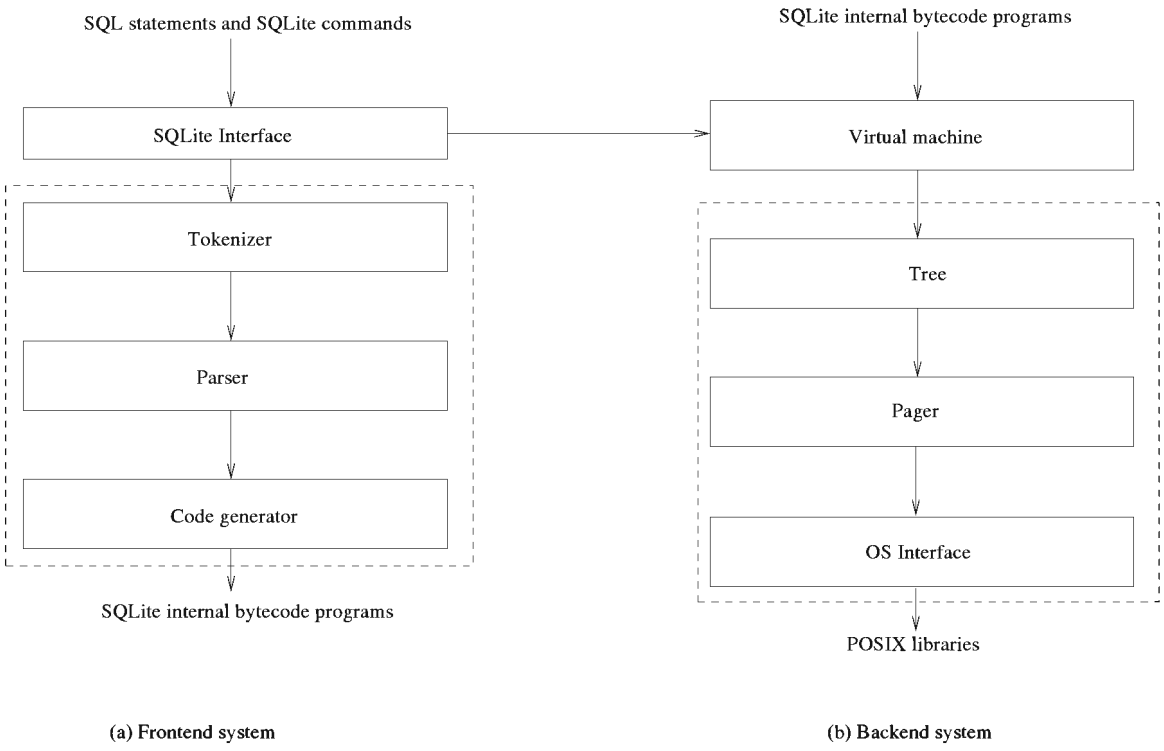


Figure 2.10: Components of SQLite.

Architecture: The architecture of a system provides a model about how the system is compartmentalized into subsystems and how these subsystems communicate among themselves. Said differently, the arrangement of subsystems and their relationships is the architecture. ◁

2.6.1 Frontend

The frontend preprocesses SQL statements and SQLite commands that are sent to it as input by applications. It parses these statements (and commands), optimizes them, and generates equivalent SQLite's internal bytecode programs that the backend can execute. The frontend division is composed of three modules: tokenizer, parser, and code generator.

- *The tokenizer*: It splits an input SQL statement into tokens.
- *The parser*: It analyzes the structure of an SQL statement by analyzing the tokens produced by the tokenizer, and generates a parse tree from the tokens. The parser also includes an optimizer that restructures the parse tree, and finds an equivalent parse tree that will produce an efficient bytecode program.
- *The code generator*: It traverses the parse tree, and generates an equivalent bytecode program that, when executed by the backend, will produce the effect of the SQL statement.

The frontend implements the `sqlite3_prepare` API function. During the function execution, the parsing and code generation steps get interwoven.

2.6.2 Backend

The backend is the engine that executes those bytecode programs generated by the frontend. The engine does the actual database processing work. The backend division is composed of four modules: virtual machine, tree, pager, and operating system interface.

- *The Virtual Machine (VM)*: It executes bytecode programs to carry out the work of the corresponding SQL statements and SQLite commands. It is the ultimate manipulator of data from databases. It sees a database as a collection of tables and indexes, where a table or index is a set of *tuples* or *records*.

- *The tree*: It organizes each tuple set into an ordered tree data structure; tables and indexes in separate B^+- and B-trees, respectively. It helps the VM to search, insert, delete, and update tuples in trees. It also helps the VM to create new trees, and to delete old trees.

- *The pager*: It implements a page-oriented database file abstraction on the top of the native byte-oriented files. It manages an in-memory cache (of database pages) that the tree module uses, and, in addition, it manages file locking and page journaling to implement the transactional ACID properties. It is the data-, lock-, log-, and transaction manager in SQLite.

- *The operating system interface*: It provides a uniform interface to different native operating systems. It is a very thin layer, and it makes SQLite applications independent of the native operating systems. It implements routines for file I/O, thread mutex, sleep, time, random number generation, etc.

The backend implements `sqlite3_bind_*`, `sqlite3_step`, `sqlite3_column_*`, `sqlite3_reset`, and `sqlite3_finalize` API functions.

2.6.3 The interface

Applications cannot directly access the frontend or the backend's (internal) APIs. The former need to channel their requests to the latter via the topmost SQLite interface layer. This is the only means via which database applications interact with the SQLite library. The interface routes their requests to either the frontend or the backend.

2.7 SQLite Source Organization

SQLite source code is organized into a single main directory (named `sqlite`) and seven major subdirectories: `art`, `contrib`, `doc`, `ext`, `src`, `test`, and `tool`. The `art` subdirectory contains many gif files related to the SQLite logo. The `contrib` subdirectory contains a TCL/TK console widget. The `doc` subdirectory contains programmer documentation on Lemon parser generator. The `ext` subdirectory contains version loadable extensions such as async I/O, rtree, fts (full text search), icu (internationalization components for unicode). The `src` subdirectory contains the source code that builds the SQLite library and the sqlite3 executable. It has about 94 of C code and header files (73 .c and 15 .h; 6 more are generated during compilation). There are additional 11 files for FTS3 and RTREE extensions. There are about 114K lines of text (68K code and 46K comments) in those files as of the SQLite 3.7.8 release. The `test` subdirectory contains many regression tests that are geared toward certifying the reliability of the SQLite library. The `tool` subdirectory contains sources for code generators. It contains the source code of the Lemon parser generator: lemon.c and lempar.c, and the source code to the keyword hash table generator used by the tokenizer: mkkeywordhash.c. The top level `sqlite` directory contains several control files. (1) The makefiles and utilities are used for building the SQLite library and the sqlite3 executable from the `src` directory. (2) The VERSION file contains the version number of the release.[6] (3) The configure, configure.ac, Makefile.in, and other files are used by GNU autoconf. (4) Alternative makefiles for Linux, vxworks, and arm and main.mk give more control, and is used for cross-compilation. (5) The publish.sh, that is a shell-script, builds a release for the SQLite website.

In the following subsections, I discuss how SQLite component subsystems/modules are crafted out of the source files in the `src` directory. The modules (see Fig. 2.10 on page 72) are presented in the top-down fashion. Most modules export their own interfaces. SQLite applications must not use those interfaces except the ones whose names start with the `sqlite3_` prefix.

2.7.1 SQLite APIs

A major number of public APIs to the SQLite library are implemented in the main.c, legacy.c, and vdbeapi.c source files. Some APIs are implemented in other source files where they can have access to the locally defined data structures within the file scope. For example, `sqlite3_mprintf` routine is implemented in printf.c, and TCL interface in tclsqlite.c. See the SQLite webpage http://www.sqlite.org/capi3ref.html for more information on SQLite APIs. All SQLite API function names are prefixed by sqlite3_ and API constant names by SQLITE_ prefix.

[6] The format of the version string is "X.Y.Z<trailing string>", where X is the major version number, Y is the minor version number, and Z is the release number. The trailing string is often "alpha" or "beta"; for example "3.3.0beta". In very special circumstances, a version string can be of four orders such as 3.6.23.1.

2.7.2 Tokenizer

When an application sends an SQL statement or an SQLite command string to the SQLite interface for compilation or execution, this passes on that string to the tokenizer. The tokenizer breaks up the original input string into individual tokens and feed those tokens to the parser one by one. The tokenizer code is defined in the tokenize.c source file.

Note: The tokenizer calls the parser in SQLite. People who are familiar with YACC and/or BISON are accustomed to doing things the other way around, that is, the parser calls the tokenizer. Richard Hipp, the architect and the lead developer of SQLite, has experimented it both ways and found out that it works nicer for the tokenizer to call the parser. ◁

2.7.3 Parser

A parser assigns meanings to the tokenizer generated tokens based on their context of use. The SQLite parser is generated using Lemon LALR(1) parser generator. Lemon does the same work as done by the more familiar YACC/BISON. It generates parsers that are reentrant and thread-safe and memory-leak proof. The source file that drives Lemon is found in parse.y. This file defines the SQL grammar that SQLite implements; it also defines SQLite specific commands. The Lemon generates parse.c and parse.h files. The parse.h contains the numeric codes for all token types, and the parse.c implements the SQLite parser.

Note: Lemon parser generator program is not normally found on development machines. The complete source code to Lemon (just one `C` file, lemon.c) is included in the `tool` subdirectory. Documentation on Lemon is found in the `doc` subdirectory. ◁

2.7.4 Code generator

After the parser receives and assembles all tokens of an SQL statement from the tokenizer, it calls the code generator to produce a bytecode program that, when executed by the virtual machine, will produce the result that the SQL statement has requested. There are many files involved in the code generation work: attach.c, auth.c, build.c, delete.c, expr.c, insert.c, pragma.c, select.c, trigger.c, update.c, vacuum.c, and where.c. These are the files where most of the SQLite arithmetic and logic reside. The expr.c file handles code generation for expressions, and where.c code generation for WHERE clauses on SELECT, UPDATE, and DELETE statements. The files attach.c, delete.c, insert.c, select.c, trigger.c, update.c, and vacuum.c handle the code generation for SQL/SQLite statements with the same names. (Each of these files makes calls to routines in expr.c and where.c as necessary.) All other SQL statements are coded out of build.c. The auth.c file implements the functionality of `sqlite3_set_authorizer` API function.

2.7.5 Virtual machine

Bytecode programs produced by the code generator are executed by the virtual machine (VM). A bytecode program is much like a machine-language program—a linear sequence of bytecode instructions. Each bytecode-instruction contains an opcode and up to five operands. The VM reads, decodes, and executes the bytecode instructions one at a time, and thereby, implements an abstract computing machine that is specifically designed to manipulate database files and handle transactions.

The VM itself is entirely contained in the vdbe.c source file. (Vdbe stands for `Virtual DataBase Engine`. VM and VDBE are synonymous in this book.) The VM also has its own header files: vdbe.h defines an interface between the VM and the rest of the SQLite library, and vdbeInt.h defines various data structures that are private to the VM. The vdbeaux.c file contains helper functions that are used by the VM, and interface modules that are used by the rest of the library to construct bytecode programs. The vdbeapi.c file contains external interfaces to the VM such as the `sqlite3_bind_*` family of API functions. Individual values (strings, integers, floats, and BLOBs) are stored in an internal object named 'Mem' which is implemented in vdbemem.c.

SQLite implements SQL functions using callbacks to `C` language routines. Even the built-in SQL functions are implemented this way. Most of the built-in SQL functions (e.g., `coalesce`, `count`, `substr`, and so forth) can be found in func.c file. Date and time conversion functions are found in date.c file.

Memory allocation and case insensitive string comparison routines are available in util.c file. Hashing functions are defined in hash.c. The utf.c source file contains Unicode conversion subroutines between UTF8 and UTF16 texts. SQLite has its own private implementation of printf function (with some extensions) in the printf.c source file and its own random number generator in random.c file.

2.7.6 The tree

The code for handling all indexes and tables that are B- and B^+-trees, respectively, is defined in the btree.c source file. A separate B^+-tree is maintained for each table, and a B-tree for each index. The interface to the tree module is declared in the btree.h source file.

2.7.7 The pager

The tree module requests information from a database file in fixed-size chunks called *database pages*, or simply *pages*. The pager is responsible for reading, writing, and caching database pages. It also provides the rollback and atomic commit abstraction, and coordinates transaction concurrency.

The tree module requests particular pages from the pager and notifies the pager when it wants to modify those pages, or commit/rollback changes. The pager handles all the necessary details of making sure that the requests are handled quickly, safely, and efficiently. It acts as the data manager, the transaction manager, the log manager, and the lock manager of a typical DBMS. The code to implement the pager is defined in the pager.c source file. The interface to the pager module is declared in the pager.h source file.

2.7.8 Operating system interface

In order to provide portability between POSIX, Windows, and other operating environments, SQLite uses an abstract interface layer to interact with various operating systems. It is a very thin layer. The interface to the operating system abstraction layer is declared in the os.h source file; it is called VFS adapter. SQLite obtains services from the platform via the adapter. Each supported operating system has its own implementation: os_unix.c for Unix (and Linux and MAC OS X), os_win.c for Windows (Win32 and WinCE), and osoperating system2.c for OS/2. For Unix, the os_unix.c file contains the locking code. The build process picks up appropriate files during the library compilation.

Linux Primitives Used by SQLite: SQLite uses a very small subset of primitives of the native operating system and file system to get their services. The subset includes open, read, write, close, fcntl, fsync, fdatasync, malloc, free, unlink, access, and some pthread APIs. One can get information about these primitives from Linux manpages. SQLite also uses Linux standard temporary file directories to create temporary files. These temporary files have names prefixed with `etilqs_` followed by 16 random alphanumeric characters, without any file extensions. ◁

I do not talk about operating system interface in this book. You may assume that this layer provides functions such as open, read, write, sync, lock, close, delete, etc., that can be applied on files.

2.8 SQLite Build Process

The build process is depicted in Fig. 2.11. The build process consists of the following six sequential tasks: (1) construction of the sqlite3.h interface file, (2) construction of SQL parser, (3) construction of VM opcodes, (4) construction of opcode names, (5) construction of SQL keywords, and (6) construction of the library.

In the build process, six C files are generated that are used to build the final library. Two intermediate C programs (lemon.c and mkkeywordhash.c) are compiled to run on the build machine to generate three `C` files: keywordhash.h, parse.h, and parse.c. The keywordhash.h file contains a

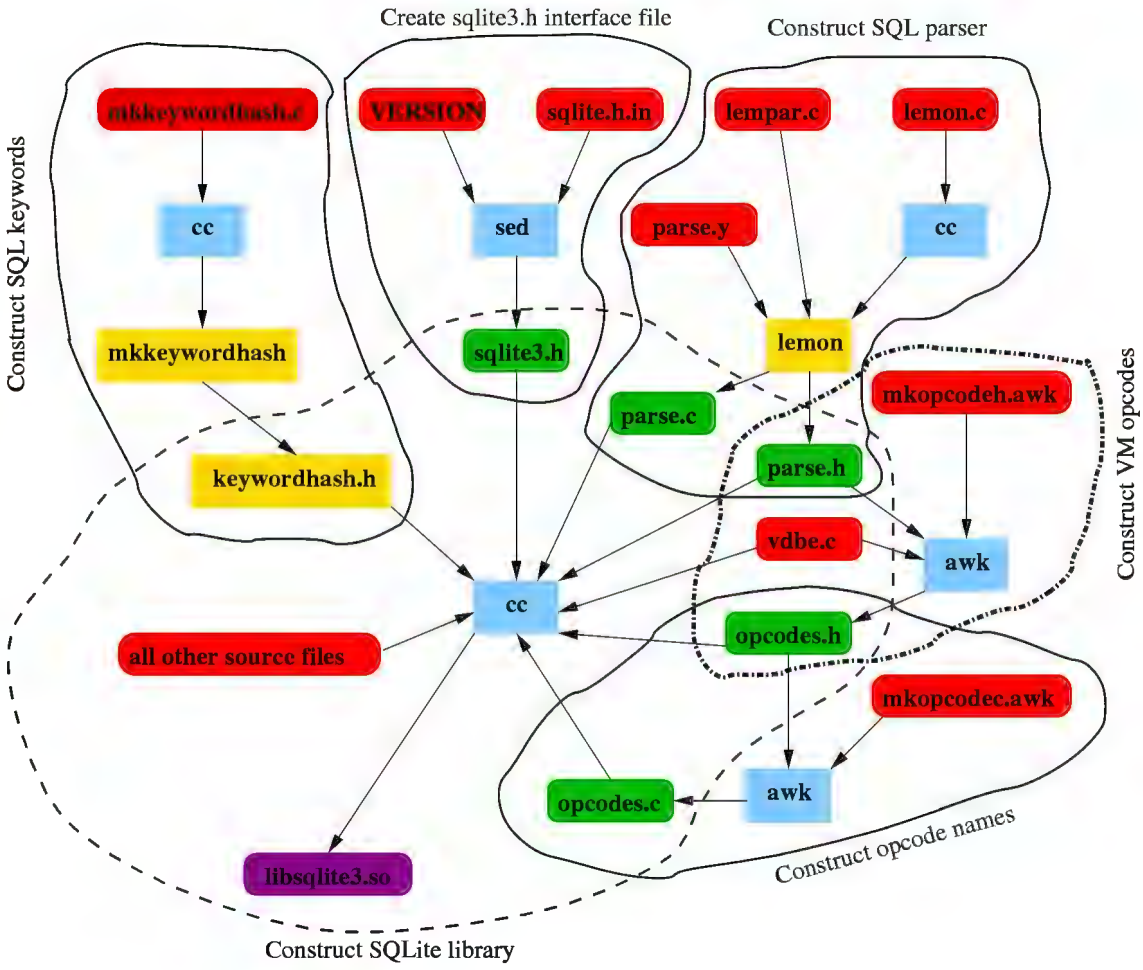


Figure 2.11: SQLite build process.

static hash table for SQL/SQLite keywords. The lemon program produces parser code. The Linux utility programs `awk` and `sed` are used to generate the other three `C` files: sqlite3.h, opcodes.h, and opcodes.c. The sqlite3.h file contains SQLite API function and constant declarations. (SQLite applications need only this file and the SQLite library.) The opcodes.c file contains text names of opcodes for bytecode programming. An awk script scans the vdbe.c source file to create the opcodes.h file that assigns numeric values to opcode symbols.

You may download SQLite source code from its homepage [22] and experiment with the build process. During the configuration of the downloaded source, a Makefile is generated in the root `sqlite` directory. The `make target_source` command does all source code generation and preprocessing, and puts the final cooked C files in a newly created `tsrc` subdirectory. The make command does everything except building the library. To build the library, you need to compile the `tsrc` subdirectory.

Lately, the SQLite development team provides one cooked amalgamation file (sqlite3.c) and the corresponding sqlite3.h file that you can use to build the library. (At the very beginning,

the sqlite3.c file also contains a copy of the sqlite3.h file.) The amalgamation files come with the default setting values for various SQLite compile time options. It has been found that libraries generated from the amalgamation files are 5–10% more efficient because the C compilers can and do more code optimizations. You can also compile the sqlite3.c file with your other .c files to make SQLite as a part of your applications. The SQLite development team highly recommends using the amalgamation files. If you want to use the command line utility `sqlite3`, you also need the shell.c source file. (See http://www.sqlite.org/howtocompile.html page for more details.)

SQLite allows making custom builds where some SQLite features may be turned off using various compilation flags. I talk about these flags in Section 10.18 on page 252.

Summary

SQLite is a SQL-92 specification based embedded relational database management system for database applications written in the C language; the entire SQLite code base is developed in ANSI C. The first SQLite release was on May 29, 2000. It has made a substantial growth since then. It is easy to use in database applications. It has commendable characteristics such as zero-configuration, custom-tailored, embeddable, thread-safe, easily maintainable, transaction-oriented, etc.

This chapter presents some simple single- and multi-threaded SQLite applications exhibiting the use of some frequently used SQLite API functions such as sqlite3_open, sqlite3_close, sqlite3_prepare, sqlite3_step, sqlite3_finalize, sqlite3_exec, sqlite3_reset, sqlite3_bind_*, sqlite3_column_*, etc. These applications show how easy it is to use SQLite to manipulate databases.

This chapter provides an overview of SQLite's way of managing transactions. Each SQL statement is executed in a transaction. The transaction is created automatically by SQLite when the application does not open one manually by executing a begin command. In the former case, we say the system is in the autocommit mode, and SQLite closes (commits or aborts) the transaction at the end of the SQL statement execution. For the latter case, the application has to manually close the transaction by executing either the commit or the rollback command. Until then all SQL statement executions become a part of the transaction. SQLite ensures serializable executions of transactions, and to implement the ACID properties, it uses a database-level locking scheme and a journal based failure recovery scheme.

Each SQLite database is stored in a single native file. The file has at least one catalog (aka, system table), namely `sqlite_master`; for a temporary database it is called `sqlite_temp_master`. The master table is created and initialized when the database itself is initialized. The table contains one row for each table, view, index, and trigger definition (except for the master table). The table has five columns: type, name, tbl_name, rootpage, and sql. Rows are added and deleted from the

table as users create and drop, respectively, database objects. The table is the anchor for the entire database. SQLite uses other optional catalogs, on need basis, such as sqlite_sequence.

SQLite DBMS has a very simple, modular software architecture. There are two divisions: the frontend parsing system and the backend engine. The frontend is composed of three modules: tokenizer, parser, and code generator. The backend is the virtual machine that gets storage support from the tree and pager modules. The lowest level module is the operating system interface that makes SQLite portable to multiple operating systems. The frontend division compiles SQL statements into internal bytecode programs that the backend engine executes.

SQLite is at the open source, and is available in the public domain. One can download the source code and binaries from the http://www.sqlite.org/download.html webpage, and use them for any purpose without any licensing concerns.

This chapter presents a very short tour of the SQLite land. All the concepts introduced in this chapter plus the new ones are elaborately discussed in the following chapters. In the next chapter I discuss the storage structures of database and journal files.

Chapter 3

Storage Organization

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- the organization of a single SQLite database
- the formats of the database and various journal files
- the concept of page in the SQLite context and the purpose of various pages
- how a database is made platform independent

Chapter synopsis

This chapter discusses how SQLite, at the lowest level, organizes the contents of database and journal files. It defines formats of these files. An entire database file is divided into fixed size pages that are used to store B/B$^+$-tree pages, freelist pages, and other pages. In the default journaling mode, a journal file stores before-image contents of database pages as log records; but, in the WAL journaling mode, a journal file stores after-image contents of database pages. This chapter discusses how these files are named and their internal organizations.

3.1 Database Naming Conventions

A database is stored in entirety in a single file, referred to as a *database file*. (The database is synonymous to the file as no other file stores any information about the database proper.) When an application tries to connect to a database by making a call to the `sqlite3_open` API function, it does so by giving the function the name of the database file as an argument. The file name can be the relative path name with respect to the current working directory or the full path name starting

from the root of the system file tree. Any regular file name accepted by the native file system is good. There are, however, two notable exceptions.

1. If the given file name is a C language NULL pointer (i.e., 0) or an empty string ("") or a string containing all white space characters, SQLite creates and opens a *new* temporary file. (But, SQLite tries to keep as much data in-memory as possible for this database.) Different instances of such case will have different temporary files. This is also the case when opening the empty or white-space string via the attach command.

2. If the given file name is ":memory:", SQLite creates an in-memory database. No files are used for this database. If the application opens ":memory:" database two or more times, there will be those many independent in-memory database copies in the process address space, and not one copy! This is also true when opening the :memory: database via the attach command.

In either exceptional case (opened explicitly via sqlite3_open or via the attach command), the database is ephemeral, aka, non-persistent and when the application closes the database connection, the database becomes extinct, i.e., automatically deleted by SQLite.

Database Filename Convention: It is advisable that you choose file names that end with the .db extension. This way you know which are database files. ◁

URI Filename: Starting with SQLite 3.7.7, SQLite support URI (uniform resource identifier) filename. The URI starts with the `file:` prefix. URIs can take query string parameter-value pairs for vfs, mode, cache. To have has this feature the SQLite library must be compiled with the SQLITE_USE_URI compile-time flag. See http://www.sqlite.org/uri.html for more information on this feature. ◁

Usage of Temporary Files: Other than the use of temporary files mentioned in the above item (1), SQLite uses them for many other purposes such as rollback journal, statement journal, multidatabase master journal, transient index, transient databases used by the vacuum command, materialization of views, and subqueries. ◁

Temporary Database File Names: SQLite chooses all temporary file names at random. The names start with the `etilqs_` prefix followed by 16 random alphanumeric characters, without any file extension. You can change the prefix to a different word by using SQLITE_TEMP_FILE_PREFIX compile macro. The files are stored in a standard native temporary file directory. SQLite tries out directories in the order (1) */var/tmp*, (2) */usr/tmp*, and (3) */tmp*. You may however use a different temp directory by setting the TMPDIR environment variable. The temporary files are automatically deleted when they are closed by the application. However, if the application or the system crashes before closing them, the files remain. ◁

In either of the above-mentioned ways of opening databases (persistent file, temporary database, or in-memory database), the database (created and/or) opened by SQLite is internally named as

the *main* database. SQLite maintains a separate temporary database for each library connection the application has opened with the `sqlite3_open` API function; the database is named the *temp* database. See Fig. 3.1, where the application opens the same main database twice via two different library connections; each open instance has its own (different) temporary file. The temp database stores temporary objects such as tables and their indexes. (Applications can use these two names, i.e., main and temp, in their queries. For example, `select * from temp.table1` returns all rows from table1 in the temp database and not from the main database. The catalog name for the temp database is `sqlite_temp_master`.) The temporary objects are only visible within the same library connection (not in other library connections, even if they open the same main database file in the same thread, process, or other processes). As shown in the figure, SQLite stores the temp database in a separate temporary file that is distinct from the main database file. The file is actually created when the application executes the very first `create temporary table table1 ...` statement on the library connection. SQLite deletes the temporary file when the application closes the library connection. The structure of the main and temp databases are the same. Users must not tamper with these temporary files on their own (either modify, delete, or rename) except through SQLite, because otherwise the database may be corrupted.

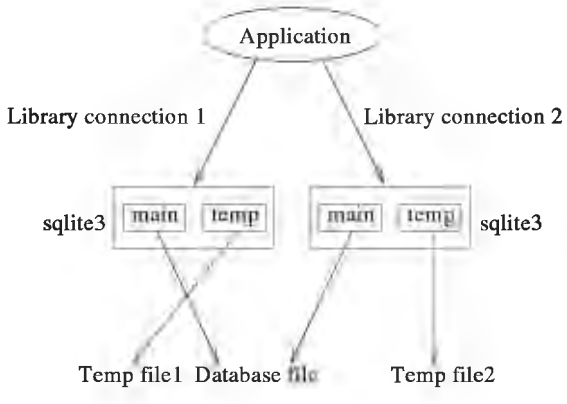


Figure 3.1: Library connections depicting temp databases.

You may note that internally the database file name is not a database name. They are two different but related concepts in SQLite. By using the SQLite attach command, you can associate any database file to a library connection as different database names (other than main, temp, and those already in use in the library connection). You can apply operations on the attached database file via these database names. For example, the SQLite command `attach /home/sibsankar/MyDB as DB1` makes the MyDB database file accessible to the library library connection; the attached database is internally named DB1. `Select * from DB1.table1` returns all rows from the table1 table in the DB1 database, that is, from the MyDB file. There are no inherent add-on temp

database files for the attachments. A database (except the main and the temp) can be released from a library connection by executing the detach command, for example, `detach DB1`.

Connection Confusion: The `sqlite3_open` API function opens a library connection even though the function takes a database file name. This library connection will have connections to the main and temp databases. These latter are referred to as database connections in this book. The 'main' database connection is a connection to the given file. The attach command adds new database connections to a library connection, and the former can be removed from the latter by the detach command. When there are no attached databases, the library connection and the main database connection are synonymous. ◁

3.2 Database File Structure

Except in-memory databases, SQLite stores an entire (persistent or temporary) database in a single database file. In its life span, a database file grows and shrinks. (A database file can grow as long as the native operating system/file system allows the file to grow.) The native file system treats a database file as an ordinary file. It does not interpret the content of the file; it sees the file as a mere byte string. It implements read/write primitives to read/write any amount of bytes in the file starting at any offset position. It also supports sync (aka, flush) operation on files. In the rest of this section I discuss how a database file is structured into (logical) pages and how are those pages organized.

3.2.1 Page abstraction

For ease in space management and reading/writing of data from database files, SQLite divides each database file (including in-memory database) into fixed-size regions called *database pages* or simply *pages*. Thus, a database file size is always a multiple of pages. The pages are numbered linearly starting with 1. The first page is called page 1, the second one page 2, and so forth. Page number 0 is treated as the NULL page or "not a page"—the page does not exist physically. Page 1 and onward are stored linearly one after another into the database file starting at the file offset 0, as shown in Fig. 3.2. You may view a database file as a variable size array of fixed size pages, and the page number is used as index into the array to access the page. (In fact, the pager module creates this abstraction on the top of the native file system.)

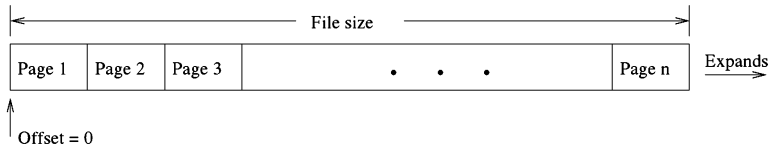


Figure 3.2: Organization of space in a database file partitioned into fixed size pages.

3.2.2 Page size

The default page size is 1024 bytes,[1] but it is a compile time customizable parameter. You can change the value when you compile the SQLite library from the source code. The page size must be a power of 2 and in the range from 512 (= 2^9) to 65,536 (= 2^{16}), both inclusive. This latter bound is a limit imposed by the necessity of storing page size in 2-byte unsigned integer variables in various places in code and external storage. A database file can have as many as 2,147,483,647 (= $2^{31} - 1$) pages; this number is hardcoded in the PAGER_MAX_PGNO macro in the pager.c source file. Thus, a database can be as large as about 140 terabytes ($\approx 2^{16} \times 2^{31} = 2^{47}$) bytes, and of course, is subjected to the limit imposed by the native file system.

Changing Page Size: Once a database file is created, SQLite uses the compile time default page size, but you can change the size by the page_size pragma command before creating the first table in the database. SQLite stores the size value as a part of the file metadata (see Section 3.2.4). It will use this page size value instead of the default one. The database will work flawlessly even if you later use a different SQLite library that is customized for a different default page size. ◁

3.2.3 Page types

SQLite keeps track of all pages that are allocated to a database file, whether or not the pages are currently in active use. It keeps the tracking information in the file itself. It accounts for all pages and there are no dangling pages floating around to be garbage collected. (There is no garbage collection scheme in SQLite.) Based on their use, pages are classified into four types: free, tree, pointer-map (used by autovacuum and incremental vacuum features, see Section 10.8 on page 235), and lock-byte (see Section 4.2.6 on page 105). Tree pages are subtyped into leaf, internal, and overflow. Free pages are inactive (aka, currently unused) pages; the others are active pages and they belong to B- or B^+-trees with exception of pointer-map and lock-byte pages. *B^+-tree internal pages* contain navigational information for searching the tree. (B-tree internal pages have both search information and data). *Leaf pages* store actual data (e.g., table rows) in B^+-trees. If a row's data is too large to fit in a single page, part of the data is stored in the tree page, and the remaining part in one or more *overflow pages.*

3.2.4 Database metadata

SQLite can use any database page for any page type. Page 1 is the lone exception, which is always a B^+-tree internal page storing the root node of sqlite_master or sqlite_temp_master. The page also contains a 100 byte file header record that is stored starting at file offset 0. See Fig. 3.3.

[1]The SQLite development team has decided on the default value by trial and error method after running many benchmark applications in year 2000.

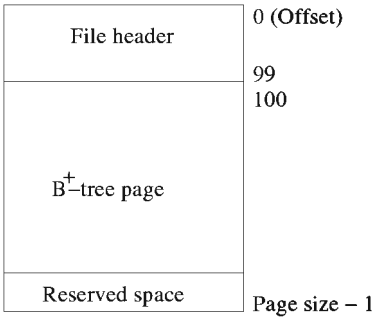


Figure 3.3: Gross structure of Page 1.

The file header information characterizes the structure of the database file. It defines various values for various database setting parameters, aka, metadata. SQLite initializes the header when it creates the file. It uses Page 1 (both the file header and the B^+-tree) as the stable anchor to track down other pages in the file. The format of the file header is given in Fig. 3.4. The first two columns in the figure are in bytes.

Offset	Size	Description
0	16	Header string: "SQLite format 3"
16	2	Page size in bytes.
18	1	File format write version
19	1	File format read version
20	1	Bytes reserved at the end of each page
21	1	Max embedded payload fraction
22	1	Min embedded payload fraction
23	1	Min leaf payload fraction
24	4	File change counter
28	4	Size of the database file in pages
32	4	First freelist page
36	4	Number of freelist pages in the file
40	4	Schema cookie number
44	56	14 4-byte meta values passed to higher layers

Figure 3.4: Structure of database file header.

Here is a description of each metadata element (aka, database setting parameter):

- *Header String*: This is the 16 byte string: "SQLite format 3" in the UTF-8 format. (You can choose a different 16-byte string, including the null terminator, at compile time by defining the SQLITE_FILE_HEADER compile macro.)

- *Page Size*: This is the size of each page in this database. The value must be a power of 2 between 512 and 32,768 (both inclusive), or 1 representing the value 65,536. As mentioned above, SQLite uses this page size when it manipulates the file. (The same SQLite library can

work with database files having different page sizes, at the same time.)

- *File Format:* The two bytes at offsets 18 and 19 are used to indicate the file format version They both have to be 1 or 2 in the current version of SQLite, or else it is an error and the database is not accessed. The value 1 is for legacy rollback journaling (prior to SQLite 3.7.0 release) and 2 for WAL journaling introduced in the SQLite 3.7.0 release. If the read or write value is greater than 2, the database cannot be read or written, respectively. If in the future the file format is changed again, these numbers will be increased to indicate the new file format version number.

- *Reserved Space*: SQLite may reserve a small fixed amount of space (<= 255 bytes) at the end of each page for its own purpose, and this value is stored at offset 20; the default value is 0. It is nonzero when a database uses SQLite's built-in (proprietary) encryption technology. The extra bytes on the end of each page store a nonce that is used by the encryption algorithm for that page. The first part of a page (page size minus reserved size) is the *usable space* where database content proper is stored. This space must be at least 480 bytes.

- *Embedded Payload*: The *max embedded payload fraction* value (at offset 21) is the amount of the total usable space in a page that can be consumed by a single entry (called a cell or record) of a standard B/B$^+$-tree internal node. A value of 255 means 100 percent. The value must be 64 (i.e., 25 percent): the value is to limit the maximum cell size so that at least four cells fit on one node. If the payload for a cell is larger than the max value, then part of the payload is spilled into overflow pages. Once SQLite allocates an overflow page, it moves as many bytes as possible into the overflow page without letting the cell size to drop below the *min embedded payload fraction* value (at offset 22). The value must be 32, i.e., 12.5 percent.

  The *min leaf payload fraction* value (at offset 23) is like the min embedded payload fraction, except that it is applicable only for B$^+$-tree leaf pages. The value must be 32, i.e., 12.5 percent. The max payload fraction value for a leaf node is always 100 percent (or 255), and hence is not specified in the header. (There are no special-purpose leaf nodes in B-trees.)

  Original purposes of these three fields are not supported of late.

- *File Change Counter*: The *file change counter* (at offset 24) is used by transactions. The counter is initialized to 0. The counter value is incremented by every successful write-transaction that writes the database. This value is intended to indicate when the database has changed so that the pager can purge its page cache. (This counter is not used when the file format indicates the WAL journaling.)

- *Database Size*: The number of pages the database holds currently is stored at offset 28.

- *Freelist*: The freelist of unused pages originates in the file header at offset 32. The total number of free pages is stored at offset 36. See the next subsection for more about freelist organization.

- *Database Schema Cookie*: A 4-byte integer number is stored at offset 40; initialized to 0. The value is incremented by one whenever the database schema changes and it is used by prepared statements for their own validity testing.

- *Other Meta Variables*: At offset 44, there are fourteen 4-byte integer values that are reserved for the tree and the VM modules. They represent values of many meta variables, including the schema format number at offset 44,[2] the (suggested) page cache size at 48, the autovacuum related information at 52 (0 for no autovacuum, otherwise the page number of the furthest root page in the database file), the text encoding at 56 (value 1 represents UTF-8, 2 represents UTF-16 LE, and 3 represents UTF-16 BE), the user version number at 60 (not used by SQLite, but by users), incremental vacuum mode at 64 (0 for no vacuum, and other values for vacuum), and version numbers at offsets 92 and 96;[3] the remaining bytes are reserved for future use and must be zeroed. You can find more information about these variables in the SQLite source files, notably btree.c.

Incremental Vacuum vs. Autovacuum: If the 4-byte integer at file offset 52 is 0, then the 4-byte value at offset 60 must be 0; no automatic vacuum will be performed on the database file. Users can though perform manual vacuum by executing the `vacuum` command. If the 4-byte integer at file offset 52 is non zero and the 4-byte value at offset 60 is zero, then autovacuum is on; otherwise, autovacuum is off and incremental vacuum is on. ◁

Cross Platform Use: SQLite stores all multibyte integer values in the big-endian (the most significant byte first) order. This lets you safely move your database files from one platform to another. For example, you can create a database on x-86 machine, and use the same database (by making a blind copy) on ARM platform without any alterations. The database works expectedly on the new platform. ◁

Backward Compatibility: The database file format is backward compatible back to version 3.0.0. It means that any later version of SQLite can read and write a database file that was originally created by version 3.0.0. This is mostly true in the other direction—version 3.0.0 of SQLite can usually read and write any SQLite database created by later versions of the library. However, there are some new features introduced by later versions of SQLite that version 3.0.0 does not understand, and if the database contains these optional new features, older versions of the library will not be able to read and understand it. ◁

[2]Different SQLite major releases may use different internal file formats. This meta information is used to detect database versus library mismatch. The value is 1, 2, 3, or 4.

[3]The version number of the SQLite library that modified the database file lately is stored at offset 96. When this value is changed, the value of the then file-change-counter (from offset 24) is saved at offset 92.

As mentioned previously, the file header is followed by a B^+-tree internal node on Page 1. The node is the root of the master catalog table, named `sqlite_master` or `sqlite_temp_master` for a regular (aka, main or attached) or temp database, respectively. As mentioned in Section 2.4 on page 67, this table stores rootpage numbers of all other trees in the same database. Thus, Page 1 helps SQLite to keep track of other tree- and overflow pages. It is the most precious page.

3.2.5 Structure of freelist

The freelist of unused pages originates in the file header at offset 32. The total number of free pages is stored at offset 36. The freelist is organized in a rooted trunk as shown in Fig. 3.5. Freelist pages come in two subtypes: trunk pages and leaf pages. The file header points to the first one on the linked list of trunk pages. Each trunk page points to multiple leaf pages. (A leaf page content is unspecified and can be garbage.)

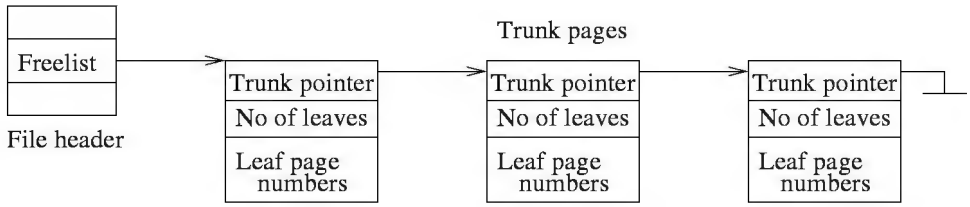


Figure 3.5: Structure of freelist.

A trunk page is formatted like the following structure, starting at the base (aka, lower offset) of the page: (1) a 4-byte page number of the next trunk page (or 0 if there is no next page); (2) a 4-byte integer value to indicate the number of leaf pointers stored on this page; and (3) zero or more 4-byte page numbers for leaf pages.

Note: Because of a bug in releases prior to 3.6.0, SQLite does not write the last 6 entries on the trunk pages. ◁

When a page becomes inactive, SQLite adds it to the freelist, and does not release it to the native file system. When you add new information to the database, SQLite takes out free pages off the freelist to store the information. (Thus, the new information may be stored anywhere in the database file.) If the freelist is empty, SQLite acquires new pages from the native file system, and appends them at the end of the database file.

In some cases, you might be concerned when the number of free pages becomes too high. You can run the VACUUM command to purge the freelist and to shrink the database file and to release the unused pages back to the file system. Databases created in the "autovacuum" mode will automatically shrink the database at each transaction commit. The freelist remains empty across transactions though a transaction may build up the freelist before the commit.

Freelist Purging: When you purge the freelist by executing the `vacuum` command, the command makes a copy of the database into a temporary file. Then, it overwrites the original database with the temporary copy, under the protection of a transaction. ◁

3.3 Journal File Structure

SQLite uses three kinds of journal files, namely rollback journal, statement journal, and master journal. (These are called legacy journals. In SQLite 3.7.0 release, the SQLite development team has introduced the WAL journaling scheme. A database file can be either in the legacy journaling exclusive-or in the WAL journaling mode.) Their structures along with log record structures for legacy journaling are presented in the next three subsections. I talk about WAL journaling in Section 10.17 on page 249.

3.3.1 Rollback journal

For every database, SQLite maintains a single rollback journal file. (In-memory databases do not use journal files. They use the main memory to store journaling information.) The rollback journal always resides in the same directory as the original (main, temp, or attached) database file does. The journal file name is the same as the database file name with '-journal' appended. You may note that the journal is a transient file in the default operating mode. SQLite creates the journal file for every write-transaction and deletes the file when the transaction is complete.

SQLite partitions each rollback journal file into variable-sized log segments, as shown in Fig. 3.6. Each log segment starts with a segment header record followed by one or more log records. These are discussed in the next two sub-subsections.

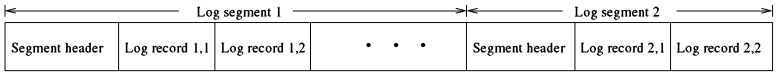


Figure 3.6: Organization of space in a rollback journal file into variable size segments.

3.3.1.1 Segment header structure

The format of the segment header is shown in Fig. 3.7. The header begins (at lower offset) with the following eight magic bytes in this order: 0xD9, 0xD5, 0x05, 0xF9, 0x20, 0xA1, 0x63, 0xD7. The sequence will be referred to as the *magic number* in the sequel. The number was chosen randomly once for all and is used for a sanity check only, and it has no other special significance. The number of records (nRec, for short) component specifies how many valid log records are there in this log segment. The nRec value is initially -1 (which is 0xFFFFFFFF as a signed value) for asynchronous

transactions (see below at the end of this sub-subsection) and 0 for synchronous transactions. The value of the random number component is used to compute the 'checksums' for individual log records. The initial database page count component notes down how many pages were there in the original database file when the current journaling started. The sector size is the size of the disk sector where the database file resides. The segment header occupies a complete disk sector. That is, the header size is equal to the sector size specified in the header itself. The page size is the size of database pages. The unused space in the header is kept there as filler. All integer numbers are stored in the big-endian format.

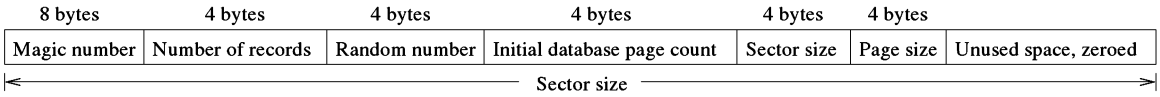


Figure 3.7: Structure of journal segment header.

Sector Size Determination: SQLite inquires the underlying file system to get the sector size. (If the file system does not have the information, 512 byte is the default sector size.) SQLite assumes that the file system does not allow changing individual bytes on a sector in place. We need to write or read sector by sector from the file system. ◁

A rollback journal file normally contains a single log segment. But, in some situations, it is a multi-segment file, and SQLite writes the segment header record multiple times in the file. (I discuss such scenarios in Section 5.4.1.3 on page 140.) Each time the segment header record is written, it is written at the sector boundary. In multi-segment journal file, the nRec field in any segment header cannot be -1 (i.e., 0xFFFFFFFF).

Journal File Retention: In the default operating mode, SQLite deletes the journal file at the transaction commit/abort time. However, you can change this behavior by using the `pragma journal_mode` command to persist the journal (header is zeroed/invalidated or the file is truncated). That is, the journal_mode is DELETE, PERSIST, or TRUNCATE; the default is DELETE. There are other journal modes that I address later in this book. If the application uses exclusive locking mode (`pragma locking_mode = exclusive`), SQLite creates the rollback journal and this journal persists until the application moves out of the exclusive locking mode. In this case, across transactions, the journal file is truncated or its header is zeroed. ◁

Journal Maintenance: A rollback journal is valid if it contains a valid segment header at offset 0. If the file size is zero or contains an invalid segment header, the journal is not used for transaction rollback. ◁

Asynchronous Transaction: SQLite supports asynchronous transactions that are faster than normal (synchronous) transactions. Asynchronous transactions neither flush the journal nor the database file at any time. The journal file will have only one log segment. The nRec value is always -1, and the actual value is derived from the size of the file. SQLite does not recommend using asynchronous transactions, but you can set the asynchronous mode by executing a pragma command. This mode is normally used at

application development phase to reduce the development time. This mode is also satisfactory for testing some applications that do not need to test for recovery from failures. ◁

3.3.1.2 Log record structure

Non-SELECT statements from the current write-transaction produce log records. SQLite uses an old value logging technique at the page level granularity. Before altering any page for the first time, the original content of that page (along with its page number) is written into the journal in a new log record. Figure 3.8 depicts the structure of an individual log record. The record also contains a 32-bit checksum. The checksum covers both the page number and the page image. The 32-bit random value that appears in the log segment header (see the third component in Fig. 3.7) is used as the checksum key. The random number is important because garbage data that appears at the end of a journal is likely data that was once in other files that have now been deleted. If the garbage data came from an obsolete journal file, the checksums might be correct. But, by initializing the checksum to a random value that is different for different journal file, SQLite minimizes that risk.

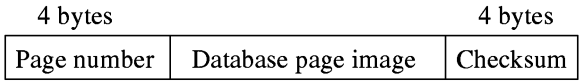


Figure 3.8: Structure of a log record.

Checksum Position: You may note that the page number is stored at the beginning of the log record and the checksum is stored at the end. This setup is important. The SQLite development team assumes that data in a file is written linearly as a byte string. So, if journal corruption occurs due to a power failure, the most likely scenario is that one end or the other of the record will be corrupted. It is very much unlikely that the two ends of the log record are correct and the middle is corrupt. Thus, this checksum scheme, though fast and simple, catches the most likely kinds of corruption. ◁

3.3.2 Statement journal

When in a user transaction, SQLite maintains a statement subjournal for the latest INSERT, UPDATE, or DELETE SQL statement execution that could modify multiple rows and that might result in a constraint violation or a raise exception within a trigger. The journal is required to recover the database from the statement execution failure. A statement journal is a separate, ordinary rollback journal file. It is an arbitrarily named temporary file (prefixed by etilqs_); it resides in a native directory for temporary files. The file is not required for crash recovery operation; it is only needed for statement aborts. SQLite deletes the file when the statement execution is complete. The journal does not have a segment header record. The nRec (number of log records) value is kept in an in-memory data structure, and so is the database file size as of the start of the statement

execution. These log records do not have any checksum information.

Statement Journal Retention: Statement journal files are deleted at the end of the statement execution. But, to implement SAVEPOINTs, SQLite retains statement journals until savepoints are released or the user transaction commits. ◁

3.3.3 Multi-database transaction journal, the master journal

You may recall that an application can attach additional databases to an open library connection by executing the ATTACH command. In this scenario, SQLite permits a user transaction to read and modify multiple databases. (See Fig. 2.8 on page 62.) If a transaction modifies multiple databases, then each database has its own rollback journal. They are independent rollback journals, and not aware of one another. Such a transaction is individually committed at each database it has updated. Thereby, the transaction may not be globally atomic. To make a (multidatabase) transaction globally atomic, SQLite additionally maintains a separate aggregate journal called the *master journal*.[4] The journal does not contain any log records that are used for rollback purposes. It instead contains the UTF8-formatted names of all the individual rollback journals that participate in the transaction. (In this context, a rollback journal is called a *child journal*.) The child journal names are full path names, and are separated by the null character in the master journal file. The master journal always resides in the same directory as the main database file does, and has the same name with '-mj' appended, followed by eight randomly chosen 4-bit hexadecimal numbers. It is a transient file; it is created when the transaction attempts to commit, and deleted when the commit processing is complete. Transaction-aborts do not create the master journal file.

Each child journal also contains the (full path) name of the master journal as shown in Fig. 3.9. (If there is no attached database, or if no attached database is participating in the current transaction for update purpose, no master journal is created, and the child journal does not contain any information about the master journal.) As shown in the figure, the master-journal-record is the last entry appended to a child journal file, and this is done at the transaction commit time. The record is aligned at the disk sector boundary. The checksum field stores the checksum of the master journal name. The length field specifies the length of the master journal name. The master journal name is always stored in UTF8 format text, excluding the terminating null character. The forbidden page number is the number of the page that contains the lock offset bytes (see Chapter 4): SQLite never writes the lock-byte page; it is reserved for working around a windows/POSIX incompatibility issue.

[4] If the main is an in-memory database, the master journal is not used.

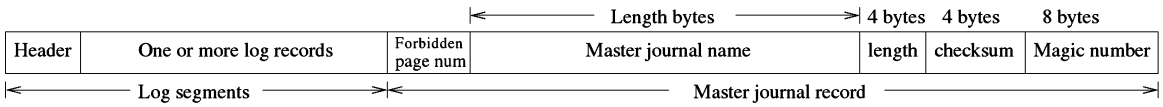


Figure 3.9: Structure of child journal file.

Summary

SQLite stores all (user and meta) information pertaining to an entire database in a single native file. The file name is synonymous to the database name. SQLite also supports temporary and in-memory databases. These databases are created and initialized when an application opens them, and they are deleted when the application closes them.

Each database file is a multiple of fixed-size pages. Logically, a database is an array of pages, and the array can expand and shrink. The default page size is 1024, but it can be set to a value between and inclusive 512 and 65,536, but the value must be a power of 2. A database file can have as many as $2^{31} - 2$ pages. There are four types of pages: free, tree, lock-byte, and pointer-map. Tree pages are subtyped as internal, leaf, and overflow.

The first page (starting at file offset 0) is the anchor page. The first 100 bytes store database metadata such as header string, page size, file format, etc., and the remaining space on the page holds the root of the sqlite_master or sqlite_temp_master catalog. All free pages are organized in a rooted trunked tree.

SQLite uses three types of journal files for the (default) legacy journaling scheme: rollback, statement, and master journals. The rollback and master journals reside where the main database does, but the statement journal resides in a temporary directory such as /tmp. The rollback journal stores variable-sized log segments, where each segment starts with a header and is followed by one or more log records. A log record is composed of a page number, the old image of the page, and a checksum. The master journal records the names of all rollback journals involved in a single multidatabase transaction. The statement journal stores logs from a single insert/update/delete statement execution.

Chapter 4

Transaction Management

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- how SQLite implements ACID properties
- the SQLite way of managing various locks and their mappings to native file locks and lock transitions
- how SQLite avoids deadlocks
- how SQLite implements journaling protocols
- how SQLite manages savepoints in user transactions

Chapter synopsis

The primary duty of database management systems (DBMSs) is helping users to apply operations on databases to manipulating data stored there. The DBMSs, in addition, protect databases from multiple concurrent users, and recovers databases to acceptable consistent states in the occurrences of application, system, or power failures. For these purposes, the DBMSs execute database operations in the abstraction of transactions. Managing transactions is crucial for maintaining database consistencies. The DBMSs implement ACID transactional properties to ensure database consistencies. SQLite relies on native file locking schemes and does page journaling to implement ACID properties. You may recall that SQLite supports only flat transactions; it does not have the nesting capability. This chapter discusses how SQLite implements ACID properties for transactions that may update a single or multiple databases. In the next chapter I will explain inner working of the SQLite transaction manager called pager.

4.1 Transaction Types

Almost all DBMSs use locking mechanisms for concurrency control, and journals to save recovery information. Before a transaction modifies a database item, a DBMS writes some log record containing recovery information (for example, old and new values of the item) in the journal. The DBMS makes sure that the log record reaches the stable storage before the item in the original database is changed. In the event of transaction abort or other failures, the DBMS has enough information available in the journal to recover the database into an acceptable consistent state that has all updates from all committed transactions and has no effects of other (aborted or failed) transactions. At the time of recovering a database, the DBMS undoes the effects of aborted/failed transactions from the database, and redoes the effects of committed transactions on the database. In SQLite, locking and logging activities depend on transaction types that I discuss in this section. Their management is discussed in the following two sections.

4.1.1 System transaction

SQLite executes each SQL statement in a transaction. It supports both read- and write-transactions. (In SQLite documentation, the use of the term transaction is a bit abstract. A transaction may mean only a read transaction or write transaction. Sometimes it is more than that: it encompasses a write transaction and one or more read transactions. In this book, whenever needs arise I will spell out the type of transactions.) Applications cannot read any data from a database except in a read- or write-transaction, and they cannot write any data except in a write-transaction. They do not need to explicitly inform SQLite to execute individual SQL statements within (appropriate) transactions. SQLite automatically does so; this is the default behavior, and the system is said to be in the *autocommit* mode. Those transactions are called *system* or *auto* or *implicit transactions*. For a SELECT statement, SQLite creates a read-transaction to execute the statement. For a non-SELECT statement, SQLite first creates a read-transaction, and then converts it into a write-transaction. Each (read- or write-)transaction is automatically committed (or aborted) at the end of the statement execution. Applications are not aware of system transactions. They do not have program fragments to handle system transactions. They submit SQL statements to SQLite that takes care of the rest as far as ACID properties are concerned. Applications receive back outcome of the SQL statement executions from SQLite.

An application can initiate concurrent executions of (same or different) SELECT statements (aka, read-transactions) on the same database connection, but it can initiate only one non-SELECT (aka, write-transaction) on the database connection. This implies that on a database connection you cannot have two concurrent write-transactions, but can have concurrent one write-transaction and multiple read-transactions; they are often collectively imagined to be a single transaction. How

these transactions are executed on the database is explained in the next paragraph.

Non-select statements are executed atomically; SQLite takes a mutex when it starts a non-select statement processing, and it releases the mutex only on the completion of the statement execution. A select statement, on the other hand, is not executed atomically, end to end; it takes a mutex on start, but it pauses and releases the mutex for each row of result. So one select statement execution can run up to its first row, then another select statement execution can run, then another one, and so on and so forth. Thereby, you can have many different select statements in various stages of their executions at any one time. And you can execute non-select statements during the pauses as well. The read- and write-transactions are though appear indivisible with respect to those transactions initiated on other connections to the same database by this thread, process, or other processes. As I will explain you later that a read-transaction and a write-transaction cannot manipulate the same table simultaneously. Thereby, read-transactions are insulated from concurrent write-transactions.

4.1.2 User transaction

The default autocommit mode might be expensive for some applications, especially for those that are highly write intensive, because SQLite requires reopening, writing to, and closing the journal file for each non-select statement execution. In addition, there is also concurrency control overhead, as applications need to reacquire and release locks on database files for each SQL statement execution. This overhead can incur a significant performance penalty (especially for large applications), and can only be curtailed by opening a user-level transaction surrounding many SQL statements. The application encompasses a sequence of SQL statements within a 'BEGIN TRANSACTION' and a 'COMMIT [or ROLLBACK] TRANSACTION' command. (The keyword TRANSACTION is optional.) A `begin—commit` or `begin—rollback` pair can enclose any number of select and non-select statements.

An application can manually start a new transaction on a library connection by explicitly executing a BEGIN command. The transaction is referred to as a *user-level* or *explicit transaction* (or simply a user transaction). This actually will open an individual user transaction on ease database connection that is a part of the library connection. When this is done, SQLite leaves the default autocommit mode; it does not invoke a commit or abort at the end of each SQL statement execution. Successive executions of non-select SQL statements on the library connection (any database connections) become a part of the user transaction; but executions of select statements are treated as separate read-transactions. (You may imagine that the user transaction is the lone write-transaction.) SQLite commits (respectively, aborts) the user transaction when the application executes the COMMIT (respectively, ROLLBACK) command on the library connection. If the user transaction commits, the write-transaction is committed but all current read-transactions

remain active. If the user transaction aborts, the write-transaction is rolled back and some of read-transactions that read tables updated by the write-transaction are aborted too. SQLite reverts to the autocommit mode on completion of the write-transaction. (The remaining read-transactions are committed independently at the end of the respective select statement executions.)

Note: SQLite supports only flat transactions. It does not have nested transaction capability. It is an error to execute a begin command inside a user transaction. An application cannot open more than one user transaction on a library connection at a time. ◁

4.1.3 Savepoint

SQLite supports the savepoint capability for user transactions. An application can execute a savepoint command inside or outside a user transaction. For the latter case, SQLite first opens a user transaction and then executes the savepoint command and commits the transaction when the application releases the savepoint. A *savepoint* is a point in a transaction execution, which is established by the application. It establishes a database state that it considers good. A transaction can set multiple savepoints. Later, it may rollback to one of the savepoints, and reestablish the database state as of the start of the savepoint.

4.1.4 Statement subtransaction

A user transaction is a little more than a flat transaction. All successive non-select statement executions are put in that transaction. (As mentioned previously, selects are treated separately.) Each such non-select statement in the transaction is executed in a separate statement level *subtransaction.* At any point in time, there can be at most one subtransaction in the user transaction. SQLite actually uses an implicit, aka, anonymous savepoint to execute the subtransaction at the end of which it releases the savepoint. This process continues until the application executes a COMMIT or an ABORT command.[1] If the current statement execution fails, SQLite does not abort the enclosing user transaction unless conflict resolver indicates a rollback (see the end of this subsection). It instead restores the database to the state prior to the start of the statement execution (by restoring the anonymous savepoint); the user transaction continues from there. The failed statement does not alter the outcome of other previously executed SQL statements, or the new ones, unless the main user transaction aborts itself. Let us study a simple SQLite application here that creates a user transaction.

```
BEGIN TRANSACTION;
    INSERT INTO table1 values(100);
```

[1] If the application closes the database connection, by default, SQLite aborts the on-going transaction along with the statement subtransaction.

```
        INSERT INTO table2 values(20, 100);
        UPDATE table1 SET x=x+1 WHERE y > 10;
        INSERT INTO table3 VALUES(1,2,3);
    COMMIT TRANSACTION;
```

The database is assumed to have three tables, namely table1, table2, and table3. The application opens a user transaction by executing the `begin transaction` command. Each of the four statements is executed in a separate subtransaction, one after another, in the said order. If, for example, a constraint violation error occurs in the middle of the UPDATE statement execution, all previous rows modified by the update execution will be restored, but the changes from the three INSERTs (surrounding the UPDATE) will be committed when the application executes the `commit transaction` command.

Conflict Resolution: When a constraint violation happens during an insert or update, there are five ways a statement can resolve a conflict. `Rollback:` abort the encompassing transaction along with the current statement subtransaction. `Abort:` annul the changes of the current statement subtransaction and stop it; the encompassing transaction remains alive. This is the default resolution. `Fail:` accept the changes of the current statement subtransaction done so far, but stop it making further progress; the encompassing transaction remains alive. `Ignore:` the particular rows causing constraint violations are not changed or inserted; the subtransaction remains alive and makes further progress. `Replace:` the rows causing constraint violations are removed; the subtransaction remains alive and makes further progress. ◁

4.2 Lock Management

A system that permits more than one concurrent (conflicting) transaction requires to have synchronization mechanisms to isolate one transaction from the effects of others. The isolation related terms that are frequently used in the literature are concurrency control, serializability, and locking. They are different but closely related terms. Gray and Reuter [9] correlates the terms as follows: concurrency control is the problem to be solved, serializability is the theory to deal with properties of the problem, and locking is a mechanism to solve the problem. The generic term is isolation, the 'I' of ACID. There are many notions of transactional isolation: read committed, cursor stable, repeatable read, serializable. See [9] for definitions of these terminologies. Serializable is the strictest isolation level, and SQLite implements it.

SQLite, toward producing serializable execution of transactions, uses a locking mechanism to regulate database access requests from transactions. It does not use shared memory among concurrent processes. Consequently, it cannot coordinate transactional concurrency in the user space. It, instead, uses file locking primitives supported by the native operating system (actually the file

system) for this purpose.

SQLite simplifies the complexity of the locking logic by severely restricting the degree of transactional concurrency. It permits any number of read-transactions to read the same database, but allows at most one write-transaction to exclusively access the database. (You may recall that on the same database connection a write-transaction and one or more read-transactions can though coexist.) SQLite lock management is very simple. As of now, SQLite does database level locking, but neither row nor page nor table level locking: it sets locks on the entire database file, and not on particular data items in the file. In addition, to produce serializable executions of transactions, it follows strict two phase locking protocol, i.e., it releases locks only on transaction completion. In the rest of this section I discuss how SQLite manages various locks, and subtleties in Linux systems.

Note: A statement subtransaction acquires locks through the container user transaction. All locks are held by the user transaction until it commits or aborts irrespective of the outcome of the subtransaction. ◁

4.2.1 Lock types and their compatibilities

From the view point of a single transaction, a database file can be in one of the following five locking states. You may note that these are SQLite locks, and not those supported by the native operating system. (I will discuss the mapping between SQLite locks and native locks in Section 4.2.6 on page 105.)

1. *NOLOCK*: The transaction does not hold any type of lock on the database file. It can neither read nor write the database file. Other transactions can read or write the database as their own locking states permit. Nolock is the default state when a transaction is initiated on a database file.

2. *SHARED*: This lock only permits reading from the database file. Any number of transactions can hold shared locks on the file at the same time, and hence there can be many simultaneous read-transactions. No transaction is allowed to write the database file while one or more transactions hold shared locks on the file.

3. *EXCLUSIVE*: This is the only lock that permits writing (and reading too) the database file. Only one exclusive lock is allowed on the file, and no other lock of any type is allowed to coexist with an exclusive lock.

4. *RESERVED*: This lock permits reading from the database file. But, it is a little more than a shared lock. A reserved lock informs other transactions that the lock holder is planning to write the database file in the future, but it is now just reading the file. There can be at most one reserved lock on the file, but the lock can coexist with any number of shared locks. Other transactions may obtain new shared locks on the file, but not other locks of any type.

5. *PENDING*: This lock permits reading from the database file. A pending lock means that the transaction wants to write to the database immediately. It is an intermediate lock on the way to acquiring an exclusive lock. The transaction is just waiting on all current shared locks to clear prior to its acquiring an exclusive lock. There can be at most one pending lock on the file, but the lock can coexist with any number of shared locks with one difference: other transactions may hold on their existing shared locks, but may not obtain new (shared or other) ones. (This lock type is internal to the lock manager and not visible to its clients.)

The compatibility matrix among SQLite locks is given in Fig. 4.1. The rows are the existing lock mode on the database held by a transaction, and the columns are a new request from another transaction. Each matrix cell identifies the compatibility between the two locks: Y indicates the new request can be granted; N indicates that it cannot.

	SHARED	RESERVED	PENDING	EXCLUSIVE
SHARED	Y	Y	Y	N
RESERVED	Y	N	N	N
PENDING	N	N	N	N
EXCLUSIVE	N	N	N	N

Figure 4.1: Lock compatibility matrix.

Discussion: A database system such as SQLite needs at least the exclusive lock; the other lock modes are used just to increase transactional concurrency. With only exclusive lock, the system will execute all (read and write) transactions serially. With shared and exclusive locks, the system can execute many read-transactions concurrently. In practice, a transaction reads a data item from a database file under a shared lock, modifies the item locally, and then requests for an exclusive lock to write the (modified) item back into the file. If two transactions do so simultaneously, there is a possibility of deadlock formation (see Section 4.2.4 on page 104), in which the transactions cannot make progress in their executions. The reserved and pending locks are conceived by the SQLite development team to minimize the formation of these kinds of deadlocks. These two locks also help to improve concurrency and to reduce the well-known *writer starvation* problem (in which read-transactions perpetually overtake write-transactions). These locks are compatible with existing shared locks, but new shared lock is not compatible with pending lock. The Unix and Window implementations provide all five locking states. ◁

4.2.2 Lock acquisition protocol

A transaction (actually, the pager module) acquires an appropriate lock on a database file before it reads or writes pages from or to the file, to make sure that two transactions do not access the file in conflicting manners. It is the pager's responsibility to obtain appropriate type of locks on the file when a transaction attempts to access the file.

A transaction starts in the nolock state. Prior to reading the very first (any) page from a database file, a transaction acquires a shared lock to indicate other transactions its intention to read pages from the file. We say it has become a read-transaction, and it can read any pages from the database file. Prior to making any changes to a database (at any page), a transaction acquires a reserved lock that indicates its intention to write in the near future. We say it has become a (semi) write-transaction, but its effects are not visible to other transactions (outside the database connection of the transaction). (Other transactions with shared locks can continue reading the file, but no other transaction can get a new reserved, pending, or exclusive lock on the file. This means the system can have any number of read-transactions, but no other semi-write or full-write-transaction on the database file.) With a reserved lock on a database file, a (semi write-)transaction can make changes to "in-cache" pages. Before writing those modified pages back to the database, it needs to obtain an exclusive lock on the database file. At this point it becomes a full write-transaction, and its effects will be visible in the database file as soon as it starts writing the file.

The locking state transition diagram is given in Fig. 4.2. (As mentioned previously, the pending lock is an intermediated internal lock, and it is not visible outside the lock manager subsystem.) The pager cannot ask the lock manager to get a pending lock; neither the lock manager entertains such requests. A pending lock is always just a temporary stepping stone to the path to an exclusive lock. The pager always requests for an exclusive lock, but the lock manger may grant a pending lock. The pending lock will be upgraded to an exclusive lock by a subsequent request for an exclusive lock.

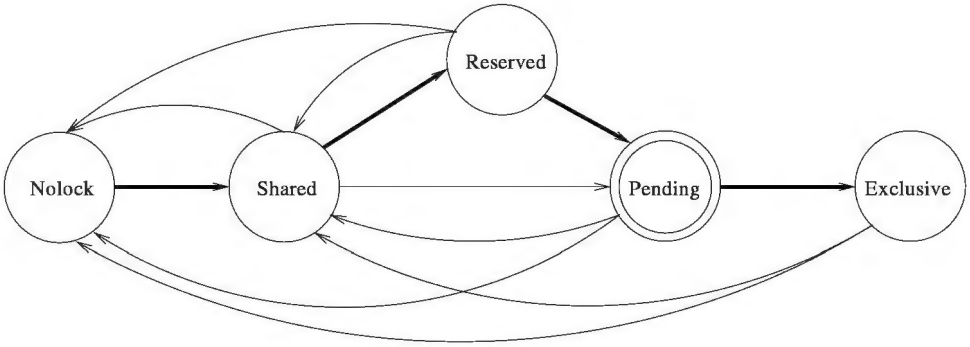


Figure 4.2: Locking state transition diagram.

For a read-transaction, the normal lock transition is from nolock to shared lock and back to nolock. For a write-transaction, normal lock transition is from nolock to shared lock to reserved lock to pending lock to exclusive lock (as shown by bold arrows in Fig. 4.2) and back to nolock. The direct shared lock to pending lock transition is taken only if there is a journal that needs rolling back when a transaction reads the database. In this case, no other transaction could make

the transition from shared lock to reserved lock. (I will revisit this in Section 5.4.2.4 on page 146.)

The lock manger implements two functions, namely `sqlite3OsLock` and `sqlite3OsUnlock` to acquire and release, respectively, SQLite locks on open database files. They are defined, for Unix systems, in os_unix.c file. The `sqlite3OsLock` function is invoked by the pager to obtain new or upgrade existing locks on database files, and `sqlite3OsUnlock` function to release or downgrade locks from the files. Thus, the pager can obtain shared, reserved, or exclusive locks by executing the `sqlite3OsLock` function, and can reset the lock to shared or nolock by executing the `sqlite3OsUnlock` function. When an open database connection is closed, the pager releases all locks obtained for the database connection. I talk about these two functions in Section 4.2.7 on page 112.

4.2.3 Explicit locking

Locking is implicit in SQLite. The system (actually, the pager module) alone decides when to lock a database file and in what mode. There are two ways that applications can explicitly direct SQLite to acquire locks by executing two variants of the `begin transaction` command. Let us study the following SQLite application that opens a user transaction a little differently.

```
BEGIN EXCLUSIVE TRANSACTION;
    INSERT INTO table1 values(100);
    INSERT INTO table2 values(20, 100);
    UPDATE table1 SET x=x+1 WHERE y > 10;
    INSERT INTO table3 VALUES(1,2,3);
COMMIT TRANSACTION;
```

By executing the `begin exclusive transaction` command, the application forces SQLite to acquire an exclusive lock on all database files (the main and attached ones) right away without waiting for the database files to be used. This makes sure that, once the `begin exclusive transaction` command execution is successful, the transaction has an exclusive lock on the database files, and will not be blocked by other transactions in its life time. This thread and no other sibling thread (via another database connection) or peer process will be able to initiate transactions to read or write any of the database files until the transaction is complete.

There are two other alternatives to the exclusive transaction: (1) begin immediate and (2) begin deferred. The `begin immediate` command starts a transaction with a reserved lock on all database files (the main and attached ones) right away without waiting for the database files to be used. After a successful `begin immediate` command execution, it is guaranteed that no other sibling thread or peer process will be able to write to any of the database files or successfully execute a `begin`

`immediate` or `begin exclusive` command involving one of those database files. Other threads and processes can however continue to read from the databases. The `begin deferred` command starts a transaction with no locks on any database files; the lock acquisition is deferred until the files are actually used: a shared lock is obtained on a database file when the file is first read, a reserved lock is obtained on the file in the next insert/update/delete operation. SQLite escalates this to an exclusive lock when needs arise (i.e., until when it actually writes the database file). Because the acquisition of locks is deferred until they are needed, it is possible that another sibling thread or peer process could create a separate transaction and write to the database after the `begin deferred` on the current thread is executed. This thread may get an SQLITE_BUSY error code when it tries to execute some SQL statements in the transaction. Application developers have been warned! If an application executes `begin transaction` statement, the default operating mode is the 'deferred transaction'. You may note that a deferred transaction may not lock all databases (main and attached); it locks selective databases on need basis.

The non exclusive `begin transaction` commands allow the presence of other transactions that may read databases at the least. Thus, an attempt to commit a write-transaction may result in getting the SQLITE_BUSY error code. When the commit fails this way, the transaction remains active and the commit can be retried later after other transactions are gone, i.e, they have cleared their shared locks.

4.2.4 Deadlock and starvation

Although locking solves the concurrency control problem, it introduces another problem. Suppose two transactions hold shared locks on a database file. They both request reserved locks on the file. One of them gets the reserved lock, and the other one waits. After a while, the transaction with the reserved lock requests an exclusive lock, and waits for the other transaction to clear the shared lock. But the shared lock will never be cleared because the transaction with the shared lock is waiting. This kind of situation is called a *deadlock*.

Deadlock is a naughty problem. There are two ways to handle the deadlock problem: (1) prevention, and (2) detection and break. SQLite prevents deadlock formation. It always acquires file locks in the nonblocking mode—either the lock manager grants the requested lock or returns an error code without blocking the requester. If it fails to obtain some lock on behalf of a transaction, it will retry only for a finite number of times. (The retry number can be preset by the application at runtime; the default is zero, i.e., no retry.) If all retries fail, SQLite returns the SQLITE_BUSY error code to the application. The application can back off and retry later, or abort the transaction. Consequently, there is no possibility of deadlock formation in the system. However, there is a possibility, at least theoretically, of starvation where a transaction perpetually tries to obtain some

lock without a success.

4.2.5 Linux lock primitives

SQLite locks are implemented via native file locks. In this subsection I review locks supported by Linux systems. The POSIX advisory locking scheme is the default locking mechanism in SQLite. Linux implements only two lock modes, namely read and write, to lock contiguous regions on files. To avoid confusion in terminologies, I will use read lock and write lock for native shared lock and exclusive lock, respectively. A write-lock excludes all other locks, both read and write. Read locks and write locks can though coexist on the same file, but at different regions. A process can hold only one type of lock on a region. If it applies a new lock to an already locked region, then the existing lock is converted to the new lock mode. (Such conversions may involve splitting, shrinking, or coalescing with an existing lock if the byte range specified by the new lock does not precisely coincide with the range of the existing lock.)

Note: Locks on a file are not a content of the file or other files. They are merely in-memory data objects maintained by the kernel. The locks do not survive system failures. If an application crashes or exits, the kernel cleans up all locks hold by the applications. ◁

4.2.6 SQLite lock implementation

SQLite implements its own locking scheme by using the file locking functions supported by the native operating system. (Lock implementation is platform dependent. I use Linux in this book to show you how SQLite locks are implemented.) As you now know that Linux supports only two lock modes, whereas SQLite has four. SQLite implements its own four lock modes using the two native Linux lock modes on different file-regions. It sets and releases the native locks on regions by making the `fcntl` system calls. This subsection discusses the mapping between SQLite locks and native locks.

- A SHARED lock on a database file is obtained by setting a read lock on a specific range of bytes on the file.[2]
- An EXCLUSIVE lock is obtained by setting a write lock on all bytes in the specified range.
- A RESERVED lock is obtained by setting a write lock on a single byte of the file (that lies outside the shared range of bytes); it is designated as the reserved lock byte.

[2]Some versions of Windows support only write-lock. There, to obtain a SHARED lock on a file, a single byte out of the specific range of bytes is write-locked. The byte is chosen at random from the range so that multiple read-transactions can probably access the file at the same time. If two read-transactions choose the same byte to set write-locks, only one of them succeeds. In such systems, read concurrency is limited by the size of the shared range of bytes.

- A PENDING lock is obtained by setting a write lock on a designated byte different from the reserved lock byte, outside the shared range.

Figure 4.3 displays this arrangement. SQLite reserves 510 bytes as the shared range of bytes, defined as the value of the SHARED_SIZE macro. The range begins at the SHARED_FIRST macro offset. The PENDING_BYTE macro (= 0x40000000, the first byte past the 1 GB boundary defines the beginning of the lock bytes) is the default byte for setting PENDING locks. The RESERVED_BYTE macro is set to the next byte after the PENDING_BYTE, and the SHARED_FIRST is set to the second byte after the PENDING_BYTE. All lock bytes will fit into a single database page, even with the minimum page size of 512 bytes (see Section 3.2 on page 84). These macros are defined in os.h source file.[3] Preconditions and steps to acquire these locks are given below.

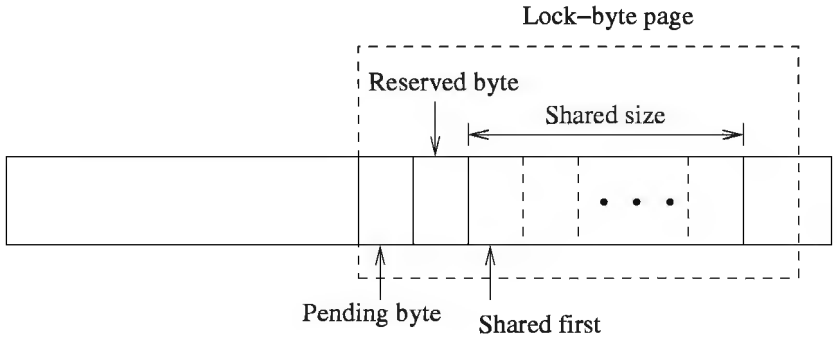


Figure 4.3: File offsets that are used to set POSIX locks.

4.2.6.1 Translation from SQLite locks to native file locks

This sub-subsection describes how SQLite sets locks on database files.

1. To obtain a SHARED lock on a database file, a process first obtains a native read-lock on the PENDING_BYTE to make sure that no other process has a PENDING lock on the file. (You may recall from the lock compatibility matrix table in Fig. 4.1 on page 101 that the existing PENDING lock and a new SHARED lock are incompatible.) If this lock setting is successful, the SHARED_SIZE range starting at the SHARED_FIRST byte is read-locked,[4] and finally the read-lock on the PENDING_BYTE is released.

2. A process may only obtain a RESERVED lock on a database file after it has obtained a SHARED lock on the file. To obtain a RESERVED lock, a write-lock is obtained on the

[3]Locking in Windows is mandatory; that is, locks are enforced for all processes, even if they are not cooperating processes. The locking space is reserved by Windows operating systems. For this reason, SQLite cannot store actual data in the locking bytes. Therefore, the pager never allocates the page involved in locking. That page is also not used in other platforms to have uniformity and cross-platform use of databases. The PENDING_BYTE is set high so that SQLite does not have to allocate an unused page except for very large databases. In this book it is referred to as the *lock-byte page*.

[4]On some Windows, a random byte from the SHARED_SIZE range is write-locked.

RESERVED lock byte. You may note that the process does not release its SHARED lock on the file. (This ensures that another process cannot obtain an EXCLUSIVE lock on the file.)

3. A process may only obtain a PENDING lock on a database file after it has obtained a SHARED lock on the file. To obtain a PENDING lock, a write-lock is obtained on the PENDING_BYTE. (This ensures that no new SHARED locks can be obtained on the file, but existing SHARED locks are allowed to coexist.) You may note that the process does not release its SHARED lock on the file. (This ensures that another process cannot obtain an EXCLUSIVE lock on the file.)

   Note: A process does not have to obtain a RESERVED lock on the way to obtain a PENDING lock. This property is used by SQLite to roll back a journal file after a failure. If the process takes the route from shared to reserved to pending, it does not release the other two locks upon setting the pending lock. ◁

4. A process may only obtain an EXCLUSIVE lock on a database file after it has obtained a PENDING lock on the file. To obtain an EXCLUSIVE lock, a write-lock is obtained on the entire 'shared byte range'. Because all other SQLite locks require a read-lock on (at least one of) the bytes within this range, this ensures that no other SQLite locks are held on the file when the process has obtained the exclusive lock. (The reason a single byte cannot be used instead of the 'shared byte range' is that some versions of Windows do not support read-locks. By write-locking a random byte from a range, concurrent SHARED locks may exist even if the native locking primitive used is always a write-lock.)

To have a clear picture on SQLite's way of acquiring native locks on a database file, the native locking state transition is drawn in Fig. 4.4. In the figure r-lock denotes a native read lock and w-lock a native write lock. The figure also reveals the relationship between SQLite locks and native locks. The representations of PENDING lock and EXCLUSIVE lock are a little awkward; they may or may not have a write-lock on the RESERVED_BYTE depending on which route SQLite has taken to set those locks.

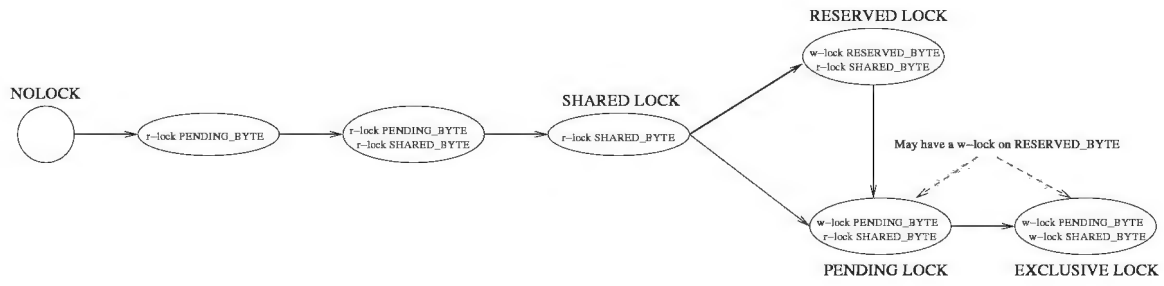


Figure 4.4: Relationship between SQLite locks and native locks in Linux.

4.2.6.2 Engineering issues with native locks

There are two engineering issues in the above defined locking scheme. The first one is due to peculiarity in Linux lock features. The second one is an SQLite specific issue. Native locks on database files are held by processes, where as SQLite locks are held by transactions that run within a process address space executed by a thread (see Fig. 4.5). In the figure, Process 1 has two threads: Thread 1 has one transaction and Thread 2 two on the same database file via three different library connections. Process 2 has one transaction. (You may recall that via a library connection a process can have on a database file one write-transaction and multiple read-transactions simultaneously.) Out of four concurrent transactions, three have shared locks as of now and one has a reserved lock on the database file. You may note that transactions are not visible to the native operating system; the operating system only sees processes and threads. We need to have a mechanism to map SQLite locks held by transactions onto native locks held by processes. If a process opens a database file once, then the process and the transaction are synonymous and there is no problem (as for Process 2 in Fig. 4.5). Complications arise when a process opens many concurrent transactions that may access the same database file via different library connections, as shown in the figure for Process 1. What native locks does Process 1 hold on the database file for its three transactions, and what happens when one transaction completes? I address these two issues in the next two sub-subsections. The only good news is that a library connection can have at most one open write-transaction and the transaction opens the database file once. (What I mean is that a process, to execute a transaction, opens a database file once for the transaction; the process can though open the same database file for other transactions.)

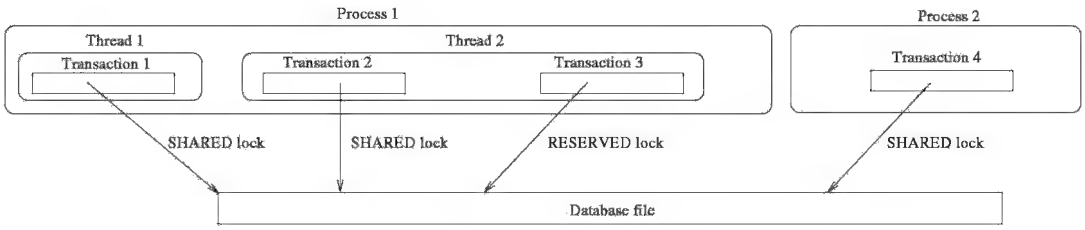


Figure 4.5: Transaction to process assignment.

4.2.6.3 Linux system issues

SQLite uses POSIX advisory locking scheme, and the fcntl API function to obtain and release locks on files. As mentioned previously, locks are in-memory objects. The first thing I would like to point out here is that Linux systems associate locks to file inodes (that represent actual files) rather than to file names. In Linux systems, two or more file names can be used for the same inode via the symbolic and hard links, and this can have some odd consequence for setting/releasing locks via

different file names. The second thing to notice is that although locks are set and released by the fcntl function via open file descriptors, there is no relationship between them. If a process (that has opened the same file multiple times) sets or clears a lock on the same file-region through two open file descriptors, the second lock operation will unwittingly override the first one because both the descriptors point to the same file inode. (The two operations can come from the same or two different sibling threads in the process.)

Let us study an example to clarify the above-mentioned point further. Suppose file1 and file2 are two file names that are really the same file (because one is a hard- or symbolic link to the other). Suppose a process opens the two files for read and write in the following way.

int fd1 = open("file1", ···);
int fd2 = open("file2", ···);

Suppose a thread T1 sets a read lock on a region of the file via fd1 first and then T1 or a different peer thread T2 attempts to obtain a write lock on the same file region via fd2. The thread gets the write lock (overriding the read lock) because both lock requests came from the same process on the same region on the same inode and Linux converts the read lock into a write lock. This means that when a process opens the same database file twice (concurrently) to do two different things in two different transactions, activities on the two open database connections may interfere one another, although unwittingly, because a process can have at most one native lock on a file region. This means that we cannot blindly use native locks to control transactional concurrency. That is, though we can synchronize transactional concurrency in different processes using native locks, we cannot use directly the native locks to synchronize accesses to the same database file in the same process that opens the file in two or more different library connections (in the same or different threads).

Thus, for a solution to the above problem, SQLite needs to keep track of all open database file descriptors of the application process internally, and needs to manage locks internally on its own before it actually tries to obtain or release the native read/write locks on file regions. This is implemented in an abstraction layer. Whenever SQLite opens a database file (for a thread), it finds the specific inode of the database file (the inode is determined by the st_dev device number and st_ino inode number fields of the stat structure that `fstat()` native API function fills in) and internally checks for whether a lock is already held by its owner process on that inode. How this is accomplished is discussed in the rest of this sub-subsection.

When locks are about to be acquired on the inode of a database file, SQLite looks at the process's internal record of locks to see whether it has previously set a lock on that same inode. For each open file (inode), SQLite maintains a piece of information (an object of type `unixInodeInfo`) to keep track of locks on the inode held by the process (see Fig. 4.6, which depicts the locking state

of Process 1 of Fig. 4.5 on page 108). An `unixInodeInfo` object encapsulates the current external locking state on the inode. In the figure, the process has opened a given database file thrice in two threads. (I will discuss subtle multithreading issues in the next sub-subsection.) They track their locks on the file via one `unixInodeInfo` object. One `unixInodeInfo` object represents one instance of SQLite lock on the database file (aka, inode). A process cannot have two or more instances of `unixInodeInfo` objects for the same file (as in Fig. 4.6).

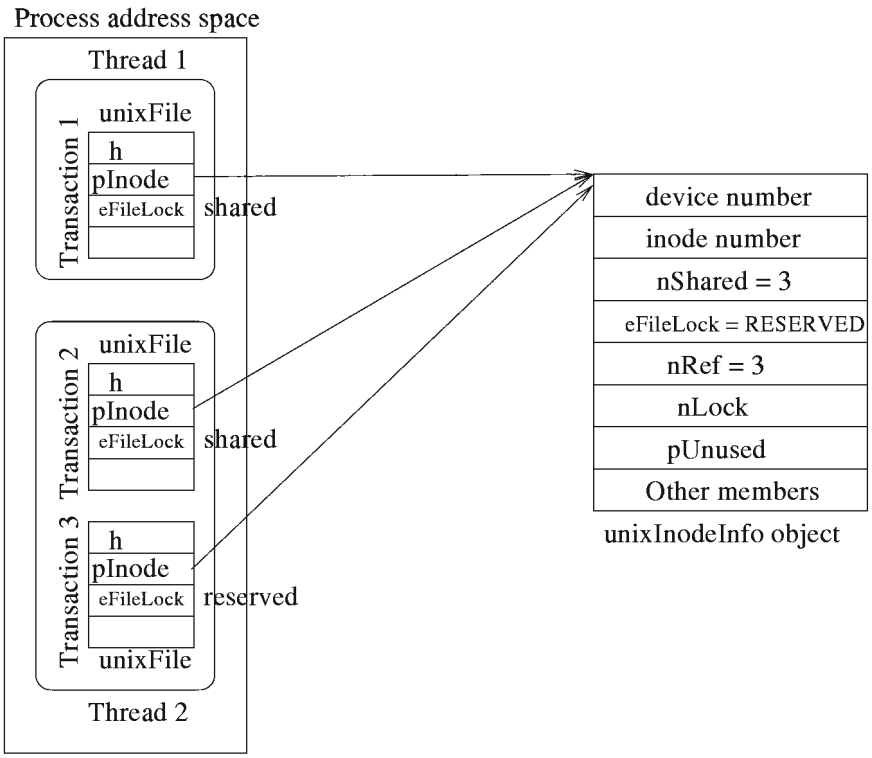


Figure 4.6: SQLite lock management structures.

All `unixInodeInfo` objects (for all files) are stored in a doubly linked list (the global `inodeList`). The device number and inode number combindly are used as the search key to search the list. The list is initially empty. The objects are created and deleted on demand. The list is protected by a (global) thread mutex.

An `unixInodeInfo` object keeps a reference count (`nRef`) to keep track of how many times the process has opened the same file, that is, how many database connections are sharing the object. The `eFileLock` member variable of the `unixInodeInfo` object indicates the highest (strongest) mode of SQLite lock currently held by the process on the database file: NOLOCK, SHARED, RESERVED, PENDING, or EXCLUSIVE. (Some transaction initiated by the process holds this SQLite lock on the database file.) The `nShared` member variable indicates the internal lock status. If `nShared` is zero, it means the file is not locked by the process; if `nShared` is greater than zero, it

means the process has obtained that many number of SHARED locks on the file. When a thread attempts to lock or unlock a file, SQLite first checks the corresponding `unixInodeInfo` object. The `fcntl` system call is invoked to set or release a native (read or write) lock on the file only if the internal locking state transition between two SQLite lock modes is indicated. For example, if the process holds a RESERVED lock on the file and the thread requests a SHARED lock on the file, SQLite increments the `nShared` member variable of the `unixInodeInfo` object. But, if the thread requests an EXCLUSIVE lock, SQLite makes a `fcntl` call to get a write-lock on the file.

SQLite maintains an object of type `unixFile` to track down locking status of one open instance of the database file. As shown in Fig. 4.6, there is an unixFile object for each open database connection. The object is synonymous to the database connection. An `unixFile` object makes a file opening independent of the native operating system's supported open file descriptor. (The unixFile structure is defined differently for different native operating systems.) All operations on the database file are performed using the object as a handle. The `h` component stores the true operating system created file descriptor for the open file. The `h` value is used as reference to read, write, close, and set/reset locks on the file. The `unixFile` object has a pointer to a `unixInodeInfo` object via which it acquires SQLite locks on the database file. The `eFileLock` field of the `unixFile` object indicates the current SQLite lock mode held on the database file through this `unixFile` object, i.e., for this connection to the database file. As one database-file-connection can have at most one open (read or write) transaction, the `eFileLock` in the corresponding `unixFile` object represents that transaction's SQLite lock type on the database file through this connection. In Fig. 4.6, Transaction 1 on Thread 1 holds a SHARED lock on the database, and Transaction 2 on Thread 2 holds a SHARED lock and Transaction 3 on Thread 2 a RESERVED lock on the same database.

If the same inode is opened twice or more by the process in any thread, there will be those many `unixFile` objects and all of them share the same `unixInodeInfo` object. This scenario is shown in Fig. 4.6 for Thread 2 that has two transactions. As mentioned earlier, all unixInodeInfo objects are stored in a shared list. When a thread attempts to lock or unlock a database file, SQLite first checks the existence of `unixInodeInfo` object for the file inode in the list. If one is not found, SQLite creates a new `unixInodeInfo` object in the list. At this time the `nRef` member variable is set to 1. If one is found, the `nRef` variable is incremented by one. When the file is no more used by the process (i.e., when `nRef` becomes 0), the `unixInodeInfo` object is removed from the list and is destroyed.

4.2.6.4 Multithreaded applications

Multithreading of SQLite applications causes some additional problems in Linux systems that support LinuxThreads. Under LinuxThreads, lock operations on a file from one thread do not override locks on the same file from other sibling threads. Only, the owner thread can manipulate its locks. Thus LinuxThreads is not POSIX compliant, and due to this problem, starting from the SQLite 3.7.0 release, the SQLite development team has stopped supporting LinuxThreads. They instead support NPTL (Native Posix Thread Library), where sibling threads can override each other's locks.

There is a complication regarding the closing of a file. When a file descriptor (for an inode) is closed, Linux systems release all locks on that inode that are owned by the current process irrespective of which threads have obtained those locks via which file descriptors. To have a solution to this problem, we need to keep track of all open file descriptors for this inode until the last file descriptor is closed. When a thread attempts to close an unixFile, at that point in time if there are other unixFiles open on the same inode that are holding locks, then the call to close the file is deferred until all of the locks are cleared. This is called lazy file closing. Lazy closing may cause withholding valuable file descriptor resources; but the SQLite development team does not have another viable alternative to this approach. The `unixInodeInfo` object keeps a list of file descriptors (`pUnused`) that need to be closed. The thread that closes the file last, closes all these deferred file descriptors.

Note: The file descriptors in `pUnused` can also be recycled by SQLite when the file is opened by the application again. ◁

4.2.7 Lock APIs

The lock manager implements two API functions, namely sqlite3OsLock and sqlite3OsUnlock, respectively, to acquire or upgrade and to release or downgrade SQLite locks on databases. In the following two sub-subsections I present them in algorithmic steps.

4.2.7.1 The `sqlite3OsLock` API

The signature of the API function is as follows: `int sqlite3OsLock(unixFile* id, int locktype)`, where `id` is an SQLite file descriptor on which an SQLite lock of `locktype` is requested. For Unix, this is implemented as the `unixLock` function in the os_unix.c source file. You may note that `locktype` value cannot be PENDING. This function can only increase the strength of a lock in the following order: NOLOCK, SHARED, RESERVED, PENDING, and EXCLUSIVE. The `sqlite3OsUnlock` API function is used to decrease lock strength. Roughly, the following steps are

carried out by the `sqlite3OsLock` function.

1. `int idLocktype = id->eFileLock;` // Locktype currently held through this unixFile id object
2. `int pLock = id->inodeInfo->eFileLock;` // Locktype held by the process on the file
3. Assertion 1: `idLocktype <= pLock.`
4. If there is already a lock of the requested type or more restrictive type on the `id`, do nothing. `if (idLocktype >= locktype) return SQLITE_OK.`
5. Assertion 2: `locktype > idLocktype.` This assertion follows from the negation of the above if-condition.
6. Assertion 3: `(locktype == SHARED) || (idLocktype` $\neq$ `NOLOCK).` To request a stronger than shared lock, the client must have some lock on the file.
7. Assertion 4: `locktype` $\neq$ `PENDING.` Clients cannot directly request pending locks.
8. Assertion 5: `(locktype` $\neq$ `RESERVED) || (idLocktype == SHARED).` One can only request a reserved lock after holding a shared lock.
9. `if (idLocktype` $\neq$ `pLock)` { // Some peer transaction has a stronger lock on the file via a different unixFile object. Test lock compatibility
   - Assertion 6: `idLocktype < pLock.` This assertion follows from Assertion 1 and the if-condition.
   - Assertion 7: `pLock >= SHARED.` This assertion follows from Assertion 6 and the values for lock types.
   - `if (pLock >= PENDING) return SQLITE_BUSY;` // Lock incompatibility. The unixFile object `id` cannot set a new lock.
   - Assertion 8: `(pLock == SHARED) || (pLock == RESERVED).` This assertion follows from Assertion 7, ordering of lock types, and $\neg$`(pLock >= PENDING).`
   - `if (locktype > SHARED)` {
     - Assertion 9: `pLock > SHARED.` From `locktype > SHARED` and Assertion 3, we have `idLocktype` $\neq$ `NOLOCK`, i.e., `idLocktype >= SHARED.` This assertion follows from `idLocktype >= SHARED` and Assertion 6.
     - From Assertion 8 and Assertion 9, we have that `pLock == RESERVED.` By lock compatibility, we cannot set a new lock stronger than SHARED lock.

- return SQLITE_BUSY;

}

- else { \ \ locktype = SHARED
  - From Assertion 2, we have `idLocktype == NOLOCK`. By Assertion 8, the id can have a SHARED lock.
  - As some one else has a lock on the same file, the `nShared` field in the corresponding `unixInodeInfo` structure is greater than 0. Increment the `nShared` field, set `id->eFileLock = SHARED`, increment `id->pInode->nLock`
  - return SQLITE_OK.

}

}

10. else, we have `idLocktype == pLock`. Thus, the `id` has the highest lock type among peer unixFile objects, and we have a case of lock escalation.

11. Assertion 10: `idLocktype == pLock` and `idLocktype < locktype`. This assertion follows from the negation of if-conditions at line 10 and Assertion 2.

12. `if (locktype == SHARED)` // `idLocktype == pLock == NOLOCK`, i.e., no lock is held
    `set a new SHARED lock on the file;`
    `elseif (locktype == RESERVED)` // The `id` holds a SHARED lock, and some other ids
    // may also have SHARED locks. Compatibility case
    `set a new RESERVED lock;`
    `else` // `locktype == EXCLUSIVE`. The `id` holds a PENDING or RESERVED lock, and some other ids may hold SHARED locks
    `if (id holds a RESERVED lock) get a new PENDING lock first;` // Lock compatible
    `if (some ids hold SHARED locks)` // We cannot get EXCLUSIVE lock
    return SQLITE_BUSY; // We may have got a PENDING lock
    `set a new EXCLUSIVE lock;`

13. If the above attempt for the file lock request succeeds, return SQLITE_OK;

14. Else return SQLITE_BUSY;

4.2.7.2 The `sqlite3OsUnlock` API

The signature of the API function is as follows: `int sqlite3OsUnlock(unixFile* id, int locktype)`, where `id` is an SQLite file descriptor on which an SQLite lock of `locktype` is requested. For Unix,

this gets translated to the posixUnlock function in the os_unix.c source file. You may note that locktype can only be NOLOCK or SHARED lock. To increase the strength of a current lock, we need to use `sqlite3OsLock` API function. The following steps are carried out by `sqlite3OsUnlock` function.

1. `int idLocktype = id->eFileLock;` // Locktype held through this file descriptor
2. `int pLock = id->inodeInfo->eFileLock;` // Locktype held by the process
3. `if (idLocktype <= locktype) return SQLITE_OK.` // Already has lower lock
4. `if (idLocktype > SHARED)` // A case of lock downgrade
   - `if (locktype == SHARED) set read-lock on SHARED_BYTES;`
   - `Clear write-locks from pending and reserved bytes.`
5. `if (locktype == NOLOCK)`
   - decrement `id->inodeInfo->nShared`; if the `nShared` is 0, clear all locks from the file
   - decrement `pInode->nLock`; if the `nLock` is 0, close all pending lazy close cases.
6. return SQLITE_OK;

4.3 Journal Management

A *journal* is a repository of recovery information that is used to recover a database when aborting a transaction or statement subtransaction, and also used when recovering the database after an application, system, or power failure. SQLite uses a single journal file per database. (It does not use journal files for in-memory databases.) It assures only rollback (undo, and not redo) of transactions, and the journal file is often called the rollback journal. The journal file always resides in the same directory as the database file does, and has the same name, but with '-journal' appended.

Transient Journal vs. Journal File Retention: SQLite permits at most one write-transaction on a database file at a time. Under the default operating mode, it creates the journal file on the fly for every write-transaction, and deletes the file when the transaction is complete. You can change this behavior by the `journal_mode` pragma to truncate, persist (but invalidate the header), memory, or off. With the memory option the journal is entirely memory resident, and with the off option journaling is not performed. I talk about wal journaling in Section 10.17 on page 249. ◁

The update operations (SQL inserts, deletes, and updates) from the current write-transaction produce log records that are written to the journal file. When a transaction wants to make a change to a database file, SQLite writes enough information in the rollback journal so that if

needs arise it can restore the database back to the state as of the start of the transaction. There are many varieties of logging schemes known to the database community; they depend on what redo/undo information is stored in log records. SQLite uses the simplest, though inefficient, of all known varieties. It uses old value logging technique at the page level granularity. (SQLite does not implement redo logic in recovering a database. Consequently, it does not save new values in log records.) Thus, before a transaction alters any page of the database file for the first time, SQLite writes the original content of that entire page along with its page number into the rollback journal as a part of new stable log record. (See Fig. 3.8 on page 92.)

Once a page image is copied into the rollback journal, the page will never be in a new log record even if the page is altered multiple times by the current transaction. One nice property of this page level undo logging is that a page can be restored by blindly copying the content from the journal file into the database file and the undo operation is idempotent. The undo operation executions do not produce any compensating log records. SQLite never saves a page in the journal that is newly added (i.e., appended) to the database file by the transaction, because there is no old value for the page. Instead, the journal notes down the initial size of the database file in the journal segment header record (see Fig. 3.7 on page 91) when the journal file is created. If the database file is expanded by the transaction, the file can be truncated back to its original size on a rollback.

Journaled Page Tracking: SQLite uses an in-memory bit map data structure to keep track of which pages are journaled by the current transaction. The memory space overhead is proportional to the number of pages the transaction updates. For small transactions, the memory overhead is negligible. ◁

Log Optimization: You may recall that a freelist 'leaf' page content is treated as garbage. When such a page is reused, the page is not logged because it does not have any useful information. ◁

If a transaction works with and modifies multiple databases (you may recall that multiple databases can be associated to a library connection by executing the ATTACH command), then each database has its own rollback journal. They are independent rollback journals, and one is not aware of the others. To bridge the gap, SQLite, in addition, maintains a separate aggregate journal called the *master journal.* The master journal file always resides in the same directory as the main database file does, and has the same name but with '-mj' appended to it followed by eight randomly chosen 4-bit hexadecimal numbers. It is a transient file. It is created when the transaction attempts to commit, and deleted when the commit processing is complete. It does not contain any log records per se that are used for rollback purpose. It instead contains the names of all the individual rollback journals that participate in the transaction. Each of the individual rollback journals also contains the name of the master journal (see Fig. 3.9 on page 94). If there is no attached database (or if none of the attached databases participates in the current transaction for update purpose) no master journal is created and the normal rollback journal does not contain any

information about a master journal. In the rest of this section I discuss log and commit protocols.

No Database Aliasing: You must not use different names (hard- or soft-links) for a database file. If different applications use different names, there will be different journal files for the same database with different rollback and master journal names and the applications would miss each other's journal file, leading to a corrupted database file. Also, you should not rename a database file without renaming the corresponding journal file. But, still there is a risk if the journal is referenced from a master journal. You have been severely warned! ◁

4.3.1 Logging protocol

SQLite follows write-ahead logging (WAL) protocol to ensure recoverability of databases in the occurrences of application, system, or power failures. SQLite implements before image logging, that is, it writes the original unaltered copies of database pages (that are about to be modified) into the journal file first before changing the pages in the database files. Writing log records in the journal file is lazy: SQLite does not flush them to the disk surface immediately. However, before writing the next page (any page) in the database file, it forces all log records to disk. This is referred to as flushing the journal. Journal flushing is done to make sure all the log records that have been written to the journal have actually reached the disk surface. (If the flush operation of the native operating system does not work this way SQLite may end up corrupting the database.) It is not safe to modify the database file in-place until after the journal has been flushed and the journal contents have become persistent. If the database is modified before the journal is flushed and a power failure occurs, the unflushed log records would be lost and SQLite would not be able to completely rollback the transaction's effects from the database, resulting in a database corruption.

4.3.2 Commit protocol

The default commit logic is both flush-log-at-commit and flush-database-at-commit. When an application commits a transaction, SQLite makes sure that all log records in the rollback journal are in the disk. At the end of the commit, the rollback journal file is finalized (i.e., deleted or truncated or invalidated depending on the operating mode), and the transaction is complete. If the system fails before reaching that point, the transaction commit has failed, and it will be rolled back when the database is read next time. However, before finalizing the rollback journal file, all changes to the database file are flushed to the disk. This is done to make sure that the database has received all updates from the transaction before the journal is finalized. This is needed because SQLite does not have a redo logic as of SQLite 3.7.8 in the legacy/rollback journaling mode.

Asynchronous Transaction and Lazy Commit: By default transactions are synchronous. SQLite strictly follows the above mentioned logging protocol (see Section 4.3.1) and the commit protocol (see

Section 4.3.2) for these transactions. Though not recommended, SQLite also permits applications to run transactions in the lazy commit mode. These are called *asynchronous transactions*. It is achieved by setting the synchronous pragma variable to zero (see Section 10.1 on page 226). SQLite does not perform journal nor database flushing at commit or other times for asynchronous transactions. Thereby, database writes and commits are very fast. But, there is certainly a risk. Upon a failure, the database may not be restored to a consistent state (because of missing log records in the journal) and becomes corrupt. Asynchronous transaction developers have been warned! For temporary databases, the default is though asynchronous, because we do not need nor we care about those databases upon failures. ◁

4.4 Subtransaction Management

A statement subtransaction acquires locks via the main user transaction. All locks are held by the transaction until it commits or aborts. But, SQLite uses a separate journal file to store log records produced by statement subtransactions. The statement journal is a temporary file, and is not required for recovery from transaction abort, nor for application, system, and power failures. SQLite writes some log records in the statement journal, and some in the main rollback journal. A log record is written in the statement journal only if the corresponding page has already been written in the main rollback journal before the statement subtransaction execution started or the page was added by a previous subtransaction. SQLite never flushes statement journal files to the disk, because they are not required for failure recovery.

Summary

SQLite executes each SQL statement either in a user or system transaction. The default operating mode is autocommit. In this mode, SQLite creates a read-transaction to execute a select statement, and a write-transaction to execute a non select (i.e., insert, delete, or update) statement. At the end of the statement execution, the transaction is committed or aborted.

The default autocommit mode is overridden by applications by creating a user transaction by executing begin commands. Successive non select SQL statement executions become a part of the user transaction until the application commits or aborts the transaction. At that point SQLite reverts back to the autocommit mode. SQLite also supports savepoints in user transactions. A transaction can set up multiple savepoints and reverts back the database state to any of the savepoints and continues its execution from there. Inside a user transaction, update statements are executed in subtransactions one after another sequentially. These subtransactions are executed via anonymous savepoints.

SQLite uses a lock based concurrency control mechanism to ensure serializable executions of

transactions. It uses database-level locking, that is, it sets locks on the entire database and there are no fine granule locks. SQLite defines five lock types: nolock, shared, exclusive, reserved, and pending. (These locks are implemented using native read- and write locks on different bytes of the database file.) A read-transaction moves from nolock to shared lock and back to nolock. A write-transaction moves from nolock to shared lock to reserved lock to pending lock to exclusive lock and back to nolock. When a database needs recovery by rolling back a journal file, the lock transition is from nolock to shared lock to pending lock to exclusive lock and back to shared lock.

When an application opens a user transaction by executing a begin transaction command, the transaction does not hold any lock on the database at that point in time. Lock acquisition is deferred until actual needs arise. There are two variants of the begin command, namely begin exclusive and begin immediate. Upon a successful begin exclusive (alternatively, immediate) command execution, SQLite sets an exclusive (alternatively, reserved) lock on all databases (the main and attached ones) right away.

SQLite implements a journal based logging scheme for recovering a database upon a transaction abort, process crash, or system/power failure. In such situations, SQLite rolls back the effect of the transaction from the database by some simple undo operations. When a transaction modifies a page (for the first time), SQLite logs the old value of the (entire) page in the rollback journal. During annulling the effects of a transaction from the database, the old page image is restored. (The recovery operation is idempotent.) At the fag end of the recovery, the database file is truncated to the size when the transaction started; this is done to eliminate all those pages that were added by the transaction. To commit a 'multidatabase transaction', SQLite uses a master journal (in addition to all individual rollback journals) to keep information about which databases participated in the transaction; also, each individual rollback journal contains a record that points back to the master journal.

In the next chapter I present the inner working of the transaction manager called pager in the SQLite world. Some concepts introduced here will be repeated there to make the chapter self contained.

Chapter 5

The Pager Module

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- what a page cache is, why it is needed, and who uses it
- generic cache management techniques
- the normal transaction processing and recovery processing steps that are carried out by SQLite

Chapter synopsis

This chapter discusses the pager module. The module implements an abstraction of page oriented database file system on the top of the ordinary native (byte oriented) files. It acts as the data manager where data items are fixed-size pages, and defines an interface for accessing pages from database files. It helps the tree module to speedup accesses to database pages by providing the module an in-memory cache for database pages. It manages the page cache. It is also the transaction manager that implements ACID properties for transactions by taking charge of concurrency control and failure recovery. It makes concurrency control and recovery absolutely transparent to the tree- and higher up modules. It is also the lock- and log manager. In essence, it implements the persistence layer of a normal database management system.

5.1 The Pager Module

Databases (except in-memory ones) normally reside on external storage devices such as the disk, in the form of ordinary native files. SQLite cannot efficiently access and manipulate data on disk. When it needs a data item, it reads it from the database file into the main memory, manipulates it in-memory, and, if needed, writes it back to the database file. Normally, databases are very large

compared to the available main memory. Because of the limited availability of the main memory, only a part of the memory is reserved to hold a (tiny) fraction of data from database file(s), and this reserved memory space is popularly called a database cache or data buffer; in SQLite terminology, it is called the *page cache*. The cache resides in the application process address space, and not in the operating system space. (The operating system has its own data caches.)

The page cache manager is called *the pager* in the SQLite world. It sees underneath random accessed byte oriented ordinary native files, and converts them into random accessed higher-level page oriented files, where pages are fixed size objects crafted out of the native files. Different higher-level files can have different page sizes. The pager defines an 'easy to use' (independent of native file systems) interface for accessing pages from database files. The tree module that resides directly on the top of the pager module always uses the pager provided interface to access databases, and never directly accesses any database or journal file. The former (tree module) sees the database file as a logical array of (uniform size) pages and reference pages by providing their array index numbers.

SQLite maintains a separate page cache for each open database file (aka, database connection). When an application process opens a database file, the pager creates and initializes a new page cache for the file. If the process opens the same database file two or more times, in the default operating mode, the pager creates and initializes those many separate page caches for the file. (SQLite supports an advance feature in which all database connections to the same database file can share the same page cache of file that is open multiple times via the same or different library connections, see Section 10.13 on page 241.) In-memory databases do not refer to any external storage devices; but, they are also treated like ordinary native files, and are stored entirely within the cache. Thus, the tree module uses the same interface to access either type of database.

The pager is the lowest layer module in SQLite. It is the only module that accesses native database and journal files using I/O APIs supported by the native operating system. It directly reads and writes database files (and journal files). It does not understand how data items in the databases are organized. It neither interprets the content of databases, nor modifies the content on its own. It only guarantees whatever information is stored in a database file can be repeatedly retrieved later without any alteration. In that sense, the pager is a passive entity. (It though may modify some information in the database file header record, such as the file change counter.) It takes the usual random access byte-oriented file system, and abstracts it into a random access page-based file system for working with database files. It defines an easy-to-use, file-system-independent interface for randomly accessing pages from database files.

For each database file, moving pages between the file and the (in-memory) cache is the basic function of the pager as the cache manager. The page movement is transparent to the tree and

higher-up modules. The pager is the mediator between the native file system and those higher-up modules. Its main purpose is to make database pages addressable in the main memory so that those modules can access the in-memory page contents directly. It also coordinates the writing of pages back to the database file. It creates an abstraction so that the entire database file appears to reside in the main memory as an array of pages. The two modules work together via a well defined page access protocol.

Apart from the cache management work, the pager does carry out many other functions of a typical database management system (DBMS). It provides the core services of a typical transaction processing system: transaction management, data management, log management, and lock management. As a transaction manager, it implements transactional ACID properties by taking charge of concurrency control and recovery. It is responsible for atomic commit and rollback of transactions. As a data manager, it coordinates reading and writing of pages in database files with the cache and does file space management work. As a log manager, it decides on the writing of log records in journal files. As a lock manager, it makes sure that transactions, before accessing a database page, have appropriate locks on the database file. In essence, the pager module implements storage persistence and transactional atomicity. The interconnections between all the submodules of the pager are shown in Fig. 5.1.

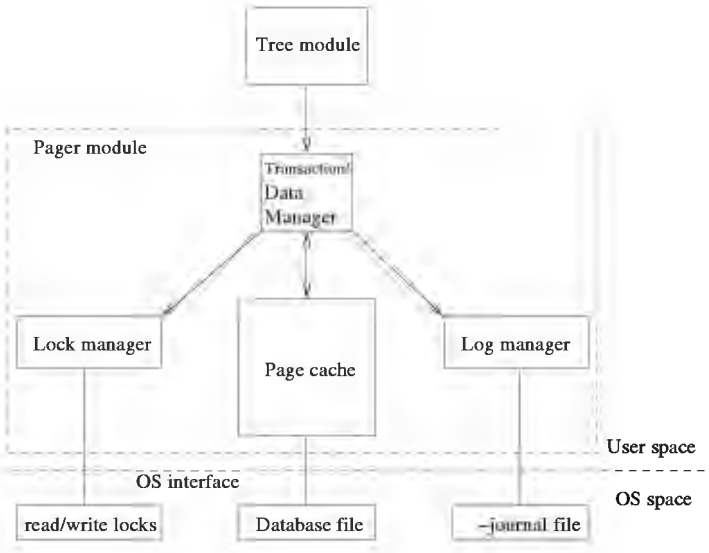


Figure 5.1: Interconnection of pager submodules.

5.2 Pager Interface

In this section I present some interface functions exposed by the pager module that the tree module uses to access databases. Before doing so, I first discuss the protocol of interactions between the pager and the tree modules.

5.2.1 Pager–client interaction protocol

All modules above the pager are completely insulated from low-level lock and log management mechanisms. In fact, they are not aware of locking and logging activities. The tree module sees everything in terms of transactions, and is not concerned with how the transactional ACID properties are implemented by the pager module. The pager module splits the activities of a transaction into locking, logging, and reading and writing of database files. The tree module requests a page from the pager by the page number. The pager, in turn, returns a pointer to the page's data loaded into the page cache. Before modifying a page, the tree module informs the pager so that it (the pager) can save sufficient information (in a journal file) for possible use in future recovery, and can acquire appropriate locks on the database file. The tree module eventually notifies the pager when it (the tree module) has finished using a page; the pager handles writing the page back to the database file if the page was modified.

5.2.2 The pager interface structure

The pager module implements a data structure named `Pager`. Each open database file is managed through a separate `Pager` object (see Fig. 5.2) and each `Pager` object is associated with one and only one instance of an open database file. (At the pager module level, the object is synonymous to the database file.) The tree module, to use a database file, creates a new `Pager` object first and then uses the object as a handle to apply all pager-level operations on the file. The pager module uses the handle to track down information regarding file locks, journal file, the status of database, the status of the journal, etc. You may note that a process can have several `Pager` objects for the same database file, one for each connection to the file; the objects are treated as independent, and not related to one another. (For shared cache mode of operation, there is though only one Pager object per database file that is shared by all database connections to the file.) In-memory databases are also accessed using `Pager` objects as handles.

Some component member variables of a `Pager` object are given in Fig. 5.3. The purposes of the member variables are discussed in the figure itself. As shown in the figure, a `Pager` object is immediately followed by a variable amount of memory space that stores various handlers such as for page cache, database file, journal files, and the database file name, and the journal file name. (I already talked about database file handle (unixFile for Linux) in Section 4.2.6.3 on page 108 and Fig. 4.6 on page 110.)

As mentioned in an earlier chapter, SQLite, when in a user transaction, executes each update (i.e., SQL insert, delete, or update) operation in a savepoint. Also the application can setup its own savepoints. There can be multiple simultaneous savepoints, aSavepoint array in Fig. 5.3. The structure of a savepoint handler is depicted in Fig. 5.4. When SQLite creates a savepoint, it sets

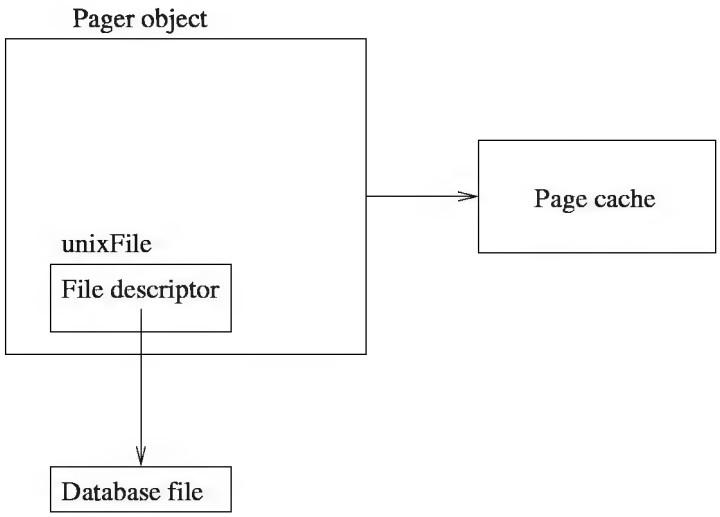


Figure 5.2: Pager interface with a database file.

the iHdrOffset to 0. But, if it opens the rollback journal and writes a (new) segment header record while the savepoint is active, it sets iHdrOffset to the byte offset immediately following the last log record into the rollback journal before the segment header. The iOffset is set to the starting offset in the rollback journal when the `PagerSavepoint` object is created.

5.2.3 The pager interface functions

The pager module implements a set of interface functions (that are used by the tree module). Some important interface functions are briefly discussed below. (These function descriptions will give give you some heads up information that will be valuable to read through the rest of this chapter.) There are many more other such functions. All function names are prefixed with `sqlite3Pager`. They are defined in the pager.c source file. They are strictly internal to SQLite, and SQLite application developers must not use them in their applications.

1. `sqlite3PagerOpen`: This function creates a new `Pager` object, opens a given database file, creates and initializes an empty page cache, and returns a pointer to the `Pager` object. Depending on the name of the database file, it creates and/or opens an appropriate file (see Section 3.1 on page 81). The database file is not locked at this time; neither a journal file is created, nor a database recovery operation is performed. (You may note that SQLite does a deferred recovery until a page is actually read from the database file.)

2. `sqlite3PagerClose`: This function destroys a `Pager` object and closes the associated open database file. If it is a temporary file, the pager deletes the file. If it is not a temporary file and a transaction has been in progress on this file when this routine is called, the transaction is forced an immediate abort and its changes are rolled back from the database file. All

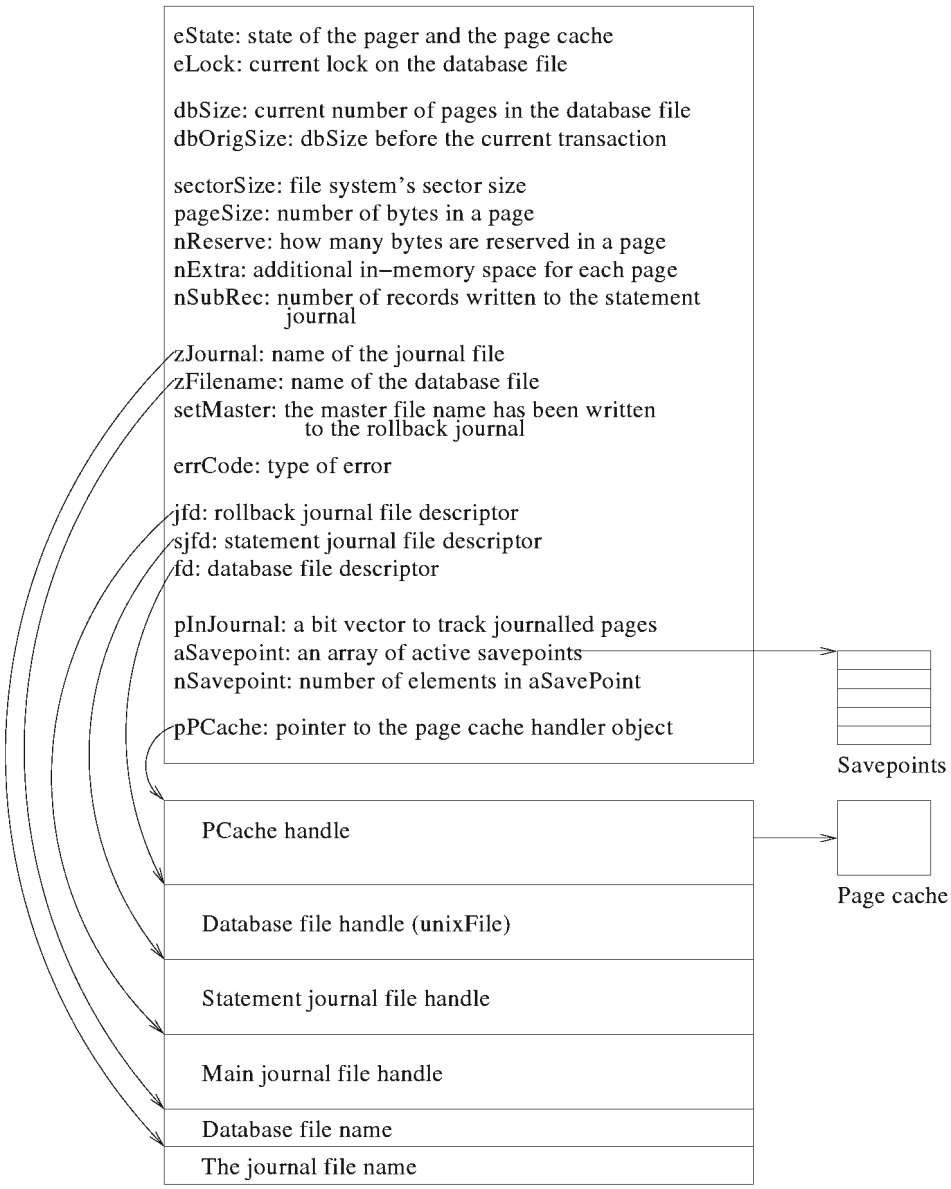


Figure 5.3: The Pager structure.

outstanding in-cached pages are invalidated and their memory is freed (i.e., released from the process address space). Any attempt to use a page that was associated with this cache, after this function returns, may result in a coredump. (In essence, all resources allocated to the `Pager` object are released including the object itself.)

3. `sqlite3PagerGet`: This function makes an in-memory copy of a database page available for the caller (aka, the tree module). The caller specifies the required page by the page number. This function returns a pointer to the in-cached copy of the page. So that the cache space is not recycled, it pins down the page copy. It obtains a shared lock on the database file at the very first time it is called. (If the lock cannot be obtained, it returns SQLITE_BUSY to the caller.) At this point it decides whether to purge the existing page cache—if the database

Savepoint structure

ioffset: starting offset of the main journal

iHdrOffset: initially 0; if a journal header is written into the main journal then the byte offset immediately following the last journal record before the new journal header

nOrig: original number of pages in the database file

iSubRec: index of the first record in sub–journal

pInSavepoint: set of pages in the savepoint

Figure 5.4: The PagerSavepoint structure.

file-change-counter (the 4-byte integer at file header offset 24) is different than when the cache was previously accessed, it purges the cache. In addition, if there is a need (in case of an existing hot journal file), the database is recovered from the journal. (I discuss hot journal and database recovery in Section 5.4.2.4 on page 146.) The function loads the required page into the cache if the page is not already there. But, if the database file is smaller than the requested page, then no actual file read is performed and the memory image of the page is initialized to all zeros. (No file access is done for in-memory databases.)

4. `sqlite3PagerWrite`: This function makes the requested database page writable for the caller (but does not write the page into the database file). It must be called on a page before its in-cached image is altered by the tree module, because otherwise the pager may not know that a cached page is altered. (You may note that, for performance sake, SQLite avoids copying pages back and forth between the pager and tree modules, and the tree module directly manipulates the contents available in in-cached pages.) If not already done in some previous function call, the pager acquires a reserved lock on the database file and creates a rollback journal. That is, it creates an implicit write-transaction. (If the lock cannot be obtained, it returns SQLITE_BUSY to the caller.) It copies the original content of the page to the rollback journal if the page is not already a part of a log record. If the original page content has already been written into the rollback journal, this function is a no-op except that it marks the page dirty. This function may also write a statement log record in the statement journal if the current query processing is happening in a user transaction and the page is already in the main rollback journal or the page was added by a previous statement subtransaction.

5. `sqlite3PagerLookup`: This function returns a pointer to the in-cached copy of the requested database page if the page is there in the cache. If the page is not in the cache, it returns NULL. In the former case, it pins down the page.

6. `sqlite3PagerRef`: This function increments the reference count on a page by 1. We say the page is pinned down by the caller. If the page is on the freelist of the cache, this function removes the page from the list.

7. `sqlite3PagerUnref`: This function decrements the reference count on a page by 1. When the count reaches zero, the page is said to be unpinned and it is freed. (The freed page though might still be held in cache as a part of the cache's freelist.) When all pages have been unpinned (i.e., on the last call to this function), the shared lock on the database file is released, and the `Pager` object is reset.

8. `sqlite3PagerBegin`: This function starts an explicit write-transaction on the associated database file. It also opens the rollback journal file if the database is not a temporary file. (For temporary files, the opening of the journal file is deferred until there is an actual need to write to the journal file.) You may note that an implicit write-transaction is started by `sqlite3PagerWrite`. Thus, if the database is already reserved for writing, this routine is a no-op. Otherwise, it obtains a reserved lock on the database file first, and if indicated in an input argument, then it obtains an exclusive lock on the file immediately instead of waiting until the tree module attempts to write into the database file.

9. `sqlite3PagerCommitPhaseOne`: This function commits the current transaction on the database file: increments the file-change-counter metadata by 1, syncs the journal file, syncs all changes (aka, dirty pages from the page cache) to the database file.

10. `sqlite3PagerCommitPhaseTwo`: This function finalizes (i.e., deletes, invalidates, or truncates) the journal file.

11. `sqlite3PagerRollback`: This function aborts the current transaction on the database file: rolls back all changes made to the database file by the transaction and downgrades the exclusive lock into a shared lock. All in-cache pages revert to their original data contents. The journal file is finalized. This routine cannot fail.

12. `sqlite3PagerOpenSavepoint`: This function creates a new savepoint handler object to establish a savepoint for the current database state.

13. `sqlite3PagerSavepoint`: This function releases or rollbacks a savepoint. For the release operation, it releases and destroys a particular savepoint handler object. For the rollback operation, it rolls back all changes made to the database since the savepoint was established, and all following savepoints are deleted.

5.3 Page Cache

Page-caches reside in the address spaces of application processes. (You may note that the same pages may be cached by the native operating system. When an application reads a piece of data from any file (residing on a block special device), the operating system normally makes its own copy of the data first, and then a copy in the application. We are not interested in knowing how the operating system manages its own cache. SQLite's page cache organization and management are independent of those of the native operating system.) Figure 5.5 depicts a typical scenario. In the figure, two processes (one is multithreaded) access the same database file. They have their own caches. Even if a thread opens the same database file two or many times, in the default operating mode, SQLite allocates separate caches to the open database connections. The caches are accessed via their different owner `Pager` objects.

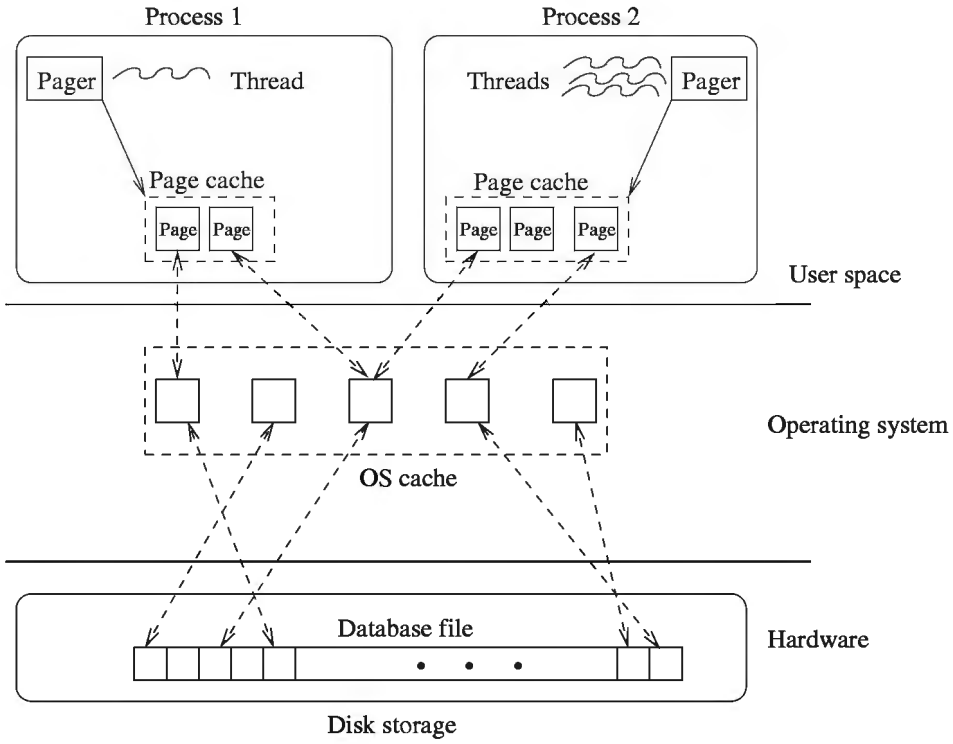


Figure 5.5: A typical scenario where two processes read the same database file.

5.3.1 Cache state

The state of a page cache (and that of the corresponding owner `Pager` object, see Fig. 5.3 on page 126) determines what the pager module can do with the cache. Two member variables, namely `eState` and `eLock`, controls the pager behavior. The page cache (and the pager) as a whole is always in one of the following seven states (the value of `Pager.eState` member variable). The state transition diagram is shown in Fig. 5.6.

1. PAGER_OPEN: When a **Pager** object is created, this is the initial state. The pager is not currently reading or writing the database file via this **Pager** object. There may not be any database page held in memory, i.e, the cache is empty. The database file may or may not be locked. There is no transaction open on the database.

2. PAGER_READER: When a **Pager** object is in this state, at least one read-transaction is open on the database connection, and the pager can read pages from the corresponding database file. (But, in the exclusive locking_mode, read-transactions may not be open.)

3. PAGER_WRITER_LOCKED: When a **Pager** object is in this state, a write-transaction is open on the database connection. The pager can read pages from the corresponding database file, but it has not made any updates on cached pages or the database file.

4. PAGER_WRITER_CACHEMOD: When a **Pager** object is in this state, the pager has given the tree module a permission to update in-cached pages, and the tree module may have made some updates.

5. PAGER_WRITER_DBMOD: When a **Pager** object is in this state, the pager has begun writing the database file.

6. PAGER_WRITER_FINISHED: When a **Pager** object is in this state, the pager has finished writing all modified pages of the current write-transaction into the database file. The write-transaction cannot make any more updates, and is ready to commit.

7. PAGER_ERROR: When a **Pager** object is in this state, the pager has seen some errors such as I/O could not be performed, no disk space available for the database or journal file, no memory can be allocated, etc.

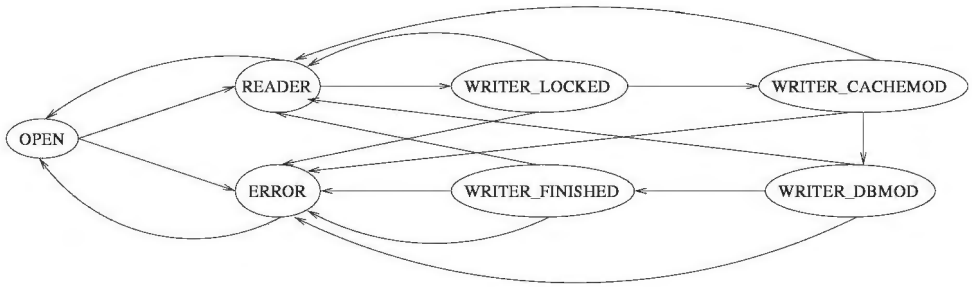


Figure 5.6: Pager state transition diagram.

Based on the value of the **eLock** member variable, a **Pager** object can be in one of the following four states.

1. NO_LOCK: The pager is not currently reading or writing the database file via this `Pager` object.
2. SHARED_LOCK: The pager has been reading pages (in arbitrary order) from the database file. There can be multiple read-transactions accessing the same database file at the same time through their respective `Pager` objects. Modifying an in-cache page is not permitted.
3. RESERVED_LOCK: The pager has reserved the database file for writing but has not yet made any changes to the file. Only one pager at a time can reserve a given database file. As the original database file has not been modified, other pagers are allowed to read the file.
4. EXCLUSIVE_LOCK: The pager has been writing pages (in arbitrary order) back into the database file. The file access is exclusive. No other pager can read or write the file while this pager continues writing the file.

A page cache comes up in the NO_LOCK state. The first time the tree module invokes the `sqlite3PagerGet` function to read any page from the database file, the pager transits to the SHARED_LOCK state. After the tree module releases all pages by executing the `sqlite3PagerUnref` function, the pager transits back to the NO_LOCK state. (At this point, it may not purge the page cache.) The first time the tree module invokes the `sqlite3PagerWrite` function on any page, the pager transits to the RESERVED_LOCK state. (You may note that `sqlite3PagerWrite` function can only be called on a page that is already read; it implies that the pager must be at the SHARED_LOCK state before it transits to the RESERVED_LOCK state.) The pager transits to the EXCLUSIVE_LOCK state before actually writing the very first (any) page to the database file. In the mid of `sqlite3PagerRollback` or `sqlite3PagerCommitPhasTwo` function execution, the pager transits back to the NO_LOCK state.

Note: For temporary and in-memory databases, the `Pager.eLock` is always set to EXCLUSIVE_LOCK because they cannot be accessed by other processes. ◁

5.3.2 Cache organization

Each page cache is managed via a PCache handler object. The pager holds a reference to this object (see Fig. 5.3 on page 126). Figure 5.7 depicts a few member variables of a PCache object. SQLite support a pluggable caching scheme that users can supply. It provides its own pluggable cache module (implemented in the pcache1.c source file) that I discuss below. Unless the user provides one, this becomes the default cache manager. The last component of the PCache object, namely `pCache`, holds a reference to a pluggable cache module object.

In general, to speed up searching a cache, the currently held in-cached items are well organized. The cache space is slotted to hold data items. SQLite uses a hash table to organize cached pages,

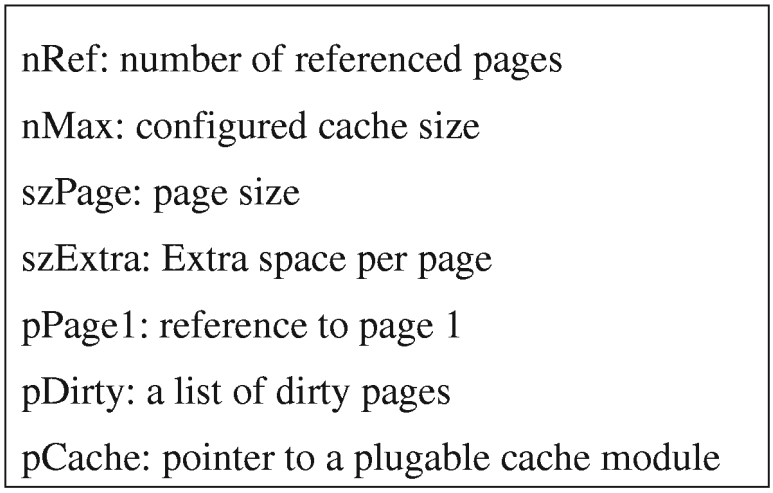


Figure 5.7: The PCache structure.

and uses *page-slots* to hold pages in the table. The cache is fully associative, that is, any slot can store any page. The hash table is initially empty. As demand for pages increases, the pager creates new slots and inserts them in the hash table. There is a maximum limit (`PCache.nMax` value) on the number of slots a cache can have. The default value is 2000 for the main and other attached databases, and 500 for the temp database. (In-memory databases have no such limit so long as the native operating system allows the application address space to grow.)

SQLite represents each page in the cache by an object of PgHdr type. The pager understands this objects though a pluggable cache can have its own page header object. Figure 5.8 depicts the layout of SQLite's own pluggable cache, represented by a `PCache1` object. Each slot in the hash table is represented by a header object of `PgHdr1` type. The pluggable component understands this type and the pager is opaque to it. The slot image is stored right before the `PgHdr1` object; the size of the slot image is determined by the value of `PCache1.szSize` variable. The slot image holds an object of `PgHdr`, a database page image, and a piece of private data that is used by the tree module to keep page-specific in-memory control information there. (In-memory databases have no journal file, so their recovery information is recorded in in-memory objects. Pointers to those objects are stored following the private part: these pointers are used by the pager only.) This (additional nonpage) space is initialized to zeros when the pager brings or constructs the page into the cache. All pages in the cache are accessible via the `PCache1.apHash` hash array; the array size is stored in the `PCache1.nHash` variable; the array is resized depending on the need. Each array element points to a "bucket" of slots; slots in each bucket are organized in an unordered singly linked list.

The `PgHdr` object is only visible to the pager module, and not visible to the tree and higher-up modules. The header has many control variables. The `pgno` variable identifies the page number of the database page it represents. The `needSync` flag is true if the journal needs a flush before writing

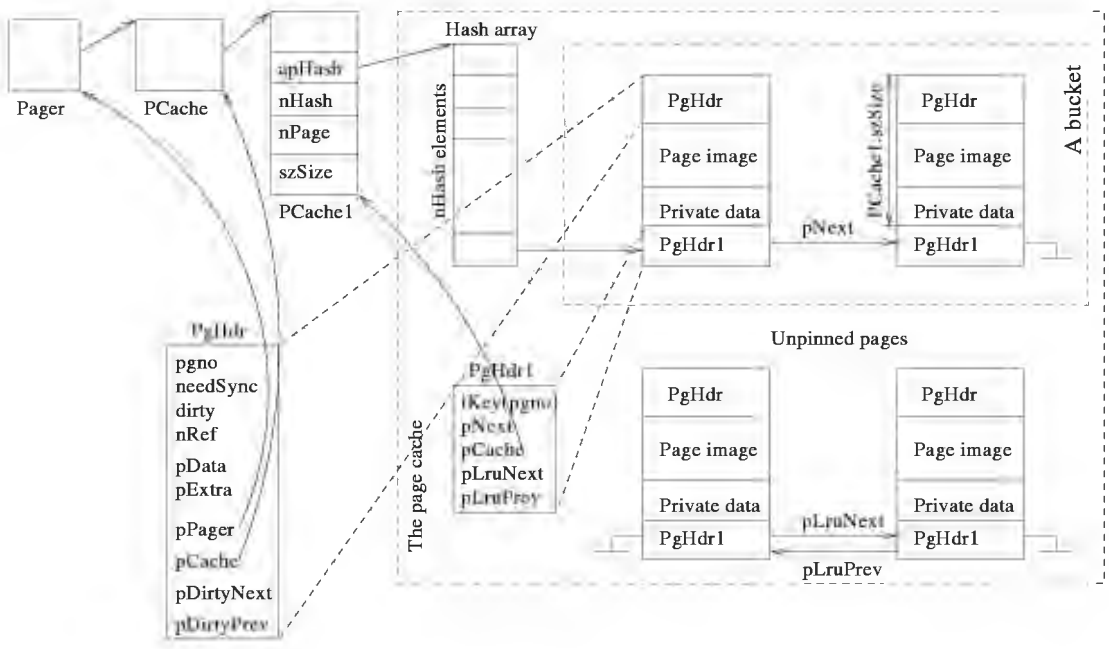


Figure 5.8: Page cache.

this page back into the database file. The `dirty` flag is true if the page has been modified, and the new value is not yet written back into the database file. The `nRef` variable is the reference count on this page. If the `nRef` value is greater than zero, the page is in active use, and we say that the page is *pinned down*; otherwise, the page is unpinned and free. The `pDirtyNext` and `pDirtyPrev` pointers are used to link together all dirty pages.

Cache Group: There is an option where SQLite puts all `PCache1` objects in a single group. The caches can recycle each other's unpinned page-slots when they are subjected to memory pressure. ◁

5.3.3 Cache read

You may recall that a cache is not a direct addressable storage unit. Cache clients cannot reference individual cache elements by providing their cache addresses. In fact, they may not know where in the cache a page copy resides, and would not know its cache address. A cache is a content addressable storage space. It is referenced by using a search key—the page number in our case. Moving pages between the cache and the database file is the basic function of the pager as the data manager. It uses the `PCache1.apHash` array to translate page number to appropriate cache slot via the cache bucket. Initially, the page cache is empty, but pages are added to the cache on demand basis. To read a page, as mentioned previously, the client (aka, the tree module) invokes the `sqlite3PagerGet` function on the page number. The function performs the following steps for a requested page P.

1. It searches the cache space.

(a) It applies a very simple hash function[1] on P to determine the index into the `apHash` array: page number modulo the size of the apHash array.

(b) It uses the index into the `apHash` array and gets the hash bucket.

(c) It searches the bucket by chasing the `pNext` pointers. If P is found there, we say a cache hit has occurred. It pins down the page (i.e., increments the `PgHdr.nRef` value by 1) and returns the base address of the page-image to the caller.

2. If P is not found in the cache, it is considered a cache miss. The function looks for a free slot that can be used to load the desired page. (If the cache has not reached the maximum limit of `PCache.nMax`, it instead creates a new free slot.)

3. If no free slot is available or can be created, it determines a slot from which the current page can be released to reuse the slot for P. This is called a *victim slot*. (Victim selection is addressed in Section 5.3.6 on page 135.)

4. If the victim (or the free slot) is dirty, it writes the page to the database file. (Following write-ahead-log (WAL) protocol, it flushes the journal file too.)

5. Two cases. (a) If P is less than or equal to the current max page in the file, it reads page P from the database file into the free slot, pins down the page (i.e., it sets the `PgHdr.nRef` value to 1), and returns the address of the page to the caller. (b) If P is greater than the current max page in the file, it does not read the page; instead, it initializes the page to zeros. In either case, it also initializes the bottom private part to zeros whether or not it reads the page from the file. It also sets the `PgHdr.nRef` value to 1.

SQLite strictly follows fetch-on-demand policy to keep the page fetch logic very simple. (See Section 5.3.5.)

When the address of an in-cached page is returned to the client (aka, the tree module), the pager does not know when the client actually works on the page. SQLite follows this standard protocol for each page: the client acquires (aka, pins down) the page, uses the page, and then releases (aka, unpins) the page. When a page address is returned to the client, the page is pinned down (the `PgHdr.nRef` is greater than zero). The page will be unpinned only when the client calls `sqlite3PagerUnref` function on the page and the `nRef` becomes zero. Pinned pages are currently in active use, and cannot be recycled by the cache manager. To avoid the scenario that 'all pages in the cache are pinned down', SQLite needs a minimum number of pages in the cache so that it always has some cache slot(s) to recycle; the minimum value is 10 as of SQLite 3.7.8 release.

[1] A *hash function* maps a larger set onto a smaller set.

5.3.4 Cache update

After acquiring a page, the client can directly modify the content of the page, but as mentioned previously it must call the `sqlite3PagerWrite` function on the page prior to making any modifications there. On return from the call, the client can update the page in place as many times as it wants.

The first time the client calls the `sqlite3PagerWrite` function on a page, the pager writes the original content of the page into the rollback journal file as part of a new log record, and sets the `PgHdr.needSync` flag on. Later, when the log record is flushed onto the disk surface, the pager clears the `needSync` flag. (SQLite follows WAL protocol: it does not write a modified page back into the database file until the corresponding `needSync` has been cleared.) Every time the `sqlite3PagerWrite` function is called on a page, the `PgHdr.dirty` flag is set; the flag is cleared only when the pager writes back the page content into the database file. Because the time when the client modifies a page is not known to the pager, updates on the page are not immediately propagated to the database file. Thereby, the pager follows a delayed write (aka, write-back) page update policy. The updates are propagated to the database file only when the pager performs a cache flush or selectively recycles dirty pages.

Note: A transaction performs direct updates on cached pages, and the cache manager does deferred updates on database files. Direct cache updating requires saving old values of pages so that they can be restored if the transaction aborts itself. Deferred updates to database file can increase a transaction's memory usage. When the memory usage crosses the upper boundary, the cache manager performs a cache replacement. ◁

5.3.5 Cache fetch policy

Cache fetch policies decide when to bring a page into the cache. A fetch-on-demand policy brings a page into the cache only when the page is required by the client. You may note that during demand fetching the client is stalled, and it cannot make any progress until the page is read from the database file. Many cache systems use sophisticated pre-fetch techniques to bring some pages to the cache in advance to reduce the frequency of stalling. SQLite strictly follows fetch-on-demand policy, and avoids any other pre-fetch policy to keep the fetch logic very simple and SQLite library size in check. Also, it reads one page at a time from the database file.

5.3.6 Cache management

In general, a page-cache is of limited size, and unless the database is pretty small, it can hold only a small number of pages from the database. The cache space may need to be recycled to hold different pages from the database at different times. Thus, the space must be managed with extreme care

to gain real performance from the cache. The basic idea is to keep in the cache those pages that are immediately required by cache clients. We need to consider three things while devising a cache management policy. (1) Whenever there is a page in the cache, there is also a master copy of the page in the database file. Whenever the cache copy is updated, the master copy may need to be updated too. (2) For a requested page that is not in the cache the master copy is referenced and a new cache copy is made from the master. (3) If the cache is full and a new page is to be placed in the cache, a replacement algorithm is invoked to remove some old page from the cache to make room for the new one.

As cache is a limited size storage space, we need to recycle the cache space to 'fit' a larger collection of pages into a small number of cache slots (see Fig. 5.9). In the figure, there are 26 master pages that we would need to fit into five cache slots by recycling the slots. Consequently, the space needs to be managed well to achieve real performance gain from the cache. Cache management is very crucial for cache performance as well as the overall system performance. As long as there are free slots available in the cache for newly requested pages, there is no hard work to do by the cache manager. Cache management becomes challenging when the cache becomes full. The duty of a cache manager is to decide what it will keep in the cache and what it will flush out off the cache when the cache is full. The effectiveness of a cache is a measure of how often requested pages are found in the cache. We need a cache with a very high hit rate. So, for cache replacements, the most crucial thing is to determine which pages are to be retained in the cache. If the decision is bad, the cache will be polluted with non immediately required, unimportant pages. I discuss cache replacement now.

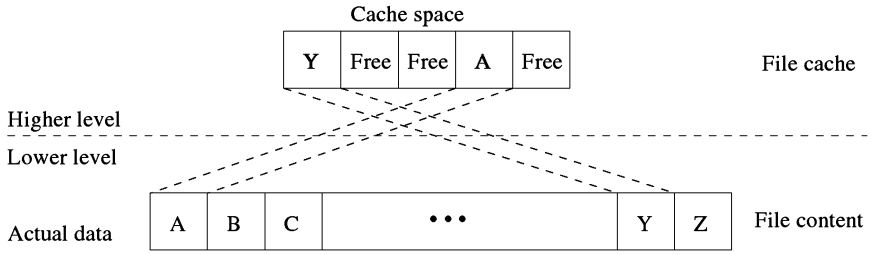


Figure 5.9: A typical cache management scheme.

5.3.6.1 Cache replacement

Cache replacement refers to the activity that takes place when a cache becomes full, and old pages are removed from the cache to make room for new ones. As mentioned in the Section 5.3.3 on page 133, when a requested page is not in the cache and a free slot is not available in the cache, the pager victimizes a slot for replacement. Victim selection may not be an easy decision to make. You may recall that the page cache is fully associative, that is, any slot is good for the new page.

As more than one slot is considered for replacement, the one that is chosen is dictated by the cache replacement policy. The main objective of a replacement policy is to keep those pages in the cache so that a large fraction of requests can be satisfied from the cache without referencing master pages. That is, the cache hit rate should be very high. If cache hit rate is low, the cache is not worth to be used for speeding up page accesses.

You may note that we do not have knowledge about future page reference patterns. Consequently, the cache manager has to take replacement decisions based on some heuristics or limited amount of past history. Thus, the replacement schemes that are normally used in practice occasionally make 'mistakes' in choosing victims—replacements that are quickly undone when the victims are recalled immediately by clients. A replacement of a page p is considered to be a poor choice if p is referenced again and at least one cache slot has not been referenced since p is replaced. The objective of a replacement policy is to minimize mistakes and maximize the times between cache misses. One cache replacement policy differs from another one how victim slots are selected for replacement. Many replacement policies have been proposed in the literature, and are implemented in a variety of contexts. First-in first-out, least recently used, least frequently used, clock schemes are widely followed cache replacement policies in both hardware and software development. SQLite uses a kind of least recently used (LRU) replacement scheme.

5.3.6.2 LRU cache replacement scheme

LRU is a very popular replacement policy. It and its variants have been implemented successfully in many areas of software and hardware cache development. It exploits temporal locality of references to pages. (Temporal locality refers to repeated accesses to the same page within a short span of time.) Locality of immediate past references is used to predict immediate future references. What it means is that if a page is accessed now, it is assumed that the page will be accessed again soon; if a page is not accessed for a long time, it is assumed that the page will not be accessed again soon. The victim is the one that has not been accessed for the longest time.

5.3.6.3 SQLite's cache replacement scheme

SQLite organizes inactive pages in a logical queue. When a page is unpinned, the pager appends the page at the tail end of the queue. (The page at the tail end of the queue is always the latest one accessed, and the one at the head end of the queue is accessed farthest in the past.) The victim is chosen from the header end of the queue, but may not always be the head element on the queue as is done in the pure LRU scheme. SQLite tries to find a slot on the queue starting at the head such that recycling that slot would not involve doing a flush of the journal file. (You may recall that following the WAL protocol, before writing a dirty page into the database file, the pager flushes

the journal file. Flushing is a slow operation, and SQLite tries to postpone the operation as long as possible.) If such a victim is found, the foremost one on the queue is recycled. Otherwise, SQLite does flush the journal file first, and then recycles the head slot from the queue. If the victim page is dirty, the pager writes the page to the database file before recycling it.

5.4 Transaction Management

The pager is also the transaction manager in SQLite, and it is responsible for ensuring transactional ACID properties by managing locks on database files and by managing log records in journal files. Although SQLite lock manager (discussed in Chapter 4) acquires and releases locks on files, the pager decides on the mode of locks and the time of acquiring and releasing the locks. It follows strict two phase locking protocol to produce serializable executions of transactions. It also determines the content of log records, and their writing to the journal file.

Like any other DBMS, SQLite's transaction management has two components: (1) normal processing and (2) recovery processing. During normal processing the pager saves enough recovery information in the journal file, and it uses the saved information at the recovery processing time when needs arise. The activities of the two processing components are presented in the next two subsections. Bits and bytes of transaction management are discussed earlier in piecemeal. Here, I consolidate them in a cohesive manner.

5.4.1 Normal processing

Normal processing involves reading pages from and writing pages into database files, committing transactions and statement subtransactions, and setting up and releasing savepoints. In addition, the pager selectively recycles page-cache slots or flushes the page-cache as a part of normal processing work.

5.4.1.1 Read operation

To act on a database page, the client (aka, the tree module) needs to apply the `sqlite3PagerGet` function on the page number. The client needs to invoke the function, even if the page is nonexistent in the database file: the new page will be created by the pager. The function will obtain a shared lock on the database file if a shared or stronger lock has not already been obtained on the file.[2] If it fails to obtain the shared lock, it means some other transaction holds an incompatible lock and the function returns the SQLITE_BUSY error code to the caller. Otherwise, it performs a cache

[2]The first time the pager acquires the shared lock, we say it has started an implicit read-transaction for the client. When the transaction commits or aborts, the pager releases the shared lock.

read operation (see Section 5.3.3 on page 133), and returns the caller a pointer to the page. As mentioned previously, the cache read operation pins down the page.

You may recall from Fig. 5.8 on page 133 that each in-memory page image is followed by a chunk of private space. This extra space is always initialized to zeros the first time the page is loaded from the database file (or created and initialized) into the main memory. This space is later reinitialized by the tree module.

You may also recall that the first time the pager acquires a shared lock on a database file, it determines whether or not the file needs a recovery. It looks for the presence of the corresponding 'hot' journal file. (I discuss determination of hot journal and failure recovery in Section 5.4.2.4 on page 146.) If the hot journal file does exist, it means that there was a failure during the previous transaction execution on the database and the pager rolls back the failed transaction and finalizes (i.e., deletes, truncates, or invalidates) the journal file before returning to the caller from the `sqlite3PagerGet` function.

As mentioned previously, a requested page may not be in the page cache. In that case the pager finds a free cache-slot and reads the page from the database file in a user transparent way. Getting a free cache slot may lead to a writing of a (victim) page into the database file, i.e., requires a cache flush (see Section 5.4.1.3).

5.4.1.2 Write operation

Before modifying a page, the client (aka, the tree module) must have already pinned down the page (by applying the `sqlite3PagerGet` function on the page). It applies the `sqlite3PagerWrite` function on the page to make the page writable. Once a page becomes writable, the client can update the page as many times as it wants to without informing the pager. Writing a page never leads to a cache flush. The pager may though need to acquire a reserved lock on the database file. The first time the `sqlite3PagerWrite` function is called on a (any) page, the pager acquires a reserved lock on the database file. The reserved lock indicates an intention to write the database file in the near future. Only one transaction at a time can hold a reserved lock. If the pager is unable to obtain the lock, it means that another transaction already has a reserved or stronger lock on the file. In that case, the write attempt fails, and the pager returns the SQLITE_BUSY error code to the caller.

The first time the pager acquires a reserved lock, we say it escalates the read-transaction into a write-transaction. (You may note that this is either a system or a user transaction.) At this point, the pager creates and opens the rollback journal. (The rollback journal is created in the same directory where the database file resides and has the same name but with '-journal' appended.) It initializes the first segment header record (see Fig. 3.7 on page 91), notes down the original size of

the database file in the record, and writes the record in the journal file.

To make a page writable, the pager writes the original content of that page (in a new log record) into the rollback journal. (You may note that newly created pages are not logged because there are no old values for the pages.) A page is written at most once to the rollback journal file. Changes to the page are not written to the database file immediately. Changes to pages are held in-cache at first. The database file remains unaltered, which means that other transactions can continue to read from the file.

Sector Logging: If the sectors in the storage device can store more than one database page, SQLite logs the entire sector instead of the page being updated. ◁

Page Logging Policy: Once a page image is copied into the rollback journal, the page will never be in a new log record, even if the current transaction invokes the `sqlite3PagerWrite` function on the page multiple times. One nice property of this logging is that a page can be restored by blind-copying the content from the journal. Thereby, the undo operation is idempotent, and it does not produce any compensating log records. SQLite never saves a new page (that is added, i.e., appended, to the database file by the current transaction) in the journal because there is no old value for the page. Instead, the initial size of the database file is stored in the journal segment header record (see Fig. 3.7 on page 91) when the journal file is created. If the database file is expanded by the transaction, the file will be truncated to its original size on a rollback. ◁

5.4.1.3 Cache flush

The cache flush is an internal operation of the pager module; the client (aka, the tree module) can never invoke a cache flush directly. There are two situations when the pager would like to flush a page out of the page cache: (1) the cache has filled up and there is a need for cache replacement, or (2) the transaction is ready to commit its changes. The pager writes some or all modified pages back to the database file. Before this writing happens, the pager must make sure that no other transaction is reading the database file. SQLite follows WAL protocol to write database files. It means that the rollback journal content may need to be flushed to the disk so that it should be possible to rollback the transaction in the event of a failure while writing pages into the database file. The pager performs the following steps:

1. It determines whether there is a need to flush the journal file. If the transaction is synchronous and has written new data in the journal file and the database is not a temporary file,[3] then the pager needs to do a journal flush. In that case, it makes an `fsync` system call on the journal file to ensure that all log records written so far have actually reached the disk surface. At this

[3]For temporary databases, we do not care if we are able to do a rollback after a system or power failure, so no journal flush occurs.

time of fsync, the pager does not write the number of log records (`nRec`) value in the current log segment header. (The `nRec` value is a precious resource for rollback operation. When the segment header is formed, the number is set to zero for synchronous transactions and to -1, aka, 0xFFFFFFFF for asynchronous ones.) After the journal is flushed, the pager writes the `nRec` value in the current log segment header, and does another fsync on the file again.[4] As disk writes are not atomic, it will not rewrite the `nRec` field any more. The pager instead creates a new log segment for the new oncoming log records. In these scenarios, SQLite uses multisegment journal files.

2. It tries to obtain an EXCLUSIVE lock on the database file. (The pager never unconditionally waits for the lock grant. It attempts for the lock in nonblocking mode. If other transactions are still holding SHARED locks, the lock attempt fails, and it returns the SQLITE_BUSY error code to the caller. The transaction is not aborted.)

3. It writes all modified pages (currently held in the page cache) or selective ones into the database file. The page writing is done in-place. It marks cache copies of these pages clean. (It does not flush the database file to disk at this time.)

If the reason for writing to the database file is because the page cache is full, then the pager does not commit the transaction right away. Instead, the transaction might continue to make changes to other pages. Before subsequent changes are written to the database file, these three steps are repeated once again by the pager.

Note: The EXCLUSIVE lock the pager has obtained to write to the database file is held until the transaction is complete. It means that this process (via a different library connection) and other application processes will not be able to open another (read or write) transaction on the database from the time the pager first writes the database file until the transaction commits or aborts. For short transactions, updates are held in-cache, and the exclusive lock will be acquired only at the commit time for the duration of the commit. But, a long transaction causes degradation of performance of other read-transactions. ◁

5.4.1.4 Commit operation

SQLite follows a slightly different commit protocol depending on whether the committing transaction modifies a single database or multiple databases.

Single database case: When the tree module is ready to commit a transaction, it invokes the `sqlite3PagerCommitPhaseOne` function first (see the first two items on the following list) and then `sqlite3PagerCommitPhaseTwo` function (the last two items on the following list). Committing a

[4]The journal file is flushed twice. The second flush leads to an overwrite of the disk block that stores the nRec field. If this overwrite is atomic, then we are guaranteed that the journal will not be corrupted at this point of flushing. Otherwise, we are at some minor risk.

read-transaction is easy. The pager releases the shared lock from the database file (if there are no other read- or write-transactions on the database) and returns to the NO_LOCK state; it does not have to purge the page cache. (The next transaction starts with a warm page-cache.) To commit a write-transaction, the pager performs the following steps in the order listed:

1. It obtains an EXCLUSIVE lock on the database file. (If the lock acquisition fails, it returns SQLITE_BUSY to the caller of the `sqlite3PagerCommitPhaseOne` function. It cannot commit the transaction now, as other transactions from other database connections are still reading the database.) It increases the database metadata file-change-counter. It writes all modified (in-cached) pages back to the database file following the algorithmic steps 1–3 of Section 5.4.1.3. This is called flush-log-at-commit, and is done to save sufficient information at the rollback journal to remove the effects of the entire transaction.

2. Many operating systems such as Linux cache these writes in-memory in the operating system space, and may not send them to the disk right away. To overcome this situation, the pager makes an `fsync` system call on the database file to flush the file to the disk. This is called flush-database-at-commit, and is done to eliminate the redo logic at system restart time.

3. It then finalizes (i.e., deletes, truncates, or invalidates) the journal file.

4. Finally, it releases the EXCLUSIVE lock from the database file. If there are concurrent select operation executions (i.e., read-transactions), it returns to the SHARED_LOCK state; otherwise, it returns to the NO_LOCK state; it does not have to purge the page cache.

Commit Point: The transaction commit point occurs at the instant the rollback journal file is finalized. Prior to that, if a power failure or system crash occurs, the transaction is considered to have failed during the commit processing. The next time SQLite reads the database, it will roll back the transaction's effects from the database. SQLite does assume that the journal finalization by the native operating system is an atomic operation. ◁

Multidatabase case: The commit protocol is a little more involved, and it resembles a transaction commit in a distributed database system. The VM module (the `VdbeCommit` function) actually drives the commit protocol as the commit coordinator. The pager of each database does its own part of "local" commit on its database. For a read-transaction or a write-transaction that only modifies a single database file (the `temp` database is not counted), the protocol executes a normal commit on each database involved. If the transaction modifies more than one database file, the following commit protocol is performed:[5]

[5]If the main database is ":memory:", SQLite does not ensure atomicity of multidatabase transactions. Instead, it follows the simple commit on individual database files.

1. Release the SHARED lock from those databases the transaction did not update (if other read-transactions are not active from this thread).

2. Acquire EXCLUSIVE locks on those databases the transaction has updated. Increment the file-change-counter metatdata for the database file.

3. Create a new master journal file. (The master journal is always in the same directory as the main database and has the same name but with '-mj' appended followed by eight randomly chosen 4-bit hexadecimal numbers. This happens even when the committing transaction does not modify the main database.) Fill the master journal with the names of all individual rollback journal files, and flush the master journal and the journal directory to disk. (The temp database name is not included in the master journal.)

4. Write the name of the master journal file into all individual rollback journals in a master journal record (see Fig. 3.9 on page 94), and flush the rollback journals. (A pager may not know that it has been a part of multidatabase transaction until the transaction commit time. Only at this point it comes to know that it is a part of a multidatabase transaction.)

5. Flush individual database files.

6. Delete the master journal file and flush the journal directory.

7. Finalize (delete or truncate) all individual rollback journal files.

8. Release EXCLUSIVE locks from all database files. All pagers return to the SHARED_LOCK or NO_LOCK state. The pagers do not have to purge their page-caches.

Commit Point: The transaction is considered to have been committed when the master journal file is deleted. Prior to that, if a power failure or system crash occurs, the transaction is considered to have failed during the commit processing. When SQLite reads these databases next time, it will recover them to their respective states prior to the start of the transaction. ◁

Rollback Journal Finalization: When the `journal_mode` is persist, the journal file is truncated to the zero size instead of invalidating the journal header. ◁

Warning!: If the main database is a temporary file (or in-memory), SQLite does not guarantee atomicity of the multidatabase transaction. That is, the global recovery may not be possible. It does not create a master journal. The VM module follows the simple commit on individual database files, one after another. The transaction is thus guaranteed to be locally atomic within each individual database file. Thus, on occurrence of a failure, some of those databases might get the transaction's updates and some might not. You have been warned! ◁

Commit Failure: User-level transactions are committed by applications themselves executing the COMMIT command and SQLite attempts to finish the transaction. As mentioned previously, an attempt to execute the COMMIT command may fail due to lock conflict and might result in an SQLITE_BUSY return code. This indicates that another transaction holds a shared lock on the database, which prevents the COMMIT to succeed. When COMMIT fails in this way, the transaction remains active and the COMMIT can be retried by the application later after the other transactions have had a chance to clear their shared locks. SQLite does not automatically retry the commit. The application has to do it by itself. ◁

5.4.1.5 Statement operations

Statement subtransactions are implemented as anonymous savepoints that are released at the end of the subtransactions. Normal operations at the level of statement subtransaction are read, write, and commit. These are discussed below.

Read operation: A statement subtransaction reads pages through the encompassing user transaction. All rules are followed as those for the user transaction.

Write operation: There are two parts in a write operation: locking and logging. A statement subtransaction acquires locks through the encompassing user transaction. But statement logging is a little different, and is handled by using a separate temporary statement journal file. (The statement journal is an arbitrary named, prefixed with `etilqs_`, temporary file.) The pager writes some log records in the statement journal, and some in the main rollback journal. It performs one of the following two alternative actions when a subtransaction tries to make a page writable via the `sqlite3PagerWrite` operation:

1. If the page is not already in the rollback journal, the pager adds a new log record to the rollback journal. (But, a newly added page is not logged.)

2. The pager adds a new log record to the statement journal if the page is not already in this journal. (The pager creates the statement journal file when the statement subtransaction writes the first log record in the file.)

The pager never flushes a statement journal, because this is never required for failure recovery. If a system failure or power loss occurs, the main rollback journal will take care of database recovery. You may note that when a page is a part of both the rollback journal and the statement journal, the rollback journal has the oldest page image.

Commit operation: Statement commit is very simple. The pager deletes the statement journal file. (But, see the following sub-subsection.)

5.4.1.6 Setting up savepoints

When a user transaction establishes a savepoint, SQLite enters into the savepoint mode. In this mode, SQLite no more deletes a statement journal on the statement commit. It retains the journal until the transaction releases all savepoints or it commits or aborts itself. In the savepoint mode, logging is a bit different: if a page was added by a previous statement, the page is again added to the current statement journal. Thereby, the statement journal can have several log records for the same database page.

5.4.1.7 Releasing savepoints

When the application executes a `release sp` command, SQLite destroys the corresponding `PagerSavepoint` object and those that were created after the `sp` savepoint was established. The application can no more reference these savepoints.

5.4.2 Recovery processing

Most transactions and statement subtransactions commit themselves. But occasionally, some transactions or statements abort themselves. In rare cases, there are applications and system failures. In either case, SQLite may need to recover the database into an acceptable consistent state by performing some rollback actions. In the former two cases (of statement and transaction aborts), in-memory reliable information might be available at the time of recovery. In the latter case (failures), the database may be corrupt, and there is no in-memory information. There is an in-between case where a transaction reverts to a previous savepoint. I discuss these four cases in the following four sub-subsections.

5.4.2.1 Transaction abort

Recovery from abort is very simple in SQLite. The pager may or may not need to remove the effects of the transaction from the database file. If the transaction holds only a RESERVED or PENDING lock on the database, it is guaranteed that the file is not modified; the pager finalizes the journal file, and discards all dirty pages from the page cache. Otherwise, the transaction holds an exclusive lock on the database file and some pages may have been written back into the database file by the transaction, and the pager performs the following rollback actions.

The pager reads log records one after another from the rollback journal file, and restores the page images from the records. (You may recall that a database page is logged at most once by the transaction and the log record stores the before image of the page.) Thus, at the end of the journal scan, the database is restored to its original state prior to the start of the transaction. If

the transaction has expanded the database, the pager truncates the database to the original size. It (the pager) then flushes the database file first and finalizes the rollback journal file next. It releases the exclusive lock, and purges the page cache.

5.4.2.2 Statement subtransaction abort

As noted in Section 5.4.1.5, a statement subtransaction may have added log records to both the rollback journal and the statement journal. SQLite needs to roll back all log records from the statement journal, and some from the rollback journal. As mentioned previously, each statement is treated as an anonymous savepoint. So, a statement abort is equivalent to restoring the anonymous savepoint. I discuss it now in the next sub-subsection.

5.4.2.3 Reverting to savepoints

As noted in Section 5.4.1.6 on page 145, when in the savepoint mode, a transaction does not delete the statement journal. When the transaction executes a `rollback to sp` command, SQLite also plays the log record from the statement journal that are produced after the `sp` point has been established. Three member variables of the corresponding of `PagerSavepoint` object play a crucial role: iOffset, iHdrOffset, and iSubRec. It first plays all the log records from the main rollback journal starting at the one at iOffset until the end of the journal file. It then plays all log records from the statement journal starting at iSubRec until the end of the file are used to restore the savepoint. However, in the previous case, if the iHdrOffset is non-zero, the playing of log records from the rollback journal is done in two steps: (1) from iOffset to iHdrOffset and (2) all the subsequent log segments. During the restore process, the pager keeps an account of which pages are rolled back and makes certain that a page is not rolled back more than once. Pager.dbSize is restored to that of the start of the savepoint (PagerSavepoint.nOrig). For reverting an entire transaction, only the rollback journal is used. SQLite destroys all the `PagerSavepoint` objects created after the `sp` savepoint, but not including its one. These savepoints are no more accessible to the application.

5.4.2.4 Recovery from failure

After a process crash or system failure, inconsistent data may have been left in a database file. When no application is updating a database, but there is the rollback journal file, it means that the previous transaction may have failed, and SQLite may need to recover the database from the effects of the failed transaction before the database can be used for normal business. A rollback journal file is said to be *hot* if the corresponding database file is unlocked or shared locked. A journal becomes hot when a write-transaction is progressing toward its completion and a failure

prevents the completion. However, a rollback journal is not hot if it is produced by a multidatabase transaction and there is no master journal file; this implies that the transaction is committed by the time the failure had occurred. A hot journal implies that it needs to be rolled back to restore the database consistency.

Hotness Determination: There are two cases. (1) A master journal is *not* involved, that is, a master journal record does not appear in the rollback journal file. A rollback journal is *hot* if it exists and it is valid (i.e., the journal header is well-formed and not zeroed) and the database file does not have a reserved or stronger lock and database is not empty (size = 0). (You may recall that a transaction with a reserved lock creates a rollback journal file; this file is not hot.) (2) A master journal name appears in the rollback journal. The rollback journal is hot if the master journal exists and has a reference to this rollback journal and there is no reserved or stronger lock on the corresponding database file. ◁

When a database starts, in most DBMSs, the transaction manager initiates a recovery operation on the database immediately. SQLite does a deferred recovery. As explained in Section 5.4.1.1, when the first read of a (any) page in the database is performed, the pager goes through the recovery logic and recovers the database only if the rollback journal is hot.

Warning!: If the current application only has the read permission on the database file, and no write permission on the file nor on the containing directory, the recovery fails and the application gets an unexpected error code from the SQLite library. ◁

When the pager wants to read from a database file for the first time, it performs the following sequence of recovery steps before it actually reads a page from the file.

1. It obtains a SHARED lock on the database file. (If it cannot get the lock, it returns the SQLITE_BUSY error code to the application.)

2. It checks if the database has a hot journal. If the database does not have a hot journal, the recovery operation finishes. If there is a hot journal, the journal is rolled back by the following steps.

3. It acquires an EXCLUSIVE lock on the database file. (The pager does not acquire a RESERVED lock because that would make other pagers think the journal is no longer hot and they would read the database. It needs an exclusive lock because it is about to write to the database file as a part of the recovery work.) If it fails to acquire the lock, it means another pager is already trying to do the rollback. In that case, it releases all locks and returns SQLITE_BUSY to the application.

4. It reads all log records from the rollback journal file and undoes them. This step restores the database to its original state prior to the start of the crashed transaction, and hence, the

database is in a consistent state now. If needed, it truncates the database file to the size as of the start of the failed transaction.

5. It flushes the database file. This protects the integrity of the database in case another power failure or crash occurs.

6. It finalizes (i.e., deletes, truncates, or invalidates) the rollback journal file.

7. It deletes the master journal file if it is safe to do so. (This step is optional. Discussed below.)

8. It reduces the lock strength to SHARED. (This is because the pager performs the recovery in the `sqlite3PagerGet` function.)

After the above algorithm terminates successfully, the database file is guaranteed to have been restored to the state as of the start of the failed transaction. It is safe to read from the file now.

Stale Master Journal: A master journal is considered *stale* if no individual rollback journal references it any more. The pager first reads the master journal and obtains the names of all rollback journals. It then examines each of those rollback journals individually. If any of them exists and points back to the master journal, then the master journal is not stale. If all rollback journals are either missing or they refer to other master journals or to no master journal at all, then the master journal is stale and the pager deletes the master journal. There is no requirement that stale master journals be deleted. The only reason for doing so is to free up disk space occupied by them. ◁

5.4.3 Other management issues

This subsection briefly discusses other database related issues.

5.4.3.1 Checkpoint

To reduce workload at failure recovery time, most DBMSs perform checkpoints on databases at regular intervals. You may recall that SQLite can have at most one write-transaction on a database file at a time. The journal file contains log records from only that transaction, and SQLite deletes (or truncates or invalidates) the journal file when the transaction completes. Consequently, SQLite does not accumulate logs forever, and does not need to perform checkpoints, and it does not have any checkpoint logic embedded in it. When a transaction commits, SQLite makes sure that all updates from the transaction are in the database file before finalizing (i.e., deleting, truncating, or invalidating) the journal file. (In SQLite 3.7.0 release, the SQLite development team has introduced the WAL journaling feature. In this journaling mode, checkpoints are performed. I discuss this journaling in Section 10.17 on page 249.)

5.4.3.2 Space constraint

The most troublesome problem in some DBMSs is that the journal gets out of space. That is, the file system does not have sufficient space to grow the journal file to write new log records any more. There are DBMSs in which aborting transactions produce (compensating) log records while undoing some updates, aggravating the situation further. Lack of journal space may create problems in those systems for transaction aborts and system restarts. SQLite does not have the log space problem, because aborting transactions do not produce any new log records. System restarts could though be a problem, but only in the following extreme scenario: a transaction shrinks a database file and the released space is already allocated by the native file system for some other purposes. In such cases, recovery would fail because SQLite is not able to take the database back to the original size, and the database stalls until the required space is available for the database file to grow back to the original size.

There is another related problem: there is no space for database file to grow. In this case, the pager returns SQLITE_FULL error code to the application that may abort the transaction. So, this does not cause a problem in SQLite either.

Summary

This chapter presents the most important module in minute details. The pager directly accesses database and journal files and manages SQLite lock acquisitions and releases on database files. (No other module in SQLite directly accesses these resources bypassing the pager module.) Overall, it implements the ACID properties.

The pager uses a small amount of space in the process address space (actually, on the process heap space) to hold parts of the database file. The space is called a page cache in the SQLite world. The space is slotted, where each slot can hold precisely a database page and some control information. The page cache abstraction eases the database file access by the tree module irrespective of the type of database it accesses.

The page cache management is quite flexible. The caching scheme is a pluggable module, and users can supply their own caching modules though SQLite supplies a default one. This one uses an expandable/shrinkable hash array to organize cache slots into buckets. The page number is used as the search key. SQLite uses a kind of LRU cache replacement scheme. It maintains a queue of unpinned pages with least recently accessed ones at the header side. The victim may not be the head element. SQLite tries to find the first slot that would not cause a journal flush. If such a slot is found, it is used for replacement. Otherwise, the head slot is replaced.

This chapter also describes various transaction processing related (internal) steps toward imple-

menting ACID properties. The internal processing is partitioned into two parts: normal processing and recovery processing. The normal processing includes reading a page, writing a page, flushing the page-cache, and committing a transaction or subtransaction. When in a user transaction, SQLite executes each SQL statement in the abstraction of an anonymous savepoint. Recovery processing includes subtransaction or transaction abort, savepoint restoration, and handling system failure.

Chapter 6

The Tree Module

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- how SQLite organizes tables into separate B^+-trees and indexes into separate B-trees.
- how trees are constructed in database files, and tuples are inserted and deleted, respectively, into and from the trees
- B- and B^+-tree structures and algorithms to manipulate them
- structures of internal, leaf, and overflow pages

Chapter synopsis

Data in a database file can be organized in many ways such as entry sequence, relative, hash, key sequence. SQLite uses B^+-trees to organize the content of tables, and B-trees to organize indexes on tables. They are key sequence data structures. This chapter discusses an implementation of B/B^+-tree that is normally used to implement ordered indexes on external (disk-based) databases in other DBMSs. The algorithm implemented in SQLite is the one presented by Donald E. Knuth in his famous book, The Art of Computer Programming, Volume 3: "Sorting And Searching".

6.1 Preview

In the previous chapter, I have discussed how the pager module implements a page-oriented file abstraction on the top of a native byte-oriented file. In this chapter I will discuss how the tree module implements a tuple (aka, row) oriented file abstraction on the top of a page-oriented file. The ultimate user of tuples (aka, the database file) is the VM module that accesses tuples via

the interface provided by the tree module. The VM sees the database as a (key sequenced) tuple oriented file.

This chapter presents the tuple management scheme of SQLite. You may recall that each relation is a set of tuples. Tuples of all relations in a database are stored in the same database file. The VM must have a means to cluster and organize tuples of a relation, and separate them from those of other relations. In addition, the VM must be able to store, retrieve, and manipulate tuples efficiently. The tree module helps the VM in doing so; the tree module does the translation of tuples to pages.

Tuples of a relation can be organized in many ways such as entry sequence, relative, hash, key sequence. Each organization has its own mechanisms to insert new tuples into relations and to retrieve and delete tuples from relations based on values of some attributes (of the relation). It is a kind of content addressable tuple-based storage organization as seen by the VM. Different relations can have different tuple organizations. SQLite uses a single B^+-tree to organize all tuples of a relation, and different ones for different relations. It treats the content of an index as a kind of relation and stores the content in a B-tree, and different indexes in different B-trees. It does not use any other tuple organization techniques. Consequently, an SQLite database is a collection of B- and B^+-trees. All these trees are stored in a single file, and they are spread across database pages, and they can be interspersed. But, no database page stores tuples from two or more relations or indexes. It is the duty of the tree module to organize pages of a tree so that tuples can be stored and retrieved efficiently. The module sees underneath a page oriented file, and converts the file into a tree oriented file (of tuples).

B- and B^+-trees are similar kind of key sequence data structures. In the rest of this chapter I restrict myself primarily to B^+-tree. I will use B^+-tree for generic purpose. The tree module implements primitives to read, insert, and delete individual tuples from trees, and, of course, to create and delete trees. It is a passive module as far as the internal structures of tuples are concerned, and it treats tuples as variable length byte strings. The mapping information for relation or index names to trees (actually, the roots of trees) is kept in the `sqlite_master` (or `sqlite_temp_master`) catalog that is stored in a predetermined B^+-tree.

6.2 The Tree Interface Functions

The tree module implements a set of interface functions (that are used by the VM module). Some important functions are briefly discussed below. There are many more other such functions. They are strictly internal to SQLite, and SQLite application developers must not use them in their applications. They are defined in the btree.c source file. All function names are prefixed with `sqlite3Btree`.

1. `sqlite3BtreeOpen`: This function opens a connection to a database file, and not to a single B- or B^+-tree in the database. It ends up calling the `sqlite3PagerOpen` pager function to open the file. By doing so, it establishes a new connection between the application and the file. It creates and returns a pointer to an object of type `Btree` (see Section 6.5.1.1 on page 165) that is used by the VM as the handle in other tree interface functions.

2. `sqlite3BtreeClose`: This function closes a previously opened database connection and destroys the `Btree` object. It ends up calling the `sqlite3PagerClose` pager function to destroy its pager object. However, before doing so, it rolls back any pending transaction, closes and frees up all cursors (see the next item on this list), and frees other resources allocated to the `Btree` object.

3. `sqlite3BtreeCursor`: This function creates a new cursor on a particular tree, that is, it opens the tree for reading or writing it. (The tree is identified by its root page.) The cursor can be either a read-cursor or a write-cursor, but not both. There can be many open cursors on the same tree, each created by a separate calls to the `sqlite3BtreeCursor` function. However, read and write cursors may not coexist on the same tree. (You may note that SQLite puts a restriction that a transaction cannot concurrently both read and write the same tree using different cursors.) On the creation of the first cursor, this function obtains a shared lock on the database file (via its pager).

4. `sqlite3BtreeCloseCursor`: This function closes a previously opened cursor. The shared lock on the database file is released when the last cursor (that can be on any tree) is closed.

5. `sqlite3BtreeClearCursor`: This function basically makes the cursor invalid.

6. `sqlite3BtreeFirst`: This function moves a cursor to the first element in the tree, that is, to the left most descendant node of the tree.

7. `sqlite3BtreeLast`: This function moves a cursor to the last element in the tree, that is, to the right most descendant node of the tree.

8. `sqlite3BtreeNext`: This function moves a cursor to the next element after the one it is currently pointing to.

9. `sqlite3BtreePrevious`: This function moves a cursor to the previous element before the one it is currently pointing to.

10. `sqlite3BtreeMovetoUnpacked`: This function moves the cursor to the element that matches the key value passed as an argument. If an exact match is not found, then the cursor always points at a leaf page which would hold the entry if it were present and the cursor might point to an entry that comes before or after the key.

11. `sqlite3BtreeBeginTrans`: This function starts a new transaction on the database file. If indicated in an argument, it starts a write-transaction; otherwise, it starts a read-transaction. The caller may request for an exclusive transaction, meaning that no other process or thread is allowed to access the database.

12. `sqlite3BtreeCommitPhaseOne`: This function performs the first phase of SQLite's two phase commit. It does everything to commit a transaction such as flushing the rollback journal, flushing the database file, but it does not finalize the rollback journal nor release the locks.

13. `sqlite3BtreeCommitPhaseTwo`: This function actually commits the transaction currently in progress on the database file. It finalizes the rollback journal file, and downgrades the exclusive lock into a shared lock on the database file, and if there are no active cursors, it also releases the shared lock.

14. `sqlite3BtreeRollback`: This function rolls back the current write-transaction. All cursors accessed by the current write-transaction are invalidated by this function. Any attempt to use these cursors later results in an error. (The function downgrades the exclusive lock into a shared lock on the database file, and if there are no active cursors, it also releases the shared lock.)

15. `sqlite3BtreeBeginStmt`: This function starts a statement subtransaction. (The VM module must start a transaction before starting a subtransaction.) Only one subtransaction may be active at a time. It is an error to try to start a new subtransaction if another subtransaction is already active. Each statement subtransaction is an anonymous savepoint.

16. `sqlite3BtreeCreateTable`: This function creates a new, empty tree in the database file. The type (B or B^+) of the tree is determined by an input argument. Only two alternative values are allowed as of SQLite 3.7.8 release: BTREE_INTKEY (used for SQL tables, the new B^+-tree has integer keys and arbitrary size data) or BTREE_BLOBKEY (used for SQL indexes, the new B-tree has arbitrary size keys and no data). There must not be any cursor open on the database at the time of this function invocation; otherwise an error code is returned.

17. `sqlite3BtreeDropTable`: This function destroys a B- or B^+-tree, and releases all its pages. However, it never releases the root node that resides at Page 1.

18. `sqlite3BtreeClearTable`: This function removes all the data from an existing B- or B^+-tree but keeps the tree itself around. It frees all the corresponding tree pages and overflow pages, except the root page. The root page of the tree is empty on return from this function, as if the tree is just created.

19. `sqlite3BtreeDelete`: This function deletes the entry that the cursor is currently pointing to. The cursor is left pointing at a random location after the delete.

20. `sqlite3BtreeInsert`: This function inserts a new element at the appropriate place in the B- or B^+-tree via a cursor. The key is given by a pair ⟨pKey, nKey⟩ and the data by pair ⟨pData, nData⟩. For a B^+-tree (i.e., SQL table), only the nKey value of the key pair is used; the pKey is ignored. For a B-tree (i.e., SQL index), the pData and nData are both ignored. The cursor is used only to define what table the entry should be inserted into. The cursor is left pointing at a random location after the insert.

21. `sqlite3BtreeKeySize`: This function returns the number of bytes in the key of the entry that a given cursor is pointing to. If the cursor is not pointing to a valid entry, it returns 0.

22. `sqlite3BtreeKey`: This function returns the key of the entry that a given cursor is pointing to.

23. `sqlite3BtreeDataSize`: This function returns the number of bytes in the data of the entry that a given cursor is pointing to. If the cursor is not pointing to a valid entry, it returns 0.

24. `sqlite3BtreeData`: This function returns the data of the entry that a given cursor is pointing to.

25. `sqlite3BtreeGetMeta`: This function returns the value of a database setting parameter, aka, metadata. This function must be called inside a read- or write-transaction.

6.3 B^+-tree Structure

B-tree, where 'B' stands for 'balanced', is by far the most important index structure in many external storage based DBMSs I am aware of. It is a way of organizing a collection of similar data records in a sorted order by their keys. (A *sort order* is any total order on the keys.) Different B-trees in the same database can have different sort orders. B-tree is a special kind of height balanced n-ary, $n > 2$, tree, where all leaf nodes are at the same level. Entries[1] and search information (i.e., key values) are stored in both internal nodes and leaf nodes. B-tree provides near optimum performance over the full range of tree operations, namely insertion, deletion, search, and search next.

A *B^+-tree* is a variant of B-tree, where all entries reside in leaf nodes. The entries are ⟨key value, data value⟩ pairs; and they are sorted by the key values. Internal nodes contain only search

[1]To avoid confusion, I use the term 'entry' for a tuple or data item here. An entry consists of a key and other optional data.

information (key values) and child pointers. Keys in an internal node are stored in the sort order, and are used in directing a search to appropriate child node.

For either kind of trees, internal nodes can have a variable number of child pointers within a preset range. For a particular implementation, there are lower and upper bounds on the number of child pointers an internal node can have. The lower bound is normally equal to or more than half of the upper bound. The root node can violate this rule; it can have any number of child pointers (from zero up to the upper bound limit). All leaf nodes are at the same lowest level, and in some implementations, they are linked in an ordered chain. The root node is always an internal node for B^+-tree.

For a given upper bound of $n+1$, $n > 1$, in an $n+1$-ary B^+-tree, each internal node contains at most n keys and at most $n+1$ child pointers. The logical organization of keys and child pointers is shown in Fig. 6.1. For any internal node, the following holds:

- All of the keys on the left most child subtree that Ptr(0) points to have values less than or equal to Key(0);
- All of the keys on the child subtree that Ptr(1) points to have values greater than Key(0) and less than or equal to Key(1), and so forth;
- All of the keys on the right most child subtree that Ptr(n) points to have values greater than Key($n-1$).

Key values in internal nodes are merely used for search routing. Finding a particular entry for a given key value requires traversing $O(\log m)$ nodes, where m is the total number of data items in the tree.

Figure 6.1: Structure of a B^+-tree internal node.

6.3.1 Operations on B^+-tree

In the following sub-subsections I present four operations, namely search, search next, insert, and delete. I briefly discuss them below; details about them can be found in any text book on data structures and algorithms, especially the one by Knuth [14]. You may note that in SQLite, all B^+-trees are unique—key values are not duplicated; also it does not link together leaf nodes. In the

following sub-subsections I use the B^+-tree sample of Fig. 6.2 as a reference example. I associate a name with each node of the tree, that is printed on the top of the node: N0–N8. The names are used solely for reference purposes in the following sub-subsections.

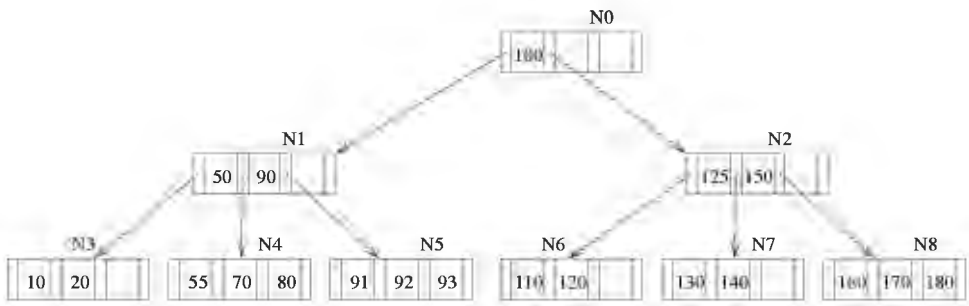


Figure 6.2: A typical text book example of a B^+-tree.

6.3.1.1 Search

Consider a search operation for key value k. The search starts at the root and terminates at a leaf. The root is always an internal node. Suppose there are n keys, $Key(0), \ldots, Key(n-1)$ and $n+1$ pointers $Ptr(0), \ldots, Ptr(n)$, on the node. The keys and pointers are logically structured as shown in Fig. 6.1. If $k > Key(n-1)$, then the search is continued at the subtree rooted at $Ptr(n)$. Otherwise, there is a $j = 0, \cdots, n-1$, such that $Key(j-1) < k \leq Key(j)$, and the search is continued at the subtree rooted at $Ptr(j)$. (I assume here $Key(-1) < k$, for any k.) If the root of the subtree is an internal node, apply the same procedure on the internal node. Eventually, the search reaches a leaf node. Search the leaf node for the existence of an entry with key value k. SQLite uses the standard binary search technique on the entities to do the local search on individual leaf nodes.

Suppose you search for 93 in our reference B^+-tree of Fig. 6.2. The search starts at the root node (N0), takes the left pointer because $93 < 100$, that is, the search restarts at node N1. The search takes the right most pointer there because $93 > 90$, and the search restarts at node N5. N5 is a leaf node, and the search terminates there with a success. If you search the tree for 94, the search would terminate at N5 with a failure.

6.3.1.2 Search next

This operation is a little more involved in SQLite as it does not link together leaf nodes. Suppose the previous search stopped on a leaf node N at index i. If i is not the last index on the node, return the $i+1$ st entry from N. Otherwise, we need to move to another leaf node that is logically

the next leaf to N. The move is a simple tree traversal logic. If N is the the last node of the tree, then there is no more entry in the tree to search and it returns EOT (end-of-tree). Suppose N is not the right most descendant on the tree. We chase parents starting at node N until we reach a node N' that is not the right most child of its parent. Suppose N' is the P_j th child of its parent P. The search moves to the left most descendant (a leaf node) of P_{j+1} subtree of P. It returns the first entry from that leaf node.

Suppose in our sample example in the last sub-subsection, the previous search terminated on node N5 at key 93. The search-next starts at N5. There is no more entry at N5 greater than 93 and the node is the right most child of its parent (N1). N1 is not the right most child of its parent (N0). So, the search moves to the left most descendant of the right most subtree of N0. That is, the search restarts at N6, and returns the first entry (110) from N6. If we call search-next again, it will return the next entry (120) from N6.

6.3.1.3 Insert

Suppose we want to insert a key k (and data) in a B^+-tree. First, we invoke the normal search to find the leaf where k could reside. If k is already there, we reject the insert operation because we have a unique B^+-tree. Suppose k is not there. If there is space on the leaf node, we insert the key and the data on the node in the desired sort position. The insert terminates then and there. If there is no space on the leaf node, we would split the existing entries residing there and the new one (being inserted) into two equal halves: lower and upper halves. (Keys on the upper half are strictly greater than those on the lower half.) We allocate a new leaf node, and move the upper half into the new node. We now need to add the new node into the tree. We add a pair $\langle K, P \rangle$, where K is the max key on the splitting node and P is the pointer to the new node, into the parent of original leaf (that got split). If the parent has space for the new pair, the pair is stored there and the insert terminates then and there. Otherwise, the parent is split in the same way. The split propagates until we find some ancestor with sufficient space for a new pair or the root is split.

Splitting the root requires extra care, because there is no higher level node where a split pair can be inserted. In addition, the rootpage of a B^+-tree is never relocated. Root splitting is normally done in the following way. Let N be the root node. First allocate two nodes, say L and R. Move lower half of N into L and the upper half into R. Now N is empty. Add $\langle L, K, R \rangle$ in N, where K is the max key in L. Page N remains the root. Note that the depth of the tree has increased by one, but the new tree remains height balanced without violating any B^+-tree property.

Suppose we insert 200 in the B^+-tree of Fig. 6.2. We first search for the leaf node where the entry can be inserted. The search terminates at node N8. The node is full. So, we allocate a new node, say N9, and transfer 180 from N8 to N9 and insert 200 in N9. We then insert $\langle 170, N9 \rangle$ into

N2 that has space to store the pair. The final configuration is shown in Fig. 6.3.

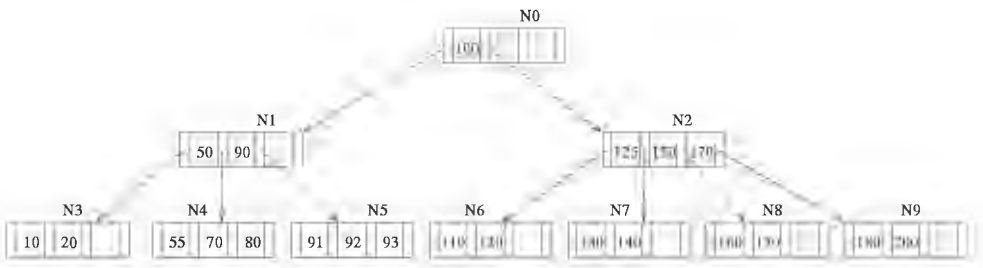


Figure 6.3: The configuration of the B^+-tree of Figure 6.2 after the insertion of 200.

6.3.1.4 Delete

Suppose we want to delete an entry with key value k from a B^+-tree. First, we invoke the normal search routine to find the leaf where k could reside. If the key value is not found on the leaf, the delete terminates immediately. Suppose the key value is there, and i be the corresponding index at the entries. All entries with index greater than i are moved down one element. If the number of remaining entries does not fall below the lower bound of page occupancy, the delete terminates then and there. Suppose the leaf occupancy falls below the lower bound. We need to restructure the tree (by merging sibling nodes) bottom up starting from this node. We do the following until the restructuring process terminates.

Suppose a node N is under restructuring process. We redistribute entries on N and up to two siblings of N so that all nodes have about the same amount of free space. Usually one siblings on either side of N is used in the balancing action, though siblings only come from one side if N is the left most or the right most child of its parent. If N has fewer than two siblings (something which can only happen if N is the root page or a child of root) then all available siblings participate in the restructuring process.

The number of siblings of N might be decreased by one or two in an effort to keep nodes (nearly) full. The root node is special and is allowed to be (nearly) empty. If N is the root node, then the depth of the tree might be decreased by one, as necessary, to keep the root node from being overfull. In the course of balancing the siblings of N, the parent of N might become underfull. If that happens, then restructuring is called recursively on the parent until we restructure the root.

6.3.2 B^+-tree in SQLite

A tree is created by allocating its root page. The root page is not relocated. Each tree is identified by its root page number. The number is stored in the master catalog table, whose root always

resides on Page 1.

SQLite stores tree nodes (both internal and leaf) in separate pages—one node in one page (see Fig. 6.4). The pages are accordingly referred to as internal and leaf pages, respectively. Actual data entries partly reside on the leaf pages, and party on overflow pages. The tree module may not understand internal structures of data or key. It stores data and key in raw byte strings and their sizes. For each node, the key and data for any entry are combined together to form a *payload*. A fixed preset amount of payload is stored directly on the page. The module tries to put as much of the payload as it can on a single leaf page. If a payload is larger than the amount, the module spills the surplus bytes into one or more overflow pages: excess payload is sequentially stored in a singly linked list of overflow pages. Internal pages, though not shown in the figure, can have overflow pages, too.

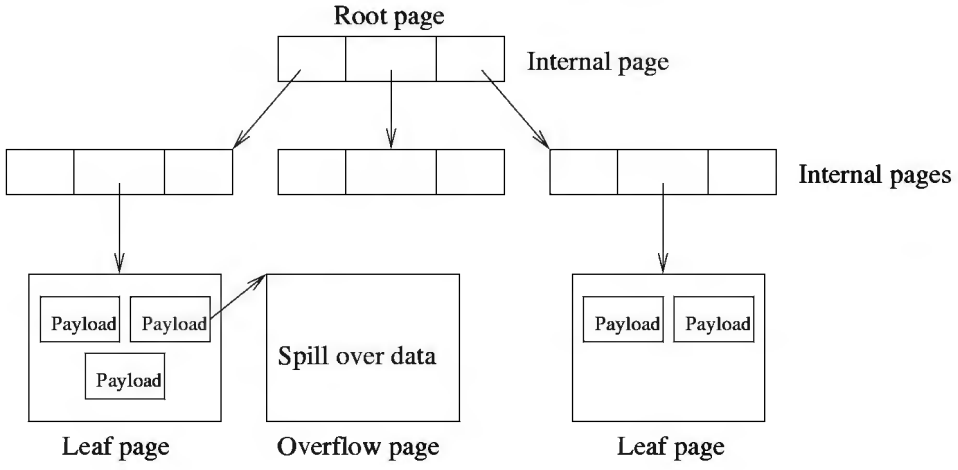


Figure 6.4: A typical B^+-tree.

6.4 Page Structure

You may recall that each database file is divided into fixed size pages. The tree module manages all those pages. Each page is either a tree page (internal, leaf, or overflow page), or a free page. (Exceptions are the lock-byte and pointer-map pages.) Figure 3.5 on page 89 depicts how free pages are organized into a single trunk list. In this section, I present structures of internal, leaf, and overflow pages.

Any database page can be used for any purpose, except that the first page (Page 1) of the database file is always a B^+-tree internal page holding a root node. But, the first 100 bytes of the first page contain a special header (called the "file header") that describes properties of the file. Figure 3.4 on page 86 presents the structure of the file header. The remaining space on the page is used for a B^+-tree root node. Every other database page is consumed wholly to store a single tree

node or overflow content.

6.4.1 Tree page structure

The logical content of each internal/leaf page is partitioned into what are called *cells*. For example, for a B^+-tree internal node, a cell consists of a key value and the child pointer preceding the key; for a leaf node, a cell contains (a part of) a payload and no child pointer. I discuss cells in Section 6.4.1.3 on page 163. Cells are units of space allocation and deallocation on tree pages. For the purpose of space management on each internal or leaf page, it is divided into four sections as shown in Fig. 6.5:

1. The page header,
2. The cell content area,
3. The cell pointer array, and
4. Unallocated space.

(You may recall that the Page 1 also has a 100-byte file header that resides before the page header.) The cell pointer array and the cell content area grow toward each other (via the middle unallocated space) just like two stacks are placed facing one another. The cell pointer array acts as the page-directory that helps in mapping logical cell order to their physical cell storage in the cell content area.

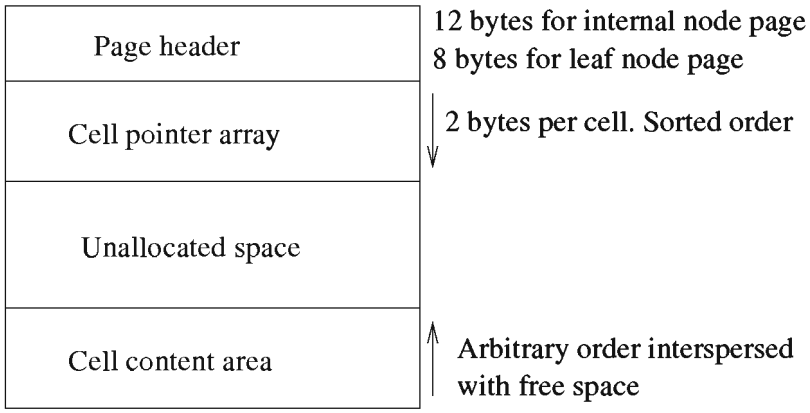


Figure 6.5: Structure of a tree page.

6.4.1.1 Structure of page header

A page header contains the management information only for that page. The header is always stored at the beginning of page (low address). (Page 1 is an exception: the first 100 bytes contain

the file header.) The structure of the page header is given in the table in Fig. 6.6. The first two columns are in bytes; multiple-byte integers are stored in the big-endian format. The flags at offset 0 define the format of the page (aka, the type of the tree page). There are four possibilities: (1) a table B^+-tree internal page, (2) a table B^+-tree leaf page, (3) an index B-tree internal page, and (4) an index B-tree leaf page. For internal pages, the header also contains the rightmost child pointer at offset 8. The other components of the header are explained in the next sub-subsection.

Offset	Size	Description
0	1	Flags. 2: internal index-tree page, 5: internal table-tree page, 10: leaf index-tree page, 13: leaf table-tree page
1	2	Byte offset to the first free block
3	2	Number of cells on this page
5	2	Offset to the first byte of the cell content area
7	1	Number of fragmented free bytes
8	4	Right child (the Ptr(n) value). **Omitted on leaves**

Figure 6.6: Structure of tree page header.

6.4.1.2 Structure of storage area

Cells are stored at the very end of the page (high address), and they grow toward the beginning of the page. The cell pointer array begins on the first byte after the page header, and it contains zero or more cell pointers. See Fig. 6.7. The number of elements in the array is stored in the page header at offset 3 (see Fig. 6.6). Each cell pointer is a 2-byte integer number indicating an offset (from the beginning of the page) to the actual cell within the cell content area. The cell pointers are stored in sorted order (by the corresponding key values), even though cells may be stored unordered. The left entries have smaller key values than the right entries. Cells are not necessarily contiguous or in order. SQLite strives to keep free space after the last cell pointer so that new cells can be easily added without having to defragment the page.

Because of random inserts and deletes of cells on a page, the page may have cells and free space interspersed (inside the cell content area). The unused space within the cell content area is collected into a singly linked list of free blocks. The blocks on the list are arranged in ascending order of their addresses. The head pointer (a 2-byte offset) to the list originates in the page header at offset 1 (see Fig. 6.6). Each free block is at least 4 bytes in size. The first 4 bytes on each free block store control information: the first 2 bytes for the pointer to the next free block (a zero value indicates no next free block), and the other 2 bytes for the size of this free block (including the header). Because a free block must be at least 4 bytes in size, any group of 3 or fewer unused bytes (called a *fragment*) in the cell content area cannot exist on the free block chain. The cumulative size of all fragments is recorded in the page header at offset 7 (see Fig. 6.6). (The size can be at

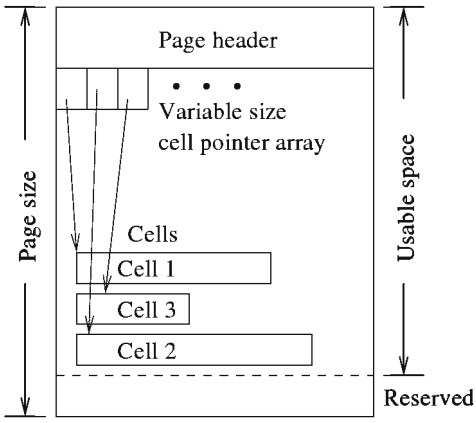


Figure 6.7: Location of cell pointer array in a page.

most 255. Before it reaches the maximum value, SQLite defragments the page.) The first byte of the cell content area is recorded in the page header at offset 5 (see Fig. 6.6): If the value is 0, it is treated as offset 65,536. The value acts as the border between the cell content area and the unallocated space.

6.4.1.3 Structure of a cell

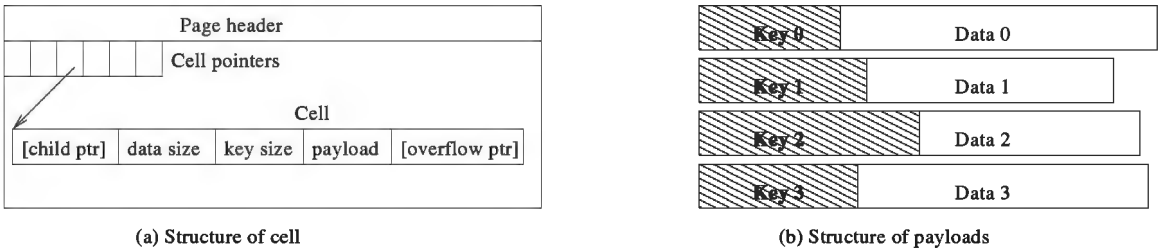
Cells are variable length byte strings. A cell stores (a part of) a single payload, and it is a composition of a key and an optional data. The structure of a cell is given in the table in Fig. 6.8. The size column is in bytes. The var entry int he size column is a variable length integer, represented in 1 to 9 bytes; it will be referred to as a *variant.*

Size	Description
4	Page number of the left child. **Omitted on leaf page**
var(1–9)	Number of bytes of data. **Omitted on index-tree page or internal table-tree page**
var(1–9)	Number of bytes of key. **Or the key itself for table-tree page**
*	Payload
4	First page of the overflow chain. **Omitted if no overflow**

Figure 6.8: Structure of a cell.

For internal pages, each cell contains a 4-byte child pointer; for leaf pages, cells do not have child pointers. Next come the number of bytes in the data image and the number of bytes in the key image. (The data and key images do not exist on internal table-tree pages. The key length bytes instead store the integer key value. The data image is nonexistent for the index-tree.) Thereby, (1) a cell on a table tree's internal node has a 4-byte left child page number and a variant for the rowid value; (2) a cell on a table tree's leaf node has a variant for the total length of a table row

record, a variant for the rowid value, a part of the row record, and a 4-byte overflow page number; (3) a cell on an index tree's internal node has a 4-byte left child page number, a variant for the total length of the key value, a part of the key, and a 4-byte overflow page number; (4) a cell on an index tree's leaf node a variant for the total length of the key value, a part of the key, and a 4-byte overflow page number. Figure 6.9 depicts the cell format: part(a) structure of a cell, and part(b) structure of payloads. The key or the data or both may be absent in a payload.



(a) Structure of cell

(b) Structure of payloads

Figure 6.9: Cell organization.

Variant Integer Number: As depicted in Fig. 6.8, SQLite uses variable length integers to represent integer sizes (and integer key values). A variable length integer is 1–9 bytes long, where the lower 7 bits of each byte are used for the integer value calculation. The integer consists of all consecutive bytes that have bit 8 set and the first byte with bit 8 clear. The most significant byte of the integer appears first. A variable-length integer is at most 9 bytes long. As a special case, all 8 bits of the 9th byte are used in the value determination. This representation allows SQLite to encode a 64-bit integer in at most 9 bytes. It is called the Huffman code, and it was invented in 1952. SQLite prefers the variable length Huffman encoding over fixed length 64-bit encoding because only one or two bytes are required to store the most common cases, but up to 64-bits of information can be encoded (in 9 bytes) if required. ◁

You may recall that SQLite puts a restriction in storing payloads on a tree page. A payload might not be stored in its entirety on a page even if there is sufficient free space on the page. The maximum embedded payload fraction is the amount of the total usable space in a page that can be consumed by a single cell of an internal page. This value is available in the database file header at offset 21 (see Fig. 3.4 on page 86). If the payload of a cell on an internal page is larger than the maximum allowed size, SQLite spills the extra payload into a chain of overflow pages. Once an overflow page is allocated, as many bytes as possible are moved into the overflow pages without letting the cell size drop below the min embedded payload fraction value (in the database file header at offset 22). The min leaf payload fraction (in the file header at offset 23) is like the min embedded payload fraction, except that it applies to leaf pages. The maximum payload fraction for a leaf page is always 100 percent, and it is not specified in the header.

Overflow Calculation: Let p be payload size and u be usable area on pages.

Leaf table-tree page: if $(p < (u - 36))$, then put the entire payload on the leaf; otherwise, let $M = (((u - 12) * 32/255) - 23)$; store $\min\{u - 35, (M + (p - M)\%(u - 4))\}$ bytes on the page and rest on overflow pages.
Internal table-tree page: has no payload and no overflow;
Leaf/internal index-tree page: if $(p < (((u - 12) * 64/255) - 22))$, then put the entire payload on the page; otherwise, let $M = (((u - 12) * 32/255) - 23)$; store $\min\{(M + (p - M)\%(u - 4)), (((u - 12) * 64/255) - 23)\}$ bytes on the page and rest on overflow pages. $\triangleleft$

6.4.2 Overflow page structure

Multiple small entries can fit on a single tree page, but a large entry can span multiple overflow pages. Overflow pages (for a payload) form a singly linked list. Each overflow page except the last one is completely filled with data of length equal to usable space minus four bytes: the first four bytes store the next overflow page number. The last overflow page can have as little as one byte of data. (The remaining space on the page is internal fragmentation.) An overflow page never stores content from two payloads.

6.5 The Tree Module Functionalities

The tree module helps the VM to organize all tables and indexes into B- and B^+-trees: one B^+-tree for each table, and one B-tree for each index. Each tree consists of one or more database pages. The VM can store and retrieve variable length records in any tree. It can delete records from a tree any time. The tree module does self-balancing on insert and delete, and it performs automatic reclamation and reuse of free space. For a tree with m tuples, the module provides the VM with $O(\log m)$ time-bound lookup, insert, and delete of records, and O(1) amortized bidirectional traversal of records.

6.5.1 Control data structures

The tree module is a passive entity in the sense that it does not interpret records (keys and/or tuple images) that are stored in B- and B^+-trees. The VM is the sole interpreter of records. The tree module accesses each database page via the pager module. It creates many in-memory control objects of four data structures (`Btree`, `BtShared`, `MemPage`, and `BtCursor`) to manage in-memory copies of database pages. These four data structures are discussed in the next four sub-subsections.

6.5.1.1 `Btree` structure

When the VM opens a database file by calling the `sqlite3BtreeOpen` function, the function creates an object of `Btree` type to apply further operations on the file at the tree module layer. The VM

uses the object as a handle to manipulate the file. Everything the VM needs to know about the file is summarized in the object. The object is synonymous to the database connection for the VM. The object has the following member variables: (1) db: a pointer to the library connection holding this Btree object, (2) pBt: a pointer to one `BtShared` object via which the tree module accesses pages of the database file; this object holds a Pager object (see Section 6.5.1.2); (3) `inTrans`: indicates whether or not a transaction is in progress currently on the database file via this database connection; and (4) many other control variables. The value of `inTrans` determines the state of the `Btree` object; it can be in one of the following three states: TRANS_NONE, TRANS_READ, and TRANS_WRITE, indicating what type of transaction is currently in progress on the database file via the `Btree` object.

6.5.1.2 `BtShared` structure

At the tree module layer, an instance of `BtShared` object represents the state of a single database file. The object has the following member variables: (1) pPager: it points to a Pager object that manages this database and journal files; (2) `pCursor`: a list of open cursors on trees of the database; (3) `pageSize`: it indicates the total number of bytes on each page; (4) nTransaction: number of open (read and write) transactions; (5) inTransaction: the transactional state; (6) pSchema: pointer to the schema cache (of schema objects); (7) db: a pointer to the library connection that is currently using this object; (8) pPage1: a pointer to the in-memory copy of MemPage object for database Page 1; (9) mutex: access synchronizer; and (10) many other control variables. When the shared caching feature is disabled, each `BtShared` object is owned by a single `Btree` object. When the feature is enabled, a `BtShared` object can be owned by multiple `Btree` objects. I discuss shared page caching in Section 10.13 on page 241.

6.5.1.3 `MemPage` structure

As shown in Fig. 5.8 on page 133, for each page image in the page-cache, the pager allocates an extra amount of space right below the page. This space is used by the tree module to store page specific control information there. The pager initializes the space to zeros when it brings the page into the cache. The tree module reinitializes the space according to its need. This space holds an object of type `MemPage`. Figure 6.10 presents some key member variables of `MemPage`. The variables are self explanatory. A `MemPage` object stores information about the page that is decoded from the raw file page content. The `pParent` variable points back to the in-cache copy of the parent page. This variable allows us to traverse the tree from any node up to the root of the tree. The `aData` points to the beginning of the page copy in the cache.

Member	Description
pBt	Pointer to `BtShared` object to which this page belongs to.
pDbPage	Pointer to PgHdr that holds this page.
pgno	Page number for this page.
aData	Pointer back to the start of the in-cache page image.
intKey	True if intkey flag is set (for table B^+-tree).
leaf	True if leaf flag is set.
hasData	True if this page stores data.
aOvfl	Cells that will not fit on aData.
nOverflow	Number of overflow cell bodies in the cell array.
nCell	Number of cells on this page, local and overflow.
nFree	Number of free bytes on the page.
...	Other variables.

Figure 6.10: Structure of `MemPage` object.

6.5.1.4 `BtCursor` structure

To operate on a specific tree in the database, the VM has to 'open' the tree first by a way of creating a cursor on the tree (by invoking the `sqlite3BtreeCursor` function). Cursor (many authors call it a scan) is an abstraction of applying operations on trees. A cursor acts as a logical pointer to a particular entry in a tree. For each open tree, the tree module creates an object of `BtCursor` type that is used as a handle to read, insert, or delete tuples from the tree. A cursor here is a kind of a representation of a single SQL statement execution on a single tree. A BtCursor object cannot be shared by multiple database connections (i.e., Btree objects).

Figure 6.11 presents some key member variables of `BtCursor`. The variables are self explanatory. The `apPage` is an array of `MemPage` objects. These objects contain all pages from the root down to the current page; the entry the cursor is currently pointing to. The `aiIdx[]` contains the corresponding indexes of the cell pointer arrays on those pages. The `eState` is the state of the cursor: valid (points to a valid entry), invalid (does not point to a valid entry), requireseek (tree got modified by someone else), or fault (some error such as no memory occurred).

The VM can have many cursors open on the same tree. A cursor can be either a read-cursor or a write-cursor, but not both. We can only read cells via read-cursors, but can both read and write via write-cursors. (The output of a read cursor are tuples ordered by the natural tree order.) Read- and write-cursors may not coexist on the same tree. (Read- and write-transactions cannot concurrently read and write the same tree via two different cursors.) Thereby, read-cursors are guaranteed to have the repeatable read property. A read-cursor is a kind of 'read lock' on the tree. (This is irrespective of whether or not the database connection has an exclusive lock on the database file.) This allows read-cursors to do sequential scans of the tree without having to worry about entries being inserted into or deleted from the tree during the scan. The VM opens read-

Member	Description
pBt	Pointer to the `BtShared` that owns this cursor.
pBtree	Pointer to `Btree` object to which this cursor belongs to.
pgnoRoot	The root page of the tree the cursor represents.
apPage[]	Pages from the tree root down to the current page.
aiIdx[]	current indexes in apPage array.
iPage	Index of the current page in apPage[].
wrFlag	True if we can write through this cursor.
eState	The current state of the cursor.
...	Other variables.

Figure 6.11: Structure of `BtCursor` object.

cursors only when it intends to perform a range search query on the tree. All other cursors are write-cursors that can both read and write the tree.

You may note that a tree can have its own key comparison function. The comparison function must be logically the same for every cursor on the same tree. The default comparison function is the integer comparison for table B^+-trees and the native `memcmp` API for index B-trees.

6.5.1.5 Integrated control structures

The interlinks between the above four data structures and those defined in Chapter 5 are shown in Fig. 6.12. A `Btree` object and its associated control structures summarize the current state of one database file as seen by the VM via this Btree object. (Many Btree objects can share the same `BtShared` object. I discuss this in Section 10.13 on page 241.)

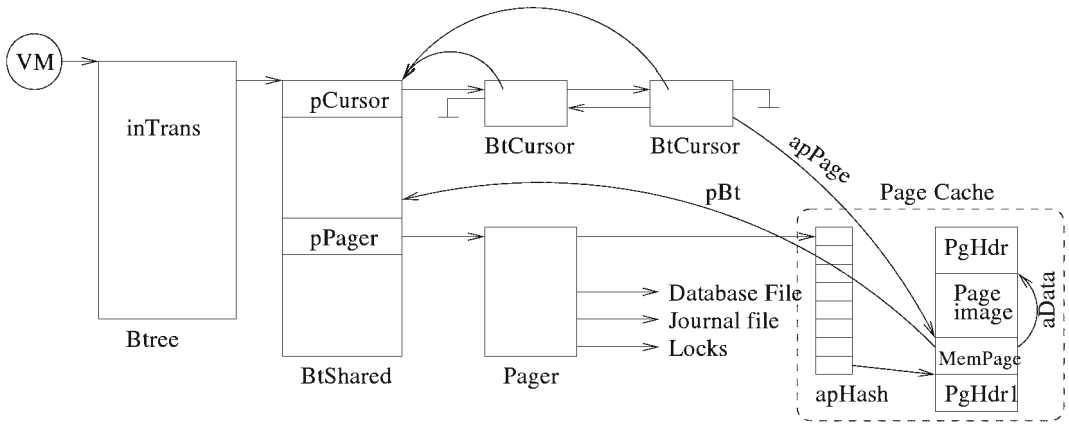


Figure 6.12: Integration of control structures.

6.5.2 Space management

The tree module receives requests for payload insertions and deletions at random from the VM module. An insert operation requires allocation of space in the tree (and overflow) pages. A delete operation frees up occupied space from tree (and overflow) pages. The management of free space on each page is very crucial for effective utilization of space allocated to the database. These are discussed in the rest of this subsection.

6.5.2.1 Management of free pages

When a page is removed from a tree, the page is added to the file freelist for later reuse. (The freelist originates in the file header at offset 32; see Fig. 3.4 on page 86.) When a tree needs expansion, a page is taken off the freelist and added to the tree. If the freelist is empty, a page is taken from the native file system. (Pages taken from the native file system are always appended at the end of the database file.)

You can purge the file freelist by executing the `vacuum` command. The command shrinks the database file appropriately by compacting the file. Databases created in the 'autovacuum' mode will automatically shrink the database at each COMMIT instead of keeping the free pages (if any) with the database itself. I discuss autovacuum feature in Section 10.8 on page 235.

6.5.2.2 Management of page space

There are three partitions of free space on a tree page:

1. The space between the cell pointer array and the top of the cell content area. The top value is stored in the page header at offset 5 (see Fig. 6.6 on page 162).
2. The free blocks inside the cell content area. The blocks are linked together, and the link head pointer is stored in the page header at offset 1. The blocks are ordered by increasing their addresses.
3. Scattered fragments in the cell content area. The total fragmented space amount is stored in the page header at offset 7.

The space allocation is done from the former two partitions. At each allocation or deallocation, the corresponding affected partition is updated accordingly. One responsibility of the space manager is to make sure that cell pointer array and cell content area do not overlap. Allocation and deallocation steps are discussed below.

Cell allocation

Space allocator does not allocate space less than 4 bytes; such requests are rounded up to 4 bytes. Suppose a new request for *nRequired*, $nRequired \geq 4$, bytes comes on a page when the page has a total of *nFree* bytes. If $nRequired > nFree$, the request fails. Suppose $nRequired \leq nFree$. The allocator performs the following steps to satisfy the request:

1. It traverses down the free block list to see whether there is a large enough block to satisfy the request. This is a first fit search. If a suitable block is found, it does one of the following:
   (a) If the block size is less than $nRequired + 4$, it removes the block from the free block list, satisfies the request from the beginning of the block, and puts the remaining space (≤ 3 bytes) in the fragment partition.
   (b) Otherwise, it satisfies the request from the bottom part of the block and reduces the size of the block by *nRequired* bytes.
2. Otherwise, there is no sufficiently large free block to satisfy the request. If there is not much space in the middle unallocated partition or there are too many fragments, the space allocator defragments the page first. It runs a compaction algorithm on the page to consolidate the entire free space in the middle. During the compaction, it transfers the existing cells, one after another, to the bottom of the page.
3. It allocates *nRequired* bytes from the bottom of the free space area, and increases the top value by *nRequired* bytes.

Cell deallocation

Suppose a request comes to release *nFree* (≥ 4) bytes that was previously allocated by the allocator. The allocator creates a new free block of size *nFree* bytes, and inserts the block in the free block list at appropriate position. Then, it tries to merge free blocks in the neighborhood of the released one. If there is a fragment in between two adjacent free blocks, it also merges the fragment with the blocks. If there is a free block right at the top pointer, it merges the block with the space in the middle unallocated partition and increases the value of top.

Summary

The tree module obtains services from the pager module, and implements an abstraction of tuple oriented file system that is used by the VM module. Each table or index is considered a sorted set of tuples. These tuples are organized into a B^+-tree or B-tree, respectively. Each active database page belongs to a tree as an internal, leaf, or overflow node. (The exceptions are the lock-byte and pointer-map pages.) No page stores information from more than one tree.

The tree module implements functions for creation and deletion of trees, and read, insertion, and deletion of individual tuples in trees. This module does not alter the internal structure of tuples, and it treats them as binary string.

The root page information of all trees are stored in the sqlite_master (or sqlite_temp_master) catalog. It is a B^+-tree, just like another table tree. It is stored on the first database page after leaving the first 100 bytes for the file header.

The B^+-tree algorithm used in SQLite is the one described in Knuth's famous book The Art of Computer Programming, Volume 3: "Sorting And Searching". The internal nodes store search navigational information and leaf nodes tuples. There is an upper and a lower bound on the number of entries an internal node can have. This chapter gives an overview of how point lookup, search next, insert, and delete operations are performed on B^+-trees.

Chapter 7

The Virtual Machine Module

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- the five storage datatypes and their representations
- internal representation of a compiled SQL statement
- how data are converted between SQL types and C types
- the logical structures of table and index records
- the advantages of manifest typing in encoding data values

Chapter synopsis

This chapter discusses about how SQLite virtual machine (VM) interprets application programs written in the internal bytecode programming language. The VM is the ultimate manipulator of data stored in database files. It supports five primitive datatypes, namely, NULL, integer, real, character string, and blob to represent primitive data items in files as well as in-memory. Translation between user data and internal representation is exclusively done by the VM. It also converts data from one type into another as required during expression evaluations. This chapter discusses how the VM works, and how it formats index and table records before storing them in B- and B^+-trees.

7.1 Virtual Machine

The topmost module of the backend is popularly called the *virtual database engine*, or the *virtual machine* (VM) in SQLite terminology. The VM is the heart of SQLite, and is the interface between the frontend and the backend. Core information (arithmetic and logic) processing happens only at

this module because the lower down modules are passive as far as stored information is concerned. The VM module implements an abstraction of a new machine on the top of the tuple oriented file system, and it executes application programs written in the SQLite's internal bytecode programming language. This programming language is specifically designed to search, read, and modify tuples in trees. The VM accepts bytecode programs (generated by the frontend), and executes the programs. (You may recall that bytecode programs are prepared statements in SQLite.) The VM uses the infrastructures provided by the tree module to execute bytecode programs and to produce outputs of program executions.

The SQLite development team believes that the use of VM in SQLite has been a great benefit to the library's development and debugging. The VM provides a well-defined glue between the frontend (the part that parses SQL statements and generates VM applications) and the backend (the part that executes bytecode applications and computes results). Bytecode programs are easily readable than interpreting complex mesh of integrated data objects. The VM also helps the SQLite engine developers to see clearly what the engine is trying to do with each SQL statement it compiles.

Bytecode Execution Tracing: To mortal human, it is easier to read bytecode programs than interpret data structure values. Depending on how the SQLite source code is compiled, it also has the capability of tracing the program execution of each VM application—printing each bytecode instruction and the results as the execution evolves. ◁

A bytecode application program is represented by an in-memory object of type `sqlite3_stmt` (internally called `Vdbe`). The following SQLite API functions can be applied on the object to associate input values to SQL parameters, to execute the bytecode program, and to retrieve output produced by the program. Details about these functions can be found in Section 2.2.2 on page 51.

1. `sqlite3_bind_*`: They assign values to SQL parameters that become input to the bytecode program.

2. `sqlite3_step`: It advances the bytecode program execution to the next breakpoint, or to the halting point.

3. `sqlite3_reset`: It rewinds the program execution back to the beginning and makes the same program ready for a new execution with the same bounded parameter values.

4. `sqlite3_column_*`: They extract results, column by column, from the current output row produced by the program.

5. `sqlite3_finalize`: It destroys the `sqlite3_stmt` object along with the bytecode program.

The internal state of a `Vdbe` object (aka, prepared statement) includes the following:

- a bytecode program,
- names and datatypes for all result columns,
- values bound to input parameters,
- a program counter,
- an arbitrary amount of "numbered" memory cells (referred to as register locations),
- other run-time state information (such as open Btree objects, sorters, lists, sets).

The VM does not perform any query optimization work. It blindly executes bytecode programs. In doing so, it converts data from one format into another on demand. On-the-fly data conversion is the primary task of the VM; everything else is controlled by the bytecode programs it executes. A major part of this chapter presents data conversion and manipulation tasks. Before doing so, I present an overview of bytecode programming language, bytecode program, and program execution logic in the next section.

7.2 Bytecode Programming Language

SQLite defines an internal programming language to prepare bytecode programs. The language is akin to the assembly language used by physical as well as virtual machines such as Java: it defines bytecode instructions. Each bytecode instruction does a small amount of information processing work or makes logical decisions. A bytecode program consists of one or more bytecode instructions, in a linear sequence of instructions. A bytecode instruction can have up to five operands, and is of the form $\langle opcode, P1, P2, P3, P4, P5 \rangle$, where *opcode* identifies a specific bytecode operation, and P1, P2, P3, P4, and P5 hold operands to the operation or register names that contain operands. The P1, P2, and P3 operands are a 32-bit signed integer. The P2 operand is always the address of the jump destination in any operation that might cause a jump. It is also used for other purposes. The P4 operand is a 32/64-bit signed integer or a 64-bit float or a pointer to a null terminated string, a blob, a collation comparison function, an SQL function, etc. The P5 operand is an unsigned character. Some opcodes use all five operands, some typically ignore one or two operands, and some ignore all five operands.

Note: Opcodes are internal VM operation names, and they are not a part of the SQLite interface specification. Consequently, they themselves or their operational semantics may change from one release to another. The SQLite development team does not encourage SQLite users to write bytecode programs on their own. The bytecode programming language is strictly for internal use. ◁

Figure 7.1 displays a typical bytecode program that is equivalent to this SQL query statement: `SELECT * FROM t1`. The table `t1` has two columns, namely `x` and `y`. (You may note that the top line on the figure is not a part of the program. Every other line is a bytecode instruction. There are 14 bytecode instructions in the example.) The bytecode program execution starts from instruction at address 0, and continues until a halt instruction is executed or the program execution control goes beyond the last instruction. The meanings of the opcodes are presented in the next subsection.

addr	opcode	P1	P2	P3	P4	P5
0	Trace	0	0	0		00
1	Goto	0	10	0		00
2	OpenRead	0	2	0	2	00
3	Rewind	0	8	0		00
4	Column	0	0	1		00
5	Column	0	1	2		00
6	ResultRow	1	2	0		00
7	Next	0	4	0		01
8	Close	0	0	0		00
9	Halt	0	0	0		00
10	Transaction	0	0	0		00
11	VerifyCookie	0	1	0		00
12	TableLock	0	2	0	t1	00
13	Goto	0	2	0		00

Figure 7.1: A typical bytecode program.

7.2.1 Bytecode instructions

There are currently about 142 opcodes. Opcodes are classified into five categories: (1) arithmetics and logic, (2) data movement, (3) control flow, (4) B- and B^+-tree related, and (5) specialized ones. Category 1 opcodes include add, subtract, multiply, divide, remainder, bitwise OR, bitwise AND, one's complement, two's complement, right shift, left shift, and string concatenation. Category 2 opcodes move values between memory cells. Category 3 opcodes include goto, gosub, return, halt, and conditional branch. Category 4 opcodes include (i) create, destroy, and clear B/B^+-trees, (ii) open and close cursors on B/B^+-trees, (iii) move forward and backward cursor or move to specific keys, (iv) branch on cursor movement, (v) insert, delete records from B/B^+-tree, (vi) begin, commit, and rollback transactions. Category 5 opcodes include (a) get a rowid that is currently not in use, (b) combine n elements from memory cells to form a record, (c) extract i-th column from a table row, etc.

Each bytecode instruction is internally represented by an object of type `VdbeOp`. It is a simple object that have the following members: (1) `opcode` indicating what operation to perform, (2) `p1`

holding the first operand, (3) `p2` holding the second operand, (4) `p3` holding the third operand, (5) `p4` holding the fourth operand, (6) `p5` holding the fifth operand, and (7) `p4type` indicating the type of `p4` operand. There are thirteen types of `p4` operands. A VM application is in fact a linear array of `VdbeOp` objects.

The semantics of some opcodes (those used in Fig. 7.1) are presented below. All opcodes of the latest SQLite release are documented on the SQLite webpage `http://www.sqlite.org/opcode.html`. In SQLite source code, each opcode name is prefixed with OP_, and is assigned a different integer literal.

1. `Trace`: This opcode checks whether or not the SQLite library has the tracing mode turned on. If on, the P4 content (an UTF8 string) is output on each trace callback. (You can enable tracing using the `sqlite3_trace` API function.)

2. `Goto`: An unconditional jump to the address specified by the P2 operand. The next instruction executed by the VM will be the one at offset P2 from the beginning of the program.

3. `OpenRead`: Open a read-only cursor on a B/B$^+$-tree whose root page is identified by the P2 operand. (If P5 $\neq 0$, then the content of register P2 has the root page number, and not the P2 value itself.) The database file is determined by the P3 operand—the value is 0 for the main database, 1 for the temp database, or greater than 1 for an attached database. The value of P1 (a non negative integer) operand becomes the identifier for the new cursor. The P4 value is either an integer or a pointer to a KeyInfo structure. The KeyInfo structure defines the content and collating sequence if the cursor is open on a B-tree (an SQL index). If P4 is an integer value, it defines the number of columns of the table.

   If the database is unlocked when the read cursor is opened, a shared lock is acquired as part of this instruction execution. If it cannot get a shared lock, the VM terminates the bytecode execution with the SQLITE_BUSY error code.

4. `Rewind`: Reset the cursor P1. The cursor will refer to the first entry in the table or index, i.e., the least entry in the corresponding tree. If the tree is empty and P2 > 0, then jump immediately to P2 address. Otherwise, fall through to the following instruction.

5. `Column`: Get the P2-th column from in the record the cursor P1 points to. (Interpret the data that cursor P1 points to as a structure that is built using the `MakeRecord` instruction. See the MakeRecord opcode below for additional information about the format of the data record.) If the record contains fewer than P2 values, then extract a NULL; if P4 is of type P4_MEM, then use the P4 value as the result. The returned value is stored in the P3 register.

6. `MakeRecord`: Convert P2 number of registers beginning with the register P1 into a single entity that is suitable for use as a data record in a database table or as a key in an index. That is, register P1 contains the first data item, P1+1 the second data item, and so forth.

7. `ResultRow`: The registers P1 through P1+P2−1 contain a single row of results. This instruction is executed in the process when the application executes the sqlite3_step function and the function execution returns with SQLITE_ROW.

8. `Next`: Advance cursor P1 so that it points to the next entry (key–value pair) on the tree. If there is no more entry then fall through to the following instruction. Otherwise, jump immediately to the P2 address.

9. `Close`: Close a cursor previously opened as P1. If cursor P1 is not currently open, this instruction is a no-op.

10. `Halt`: Exit immediately after closing all open cursors, FIFOs (aka, RowSet objects), etc. P1 is the result code that will be returned by `sqlite3_exec`, `sqlite3_reset`, or `sqlite3_finalize` API functions. For a normal halt, the return code is SQLITE_OK (= 0). For errors, it can be some other value. If P1 ! = 0, then the P2 value determines whether rollback the current transaction needs a rollback. Do not rollback if P2 = OE_Fail. Do the rollback if P2 = OE_Rollback. If P2 = OE_Abort, then back out all changes that have occurred during this execution, but do not rollback the transaction. If P4 is not null, then it is an error message string.

    There is an implied "Halt 0 0 0 0 0" instruction inserted at the very end of every bytecode program. So a jump past the last instruction of the program is the same as executing the Halt.

11. `Transaction`: Open a new transaction. P1 is the index of the database file on which the transaction is started. The value is 0 for the main database file, 1 for the temp database, or greater than 1 for an attached database. If P2 is zero, then a shared-lock is obtained on the database file. If P2 is non-zero, then a write-transaction is started, that is, a RESERVED lock is obtained on the database. If P2 is 2 or greater, then an EXCLUSIVE lock is also obtained on the file. Starting a write-transaction also creates a rollback journal.

12. `VerifyCookie`: Check the value of global database parameter number 0 (i.e., the schema version cookie) and make sure it is equal to the P2 value and the generation counter on the local schema parse is equal to P3 value. P1 is the database number which is 0 for the main database, 1 for the temp database, or some higher number for attached databases. (You may recall that the cookie value is changed whenever the database schema changes.)

This verify operation is used to detect when the cookie has changed and that the current process needs to reread the schema. Either a transaction needs to have been started or an OpenRead/OpenWrite needs to be executed (to establish at least a shared lock on the database) before this opcode is executed.

13. `TableLock`: Obtain a lock on the table whose root page is P2 on database P1. If P3 is 0, take a read lock; if P3 is 1, take a write lock. P4 contains a pointer to the table name.

In the next two subsections, I present VM's execution logic for SQL insert and join processing that would give you a cue how bytecode programs are generated for SQL statements.

7.2.2 Insert logic

Suppose you have a table T1 with two columns: c1 text and c2 integer; the table does not have any index. If you execute `insert into T1 values('Hello, World!', 2000)` statement, the VM performs the following algorithmic steps.

1. Open a write-transaction on the main database.
2. Check the database schema version to make sure that the schema has not been changed after the bytecode program has been generated.
3. Open a new write-cursor on table "T1" B^+-tree.
4. Create a new rowid and make a table record entry along with 'Hello, World!' and 2000 values.
5. Insert the record entry into the B^+-tree via the open cursor.
6. Close the cursor.
7. Return the execution status code to the caller.

If there are indexes on T1, the VM would open a write-cursor on each index at Step 3, and prepare and insert index records at Steps 4 and 5, respectively.

7.2.3 Join logic

In a join operation, two or more tables are combined to produce a single result table. The resulting table consists of every possible combination of rows from the tables being joined. The easiest and most natural way to implement the join is with nested loops. SQLite does loop-joins only, not merge-joins. The left-most table in the FROM clause forms the outer loop. The right-most table forms the inner loop.

Consider the following SQL select statement: `select * from t1, t2 where some-condition`. Suppose there are no indexes on the two tables. The pseudo code for the select statement processing is something like the following.

1. Open a read-transaction on the main database.
2. Check the database schema version to make sure that the schema has not been changed after the bytecode program has been generated.
3. Open two read cursors, one for T1 table and the other for T2 table.
4. For each record in T1, do:
   For each record in T2, do:
   If the WHERE's some-condition evaluates to TRUE,
   Compute all columns for the current row of the result.
   Invoke the default callback function for the current row of the result.
5. Close both cursors.

7.2.4 Program execution

The VM begins an execution of a bytecode program starting at instruction number 0. The execution continues until (1) an explicit Halt instruction is processed, or (2) the program counter refers to past the last instruction (aka, an implicit Halt instruction), or (3) there is an execution error. When the VM halts, all memory that has been allocated to it is released and all cursors it opened are closed. If the execution stopped due to an error, the pending transaction is terminated and changes made to the database are rolled back.

The structure of the VM interpreter, in C code, is given in Fig. 7.2. The interpreter (the sqlite3VdbeExec function taking a Vdbe object pointer) is a simple for-loop containing a massive switch statement that contains a large number of cases. Each case statement implements one bytecode instruction. (In the source code, opcode names start with the `OP_` prefix. Numeric values to opcode names are not statically numbered; they are assigned at the time of SQLite source code compilation, see Section 2.8 on page 77. The numbers may vary from one SQLite release to another.) In each iteration, the VM fetches the next bytecode instruction from the program, i.e., from `aOp` array using `pc` (both are members of `Vdbe` object) as index into the array. It decodes and carries out the operation specified by the instruction. Normally program execution evolves from one bytecode instruction to the next one (pc++), but the pc can be changed by jump instructions. The for-loop continues until the VM processes a halt instruction or loop-condition fails (that is,

```
for (int pc = 0; pc < nOp && rc == SQLITE_OK; pc++){
    switch (aOp[pc].opcode){
        case OP_Add:
            /* Implementation of the ADD operation here */
            break;

        case OP_Goto:
            pc = op[pc].p2-1;
            break;

        case OP_Halt:
            pc = nOp;
            break;

        /* other cases for other opcodes */
    }
}
```

Figure 7.2: Structure of the VM interpreter.

pc becomes greater than or equal to nOp), and we say that the bytecode program has terminated. This is a normal termination.

The `sqlite3_stmt` structure pointer that is returned by the `sqlite3_prepare` API function is actually a pointer to an object of type `Vdbe`. (It stands for Virtual DataBase Engine.) The object contains the complete state of the VM. Some components of the object are shown in Fig. 7.3. The `aOp` array holds the opcodes. All memory required by an execution of this program resides on the `aMem` array whose size is `nMem`.

Member	Description
db	The library connection (sqlite3 object).
aOp	Space to hold the VM program.
nOp	Number of instructions at aOp.
apCsr	One element of this array for each open cursor.
nCursor	Number of slots in apCsr[].
aMem	The memory locations.
nMem	Number of memory locations currently allocated.
rc	Value to return.
pc	The program counter.
...	Other variables.

Figure 7.3: The `Vdbe` structure.

The VM accesses a database using cursors. (These are different from `BtCursors` used at the tree module.) It can have zero or more open cursors on the database. Each cursor is a pointer into a single table or index tree in the database. The cursor can seek to an entry with a particular key, or loop over all entries of the tree. The VM inserts new entries, retrieves the key/value from the

current entry on the cursor, or deletes the entry.

There can be multiple cursors pointing at the same index or table. All cursors operate independently, even if they point to the same index or table. Every cursor that the VM has open is represented by a `VdbeCursor` object. The structure of `VdbeCursor` is shown in Fig. 7.4. Instructions in the bytecode program can create a new cursor (OP_OpenRead or OP_OpenWrite), read data from the cursor (OP_Column), advance the cursor to the next entry in the table or index (OP_Next), and many other operations. All cursors are automatically closed when the VM terminates the bytecode program execution.

Member	Description
pCursor	The cursor structure (BtCursor) of the backend.
iDb	The database where the cursor is.
pBt	Separate file holding a temporary table for this cursor.
pKeyInfo	Info about index keys needed by index cursors.
aType	Type values for all entries in the record.
aOffset	Cached offsets to the start of each columns data.
aRow	Data for the current row, if all on one page.
...	Other variables.

Figure 7.4: The `VdbeCursor` structure.

7.3 Internal Datatypes

The VM uses an arbitrary amount of numbered memory locations to hold all intermediate results. Each memory location holds a single data value. Each data value processed by the VM has one of the following five datatypes:

1. INTEGER: a signed integer number;
2. REAL: a signed floating point number;
3. TEXT: a character string value;
4. BLOB: a byte image;
5. NULL: an SQL NULL value.

These are the only five primitive datatypes the VM supports. The type determines how a value will be represented physically. Every data value stored in the database file or in-memory must be one of these types. You may note that some values may have more than one representation at a time. For example, 123 can be an integer number, a floating point number, and a string all at once.

BLOB and NULL values cannot have another representation. An implicit conversion from one type to another occurs as necessary. You may use the built-in SQL function `typeof` to determine the type of a value—'`select a, typeof(a) from t1`' returns the values of column `a` along with their storage types. The API function `sqlite3_column_type` returns the storage type of a column value when the `sqlite3_step` function returns a row.

Internally, the VM manipulates nearly all values as `Mem` object (see Fig. 7.5). Each `Mem` object may cache multiple representations (string, integer, etc.) of the same value. A value (and therefore the corresponding `Mem` object) has the following properties: Each value has precisely one of the above mentioned five storage types. (Each `Vdbe.aMem` array element is a `Mem` object.)

Member	Description
type	The type of the value represented by this `Mem` object.
i	Integer value.
r	Real value.
z	String or BLOB value.
n	Size at z.
enc	UTF type in case z is a character string.
···	Other variables.

Figure 7.5: The `Mem` structure.

7.4 Record Format

The VM composes data values into records, and stores them in B- and B^+-trees. Each record consists of a key and optional value. The VM is solely responsible for maintaining internal structures of keys and values. (Although the tree module may split a single record across leaf or internal and multiple overflow pages, the VM sees the record as a logically contiguous byte string.) The VM uses two different but very similar record formats for table and index records.

There are two ways to format value/key records: fixed length and variable length. For fixed length format, the same amount of space is used for all records (of a table or index); the size of each individual column is known at table/index creation time. In variable-length formatting, the space for an individual column may vary from one record to another. SQLite uses a variant of variable-length record formatting because this has several advantages. It results in smaller database files because there is no wasted space due to padding. It also makes the system run faster, as there are fewer bytes to move between the main memory and the external storage devices such as the disk. In addition, the use of variable-length records allows SQLite to employ manifest typing instead of static typing. I discuss the manifest type in the next two subsections before I discuss record formats in the following subsections.

7.4.1 Manifest type

Each primitive data value, whether stored in a database file or in-memory, has a datatype associated with it. This is called the *storage type* of the data value. Most SQL databases use static typing: A datatype is associated with each column in a table, and only values of that particular datatype are allowed to be stored in that column. This is very rigid and has its own advantages and disadvantages. SQLite relaxes this restriction by using *manifest typing* [8].[1] In manifest typing, the datatype is a property of the value itself, not of the column or variable in which the value is stored. This enables you storing any data value in any variable or column, and the actual type of the value is not lost. SQLite uses manifest typing, where the storage type information is stored as a part of data value. It allows you to store any value of any datatype into any column regardless of the declared SQL type of that column. (There is an exception to this rule: an `integer primary key` column stores only integers in the range -2^{63} to $2^{63}-1$.) Although this feature is controversial and disputed, the SQLite development team is proud of it; they feel very strongly about the feature.

7.4.2 Type encoding

Storage types are encoded as integers. The integer value encodings of the types are given in the table of Fig. 7.6. The beauty of this encoding is that the data length becomes a part of the type encoding. The NULL type represents the SQL NULL value. For the INTEGER type, the data value is a signed integer number (in 2's complement), and is stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value. For the REAL type, the data value is a floating point number that is stored in 8 bytes, as specified in the IEEE floating point number representation standards. The type encoding values 8 and 9 represent integer constants 0 and 1, respectively. For the TEXT type, the data value is a text string, stored using the default encoding (UTF-8, UTF-16BE, or UTF-16LE) format of the database file; for the latter two, the native byte ordering is either big-endian or little-endian, respectively. (Each database file stores text data in only one UTF format.) For the BLOB type, the data value is a blob, stored exactly as it was input.

Boolean Values: Boolean value true has datatype 9 (Integer 1) and false datatype 8 (Integer 0). ◁

7.4.3 Table record format

The format of table records (aka, row data) is depicted in Fig. 7.7. It has two parts: header and record image. The header starts with a size field, followed by type fields. The header is followed by individual data items of the record. (SQLite does not alter the order of columns declared in create statements. It is advisable that schema designers put small and frequently used columns early in the record to minimize the need to follow the overflow chain.)

[1] Other authors may use some other terminologies instead of manifest typing. Be aware about the name confusion.

Type value	Meaning	Length of data
0	NULL	0
$N \in \{1..4\}$	Signed integer	N
5	Signed integer	6
6	Signed integer	8
7	IEEE float	8
8	Integer constant 0	0
9	Integer constant 1	0
10, 11	Reserved for expansion	N/A
$N >= 12$ and even	Text	$(N - 12)/2$
$N >= 13$ and odd	Blob	$(N - 13)/2$

Figure 7.6: Storage types and their meanings.

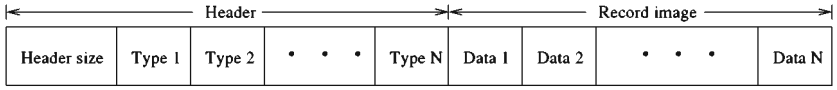


Figure 7.7: Format of table record.

The header size is the number of bytes before the Data 1 field. The size is represented as a variable-length 64-bit integer in Huffman code, and it includes the number of bytes occupied by itself. The size effectively acts as a pointer to the Data 1 item. After the header size field comes the datatype fields, one for each data value in the record in the sequence of their appearance in the record. Each Type i field is a variable-length unsigned integer (the maximum value is 2^{64}) that encodes the storage type for Data i.

Zero Length Data: For type values 0, 8, 9, 12, and 13, the length of data is zero, and hence, the data is not stored in the record. ◁

7.4.4 Table key format

In SQLite, every B^+-tree must have a unique key. Although a relation (table), by definition, does not contain identical rows, in practice, users may store duplicate rows in the relation. The database system must have means to treat them as different, distinct rows. The system must be capable of associating additional information for the differentiation purpose. It means that the system provides a new (unique) attribute for the relation. Thus, internally, each table has a unique primary key, and the key is either defined by the creator of the table or by SQLite. The primary key is an integer called rowid.

7.4.4.1 Rowid column

For every SQL table, SQLite designates one column as the *rowid* (also called by `oid` and `_rowid_`) whose value uniquely identifies each row in the table. It is the implicit primary key for the table, the

unique search key for the table B^+-tree. If any column of a table is declared as **integer primary key**, then SQLite teats that column itself (aka, aliased) as the rowid for the table. Otherwise, SQLite creates a separate unique integer column for rowid, whose name is **rowid** itself. (If the table has columns with the same three names, aka, rowid, oid, or _rowid_, the names will refer to those columns and not the internal rowid column.) Thus, every table, with or without the declaration of an **integer primary key** column, has a unique integer key, namely **rowid**. For this latter case, the rowid itself is internally treated as the integer primary key for the table. In either case, a rowid is a 64-bit signed integer value in the range $-2^{63} \ldots 2^{63} - 1$. (There is a compile time flag to restrict rowids to 32-bit values.) The sort order on table B^+-trees is the natural numerical order of integers (as mathematical objects), and we cannot have any other sort order. These B^+-trees are the primary indexes on tables. The rowids are stored in secondary indexes, if there are any, as soft-pointers back to table B^+-trees (see Section 7.4.5).

Figure 7.8 displays the contents of a typical SQL table, where the table is created by SQL statement: **create table t1(x, y)**. The rowid column is added by SQLite. The rowid value is normally determined by SQLite. Nevertheless, you can insert any integer value for the rowid column, as in **insert into t1(rowid, x, y) values(100, 'hello', 'world')**. The rows are stored in the order by their rowid values.

rowid	x	y
-5	abc	xyz
1	abc	12345
2	456	def
100	hello	world
54321	NULL	987

Figure 7.8: A typical database table with rowid as key and x and y as data.

7.4.4.2 Rowid value

If rowid is an alias column (i.e., declared as INTEGER PRIMARY KEY), database users are aware of the column. If the rowid column is added by SQLite, they may or may not be aware of the column. In either case, users can define rowid values or SQLite can define the values for users. It guarantees their uniqueness. When a row, without specifying the rowid column value, is inserted into the table, SQLite visits the table B^+-tree and finds an unused integer number for the rowid. It is usually one greater than the largest value in the tree. However, if the largest value is 9,223,372,036,054,775,807, then it chooses an unused number randomly. If no number is found, it returns SQLITE_FULL error code.

7.4.4.3 Rowid representation

Rowid values have different representation depending on who has created the column. If the rowid column is created by SQLite, then the table record does not have it. Otherwise, a NULL (type value 0) is stored in every table record. SQLite gets the actual value from the key. The value is represented as a variable-length Huffman code. Negative rowids are allowed, but they always use nine bytes of storage, and so their use is discouraged. When rowids are generated by SQLite, they will always be non-negative, though you can explicitly give negative rowid values; the -5 in the previous table is such an example. It is in general difficult to say (except for negative values) which ones are generated by the system and which ones supplied by users.

7.4.5 Index key format

In the previous two subsections, you have seen that the key to each table's B^+-tree is an integer, and the data record is one row of the table. Indexes reverse this arrangement. For an index entry, the key is the combined values of all indexing columns of the row being stored in the indexed table, and the data is an integer value that is the rowid of the row. To access row (from a table) that has some particular values for indexing columns, the VM first searches the index to find the corresponding integer value (rowid), then uses that integer value to look up the complete record in the table's B^+-tree. Although many people consider referencing rows by primary key is too expensive, SQLite uses it for its simplicity.

SQLite implements each index (on a table) by creating a separate 'index-table' that is stored in a separate B-tree. An index B-tree maps a search key to a rowid that, in turn, is used to search the indexed table B^+-tree. The index B-tree has its own key comparison, i.e., ordering comparator function to order index entries. A pointer to appropriate key comparison function is supplied to the tree module by the VM.

SQLite automatically creates an index for every UNIQUE column, including PRIMARY KEY columns, specified in a CREATE TABLE statement. You cannot drop these indexes without dropping the indexed table. You may explicitly create indexes on a non catalog table using the CREATE INDEX statement. You can drop these indexes. When a table is dropped, all its indexes are automatically deleted. There are various ways to create indexes in SQLite. The following three examples create an index on the `a` and `b` columns of a table `T1(a, b, c)`.

1. Declare the index explicitly.

   - CREATE TABLE T1(a, b, c)
   - CREATE INDEX idx1 ON T1(a, b)

2. Declare columns to be UNIQUE.
   - CREATE TABLE T1(a, b, c, UNIQUE(a, b))
3. Declare columns to be PRIMARY KEY.
   - CREATE TABLE T1(a, b, c, PRIMARY KEY(a, b))

Note: INTEGER PRIMARY KEY is special and does not generate a separate index. The table B^+-tree is ordered by the INTEGER PRIMARY KEY column aliased as rowid. ◁

SQLite allows creation of multiple indexes on the same column. Consider the following example.

- CREATE TABLE T2(x VARCHAR(5) UNIQUE PRIMARY KEY, y BLOB);
- CREATE INDEX idx2 ON T2(x);

The above example creates three identical (and independent) indexes on column x of table T2: One implicitly by the key word unique, another one implicitly by the key word primary key, and one explicitly by the user. You may note that extra indexes slow down INSERTs, UPDATEs, and DELETEs, and make the database file larger. Schema designers have been warned!

As mentioned earlier, SQLite treats an index as a kind of table, and stores the index in its own B-tree. It maps a search key to a rowid. The format of index records is depicted in Fig. 7.9. The entire record serves as the B-tree key; and, there is no data part. The encoding for index records is the same as that for the indexed table records, except that the rowid is appended at the end, and the type of rowid does not appear in the record header because the type is always signed integer and is represented in the Huffman code (and not in an internal integer type). The other data values and their storage types are copied verbatim from the indexed table. The content of an index on column x of the table of Fig. 7.8 is given in Fig. 7.10.

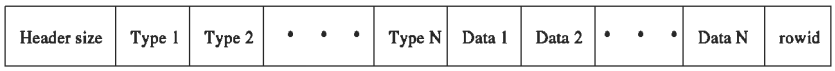


Figure 7.9: Format of index record.

x	rowid
NULL	54321
456	2
abc	-5
abc	1
hello	100

Figure 7.10: An index on column x.

SQLite also supports multicolumn indexing. Figure 7.11 presents the content of an index on the table of Fig. 7.8 with columns y and x. The entries in the index are ordered by their first column; the second and subsequent columns are used as tie-breaker.

y	x	rowid
987	NULL	54321
12345	abc	1
def	456	2
xyz	abc	-5
world	hello	100

Figure 7.11: An index on columns y and x.

Indexes are primarily used to speed up database search. For example, consider the following query: `SELECT y FROM t1 WHERE x = 456`. Suppose there is an index on the x column. The VM does an index search on that index B-tree, and finds all rowids for x = 456; for each rowid it searches the table t1 B^+-tree to obtain the value for the y column for the row that satisfies the search. It uses two tree-searches instead of one, so it takes about twice longer than a lookup against a rowid based search. But, this is definitely better than doing a full table scan.

Transient Index: SQLite may create temporary indexes on the fly in executing SQL statements with order by or group by clause, or distinct selection in aggregate query, or compound select using union, except, or intersect. Those temporary indexes are stored in temporary files. ◁

7.5 Datatype Management

Data processing happens solely in the VM module and this module drives the backend to store and retrieve data into and from databases. The VM is the sole manipulator of data stored in databases; everything else is controlled by bytecode programs that it executes. It decides what data to store in the databases, and what data to retrieve from there. Assigning appropriate storage types to data values and doing necessary data value conversions are the primary tasks of the VM. There are three places of data exchange where data value conversions may take place: from SQLite application to engine, from engine to SQLite application, and from engine to engine. For the first two cases, the VM assigns types to user data. The VM attempts to convert user-supplied values into the declared SQL type of the column whenever it can, and vice versa. For the last item, the data conversions are required for expression evaluations. I discuss these three data conversion issues in the next three subsections.

7.5.1 Assigning types to user data

In Section 7.4, I have discussed storage formats for records of tables and indexes. Each record column value has a storage type. You may note that applications send column values to SQLite in two ways: (1) literals embedded in an SQL statement and (2) values bound to a prepared statement. (The VM also derives column values by executing expressions.) Before the VM executes the prepared statement it assigns an storage type to every such input value. The storage type is used to encode the input value to appropriate physical representation.

The VM decides on the storage type for a given input value of a column in three steps: it first determines the storage type of the input data, then the declared SQL type of the column, and finally, if required, it does the data conversion. Under circumstances described next, the VM may convert data between numeric storage types (INTEGER and REAL) and TEXT during query evaluation. (You may note that SQL standards do not provide any guidelines for data encoding, with a few exceptions such as date, time, timestamp.) These are discussed in the rest of this subsection.

7.5.1.1 Storage type determination

SQLite is 'typeless', i.e., there is no domain constraint. (The SQLite development team wants to keep it typeless forever.) The typelessness permits you storing any type of data in any table column irrespective of the declared SQL type of that column. (Exception to this rule is the `integer primary key`, aka, the rowid column; this column only stores integer values; any other value is rejected by the VM.) You may note that SQLite permits no SQL type declaration in `create table` statements! For example, `create table T1(a, b, c)` is a valid SQL statement in SQLite. The question is how does the VM assign a storage type to a given input value before storing it in a given column? I answer this question below.

The VM assigns initial storage types to user supplied values as follows. As mentioned earlier, there are two ways to provide SQLite input values.

1. A value specified as literal as part of an SQL statement is assigned one of the following storage types:

   - TEXT if the value is enclosed by single or double quotes
   - INTEGER if the value is an unquoted number with no decimal point nor exponent
   - REAL if the value is an unquoted number with a decimal point or exponent
   - NULL if the value is the character string NULL without any quote surrounding it

- BLOB if the value is specified using the X'ABCD' notation, where A, B, C, D are hexadecimal digits

Otherwise, the input value is rejected by the VM, and the query evaluation fails.

2. Values for SQL parameters that are supplied using the `sqlite3_bind_*` API functions are assigned the storage types that most closely match the native type bound. For example, `sqlite3_bind_blob` binds a value with storage type BLOB.

The storage type of a value that is the result of an SQL scalar operator depends on the outermost operator of the expression. User-defined functions may return values with any storage type. It is not generally possible to determine the type of the result of an expression at SQL statement preparation time. The VM assigns the storage type at runtime upon obtaining the value.

7.5.1.2 Column affinity determination

You may recall that SQLite permits any column (except the integer primary key) to store any type of value. Thereby, the type information is stored along with the value. Other SQL database engines that I am aware of use the more restrictive system of static typing, where the type is associated with the container, and not with the value. SQLite is a little more flexible. To maximize compatibility between SQLite and other database engines, SQLite supports the concept of *type affinity* on columns in tables. Each input value may have an affinity of the column's declared SQL type. The type affinity of a column is the recommended type for values stored in that column: note that "it is recommended, not a definite requirement".

The preferred type for values of a column is called its (the column's) affinity. Affinity type of a column is different from its declared type, though the former is derived from the latter. Each column has one of these five affinity types: TEXT, NUMERIC, INTEGER, REAL, and NONE. (You may note a bit of naming conflict: "text", "integer", and "real" are names used for internal storage types, too. You can, however, determine the type category from the context of its use.) SQLite follows the following rules in determining the affinity type of a column, depending on the column's declared SQL type in the CREATE TABLE statement.

1. If the SQL type declaration contains the substring INT, then the column has the INTEGER affinity.

2. If the SQL type declaration contains any of the substrings CHAR, CLOB, or TEXT, then the column has the TEXT affinity. (You may note that the SQL type VARCHAR contains the string CHAR, and therefore has the TEXT affinity.)

3. If the SQL type declaration contains the substring BLOB or if no type is specified, then the column has the NONE affinity.

4. If the SQL type declaration contains any of the substrings REAL, FLOA, or DOUB, then the column has the REAL affinity.

5. Otherwise, the affinity type of the column is NUMERIC.

The VM evaluates the rules in the same order as given above. The pattern matching is case insensitive. (You may note that SQLite is not at all strict about spelling errors in SQL type declarations.) For example, if the declared SQL type of a column is BLOBINT, the affinity is INTEGER, and not NONE.

Note: If an SQL table is created using a `create table table1 as select...` statement, the declared type of each column is determined by the affinity type of the corresponding expression in the select part of the create table statement. If an expression affinity is text, numeric, integer, real, or none, then the declared type of the column is text, num, int, real, or "" (empty string), respectively. The default value of each such column is SQL NULL. The type of implicit rowid is always integer and cannot be NULL. ◁

7.5.1.3 Data conversion

SQLite defines a relationship between affinity types and storage types. If a user-supplied data value for a column does not satisfy the relationship, the value is either rejected or converted into the appropriate format. When a value is about to be inserted into a column, the VM first assigns it the most suitable storage type (see Section 7.5.1.1), and then it determines the affinity type of the column (see Section 7.5.1.2), and finally makes an attempt to convert the value with the initial storage type into the format of its affinity type. It does so using the following rules:

1. A column with TEXT affinity stores all values that have NULL, TEXT, or BLOB storage types. If a numerical value (integer or real) is inserted into the column, the value is converted into text form, and the final storage type becomes TEXT.

2. A column with NUMERIC affinity stores values with all five storage types. When a text value is inserted into a NUMERIC column, the VM attempts to convert the value to an integer or real number. If the conversion is successful (i.e., lossless and reversible),[2] then the converted value is stored using the INTEGER or REAL storage type accordingly. If the conversion cannot be performed, the value is stored using the TEXT storage type. The VM does not attempt to convert NULL or BLOB values.

[2]To be lossless, the first 15 significant decimal digits of the number must be preserved.

3. A column with INTEGER affinity behaves in the same way as a column with NUMERIC affinity, except that if a real value with no floating point component (or text value that converts to such) is inserted, the VM converts the value into an integer and the final storage type becomes INTEGER.

4. A column with REAL affinity behaves like a column with NUMERIC affinity, except that it forces integer values into floating point representation. (But, SQLite performs an optimization—small values without fractions are stored on disk as integers to take up less space, and are only converted to floating point as the values are read out of the table.)

5. A column with affinity NONE stores values with all five storage types, and does not prefer one storage type over another. The VM does not convert any input value.

Note: All SQL database engines do data conversions. They reject input values that cannot be converted into desired column types. SQLite however may store values even if a format conversion is not possible. For example, if you have a table column declared as SQL type INTEGER and you try to insert a string (such as "123" or "abc"), the VM will look at the string and analyze whether it looks like a number. If the string does look like a number (as in "123"), it is converted into a number (and into an integer if the number does not have a fractional part), and is stored with real or integer storage type. But, if the string is not a well-formed number (as in the case "abc"), it is stored as a string with TEXT storage type. A column with TEXT affinity tries to convert numbers into an ASCII text representation before storing them. But, BLOBs are stored in TEXT columns as BLOBs because SQLite cannot in general convert a BLOB into text. SQLite allows inserting string values into integer columns. This is a feature, not a bug. The SQLite development team is proud of this feature. ◁

7.5.1.4 A simple example

Let us study a very simple example for more clarity about storage type, affinity type, and type conversion. Figure 7.12 presents a typical table whose all columns are typeless. Suppose you execute the SQL insert statement of the figure. The figure also depicts the row record that the VM inserts into the T1 table B^+-tree. The initial storage types of input values for the a, b, and c columns are integer, NULL, and text, respectively (see Section 7.5.1.1). The affinity type of all columns is NONE (see Section 7.5.1.2), and the VM does not convert these initial storage types (see Section 7.5.1.3). In the figure, the row record (header plus data items) consists of 11 bytes as follows. (All numbers in the record are given in hexadecimals. You may recall that SQLite does not reorder the position of columns: they occur in the same sequence they appear in the create statement.)

1. The header is 4 bytes long: one byte for the header size, plus one byte for each of three manifest types of column values. The value 4 is encoded as the single byte 0x04.

2. Type 1 (the type of the column `a` value) is the number 2 (representing a 2-byte signed integer) encoded as a single byte 0x02.

3. Type 2 (the type of the column `b` value) is the number 0 (NULL) encoded as a single byte 0x00.

4. Type 3 (the type of the column `c` value) is the number 22 (a text, $(22 - 12)/2 = 5$ bytes long) encoded as a single byte 0x16.

5. Data 1 is a 2-byte integer 00B1, which is the value 177. (You may note that 177 could not be encoded as a single byte because B1 is −79, not 177.)

6. Data 2 is NULL, and it does not occupy any byte in the record.

7. Data 3 is the 5-byte string 68 65 6C 6C 6F. The zero-terminator is omitted.

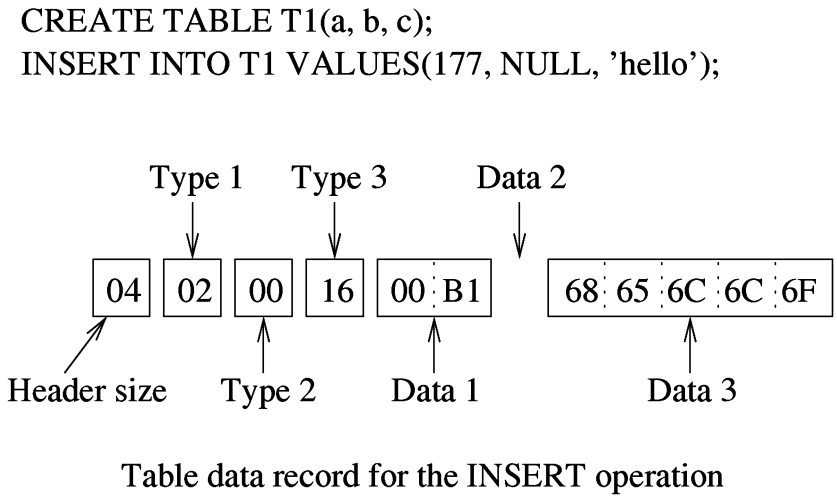


Figure 7.12: A typical table data record.

7.5.1.5 Column affinity example

Consider a table T1 that is created by this SQL statement: `create table T1(t TEXT, n NUMERIC, i INTEGER, r REAL, b BLOB)`. Suppose you execute this SQL statement: `insert into T1 values('1.0', '1.0', '1.0', '1.0', '1.0')`. The initial storage type for all input values is TEXT, because they all are single-quoted. The affinity types for columns `t`, `n`, `i`, `r`, and `b` are TEXT, NUMERIC, INTEGER, REAL, and NONE, respectively, based on the rules defined in Section 7.5.1.2 on page 191. The final storage types for `t`, `n`, `i`, `r`, and `b` values are TEXT, INTEGER, INTEGER,

REAL, and TEXT, respectively. For the numeric affinity type, '1.0' looks like a valid number 1.0 that is numerically equal to integer value 1, and hence the final storage type is integer. The blob column has none affinity and hence stores the value without any change in the initial storage type. For the SQL statement `insert into T1 values(1.0, 1.0, 1.0, 1.0, 1.0)`, the initial storage type of all input values is real and the final storage types for the columns are TEXT, INTEGER, INTEGER, REAL, and REAL, respectively. For the text column, the real value 1.0 is converted into text "1.0" as the text storage type. For the SQL statement `insert into T1 values(1, 1, 1, 1, 1)`, the initial storage type of all input values is integer and the final storage types for the columns are TEXT, INTEGER, INTEGER, REAL, and INTEGER, respectively. You can find more examples on column affinity at http://www.sqlite.org/datatype3.html.

7.5.1.6 Other affinity modes

The above sub-subsections describe the operation of the database engine in the 'normal' and default affinity mode. SQLite supports two other affinity-mode related features, for two extremes.

- **Strict affinity mode.** In this mode if a conversion between initial storage type and affinity type is ever required, the engine returns an error and the current statement execution fails.
- **No affinity mode.** In this mode no conversions between storage types are ever performed by the VM.

7.5.2 Converting engine data for applications

Applications invoke `sqlite3_column_*` API functions to read data out of the SQLite engine. These functions attempt to convert the data value where appropriate. For example, if the internal representation is REAL, and a text result is requested (via the `sqlite3_column_text` function), the VM uses the sprintf() library function internally to do the conversion before returning the value to the application. The table in Fig. 7.13 presents data conversion rules that the VM applies on internal data to produce output data for applications.

7.5.3 Assigning types to expression data

The VM may convert internal data before comparing them one against another or in evaluating expressions. In the rest of this subsection, I discuss how the VM handles internal data.

7.5.3.1 Handling SQL NULL values

A SQL NULL value can be used for any table columns except the primary key columns. The storage type of the NULL value is "NULL". The NULL value is distinct from all valid values for

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is NULL pointer
NULL	BLOB	Result is NULL pointer
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as for INTEGER⟶TEXT
FLOAT	INTEGER	Convert from float to integer
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	Same as FLOAT⟶TEXT
TEXT	INTEGER	Use `atoi()`
TEXT	FLOAT	Use `atof()`
TEXT	BLOB	No change
BLOB	INTEGER	Convert to TEXT then use `atoi()`
BLOB	FLOAT	Convert to TEXT then use `atof()`
BLOB	TEXT	Add a \000 terminator if needed

Figure 7.13: Data conversion protocol.

a given column irrespective of their storage types. SQL standards are not often very clear about how to handle NULL values from columns in expressions, i.e., how to handle NULL values in all circumstances. For example, how do we compare NULL values with one another, and with other values? SQLite handles NULL values in the way many other RDBMSs (such as Oracle and PostgreSQL) do. NULL values are considered indistinct (i.e., the same value) in the SELECT DISTINCT statement, the UNION operator in a compound SELECT, and GROUP BY. NULL values are, however, considered distinct in a UNIQUE column. NULL values are handled by the built-in SUM function as specified by the SQL standard. Arithmetic operations on NULL always produce NULL.

7.5.3.2 Types for expressions

SQLite supports four kinds of comparison operations:

- binary comparison operators =, ==, <, <=, >, >=, <>, and ! =
- the ternary comparison operator 'BETWEEN'
- the set membership operators 'IN' and 'NOT IN'
- the null checking operators 'IS NULL' and 'IS NOT NULL'

Note: The general IS and IS NOT operators work like = and ! = operators, respectively. ◁

The outcome of a comparison operation depends on the storage types of the two values being compared, according to the following rules:

1. A value (the left-hand side operand) with storage type NULL is considered less than any other value (including another value with storage type NULL).

2. An INTEGER or REAL value is less than any TEXT or BLOB value.

3. When an INTEGER or REAL is compared to another INTEGER or REAL, a numerical comparison is performed.

4. A TEXT value is less than a BLOB value.

5. When two TEXT values are compared, the standard C library function **memcmp** is usually used to determine the result. However you can override it by system or user-defined collation functions. (I will discuss collation in Section 10.16 on page 247.)

6. When two BLOB values are compared, the result is always determined by using the **memcmp**.

However, before applying the above rules, the first and foremost task for the VM is to determine the final storage types of the operands of the comparison operator. The VM first determines the preliminary storage types of operands, and then, (if necessary) converts values between types based on their affinity. Finally, it performs the comparison using the above rules. As mentioned earlier, the VM converts values between text and integer/real storage types.

If an expression is a column or a reference to a column via an AS alias or a subselect with a column as the return value or rowid, then the affinity of that column is used as the affinity of the expression. (See the sqlite3ExprAffinity function in the expr.c source file. You may note that each cast expression explicitly specifies the affinity type.) Otherwise, the expression's affinity is NONE. The VM may convert values between the numeric storage types (INTEGER and REAL) and TEXT before performing a comparison operation. For binary comparisons, this is done in the cases bulleted next (see the sqlite3CompareAffinity function in expr.c). The term "expression" in the following bullets means any SQL scalar expression or literal other than a column value.

- When two values are compared, if any one is a column value that has INTEGER, REAL, or NUMERIC affinity, then that affinity is preferred for both values. That is, the VM attempts to convert the other column value before the comparison.

- If one value has TEXT affinity and the other has NONE, then the TEXT affinity is applied on the other value.

- Otherwise, no affinity is applied and no conversions occur. The values are compared following the above-mentioned standard rules: for example, if a string is compared to a number, the number will always be less than the string.

In SQLite, the expression `a BETWEEN b AND c` is equivalent to `a >= b AND a <= c`. The two clauses of the AND-operator are evaluated independently using comparison operators, and different affinities can be applied to `a` in each of the two comparisons required to evaluate the expression.

Expressions of the kind `a IN (SELECT b from ...)` are handled by the rules enumerated above for the equality binary operator (e.g., `a = b`). For example, if `b` is a column value and `a` is an expression, then the affinity of `b` is applied to `a` before any comparisons take place. SQLite treats the expression `a IN (x, y, z)` as equivalent to `a = x OR a = y OR a = z` and no affinities are applied to `x, y, z`.

The expression `a IS NULL` returns true if `a` is a NULL value; otherwise, it returns false.

Here are a few simple examples; you can find them and other at the http://www.sqlite.org/datatype3.html webpage. Suppose you have a table t1 that is created by CREATE TABLE t1(a TEXT, b NUMERIC, c BLOB, d). You insert a single row in the table by executing INSERT INTO t1 VALUES ('500', '500', '500', 500). The final storage types for a, b, c, and d column values are TEXT, INTEGER, TEXT, and INTEGER, respectively. The following comparison (select) statements and their output reveal some subtleties.

- `SELECT a < 600, a < 60, a < 40 FROM t1` converts 600, 60, and 40 to "600", "60", and "40", respectively, before the comparison, because the `a` column has TEXT affinity and the values are compared as TEXT; and the statement returns `1 | 1 | 0` as output because "500" is less than both "600" and "60" as text, and is not less than "40".

- `SELECT b < 40, b < 60, b < 600 FROM t1` does not convert any literal values because b has NUMERIC affinity, and compares values as NUMERIC, and returns `0 | 0 | 1` as output because 500 is numerically less than both 40 and 60 but is not less than 600.

- `SELECT c < 40, c < 60, c < 600 from t1` does not convert 40, 60, and 600 because c has NONE affinity. The three literal values (storage class NUMERIC) are less than "500" (storage class TEXT), and the statement returns `0 | 0 | 0` as output.

- `SELECT d < 40, d < 60, d < 600 from t1` does not convert 40, 60, and 600 because `d` has NUMERIC affinity, and returns `0 | 0 | 1` as output because the values are compared as integers.

7.5.3.3 Operator types

All mathematical operators (except the concatenation operator ||) apply NUMERIC affinity to all operands prior to their evaluation. But, if any one operand is NULL, then the result is NULL; other non number operands are converted to 0 or 0.0 before performing the mathematical operation. For the concatenation operator, TEXT affinity is applied to both operands. If the VM cannot convert either operand to TEXT (because it is NULL or BLOB), then the result of the concatenation is NULL.

7.5.3.4 Types in order by

When values are sorted by an ORDER BY clause in select queries, no storage type conversions take place before the sort. The previously stated standard comparison rules are followed: values with storage type NULL come first, followed by INTEGER and REAL values interspersed in numeric order, followed by TEXT values in `memcmp()` order or system or user-defined collation order, and, finally, BLOB values in `memcmp()` order. (See Section 10.16 on page 247 to learn how to use collations in SQL statements.)

7.5.3.5 Types in group by

For grouping values using the GROUP BY clause in select queries, no storage type conversions take place before grouping. Values with different storage types are considered distinct, except for INTEGER and REAL values, which are considered equal if they are numerically equal. No affinities are applied to any values as the result of a GROUP BY clause.

7.5.3.6 Types in compound SELECTs

The compound SELECT operators (namely, UNION, INTERSECT, and EXCEPT) perform implicit comparisons between values. Before their corresponding comparisons are performed, the VM does not perform any affinity related value conversions. Values are compared as they are.

Summary

The VM module is the backend database engine. The data manipulation logic resides in this module. It executes programs written in a special bytecode programming language. (This is akin to any other assembly language.) A bytecode program is a linear sequence of bytecode instructions. A bytecode instruction has an opcode and can have upto five operands or register names that hold operands.

A bytecode program (that is generated by the frontend system) is represented by a `Vdbe` object (externally, by `sqlite3_stmt` pointer). Applications apply SQLite APIs on the object to execute the bytecode program. For example, when the application executes the sqlite3_step API function, it makes an iteration of the program execution until the execution halts or breaks with an output row.

The VM supports five datatypes: integer, real, text, blob, and SQL NULL. Each data in the database or in-memory must be of one of these five types. This is called the storage type of the data. When needed, the VM converts data from one type into another types. Applications can use the `typeof` SQL function to determine the datatype of a value.

The VM is the sole manipulator of data. It maintains records in table and index trees. Their record formats are very similar. SQLite uses a variant of variable-length record formatting scheme. It uses manifest typing to represent individual value in a record. In manifest typing, for each value its storage type information is also stored in the record. This scheme allows to storing any storage-typed values in any column irrespective of the declared SQL type of the column with one exception that integer primary key columns store integer values only. The storage type of a value is encoded as an integer, and the integer also encodes the (byte) length of the value. Each table record starts with a header size (that is a variable-length Huffman code), followed by the manifest types of data values, followed by individual data values.

Datatype management is a little complicated in SQLite. SQLite assigns affinity types for columns based on their declared SQL types. Input data values are first attempted to be converted into affinity types. If the conversion is not successful, the original value is stored in the column anyway. SQLite does similar data conversion for evaluation of expressions and functions.

Chapter 8

The Frontend Module

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- how SQLite frontend system is composed of tokenizer, parser, optimizer, and code generator
- various code optimization techniques employed by SQLite

Chapter synopsis

In order to execute an SQL statement (or an SQLite command), SQLite first preprocesses and analyzes the statement, and then generates a bytecode program that the VM can execute to produce the output of the statement. The frontend subsystem preprocesses SQL statements, optimizes them, and generates bytecode programs. This chapter presents inner workings of the frontend.

8.1 Frontend

Every SQL database system compiles each SQL statement into some kind of internal representation that is used by the backend engine to carry out the work specified by the statement. In most SQL RDBMSs, the internal representation is a complex web of interlinked objects or at least a tree of objects. In SQLite, the representation is a machine-language like program. This translation is done by the frontend. (For any SQL statement, you can view the corresponding bytecode program by prepending the EXPLAIN keyword to the statement. A sample bytecode program is given in Fig. 7.1 on page 176.)

The SQLite frontend is composed of four subsystems: (1) tokenizer, (2) parser, (3) optimizer, and (4) code generator. The tokenizer breaks an input string into a number of distinct tokens. The parser assigns a structure to an input SQL statement by organizing the corresponding tokens

into a parse tree. The optimizer restructures a parse tree and produces a new parse tree that will produce an equivalent, but efficient bytecode program. The code generator traverses the tree and produces a bytecode program that is considered equivalent to the operation specified by the input SQL statement. The frontend implements the `sqlite3_prepare` API function that converts SQL statements and SQLite commands into bytecode programs. You may recall that each bytecode program is internally represented by a `Vdbe` object. In the rest of this chapter, I discuss the four frontend subsystems. The bulk of this chapter though deals with query optimization.

8.2 The Tokenizer

The tokenizer subsystem is the topmost module of the frontend (see Fig. 2.10 on page 72). The tokenizer looks at an input string and splits the string into a sequence of tokens, and sends those tokens one-by-one over to the parser. (A *token* is a sequence of characters that has a special meaning. Examples of tokens include left-parenthesis, literal, keyword, identifier, etc. Literals are string, numeric, and binary constants. Keywords have special meanings in SQL/SQLite. Identifiers refer to specific objects such as column, table, index.) The tokenizer discards all white-space and comment tokens. It assigns each token a token class. There are about 140 different token classes. (SQLite uses Lemon parser generator, and this defines token classes and writes them into parse.h file during the SQLite source compilation process; token class names are prefixed with the TK_ string.) The tokenizer source code is defined in the tokenize.c file. It also uses an SQL keyword hash table from keywordhash.h (this file is also generated at the SQLite source compilation time). A typical example of what the tokenizer does with one SQL input string is shown in Fig. 8.1.

The tokenizer partitions tokens into two groups: keyword and non keyword. If a token is a keyword, the token code of that keyword is obtained from the keyword hash table and is sent to the parser. If the token is not a keyword, TK_ID is returned. For example, in the above table, table1 is a TK_ID. You may note that every (non space and non comment) class type other than TK_ID is a keyword.

8.3 The Parser

The parser accepts tokens from the tokenizer, analyzes the structure of the SQL statement, and organizes those tokens into a parser tree. (A *parse tree* is a data structure that describes an SQL statement.) A parse tree is formed from several primary data structures that I describe below.

1. **Token**: A single token from the input SQL statement. A `token` object is used to hold the text value of a literal or the name of a table or a column. It is used to pass on the value from the tokenizer to the parser.

```
CREATE TABLE table1(
    a BLOB PRIMARY KEY,
    b TEXT
);
```

Token Text	Token Class	What Happens
"CREATE"	TK_CREATE	Sent to parser
" "	TK_SPACE	Discarded
"TABLE"	TK_TABLE	Sent to parser
" "	TK_SPACE	Discarded
"table1"	TK_ID	Sent to parser
"("	TK_LP	Sent to parser
"\n"	TK_SPACE	Discarded
"a"	TK_ID	Sent to parser
" "	TK_SPACE	Discarded
"BLOB"	TK_ID	Sent to parser
And so forth....		

Figure 8.1: A typical tokenizer output.

2. **Expr**: A single operator or operand in an expression. A tree of `expr` structure describes a complete expression.

3. **ExprList**: A list of one or more expressions, each with an optional identifier and an ascending/descending flag.

4. **IdList**: A list of one or more identifiers.

5. **SrcList**: A list of one or more data sources. A data source can be an SQL table or a view or a subquery—basically anything that generates rows of data. (The list size is one for insert, delete, and update statements.)

6. **Select**: A SELECT statement such as you find in a subquery.

The parser checks the syntax of the query, and also does a kind of semantics check, for example, whether or not the tables referenced in the query are present in the database and all attributes are in their schema. Here are some examples of how the above mentioned data structures are used to parse SQL statements.

1. DELETE FROM ⟨srclist⟩ WHERE ⟨expr⟩;

2. UPDATE ⟨srclist⟩ SET ⟨exprlist⟩ WHERE ⟨expr⟩;
   UPDATE ⟨srclist⟩ SET ⟨id=expr, id=expr, id=expr⟩ WHERE ⟨expr⟩;

3. INSERT INTO ⟨srclist(idlist)⟩ VALUES(⟨exprlist⟩);
   INSERT INTO ⟨srclist(idlist)⟩ select;

4. SELECT ⟨exprlist⟩ FROM ⟨srclist⟩ WHERE ⟨expr⟩ ORDER BY ⟨exprlist⟩;

When the parser is called (via `sqlite3_prepare` function), it first creates and initializes an object of type `Parse` (see Fig. 8.2). The object represents an SQL parse context. The object is passed through the parser and down into all the parser action routines in order to carry around information that is global to the entire parsing process.

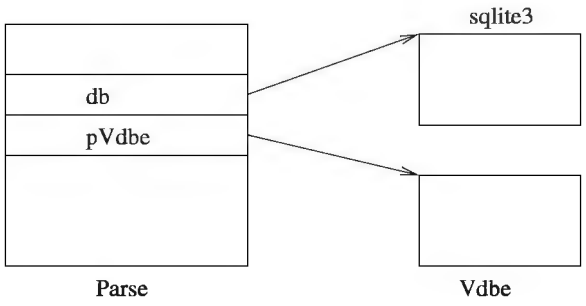


Figure 8.2: The `Parse` object.

A `Parse` object contains a pointer to a `Vdbe` object. As shown in Fig. 7.3 on page 181, the `Vdbe` object contains a space to hold a bytecode program. Initially the space is empty. It is filled up by the code generator as the parsing and code generation activities evolve. You can study the build.c source file to see how parsing and code generation are carried out for create table/index statements.

8.4 The Code Generator

The code generator accepts parse trees from the parser and produces bytecode programs that the VM can execute. Many (almost all) RDBMSs generate a tree of data structures that the VM traverses in order to evaluate the program. SQLite generates bytecode procedural programs that resemble assembly language programs. The VM interprets bytecode programs.

The code generator accepts an (optimized) parse tree from the query optimizer, and generates bytecode instructions to do the work of the SQL statement that the parse tree represents. It is the largest and the most complex subsystem within SQLite; it occupies 40% of SQLite library code. Unlike other subsystems, it does not have a well defined interface. It is tightly coupled to the parser/optimizer. The following internal functions help in the code generation process.

1. `sqlite3GetVdbe:` It returns a pointer to the Vdbe object for the current parse tree. It creates the object if it does not already exist.

2. `sqlite3VdbeAddOp0–4:` It adds a new bytecode instruction to the end of the program for the given Vdbe object.

3. `sqlite3VdbeChangeP1–5:` It changes the P1–5 operand of an existing instruction.

4. `sqlite3VdbeCurrentAddr:` It returns the address of the next instruction to be inserted.

5. `sqlite3ExprCode:` It generates code to evaluate an expression.

6. `sqlite3ExprIfTrue:` It generates code that will branch if a boolean expression is true.

7. `sqlite3ExprIfFalse:` It generates code that will branch if a boolean expression is false.

8. `sqlite3CodeVerifySchema:` It generates code that will verify the schema cookie after starting a read-transaction on a database file. (It is important that the schema cookie be verified and a read-transaction be started before anything else happens in the bytecode program.)

9. `sqlite3BeginWriteOperation:` It generates code that prepares for doing an operation that might change the database. It starts a new write-transaction on a database file and the temp database if we are not already within a transaction.

10. `sqlite3ChangeCookie:` It generates code that will increment the schema cookie.

11. `sqlite3NestedParse:` It is used only to parse and generate code for an SQL statement as a part of another SQL statement. This function is designed to allow the INSERT, UPDATE, and DELETE operations on the `sqlite_master` table, which are needed when users execute CREATE, DROP, and ALTER statements.

The routines in where.c file generate code to implement the WHERE clause of SQL SELECT, DELETE, and UPDATE statements. The code in this file decides what search strategies and which (if any) indexes to use. The code generators for SELECT, DELETE, and UPDATE call `sqlite3WhereBegin` to create the top of a loop that will run once for each row that matches the WHERE clause. Then the caller adds bytecode to deal with a single row. Then it calls `sqlite3WhereEnd` to close the loop. The `sqlite3WhereBegin` returns a pointer which becomes the only parameter to `sqlite3WhereEnd`.

8.4.1 Name resolution

A SQL statement would not make any sense even if there is an identifier (TK_ID) that cannot be associated with meaningful information. When the name of an identifier appears in an expression, the code generator has to figure out what table, index, or column of what table the identifier refers to. For example, an identifier can resolve to a column in one of the tables in the FROM clause

of SELECT or to the table being operated on by INSERT, UPDATE, or DELETE; the `SrcList` of source tables. An identifier can also resolve to the name of one of the expressions (`ExprList`) in the result set of a SELECT statement. The `SrcList` and `ExprList` (each possibly empty) form a "NameContext" which is used to resolve identifier names in expressions. Subqueries can lead to nested NameContexts. For example, in `SELECT exprlist2 FROM srclist2 WHERE ... IN (SELECT exprlist1 FROM srclist1 WHERE ...)`, the inner NameContext ⟨exprlist1, srclist1⟩ pair holds a pointer back to the outer NameContext ⟨exprlist2, srclist2⟩ pair.

The code generator uses `sqlite3ResolveExprNames` routine (implemented in resolve.c) to resolve the names in an expression. It calls the resolver with the inner NameContext when resolving names in the subqueries and uses the outer NameContext for resolving names in the surrounding statement. When resolving names for the subqueries, if a name cannot be resolved using the inner context, it pops to the outer context and tries again. Resolving a name in an outer context means we are dealing with a correlated subquery. Non-correlated subqueries run once and their results are stored for future use. In contrast, correlated subqueries must be rerun every time their result is needed.

8.5 Query Optimizer

Given an SQL statement, there are often many different equivalent relational algebraic expressions, and hence there are many parse trees, in the sense that they all produce the same result. The code generator may produce different bytecode programs for different parse trees. The VM takes different amount of time to execute those bytecode programs. The duty of a query optimizer is to find the tree that produces the most efficient bytecode program.

Each parse tree is a tree of relational algebraic operations, and determines a plan for the query evaluation. The plan decides on the appropriate algorithm for each algebraic operation and the mechanism to store intermediate results. A RDBMS typically includes an optimizer whose job is to choose the best plan that will produce results with the least time and/or space. Finding the best plan is a computation intensive task. Practical systems are satisfied with near best plans or they at least avoid the worst plan. Most systems apply various heuristics to decide on a final plan. Query optimization is a very delicate issue, because the speed of the query execution by the engine depends on the optimization done by the frontend. In SQLite, as the VM blindly executes bytecode programs, all optimization work is done by the frontend.

There is not much an optimizer can do for SQL insert statements and those statements without a `where` clause. In other cases, the optimizer primarily aims to reduce the number of rows retrieved from base tables. Accessing base tables is very much I/O intensive, and hence is a costly action. Therefore, the more the optimizer can reduce accesses to base tables, the better is the query

execution speed. The only way this can be achieved is by using a good access path (aka, index). For each SQLite table, the table itself is the primary index, but there can be secondary indexes on the table. The optimizer has to decide on an access path to retrieve rows from each base table: either a full table scan or an index scan. In SQLite, a *full table scan* involves reading all rows from the table B^+-tree in their rowid order. If there is no index for a scan column, then a full table scan is the only option. Otherwise, we can have an index scan that could cut down retrieval of some unwanted rows from the base table. For example, a point lookup query by rowid (such as `select * from t1 where rowid=2`) uses the table B^+-tree for the index scan purpose, and retrieves the result fast because it retrieves at most one row from the table. For `select * from t1`, SQLite performs a full table scan of the `t1` table B^+-tree. For `select * from t1 where col1 = val`, if there is no index on column col1, SQLite performs a full table scan and for each row it compares the value of col1 against val.

There are two main issues in query optimization for a given query: (1) what alternative plans are considered and (2) how the costs for the plans are estimated. SQLite does optimizations at many levels. The optimization basically involves selecting those tables and indexes that would produce results faster. You may recall that each SQL table is stored in a B^+-tree that has rowid as the key. The B^+-tree is called the primary index for the table. There can be other secondary indexes (B-trees) on the table for different search keys. Each entry in a secondary index includes the rowid of the corresponding entry in the SQL table. When doing an indexed lookup of a row, the normal procedure the VM follows is that it searches the index to find the index entry, extracts the rowid from the index, and uses that rowid to perform a search on the base table. Thus a typical via-index lookup involves two searches on two trees. If, however, the columns that are to be fetched from the table are already available in the index itself, the VM will use the values contained in the index and will never look up the base table. (You may recall from Section 7.4.5 on page 187 that the same values and their manifest type information are copied from the base table into the index.) This saves one tree search for each row and can make many queries run twice as fast.

In the rest of this chapter, I discuss query optimization issues for SELECT statements. The optimizations for DELETE and UPDATE are done similarly. The latter two each actually run in two phases. (1) In the first phase, the VM stores the rowids of affected rows in an internal FIFO object (of type RowSet defined in the rowset.c source file). (2) In the second phase, it does the actual deletion or update in the order rowids are put in the FIFO. A DELETE statement without a WHERE clause uses a special opcode (OP_CLEAR) to erase the entire table. In case you want to suppress this optimization, you must add WHERE 1 to the DELETE statement. The where.c source file has almost all the query optimization code.

You may note that SQLite does not keep track of the size and other statistical information

about SQL tables. The optimization schemes are reasonably simple. Each SQL is partitioned into a collection of query blocks, and each block is optimized independently. Some optimization schemes are described in the following subsections.

8.5.1 ANALYZE command

SQLite, by default, does not gather statistical information that is normally done in many DBMSs for query optimization purposes. But, SQLite supports `analyze` commands for this purpose so that database users can manually build some statistical information on tables and indexes. This enables the query optimizer to obtain a more accurate estimate of the work involved in using various indexes and to choose the best index for a query. (It is an advance feature and can be permanently disabled at the SQLite source code compilation time.) By default, the analyze command scans all tables and indexes in all (including attached) databases and gathers statistical information about them. You may though selectively run the command against a specific database (all its tables and indexes) or even a specific table to analyze itself and its indexes only or only a single index. The results of the analyze command execution are stored in the `sqlite_stat1` catalog table.

On the very first execution of the analyze command, SQLite creates the `sqlite_stat1` catalog. The structure of the catalog is equivalent to the following schema definition: `create table sqlite_stat1(tbl, idx, stat)`, where `tbl` is the name of a table, `idx` is the name of an index on the table, and `stat` contains statistical information about the index. The last one is a string composed of a list of integer numbers representing various statistics about the index. The first integer is the total number of entries in the index. The successive integer value represents statistics about each indexed column from the table. This integer value is a guess about how many rows of the table the index will select for a given value of the column. If d is the count of distinct values in the column and k is the total number of rows in the table, then the integer is computed as: $\lceil \frac{k}{d} \rceil$. If $k = 0$, then d is 0 and there will not be any entry in the `stat1` catalog. If $k > 0$, then it is always the case that $d > 0$. The query optimizer can use this analyze's output information to make better choices of indexes for query execution plans. You may note that the `stat1` catalog is not updated as the database changes. So, after making significant changes, you may need to rerun the ANALYZE command again. You though have a choice. The results of an ANALYZE command are only available to database connections that are opened after the ANALYZE command completes.

You can read, modify, and even erase the content of the `stat1` catalog completely, but cannot drop it. Erasing the entire content of the catalog has the effect of undoing the ANALYZE command. Changing the content of the catalog will perturb the information and can get the optimizer very confused and may cause it to make silly index choices. The SQLite development team does not recommend updating the catalog by any way except by running the ANALYZE command.

Note: There is an optional `sqlite_stat2` catalog that stores histogram data collected during the analyze command execution. This catalog is available when the SQLite source code is compiled with the SQLITE_ENABLE_STAT2 compile flag on. ◁

8.5.2 WHERE clause

The WHERE clause is the most widely used clause in SQL applications. SQLite breaks it down into multiple 'terms', where the terms are connected by the AND operator. This is called an expression in the conjunctive normal form. (You may note that if the where clause contains the OR operator, then the entire clause is a single term, and SQLite applies OR-optimization techniques; see Section 8.5.4 on page 210.) For example, in the `select * from t1 where a > 0 and b < 0` statement, `a > 0 and b < 0` is the where clause here that has '$a > 0$' and '$b < 0$' terms. SQLite tries to evaluate each term by using an index. If this is possible, no separate test is performed for the term because the index search will automatically perform the test. If this is not possible for a term, the term is evaluated against each row of the relevant input tables. (You may note that this test is used only to discard some retrieved rows; they though do not affect the total number of rows retrieved.) Sometimes a term may provide hints to an index but the term is still evaluated against each row of the base tables. During analyzing a term, SQLite might add new 'virtual' terms to the WHERE clause. Virtual terms are normally satisfied using indexes and never tested against rows retrieved from the base tables. The following constructs control the use of indexes in the optimization process.

1. Indexes will only be used when the WHERE clause consists of terms connected by AND (e.g., `select * from table1 where x = 5 AND y >= 'a' AND y < 'zzz'`).

2. An IN clause will also use an index (e.g., `select * from table where x in (5, 7)`).

3. The use of OR disables indexes (e.g., `select * from table1 where x = 5 OR y = 7`). Instead of OR, you can use a UNION in such cases (e.g., `select * from table1 where x = 5 union select * from table1 where y = 7`). If the two SELECTs are disjoint, a UNION ALL runs faster than a UNION (e.g., `select * from table1 where x = 5 union all select * from table1 where x = 7`).

A term must be of one of the following forms in order to be used in an index: (1) column OP expression, (2) expression OP column, (3) column IN (expression-list), (4) column IN (subquery), (5) column is null; where OP is either = or > or >= or < or <=.

Using single column indexes is straightforward. Let us study how SQLite decides on a multi-column index. Suppose a table table1 has an index that is created using a statement like this:

`create index idx1 ON table1(a, b, c, d, e, ..., y, z)`. For every pair of consecutive columns i and j, i is the principal column and j is the tie-breaker for the sorting order. Thereby, the columns of the index can only be used left to right order, starting with `a`. Thus, SQLite may choose the index if the initial columns of the index (columns `a`, `b`, and so forth, consecutively) appear in WHERE clause terms. For example, if column `a` does not appear in a term, then index idx1 cannot be used. All index columns (start from the left most column) must be used with the equality operator or the IN operator except for the right-most column which can have inequality operator and can be any column. For the right-most column, there can be up to two inequalities defining upper and lower bounding values for the column.

You may specifically note that to use an index it is not necessary for every column of the index to appear in a WHERE clause term, but there cannot be gaps in the columns of the index. For our example index, suppose in a query there is no WHERE clause term that constrains column `c`. Then terms that constrain columns `a` and `b` can be used with the index but not terms that constrain columns `d` through `z`. Similarly, no index column will be used (for indexing purposes) that is to the right of a column that is constrained only by inequalities. For our example index and WHERE clause like this: `... WHERE a = 5 AND b IN (1, 2, 3) AND c > 100 AND d = 'hello world'`, only the index columns `a`, `b`, and `c` would be usable. The `d` column would not be usable because it occurs to the right of `c` and `c` is constrained by inequalities. The `d` clause becomes a test case for the query evaluation purpose.

8.5.3 BETWEEN clause

SQLite transforms the between-clause such as `expr1 BETWEEN expr2 AND expr3` into `expr1 >= expr2` AND `expr1 <= expr3`. This substituting clause has two virtual terms. If both virtual terms can be evaluated using an index, then the original BETWEEN term is dropped and the corresponding test is not performed on input rows. Instead, SQLite performs a range search on the index with lower and upper bounds set by `expr2` and `expr3`, respectively. Otherwise, the virtual terms may be used as hints for index selection and the original clause is evaluated for each row; the `expr1` expression is only evaluated once.

8.5.4 OR clause

If a term consists of multiple subterms containing a common column name and separated by OR, such as this: `column = expr1 OR expr2 = column OR column = expr3 OR ...`, then SQLite rewrites the term as `column IN (expr1, expr2, expr3, ...)`. Note that the specified column must be the same column in every OR-connected subterm, although the column can occur on either the left or the right side of the = operator. If there is an index for the column, that index is used.

Otherwise, SQLite uses a different optimization strategy where each individual term (as if it is an entire where clause) is analyzed to see if some index can be used effectively. If all terms can be individually satisfied by indexes, the selected rows are combined and redundancies are eliminated. In the worst case, it performs a full table scan.

8.5.5 LIKE or GLOB clause

These two are pattern matching operators. The GLOB operator is always case sensitive. But, the LIKE operator has two modes that can be set by a pragma. The default mode for LIKE comparisons is insensitive to differences of case for Latin 1 characters — basically the upper and lower case letters of the English language in the lower 127 ASCII codes. Thus, by default, the following expression is true: 'a' LIKE 'A'. You can make LIKE operator case sensitive by setting pragma variable `case_sensitive_like` to 1 or by building SQLite library with default like-case-sensitive. International characters are always case sensitive unless a user supplied collating sequence uses differently.

Terms that are composed of the LIKE or GLOB operator can sometimes be used to constrain indexes. There are many conditions on this issue:

1. The left-hand side of the LIKE or GLOB operator must be the name of an indexed column with text affinity.

2. The right-hand side of the LIKE or GLOB must be a string literal or parameter bound to a string literal that does not begin with a wildcard character.

3. The ESCAPE clause cannot appear on the LIKE operator.

4. The build-in functions used to implement LIKE and GLOB must not have been overloaded using the `sqlite3_create_function` API.

5. For the GLOB operator, the column must use the default BINARY collating sequence.

6. For the LIKE operator, if `case_sensitive_like` mode is enabled then the column must use the default BINARY collating sequence, or if `case_sensitive_like` mode is disabled then the column must use the built-in NOCASE collating sequence.

8.5.6 Join table ordering

The current implementation of SQLite uses only loop joins: it is always executed as a series of nested loops. The order of the tables in the FROM clause determines the nesting of the loops. The first table in the FROM clause becomes the outer most loop and the last table becomes the inner

most loop and all intermediate tables appear in the specified order as middle loops. For example, a two table join is implemented as follows: for each row in the outer table, scan the entire table on the inner loop and produce output rows. When there is a WHERE clause with IN operator, it might result in additional nested loops for scanning through all values on the right-hand side of the IN.

The above paragraph explains the default order of nested loops. But, under circumstances, SQLite though may nest the loops in a different order if doing so helps it to select better indexes, resulting in a faster query evaluation. It uses a greedy algorithm to determine loop nesting order. When developing a query plan, the optimizer looks first for a table in the join that requires the least amount of work to be processed. This table becomes the outer loop. Then it looks for the next easiest table to process which becomes the next loop, and so forth. In the event of a tie, the specified order of the tables in the original FROM clause becomes the tie breaker. To determine how much work is required to process a table, it considers many factors such as the availability of indexes on particular columns, whether or not the use of indexes will obviate the need to sort and how selective an index is. Some indexes might reduce a search from a million rows down to a few thousand rows. Other indexes might reduce the search down to a few or a single row. The latter indexes would be preferred, of course. The purpose of the ANALYZE command (see Section 8.5.1) is to gather statistics on how selective an index is so that SQLite can make an informed guess about which indexes will reduce the search to thousands of rows versus one or two rows. If there is an index on the join columns on a table, SQLite can make it the inner loop and exploit the index. It is called index nested loop join.

Note: When selecting the order of tables in a join, SQLite uses a greedy algorithm that runs in polynomial time. If ON and USING clauses are used in an inner join, they are converted into additional terms of the WHERE clause before analyzing the WHERE clause. ◁

Inner joins can be freely reordered. As a left outer join is neither commutative nor associative, and SQLite does not reorder the tables. Tables in inner joins to the left and right of the outer join might be reordered if the optimizer thinks that is advantageous but the outer joins are always evaluated in the order in which they occur.

Nesting table reordering is automatic and usually works well enough that programmers do not have to think about it. But occasionally some hints from them are appreciated. You can force SQLite by using the 'cross' keyword in the from clause such as `table1 cross join table2`. In this case table1 is the outer loop and table2 is the inner loop.

8.5.7 Index selection

For each table in the FROM clause of a query, SQLite can use at most one index; exception is the or-clause where it can use multiple indexes. And, it strives to use at least one index on each table. When there are more than one index is available for potential use for the table, SQLite chooses one of them based on some heuristics. For example, consider the following example.

CREATE TABLE table1(x, y, z);

CREATE INDEX i1 ON table1(x);

CREATE INDEX i2 ON table1(y);

SELECT z FROM table1 WHERE x = 5 AND y = 6;

For the example SELECT statement and the database schema, SQLite can use the i1 index to lookup rows of table table1 that contain value 5 in column x and then test each matching row against the y = 6 term. Alternatively, it can use the i2 index to lookup rows of table1 that contain value 6 in column y and then test each matching row against the x = 5 term. In such situations when two or more indexes on the same table are candidate indexes, SQLite tries to estimate the total amount of work needed to perform the query using each individual index and chooses the index that gives the least estimated work. If `sqlite_stat1` catalog is available, SQLite digs out some information from the catalog to make a better decision on the index selection. You may recall that the `stat1` catalog stores information about how many rows, on the average, are expected for a column value. The index that gives the lowest expected number may be chosen.

SQLite gives application developers a choice in formulating queries in such cases. They can manually disqualify an index by prepending a unary + operator to the column name. The unary + is a no-op, but it will prevent the term from constraining an index. So, in the example above, they can rewrite the query as: SELECT z FROM table1 WHERE +x = 5 AND y = 6. The + operator on the x column would prevent that term from constraining an index on column x. This would force the optimizer to use the i2 index.

8.5.8 ORDER BY

SQLite attempts to use an index to satisfy the ORDER BY clause of a query whenever possible. When faced with the choice of using an index to satisfy WHERE clause constraints or satisfying an ORDER BY clause, SQLite does the same work analysis described in Section 8.5.7 and chooses the index that it believes will result in the answer fastest.

If the resulting records cannot be read out in the 'ORDER BY' order with the aid of an index, all records are loaded into a sorter, are sorted in the expected order, and then read out in the

specified order. SQLite uses transient indexes for all such sorting purposes, and thus have a high memory overhead. The sorting template is something like this.

```
open a sorter
where-begin
    extract columns
    bundle columns into a record
    create sort key
    add key and record to the sorter
where-end
sort
foreach sort element
    extract columns
    send result to caller
end-foreach
close the sorter
```

8.5.9 GROUP BY

Aggregate queries use an "aggregator" which is a special table with records for both key and data values. The GROUP BY terms form the key and other terms form the data. For each row, lookup the record using the GROUP BY terms, and then process them accordingly. Here is the groupby template.

```
where-begin
    compute group-by key
    focus on the group-by key
    update aggregate terms
where-end
foreach group-by
    compute result-set
    send result to caller
end-foreach
```

8.5.10 Subquery flattening

When a subquery occurs in the FROM clause of a SELECT query, the default way of evaluating the query is to execute the subquery first, store the results in a temporary table, run the outer

query on that temporary table, and finally delete the temporary table. This is problematic since the temporary table will not have any indexes and the outer query (which is very much likely a join with a where clause) will be forced to do a full table scan on the temporary table. That is, it requires two passes over the data, and thereby slows down the query processing speed. To overcome this problem, SQLite attempts to flatten subqueries in the FROM clause of a SELECT. This involves inserting the FROM clause of the subquery into the FROM clause of the outer query and rewriting expressions in the outer query that refer to the result set of the subquery.

To understand the concept of flattening a little better, let us study the following SELECT statement: `SELECT a FROM (SELECT x + y AS a FROM t1 WHERE z<100) WHERE a>5`. This statement would be rewritten using query flattening techniques as: `SELECT x + y AS a FROM t1 WHERE z<100 AND a>5`. The code generated for this simplification gives the same result but only has to scan the data once. And, because indexes might exist on the table t1, a complete table scan of the data might not be really necessary.

There is a long list of conditions that must all be met in order for query flattening to occur. Flattening is only attempted if all of the following conditions are true. The latest list can be obtained from SQLite homepage.

1. The subquery and the outer query do not both use aggregates.
2. The subquery is not an aggregate or the outer query is not a join.
3. The subquery is not the right operand of a left outer join.
4. The subquery is not DISTINCT or the outer query is not a join.
5. The subquery is not DISTINCT or the outer query does not use aggregates.
6. The subquery does not use aggregates or the outer query is not DISTINCT.
7. The subquery has a FROM clause.
8. The subquery does not use LIMIT or the outer query is not a join.
9. The subquery does not use LIMIT or the outer query does not use aggregates.
10. The subquery does not use aggregates or the outer query does not use LIMIT.
11. The subquery and the outer query do not both have ORDER BY clauses.
12. The subquery and outer query do not both use LIMIT.
13. The subquery does not use OFFSET.

14. The outer query is not part of a compound select or the subquery does not have both an ORDER BY and a LIMIT clause.

15. The outer query is not an aggregate or the subquery does not contain ORDER BY.

16. The sub-query is not a compound select, or it is a UNION ALL compound clause made up entirely of non-aggregate queries, and the parent query:

    - is not itself part of a compound select,
    - is not an aggregate or DISTINCT query, and
    - has no other tables or sub-selects in the FROM clause.

    The parent and sub-query may contain WHERE clauses. Subject to rules (11), (12) and (13), they may also contain ORDER BY, LIMIT and OFFSET clauses.

17. If the sub-query is a compound select, then all terms of the ORDER by clause of the parent must be simple references to columns of the sub-query.

18. The subquery does not use LIMIT or the outer query does not have a WHERE clause.

19. If the sub-query is a compound select, then it must not use an ORDER BY clause.

Query flattening is an important optimization when views are used because each use of a view is translated into a subquery.

8.5.11 Min/Max

Consider queries of the forms SELECT MIN(col) FROM table1 and SELECT MAX(col) FROM table1. If there is no index on the col column, we need to do a table scan. However, if there is an index on the col column, SQLite evaluates the queries in logarithmic time on the size of the table. Basically it finds the first or the last entry in the chosen index B-tree or if the column is INTEGER PRIMARY KEY, the table B^+-tree.

Summary

This chapter discusses the functionalities of the SQLite's frontend system. This consists of tokenizer, parser, optimizer, and code generator. It accepts SQL statements from the application and converts them into equivalent bytecode programs that the VM executes to produce the desired output.

Each input SQL statement is first given to the tokenizer that splits the statement into distinct tokens and send them one by one to the parser. (Unlike YACC/BISON, in SQLite the tokenizer

drives the parser.) The parser analyzes the tokens and checks the syntax of the original SQL statement, and forms a kind of parse tree out of the tokens. The optimizer and code generator work hand-in-hand to produce a bytecode program out of the parse tree. All these magic happen in implementing the `sqlite3_prepare` API function.

Query optimization is a delicate issue in any SQL database system. It is a computation intensive task. Most practical database systems use some heuristics to achieve near optimal performance. SQLite employs some simple optimization schemes. For queries with a where clause, SQLite tries to find the best indexes on tables to avoid accessing all rows from all tables. This chapter briefly discusses these schemes.

Chapter 9

SQLite Interface Handler

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- how all SQLite internal structures are interlinked with the main `sqlite3` structure

Chapter synopsis

In the previous chapters I have presented many control data structures in piecemeal that are used by various modules. In this chapter I present the main user interface structure, namely `sqlite3`, and its interrelation with other internal control data structures. This chapter provides you a complete end-to-end picture of SQLite's internal organizations of various data structures.

9.1 The Importance of Interface

As mentioned in a previous chapter, an *interface* is a contract between a system and its environment, i.e., the system users. It specifies how the system would interact with the users. In software, it is a named collection of functions and constant declarations. It also defines a protocol of communications between the users and the system, and defines behaviors for these functions. It describes the input assumptions the system makes on the environment and the output guarantees it provides. It is a mechanism that unrelated entities use to interact with one other. An *implementation* of an interface constructs all the functions declared in the interface specification. The purpose of an interface is to minimize dependencies between applications that use the interface functions and service providers that implement the interface.

After some experiments, the SQLite development team has finalized most of its interface functions and constants. They occasionally add experimental interface functions and constants though

toward adding new features. These newly added interface functions are subjected to change in later releases. Apart from functions, SQLite uses a few control structures. Control structures are subjected to continual change, but these changes do not affect SQLite applications. The main structure is `sqlite3` that I discuss in the next section.

9.2 The sqlite3 Structure

When an application invokes the `sqlite3_open` API function, the function creates and sets up a new library connection or session, and it also creates and/or opens a database file for the application. The function creates an object of type `sqlite3` and returns the application a pointer to the object. The pointer represents the new library connection for the application, and the `sqlite3` object for the SQLite library. The application must not tamper with any component member variables of the object, and it uses the pointer as a handle in further invocation of various API functions on the connection until the application applies the `sqlite3_close` API function successfully on the connection. We say the connection is closed, and the handle is gone.

Various components of an `sqlite3` object are given in Fig. 9.1. The `aDb` member of `sqlite3` is an array of objects of type `Db`. Each `Db` object represents an instance of open database file, aka, database connection. There are normally two `Db` objects on the `aDb` array. The aDb[0] is the *main* database, and aDb[1] the *temp* database. The aDb[0] object represents the database whose name has been passed down to the `sqlite3_open` API function by the application. Upon the termination of the `sqlite3_open` function call, these two Db objects are appropriately initialized. The other Db objects, if any, are created later, one for each attached database. (They are destroyed when the application detaches them from the connection.) Internally, the VM refers to each database by an index number to the `aDb` array, and not by the database name. The name to index translation is done at the code generation time. Index value 0 (respectively, 1) always refers to the main (respectively, temp) database; values greater than 1 refer to attached databases. The nDb member variable of the `sqlite3` object indicates the number of currently open databases on the library connection.

Each `Db` object has the following member variables: (1) `zName` — a pointer to the name of the database; (2) `pBt` — a pointer to one `Btree` object that is used as a handle to apply tree level functions on the database; (3) `inTrans` — type of current transaction on the database; (4) `pSchema` — a pointer to a schema object. The schema object has the following member variables: (1) `schema_cookie` — database schema version number; (2) `cache_size` — the number of pages to use in the page cache; (3) `tblHash` — for all tables; (4) `idxHash` — for all indexes; (5) `trigHash` — for all triggers; (6) `fkeyHash` — for all foreign keys; (7) `pSeqTab` — a pointer to the `sqlite_sequence` catalog; and (8) many others. (When SQLite reads a database file, it parses the

Member	Description
aDb	An array of Db objects
nDb	Number of aDb entries in use
pVdbe	List of active Vdbe objects
activeVdbeCnt	Number of Vdbes currently executing
lastRowid	ROWID of most recent insert
xCollNeeded	A collation factory
aCollSeq	All collating sequences
busyHandler	Busy callback resolution
errCode	Most recent error code
pErr	Most recent error message
flags	Various runtime flags
autoCommit	The auto-commit flag for this connection
....	Other variables

Figure 9.1: Components of **sqlite3** objects.

schema and populates the four internal hash tables, **tblHash**, **idxHash**, **trigHash**, and **fkeyHash**, respectively, for SQL tables/views, SQL indexes, triggers, and foreign keys that are defined in that database.)

Each SQL table is represented in-memory by an instance of **Table** object (see Fig. 9.2). The **aCol** is an array of objects representing **nCol** columns of the table. Each column is represented by an instance of **Column** object. This object has the following fields: (1) **zName** — the name of this column; (2) **pDflt** — default value of this column; (3) **zType** — SQL type for this column; (4) **notNull** — true if there is a NOT NULL constraint; (5) **isPrimKey** — true if this column is part of the PRIMARY KEY; and (6) many others.

Member	Description
zName	Name of the table.
nCol	Number of columns in this table.
aCol	Information about individual columns.
iPKey	If not less then 0, use aCol[iPKey] as the primary key.
pIndex	List of SQL indexes on this table.
tnum	Root page number for this table.
pSelect	NULL for tables; Points to definition if a view.
pSchema	Schema that contains this table.
....	Other variables

Figure 9.2: Components of **Table** objects.

Each index is represented in-memory by an instance of **Index** object (see Fig. 9.3). The **aiColumn** is an array of **nColumn** integers, where each integer identifies a column in the base table; the first column is 0. The **tnum** is the page number where the root of the index resides.

Member	Description
zName	Name of the index.
nColumn	Number of columns from the base table used in this index.
aiColumn	Which columns from the base table are used.
pTable	The base table on which this index is.
tnum	Root page number for this index.
autoIndex	Whether SQLite created or user created this index.
pSchema	Schema that contains this index.
....	Other variables

Figure 9.3: Components of `Index` objects.

The `lastRowid` field in an `sqlite3` object records the last inserted rowid generated by an insert statement. (Inserts on views do not affect the `lastRowid` value.) The `errCode` and `pErr` store the most recent error code and, if applicable, error string, respectively. The `flags` encode various runtime state of the `sqlite3` object. The `pVdbe` is a bucket of `Vdbe` objects, each represents a separately compiled SQL statement, aka, a bytecode program, on the library connection. Each `Vdbe` object is directly referenced by a `sqlite3_stmt` pointer at the user application.

9.3 The Final Configuration

The interlinks between an application and SQLite data structures are pictorially shown in Fig. 9.4. Applications apply SQLite API functions on the `sqlite3*` and `sqlite3_stmt*` pointers. You may note that SQLite allows applications to register various user defined callback functions with it. SQLite, if needs arise, executes these functions to manipulate data in the application space.

Not shown in Fig. 9.4, an application can have multiple library connections (aka, `sqlite3*`), and they each can have multiple open connections to the same or different databases. That is, a library connection can have multiple databases (main, temp, and other attached ones) associated with it; the Db array in Fig. 9.4. Each such database is accessed via a dedicated Btree object and the object has its own Pager object. (In the shared caching mode, the Pager object is shared by multiple Btree objects created for the same database file via the BtShared object.) The Pager object tracks down the state of the database file, journal file, lock, page cache, etc. The application can have at most one active transaction on a library connection at any one time. And, depending on the need, the library connection starts transactions on individual databases.

Each SQL statement (aka, `sqlite3_stmt*`) has its own bytecode program. When executed, the VM opens cursors via which it accesses databases; the VdbeCursor array in Fig. 9.4. A cursor refers to one BtCursor to access a single B/B^+-tree in an individual database. One BtCursor references one (table or index) tree, and can iterate over the records of the tree. There can be multiple VdbeCursors on the same tree, and they are independent of one another.

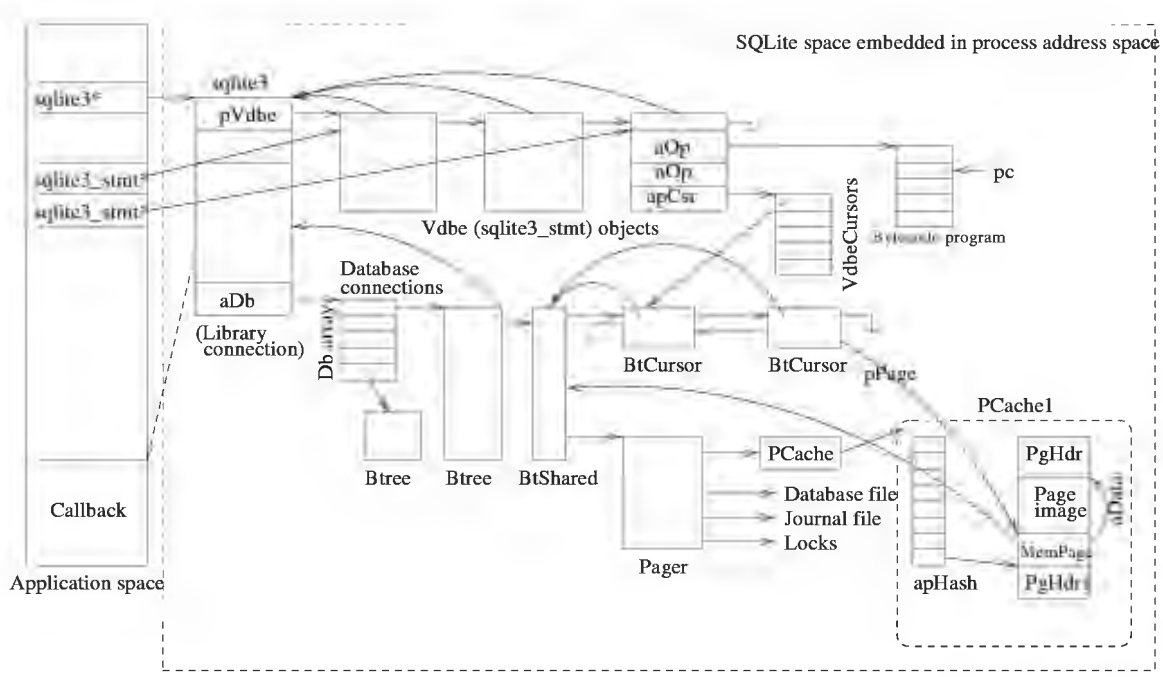


Figure 9.4: Integration of control data structures.

9.4 API Interaction

When an application successfully invokes the `sqlite3_open` API function, SQLite creates an `sqlite3` object (on the heap of the application process address space). The `pVdbe` pointer variable in the object is NULL (i.e, empty). When the application compiles an SQL statement (by invoking the `sqlite3_prepare` API function), SQLite creates a prepared statement object (of type Vdbe) and appends it to the `pVdbe` list. This prepared statement does not have any cursor associated with it. The application invokes the `sqlite3_bind_*` API functions on the object pointer (`sqlite3_stmt*`) to define values for parameters. It then invokes the `sqlite3_step` API function on `sqlite_stmt*` to execute the bytecode program. During the program execution, the VM creates cursors to apply operations on B/B^+-trees in individual databases. On termination of the program execution, the VM closes and deletes those cursors. (If the application calls the `sqlite3_reset` API function on the prepared statement, all its cursors are closed and deleted too.) The application may destroy a `Vdbe` object (along with its allocated resources) by applying the `sqlite3_finalize` API function on it. When it has finalized all prepared statements, it can destroy the `sqlite3` object by applying the `sqlite3_close` API function on it; we say the library connection or session is closed. All SQLite handles associated with the library connection become invalid.

Summary

This chapter gives you an end-to-end picture of how SQLite internal data structures are interlinked together to define the main `sqlite3` interface structure. To open a library connection an application invokes the `sqlite3_open` API function, and this function creates a `sqlite3` object. A pointer to this object is used by the application as a handle to apply further operations on the library connection. This object encompasses all database connections (the main, the temp, and attached ones).

Chapter 10

Advance Features

Scholastic Objectives

After reading this chapter, you should be able to explain/describe:

- various advance SQLite features such as pragma, view, trigger, collation, subquery, autovacuum, etc
- the usage of the `sqlite_sequence` catalog to implement the autoincrement feature
- how one can use `current_date`, `current_time`, and `current_timestamp` in SQL expressions
- how autovacuum feature is implemented
- how unicode data are handled by SQLite
- how the WAL journaling feature is implemented

Chapter synopsis

The minimum features a RDBMS should have are simple insert, delete, and select on a single table, and of course the ACID properties for transactions. With these minimal features, you can devise solutions to most database problems. But, application development may become cumbersome and time consuming. Most RDBMSs also support update operation on a single table and cross-product on two or more tables. SQLite supports them too, and in addition, like many other RDBMSs, it supports many optional advance features. One nice thing with SQLite is that every advance feature is implemented under a compile flag. These features can be individually turned off by setting appropriate flags at the time of building the SQLite library from the source code. Some of them can also be turned off at runtime. Some of these features are discussed in this chapter.

10.1 Pragma

PRAGMAs are special SQLite commands that you can use to query the SQLite library to obtain internal (non-table) metadata, or to modify the default behavior of the library. SQLite supports many pragmas, and each has a different name. A PRAGMA command is sent to SQLite using the same interface as other SQLite command and SQL statements, but it is different in the following important respects.

- SQLite does not generate any error message if you send it an unknown pragma. It ignores unknown pragmas. It means that if you make a typographic error in a pragma command SQLite does not inform you of the error. You need to be very careful in spelling pragma names.
- Some pragma commands take effect during the compilation stage instead of in the execution stage. If your application is using the `sqlite3_prepare`, `sqlite3_step`, `sqlite3_finalize` API function sequence to execute SQL statements, those pragma commands may be applied to the library during the `sqlite3_prepare` call only.
- Pragmas are very much unlikely to be compatible with any other SQL RDBMS. Porting applications to other systems may become difficult.

There are two alternative ways to specify values to pragma names: (1) PRAGMA name = value and (2) PRAGMA name(value); here the name is a pragma identifier, and the value is a string or a number. (In addition, 'PRAGMA name' returns the current value of the pragma identifier indicated by that name.) The name can be prefixed with a database name followed by a dot, e.g., pragma temp.name. The pragmas that take integer values zero and one also accept symbolic names; the strings "on", "true", and "yes" are equivalent to 1, and the strings "off", "false", and "no" are equivalent to 0. These strings are case-insensitive, and do not require quotes. An unrecognized string will be treated as 1, and will not generate an error. For example, consider a pragma command `PRAGMA synchronous = OFF`. This command turns off synchronous logging and database writing by transactions, that is, these become asynchronous transactions. When a value is returned by such a pragma it is always an integer.

All pragmas are classified into four basic categories: (1) pragmas to query the schema of the current database; (2) pragmas to modify the operation of the SQLite library in some manner, or to query for the current mode of operation; (3) pragmas to query or modify two database version values (the schema-version and the user-version); (4) pragmas to debug the library and verify that databases are not corrupted. The currently supported pragmas include auto_vacuum, cache_size, case_sensitive_like, count_changes, default_cache_size, empty_result_callbacks, encoding,

full_column_names, page_size, short_column_names, synchronous, temp_store, temp_store_directory, database_list, foreign_key_list, index_info, index_list, table_info, schema_version, user_version, integrity_check, parser_trace, vdbe_trace, vdbe_listing. We discuss a few sample pragma commands below. The SQLite webpage http://www.sqlite.org/pragma.html has more information on these and other pragmas.

1. Pragmas for querying the database schema.

   - `Pragma foreign_key_list(table-name)`. For each foreign key that references a column in the argument table, this pragma returns one row.

2. Pragmas for modifying the operation of the SQLite library.

   - `Pragma auto_vacuum = 0, 1, or 2` (aka, NONE, FULL, or INCREMENTAL, respectively) sets the value of `auto_vacuum` flag. The default value is 0 or NONE. It is only possible to modify the value of the flag before any table has been created in the database. No error message is returned if an attempt to modify the flag is made after one or more tables have been created. There are some restrictions about this pragma; see the http://www.sqlite.org/pragma.html#pragma_auto_vacuum webpage.
   - `Pragma cache_size = Number-of-pages` sets the page-cache size. When you change the cache size using this pragma, the change is effective for the current database session only. The cache size reverts to the default value when the database connection is closed and reopened. You may use the `default_cache_size` pragma to change the cache size permanently. Once you set the default cache size, the setting value is retained and reused every time you reopen the database.

3. Pragmas for querying or modifying the database versions.

   - `Pragma schema_version = integer value` and `pragma user_version = integer value` set the value of the schema_version and user_version, respectively. Both the schema version and the user version are 32-bit signed integers stored in the database header. The schema version is usually only manipulated internally by SQLite. It is incremented whenever the database schema is modified (by creating, altering, or dropping a table or index). The schema version is used by SQLite each time a query is executed to ensure that the internal cache of the schema used when compiling the SQL query matches the schema of the database against which the compiled query is actually executed. Subverting this mechanism by using `pragma schema_version` to modify the schema-version is potentially dangerous and may lead to program crashes or database corruption. Use this pragma with caution. You have been warned! The user version is not used internally by

SQLite. It may be used by applications for any purpose, for example, to keep track of the version of database file for backup purposes.

4. Pragmas for debugging the library.

   - `Pragma integrity_check`. The command does an integrity check of the entire database. It looks for out-of-order records, missing pages, malformed records, and corrupt indexes. If any problems are found, then a single string is returned which is a description of all problems. If everything is in order, "ok" is returned.

10.2 Subquery

A simple SQL select statement is `select x from y where z`, where `x` is a list of attributes or expressions, `y` a list of tables, and `z` a boolean expression. A powerful feature of SQL is that `z` can itself contain a SQL (sub)query that is another select statement. For example, in `select name from Students where sid in (select sid from Admitted_to where doj='Jan 01, 2000')`, the statement in parentheses is a simple subquery. It is an independent SQL query by itself, and can be executed on its own independently. Such subquery can also occur in the from- and groupby's having-clauses as well. In executing the full query, the subquery is executed only once.

SQLite supports correlated subquery too. A *correlated subquery* is an SQL select statement nested inside another SQL select, in which the nested one contains a reference to one or more columns from the outer select. It is a dependent subquery and cannot be executed on its own. Here is a simple example of correlated query: `select name from Students where exists (select * from Admitted_to where sid = Students.sid and doj = 'Jan 01, 2000')`. Here the nested subquery makes a reference to Students.sid column.

A correlated subquery is dependent on the outer query. The subquery will be executed many times while processing the outer query. The subquery will be run once for each candidate row selected by the outer query. The columns in the outer query that are referenced in the correlated subquery are replaced with values from the candidate row prior to each execution of the subquery. Depending on the outcome of the execution of the subquery, the VM determines whether or not the candidate row is put in the result set of the complete query.

10.3 View

A *view* is a virtual table whose rows are not explicitly stored separately in the database as a distinct existence, and instead it is derived from one or more base tables on demand. That is, views are not persistent tables; their contents are dynamically generated when they are used. It is a virtual table

for which SQLite stores a pure query definition that is used to derive the view's rowset at runtime. Views are normally used to present necessary information to users, while hiding the details in base tables. For query purposes views are almost treated like normal tables. Users often cannot tell whether they are accessing a view or a table. When you apply a query on a view, SQL behaves as if the query is applied on a new temporary table that is derived by the view definition at that point in time.

Views provide some kind of schema independence. If a base table definition is altered (for example, addition of a new column), the view definition may not be affected. Applications that access the view are not affected, but those that access the base table may be.

The SQL construct `create view view1 as select name, sid from Students` is a typical create view statement. The `create view` statement assigns a name to a pre-packaged select statement, and it inserts this row ⟨view, view1, view1, 0, `create view view1 as select name, sid from Students`⟩ into the `sqlite_master` catalog table. Once the view is created, it can be used in the FROM clause of another SELECT in place of a table name. For example, `select name from view1 where sid = 1001` returns the name of the student with `sid` value 1001. The template for a view definition is `create [temp | temporary] view [database-name.] view-name as select-statement`. (One cannot use both temp/temporary and database-name in a view definition unless the database-name itself is temp.)

Though SQL semantics allows updating views, SQLite prohibits doing so. Thus, insert, update, and delete are not applicable on views. SQLite though allows you to create a particular kind of triggers on views via which you can update base tables. See Section 10.5 on page 230.

SQLite permits creation of temporary views by putting the temp or temporary keyword between create and view keywords in the create statement. The temporary views are only visible to the library connection and is automatically deleted when the library connection is closed.

10.4 Autoincrement

In a user table, if you declare a column as INTEGER PRIMARY KEY, the column will autoincrement. What I mean by this is that whenever you insert an SQL NULL value into that column (or do not specify any value for that column) when inserting a row in the table, the NULL or absent value is automatically converted into a (64-bit signed) integer. The integer value is normally one greater than the largest value of that column over all other rows in the table, or 0 if all existing values are negative, or 1 if the table is empty. For example, suppose you have a table that is created by: `create table t1(a INTEGER PRIMARY KEY)`. The statement `insert into t1 values(NULL)` becomes logically equivalent to `insert into t1 values((select max(a) from t1)+1)`, well, in

most of the cases.

Note: The maximum possible value of a rowid is 9,223,372,036,854,775,807. If this value is in the table presently, then SQLite chooses an unused positive value at random. SQLite however makes only a few attempts to choose the value. If all attempts fail, it returns SQLITE_FULL error code. ◁

The new value will definitely be unique over all values currently in the column, but it might be one that was previously deleted from the column. If you want unique values over the lifetime of the table, you need to add the AUTOINCREMENT keyword to the INTEGER PRIMARY KEY declaration, that is, declare the column as INTEGER PRIMARY KEY AUTOINCREMENT. Then the value chosen by SQLite will definitely be one greater than the largest value that has ever existed in that column. If the largest possible value has previously existed in that column, then all new INSERTs will fail with an SQLITE_FULL error code.

SQLite maintains an optional catalog table, named `sqlite_sequence` to keep track of values for tables with the explicitly autoincrement column. The schema of the sequence catalog is as follows: `CREATE TABLE sqlite_sequence(name, seq)`, where `name` is a table name and `seq` is the highest value of the integer primary key ever used for the table. For each such table, the catalog has one row that holds the maximum value ever issued for the autoincrement column in the table. (You may note that a table cannot have more than one autoincrement column.) The sequence table may not exist by default in an SQLite database. The sequence table is created the very first time you insert a row into a table having an autoincrement column. SQLite initializes a row and inserts the row into the sequence catalog whenever a user table with autoincrement column receives the first insert. Subsequent SQL inserts into such a table may require updating the corresponding row in the sequence table. When the user table is deleted, the corresponding row from the sequence catalog is removed too. SQLite allows you to modify the content of the sequence catalog using ordinary UPDATE, INSERT, and DELETE statements. But making modifications to this table will likely perturb the AUTOINCREMENT value generation algorithm. You have been warned!

10.5 Trigger

A *trigger* is a procedure that is automatically executed by the DBMS when some event such as a specified change has occurred in the database. For example, in a university database, if a student fails in two or more courses in a semester, a termination order is initiated behind the scene.

An SQL trigger statement has three parts: (1) event, (2) condition, and (3) action. The event identifies an SQL (insert, delete, or update) statement execution before or after which the trigger becomes effective. We say the event activates the trigger. On every such activation, the DBMS evaluates the condition of the trigger, and if the condition is satisfied, it starts executing the trigger

action. A trigger action consists of one or more SQL select, insert, delete, and update statements, and they can refer to both old and new values of tuples to carry out the action.

The `create trigger` statement is used to add triggers to the database schema. The template for a trigger statement is the following: `create [temp | temporary] trigger [database-name.]` *`trigger-name`* `[ before | after | instead of]` *`database-event`* `on table-name` *`trigger-action`*. (The trigger is automatically dropped when the table is dropped.) The *trigger-name* is the name of the trigger that is activated either before or after the occurrence of the specified *database-event*. A trigger may be specified to fire whenever a DELETE, INSERT, or UPDATE on a particular database table occurs, or whenever an UPDATE on one or more particular columns of a table occurs.

There are two standard forms of trigger actions: FOR EACH ROW and FOR EACH STATEMENT. SQLite supports only FOR EACH ROW triggers as of SQLite 3.7.8 release. (Hence explicitly specifying FOR EACH ROW is optional.) The FOR EACH ROW clause is followed by an optional WHEN clause. The action of a trigger is a sequence of select, update, insert, and delete SQL statements surrounded by keywords `begin` and `end`;[1] each of these action statements is called a *trigger-step*. A trigger action must have at least one trigger-step. FOR EACH ROW implies that the trigger-steps may be executed (depending on the WHEN clause) for each database row being inserted, updated, or deleted by the statement causing the trigger to fire. Both the WHEN clause and the trigger-steps may access elements of the row being inserted, deleted, or updated using references of the form "NEW.column-name" and "OLD.column-name", where column-name is the name of a column from the table that the trigger is associated with. OLD and NEW references may only be used in triggers on trigger-events for which they are relevant, as follows:

- INSERT: only NEW references are valid,
- UPDATE: NEW and OLD references are valid,
- DELETE: only OLD references are valid.

If a WHEN clause is supplied, the trigger-steps are only executed for those rows for which the WHEN clause is true. If no WHEN clause is supplied, the trigger-steps are executed for all rows. The specified trigger-time (before or after) determines when the trigger-steps will be executed relative to the insertion, updating, or deletion of the associated row.

Triggers may be created on views, as well as on regular tables, by specifying INSTEAD OF in the CREATE TRIGGER statement. Although SQLite does not permit updating views, but if one or more INSERT, DELETE, or UPDATE triggers are defined on a view, then it is not an error to

[1] There are some minor restrictions on what queries can be in a trigger action. See http://www.sqlite.org/lang_createtrigger.html page for details.

execute an INSERT, DELETE, or UPDATE statement on the view, respectively; executing them on the view causes the associated triggers to fire. The base tables underlying the view are not modified (except possibly explicitly, by a trigger program).

Suppose you have a 'customers' table in which customer information is stored, and an 'orders' table in which order status information is stored. The following trigger ensures that all associated orders are redirected when a customer changes her address:

```
CREATE TRIGGER update_customer_address UPDATE OF address ON customers
BEGIN
    UPDATE orders SET address = new.address WHERE customer_name = old.name;
END;
```

With this trigger available in the database, when you execute statements such as `update customers set address = '364 Olive Avenue' where name = 'Sibsankar'`, the trigger is fired and it executes `update orders set address = '364 Olive Avenue' where customer_name = 'Sibsankar'` behind the scene.

You may note that presently, triggers may behave oddly when created on tables with INTEGER PRIMARY KEY fields. If a BEFORE trigger program modifies the INTEGER PRIMARY KEY field of a row that will be subsequently updated by the statement that causes the trigger to fire, then the update may not occur. To circumvent this, you need to declare the table with a PRIMARY KEY column instead of an INTEGER PRIMARY KEY column.

Recursive Trigger: In the default setting, SQLite does not support recursive triggers, but supports cascading triggers. That is, actions performed by a trigger do not cause the same trigger to fire though it may fire another trigger. You can though activate recursive triggers by using the recursive_triggers pragma. ◁

10.6 Date, Time, and Timestamp

SQLite processes all dates and times as Julian Day numbers that are real numbers representing the number of days. It stores the dates and times as the number of days elapsed since the noon in the Greenwich on November 24, 4714 B.C., according to the Gregorian calendar system. That particular time instant is Julian Day 0.0. Negative and positive numbers represent date before and after that time instant, respectively.

The date/time conversion algorithms SQLite implements are based on descriptions found in [16]. SQLite supports five date and time related functions: `julianday`, `date`, `time`, `datetime`, and `strftime`. These five functions take a time string as an argument; the time string may be followed

by zero or more modifiers. The `strftime` function in addition takes a format string as its first argument. The `date` function returns the date in this format: YYYY-MM-DD. The `time` function returns the time as HH:MM:SS. The `datetime` function returns YYYY-MM-DD HH:MM:SS. The `julianday` function returns the number of days since noon in Greenwich on November 24, 4714 B.C. The `strftime` routine returns the date formatted according to the format string specified as the first argument (see Linux manpage for strftime). The input time string to the five functions can be one of the following formats.

1. YYYY-MM-DD
2. YYYY-MM-DD HH:MM
3. YYYY-MM-DD HH:MM:SS
4. YYYY-MM-DD HH:MM:SS.SSS
5. YYYY-MM-DD**T**HH:MM
6. YYYY-MM-DD**T**HH:MM:SS
7. YYYY-MM-DD**T**HH:MM:SS.SSS
8. HH:MM
9. HH:MM:SS
10. HH:MM:SS.SSS
11. now
12. DDDD.DDDD

In Formats 5–7, the 'T' is a literal character separating the date and the time (see the ISO-8601 standard). Formats 8–10 that specify only a time relative to the default date of 2000-01-01. Format 11, the string 'now', is converted into the current date and time: Universal Coordinated Time (UTC) is used to estimate the date. Format 12 is the Julian Day number expressed as a floating point value.

In the five date/time functions, the time string argument can be followed by zero or more modifiers that alter the date or alter the interpretation of the date. The modifiers are applied in the left to right order of their occurrence in the input. The available modifiers are as follows.

1. NNN days
2. NNN hours

3. NNN minutes
4. NNN.NNNN seconds
5. NNN months
6. NNN years
7. start of month
8. start of year
9. start of week
10. start of day
11. weekday N
12. unixepoch
13. localtime
14. utc

The size modifiers (Items 1-6) add the given amount of time (±NNN) to the date specified by the given time string. The 'start of' modifiers (Items 7–10) shift the date backward to the beginning of the current month, year, or day. The 'weekday' modifier (Item 11) advances the date forward to the next date where the weekday number is N. (Sunday is 0, Monday is 1, and so forth.) The 'unixepoch' modifier (Item 12) only works if it immediately follows a time string in the DDDD.DDDDD format. This modifier causes the DDDD.DDDDD to be interpreted not as a Julian Day number as it is normally done, but as the number of seconds since 1970. This modifier allows Unix-based times to be converted to Julian Day numbers easily. The 'localtime' modifier (Item 13) adjusts the given time string so that it displays the correct local time. The 'utc' modifier (Item 14) undoes this.

The above mentioned five date and time functions are optional and can be omitted altogether at the SQLite source code compilation time. SQLite supports at the SQL level three special date and time features that are though always available: `current_date`, `current_time`, and `current_timestamp`. These three identifiers can be used in SQL expressions or to specify default values for table columns. The value is the then UTC date and/or time. For `current_time`, the format is HH:MM:SS, for `current_date` YYYY-MM-DD, and for `current_timestamp` YYYY-MM-DD HH:MM:SS.

SQLite webpage http://www.sqlite.org/cvstrac/wiki?p=DateAndTimeFunctions has more about date and time functions. A few examples of date/time functions are given below.

1. `select date(0)` returns -4713-11-24.
2. `select date('now')` returns the current date in YYYY-MM-DD format.
3. `select date('now','start of month','+1 month','-1 day')` returns the last day of the current month.
4. `select datetime(0)` returns -4713-11-24 12:00:00. This date and time are represented by Julianday 0 according to the Gregorian calendar system.
5. `select datetime(1092941466, 'unixepoch')` returns the date and time for the given Unix timestamp 1092941466.
6. `select julianday('2000-01-01 00:00:00')` return 2451544.5.
7. `select current_timestamp` returns the current date and time.
8. `create table t1(a int, b text not null default current_date)` indicates that NULL or omitted values for the `b` column will be converted to the then value of `current_date`.

Storage Types for Date, Time, and Timestamp: There are no separate storage datatypes for date, time, and timestamps. Depending on the situations, their values are stored as TEXT (ISO8601 string in the "YYYY-MM-DD HH:MM:SS.SSSS" format), INTEGER (Unix time), or REAL (Julian day number). SQLite uses built-in date and time functions to convert date/time/timestamp appropriately. ◁

10.7 Reindex

Reindex is an SQLite command that is used to delete and immediately recreate indexes from scratch in one operation. This command is very useful when the definition of a collation sequence changes. There are three forms of the reindex command. (1) `Reindex collation-name` recreates all indexes in all (including attached) databases that use the named collation sequence directly or via the base table. (2) `Reindex table-name` recreates all indexes associated with the specified table. (3) `Reindex index-name` recreates only the specified index.

10.8 Autovacuum

The default database operating mode is non autovacuum, as far as the management of free database pages is concerned. When a transaction that releases pages from active use commits, the database file remains at the same size at the end of the commit processing. Unused database pages remain on the freelist and are reused later when new data is inserted into the database. All free pages are linked together in a single trunked tree that originates in the file header record at offset 32 (see

Fig. 3.5 on page 89). In the default operating mode, you need to explicitly execute the `vacuum` command to purge the freelist and to shrink the database file, and to release those free pages back to the native file system. You may note that the `vacuum` command cannot be executed in a user transaction, that is, inside begin and end commands. It is called *manual vacuum.*

SQLite has a special feature, called *autovacuum*, that can be turned on at the source code compilation time, or at runtime through a pragma command.[2] In the autovacuum mode, if a transaction frees up some pages from active use (in SQL delete or update), SQLite returns those or those many pages back to the native file system at the time of committing the transaction. During the execution of the transaction, if it has freed up any pages, the pages are included in the normal freelist. At the very beginning of the commit processing, an equivalent amount of space from the file is released to the native file system. The `vacuum` command is not useful in a database with the autovacuum flag set.

As of now, no native file system releases space from the middle of a file and consequently, SQLite cannot release arbitrary freed up pages back to the native file system. The database file can only shrink. Thus, the pages that can be released to the native file system always have to come from the tail end of the file, and SQLite may need to relocate some valid pages from the tail end before releasing them to the native file system. Relocation is special kind of compaction where SQLite swaps valid database pages with free pages. You may note that there are other database pages that may refer to relocating pages, and hence the relocation may require updating content in other pages that hold pointers to the relocated pages. SQLite stores the page relocation information in the file itself, in separate pages. (To support this feature the database stores additional information internally in separate pages, resulting in a slightly larger database file.) These are called *pointer-map* pages. They intersperse with other database pages. The purpose of the pointer map is to facilitate moving pages from one position to another in the file as part of autovacuum. When a page is moved, the pointer to it in its parent must be updated to point to the new location. (You may recall that in a regular tree page, SQLite stores information of child pages and overflow pages, and not the other way. A pointer-map page is used to hold this 'other way' linkage information, from the child to the parent. The pointer-map is used to locate the parent page fast.)

The pointer-map pages collectively define a single lookup data structure that identifies the parent page for each child page in the database file. (The parent page is the page that contains a pointer to the child.) The beauty of SQLite design is that every page in the database has at most one parent page. (In this context 'database page' refers to any page that is not part of the pointer-map itself.) A pointer-map page contains an array of 5-byte entries. Each entry consists of a 1-byte 'type' and a 4-byte parent page number. The following are valid types.

[2] In either case, the autovacuum flag in the file header at offset 52 is set to 1.

1. PTRMAP_ROOTPAGE: The database page is the rootpage of a tree. The parent page-number is not used in this case, because a root does not have a parent. (You may recall that except Page 1 all other rootpage numbers are stored in the master catalog.) The parent page number should be 0.

2. PTRMAP_BTREE: The database page is a non-root tree page. The parent page-number identifies the parent page in the tree.

3. PTRMAP_OVERFLOW1: The database page is the first page in a list of overflow pages. The parent-page number identifies the page that contains the cell with a pointer to this overflow page.

4. PTRMAP_OVERFLOW2: The database page is the second or later page in a list of overflow pages. The parent page-number identifies the previous page in the overflow page list.

5. PTRMAP_FREEPAGE: The database page is an unused (free) page. The parent page-number is not used in this case, and the number should be 0.

Figure 10.1 displays positions of pointer-map pages in a typical database file, having a total of 415 pages. You may note that Page 1 is special, and it is not a part of the pointer-map. The rest of the pages are organized in simple (pointer-map) segments. All segments are of the same size except the last one that can have fewer pages. Each segment begins with a pointer-map page followed by n (= usable space on the page divided by 5) database pages whose mapping information is stored on the pointer-map page. Shown in the figure, the boxes in bold are the pointer-map pages. Suppose the page size is 1024 bytes and all bytes are usable; a pointer-map page can have a maximum of 1024/5 (= 204) map entries. Shown in the figure Page 2 is a pointer-map page that stores mapping information for Pages 3, 4, ..., 206. Given a database page number, SQLite defines two macros, namely PTRMAP_PAGENO and PTRMAP_PTROFFSET, that determines the pointer-map page number and an offset in the page, respectively, where the parental information for the given database page is stored. The PTRMAP_PAGENO macro is actually implemented by the `ptrmapPageno` function in the btree.c source file.

Note: In case a pointer-map page clashes with the lock-byte page, that particular pointer-map page is stored in the page following the lock-byte page. ◁

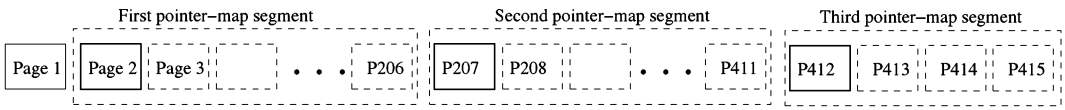


Figure 10.1: Positions of pointer-map pages in database file.

During the autovacuum, as long as there is a free page, SQLite does the following. Suppose

a page i is free, where $i > 1$ and $i \leq m$, m is the max page number. We may need to relocate the content of page m to page i. If m is a pointer-map page or a free page, remove the page from the file and there is no need of relocating the page. Otherwise, SQLite performs the following actions. Let PTRMAP_PAGENO(m) be j. That is, page j contains the pointer-map information for page m at offset PTRMAP_PTROFFSET(m). The parental information for page m is available there. If it is a rootpage, we cannot vacuum the file any more because SQLite does not relocate root pages. (See the note below.) Otherwise, we get the parent information from the pointer-map page j, and update the child pointer in the parent page accordingly. In either case, SQLite updates the pointer-map information for page i that now holds replica of the current page m. Also, update the pointer map information for all those pages that page i points to now. Remove page m from the file.

Note: When this feature is on, all B-/B$^+$-tree root pages are stored preceding all non root pages, overflow pages, and free pages. This ensures that root pages are not relocated during autovacuum. ◁

10.9 Attach and Detach

An application normally works on a single database file. It opens a connection to the SQLite library by specifying the database file name. If needs arise, you can add another one or more database files to a library connection by executing the SQLite command `attach database-filename AS database-name`, and remove the attached files from the library connection by executing the `detach database-name` command.[3] If a file being attached is non existent, it is automatically created by the attach command. The attached files are automatically detached when the library connection is closed.

SQLite permits you attaching the same database file multiple times as different database names (provided the system is not running in the shared cache mode). But, it is an error to attach many (same or different) database files with the same database name. There is a maximum limit of 62 on the number of attached database files. (The value is set in MAX_ATTACHED compile macro; the default is 10.) There is a restriction that attach and detach commands cannot be executed in a user transaction; that is, the main database must be in the autocommit mode. The default transaction recovery mode on attached databases is always synchronous even when the main database is set to asynchronous. While operational, each attached database has its own `Btree` object through which the VM accesses the database. More importantly, in the default operating mode, attached databases do not share a single page cache even if they are the same database file.

[3]If the file name contains punctuation characters it must be quoted. The database-name can be any alphanumeric string except 'main' and 'temp' that refer, respectively, to the main database connection and the database used for temporary tables. These two databases cannot be manually detached from the library connection.

The attach command allows you to work with multiple independent databases and use them together in the same query. (And, the command helps you avoid transferring data between tables from multiple databases manually.) You can read from and write to an attached database and can modify the schema of the attached database. At the SQL level, SQLite does not recognize database file names, and you need to specify the attached database names. Tables in an attached database can be explicitly referred to using the syntax `database-name.table-name`. But, in some cases the table can be implicitly referred to without providing the prefix database name. If for an attached table there is no duplicate in the main, temp, or other previously attached databases, then the table does not require a database name prefix qualifier. When a database is attached, all of its tables which do not have duplicate names (in other databases) become the 'default' table with those names. Any tables of those names attached afterwards require the database-name prefix. If the 'default' table of a given name is detached, then the first table of that name attached becomes the new default table. Having said that, SQLite follows the following search rule to resolve an implicit table name. The search order is the TEMP first, the MAIN next, and then any auxiliary databases added using the ATTACH command in the order of their attachment. This is a first find algorithm, that is, the first matching table is used and no checking for duplicate table names is done. (Application developers have been warned! Prefixing database name to the table name is always the best option.) The same rule is applicable for resolving other schema names.

When you execute a non temporary create table statement without a database name prefix, the table by default is created in the main database. You can create a new table in any attached database by providing the database name as prefix, for example, the `create table DB1.T1(a, b, c)` statement creates T1 table in the DB1 database. There are no parallel temporary databases for individual attachments.

Transactions involving multiple attached databases are atomic, assuming that the main database is not ':memory:' (nor a temporary file). If the main database is ':memory:' then transactions continue to be atomic within each individual database file. But if the application or the host computer crashes in the middle of a COMMIT where two or more database files are updated, some of those files might get the changes whereas others might not.

10.10 Table Level Locking

SQLite supports only file level locking. That is, as far as locking granularity is concerned, an entire database file is a single unit. You may avail a limited form of table level locking by modifying your database and applications. You can split your database and store different user tables in different database files. Those separate files can be attached to the main database connection by using the ATTACH command, and the combined database will function as 'logically' one. But locks will only

be acquired on individual database files as needed. So, if you redefine a 'database' to mean two or more database files, then it is certainly possible for two threads (or processes) to write the same logical database (at different tables in different files) at the same time. Transactions that access two or more attached databases are ACID, and they will have higher level of concurrency.

However there is some overhead associated with this approach. First, you have multiple database files for the same logical database. Second, there is an increase in transaction processing time that is noticeable because you need to open multiple database files and their rollback journal files. Some commit operations may be slow as they need to handle a master journal file. Each database file has its own page-cache, and hence, there is memory overhead.

10.11 Savepoint

Savepoints are named transactions, and they can be nested. You can start a savepoint by executing the `savepoint` command. The command takes the name of the savepoint as input. The name need not be unique among all existing savepoints: it overrides the previous savepoint with the same name. SQLite permits you to start a savepoint inside or outside a user transaction. For the latter situation, SQLite internally treats it as a begin deferred transaction. The application can revert the database state to a previously set savepoint by executing the ROLLBACK TO command. You may note a difference between 'rollback' and 'rollback to' commands. The latter does not abort the transaction; it only undos part of database operations. It cancels all intermediate savepoints, between the current database state and that of the specified savepoint. To commit a savepoint along with its preceding savepoints, you can execute the `release` command with the savepoint name as the argument. When you establish a savepoint, SQLite does not delete the statement journal at the end of statement subtransaction commit, because the journal is needed when you execute a 'rollback to' command.

10.12 In-memory Database

To open an in-memory database, we use the filename ":memory:" in the `sqlite3_open` API function invocation. In-memory databases do not use any files to store any kind of information; they and their logs are stored entirely in the main memory and every instance is distinct from others. So, the database operations are super fast. When the application closes an in-memory database, this is eliminated from the main memory. The in-memory databases are not sharable across processes. There are downsides with in-memory databases. You need a lot of memory to hold the entire database. In addition, using in-memory databases can be risky. If the application process or system crashes, you lose all data.

10.13 Shared Page Cache

SQLite, in the default operating mode, does not permit sharing of the page cache of a database file that is opened for different database connections, even when the connections are opened by the same thread. Consequently, SQLite uses multiple page-caches for the same database file opened via different database connections. This may cause space constraints for applications on small devices such as cell phones. When this (shared page cache) feature is enabled, if a thread opens multiple connections to the same database (via different library connections), all the database connections share a single page cache; they also share the in-memory schema cache (aka, catalog objects) created for the database. Thereby, this reduces memory pressure and I/O requirements. Of late (as of SQLite 3.5.0 release onward) the same page-cache can be shared by multiple threads in the same process. (When this feature is on, SQLite does not open the same database multiple times via the same library connection.)

Shared caching is a process-wide feature as of today. As in the default operating mode, each database connection has its own `Btree` object. But, the object does not have its own independent Pager object. These `Btree` objects share a single `BtShared` object that holds the `Pager` object (see Fig. 10.2), where the `Pager` object in Process 1 holds the shared page-cache. The `BtShared.nRef` variable counts the number of `Btree`s own this `BtShared` object. (Not shown in the figure, the schema cache is also shared.) SQLite maintains some additional information to keep track of all open databases. When the thread opens a database, SQLite searches the list of open databases. If a match is found, the new connection shares the existing `BtShared` object without actually calling the `sqlite3PagerOpen` function to open the database. (The aforementioned searching is actually done in the `sqlite3BtreeOpen` function. And, all `BtShared` objects of the process are singly linked together via the global pointer `sqlite3SharedCacheList`.)

A process can enable and disable sharing of caches by invoking the `sqlite3_enable_shared_cache` API function with parameter non-zero and zero, respectively. The process can call this function anytime, but the current database connections are not affected by this call. Its effect is materialized in the next calls to the `sqlite_open` API function.

All the database connections that share a cache effectively appear to be a single connection outside the process. Synchronization of accesses to the database by different processes (not using the shared cache) is done by the normal SQLite locking scheme. For the Fig. 10.2, SQLite uses file locking scheme to synchronize accesses to the database from the two processes. Synchronization of accesses (from all peer threads) to the same database via the shared cache is done by a different locking scheme. There are three levels to the shared-cache locking model: transaction level locking, table level locking, and schema level locking. These three models are described in the following three subsections.

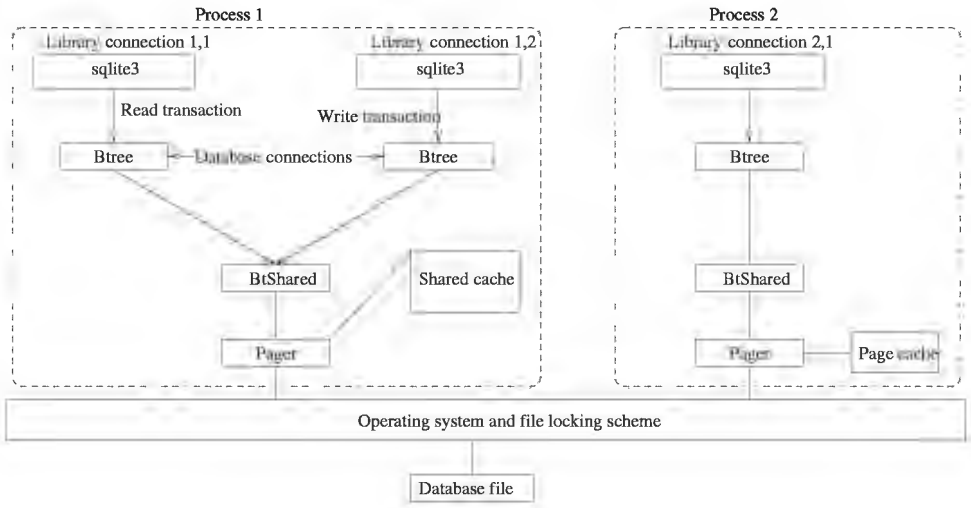


Figure 10.2: Page-cache sharing by connections in a process.

10.13.1 Transaction level locking

You may recall that on a database connection an application can open two kinds of transactions, namely read- and write-transactions. This may not be done explicitly; a transaction is started implicitly as a read-transaction until it first writes to the database, at which point it is converted into a write-transaction. A single database connection can have one write-transaction along with one or more concurrent read-transactions. However, there is a restriction here. At most one of all database connections via a single shared cache may open a write transaction at any one time. This write-transaction may however co-exist with any number of read-transactions in other connections to the same database file via the shared cache. But other connections to the same database file via different page-caches cannot have any read- or write-transactions due to conflict with the other write transaction.

Note: The strongest of all transactions is visible to the outside processes for concurrency control purpose. ◁

10.13.2 Table level locking

When many connections use a shared-cache, an internal (non file) locking scheme is used to serialize concurrent accesses on a per-table basis. There are two types of table level locks: 'read-locks' and 'write-locks'. These locks are granted to database connections — at any one time, a database connection has either a no-lock, a read-lock, or a write-lock on each database table. A lock is an object of type `BtLock`. Components of `BtLock` are given in Fig. 10.3. The `pBtree` component represents the database connection holding the `eLock` on the tree whose root resides on the `iTable`th

database page. All the `BtLock` objects form a singly linked list starting at the `BtShared.pLock` pointer.

Component	Description
pBtree	Btree holding this lock
iTable	Root page number of the tree
eLock	READ or WRITE

Figure 10.3: Structure of internal (`BtLock`) locks.

At any one time, a single table may have any number of active read-locks or a single active write lock. Before reading data from a table, a connection first obtains a read-lock on the table. Before writing data into a table, a connection first obtains a write-lock on the table. If a required table lock cannot be obtained, the request fails and SQLITE_LOCKED is returned to the caller. Table locks are released only when the current transaction completes. Thereby, for an application process, you can have multiple read-transactions or a single write-transaction active on a table. The write-transaction can never update a table being accessed by a read-transaction; this ensures the repeatable reading property as long as the read-transactions hold locks on the table.

Read-uncommitted Isolation: In this isolation mode (set by using the read_uncommitted pragma), a read-transaction does not obtain read locks on tables, and hence can read a table concurrently being written by write-transactions. As read locks are not obtained on tables, read-uncommitted transactions are nonblocking; neither they block write-transactions. ◁

10.13.3 Schema (`sqlite_master`) level locking

The `sqlite_master` table supports shared-cache read- and write locks in the same way as all other tables do (as described in the previous subsection). The following additional special rules are also applied for the master table.

- A database connection must obtain a read-lock on the `sqlite_master` table before accessing any database tables or obtaining any other read or write locks. (This rule is also applicable to read-uncommitted transactions.)
- Before executing a statement that modifies the database schema (e.g., a CREATE or DROP TABLE statement), a database connection must obtain a write-lock on the `sqlite_master` table.
- A database connection may not compile an SQL statement if any other connection is holding a write-lock on the `sqlite_master` table of the main or any attached database.

10.14 Security

A database may contain valuable information about an organization. The DBMS needs to protect the information stored in the database. Normally, for an enterprise database, not all users are authorized to access all data from the database. There are three aspects related to database security.

1. Secrecy: users must not be able to see unauthorized information.
2. Integrity: users must not be able to modify unauthorized data.
3. Visibility: a user must be able to see only what she is allowed to see.

There are two ways to ensure database security: (1) SQL control constructs and (2) encryption. An authorization constraint or security policy specifies what a user can do with a database object such as a table. SQL grant statement specifies what a user can do on what tables. Unfortunately, SQLite does not support the standard grant and revoke SQL security features because there is no concept of 'database users' per se for a SQLite database.

SQLite stores an entire database in a single, ordinary native file that can reside anywhere in the directory of the native file system. As SQLite is embeddable in the application process address space, the access permissions to the database file are governed by the native operating system/file system protection scheme. SQLite uses this protection scheme. Thereby, a user who has permission to read the file can read anything from the database. A user who has write permission on the file and the container directory can change anything in the database. The commonly used grant and revoke SQL statements are meaningless in embedded database systems. Thus the database is vulnerable to security threats.

SQLite has one API function, namely `sqlite3_set_authorizer`, to register a security callback function with the `sqlite3` connection handle. (You may note that it is not a full proof security measure.) The callback is invoked at SQL statement compilation time (and not at run-time) for each attempt to access a column of a table in the database to verify that the user has read and/or write access permissions on various fields of the database. All attached databases with the library connection use the same authorization function. The registration of the security callback function invalidates all currently prepared statements.

The signature of the security callback is: `int xAuth(void* p1, int p2, const char* p3, const char* p4, const char* p5, const char* p6)`. The first parameter is an authorization handle that is registered with the SQLite library at the time of the authorizer function invocation. The second parameter is an SQLite constant that indicates the type of authorization check; the

value signifies what kind of operation (e.g., creation/deletion of table, index, trigger, view) is to be authorized. The third and fourth parameters to the auth function are the name of the table and the column, respectively, that are being accessed or null. The fifth parameter is the name of the database (such as main, temp) where the table resides. The sixth parameter is a authorization context; it is the name of the inner-most trigger or view that is responsible for the access attempt or NULL if this access attempt is directly from input SQL statement. Except the first parameter, the values for the others are prepared by SQLite at runtime. The auth function should return one of the following values: (1) SQLITE_OK if access is allowed; (2) SQLITE_DENY if the entire SQL statement should be aborted by returning an error code; or (3) SQLITE_IGNORE if the SQL statement should run but attempts to read the specified column will return NULL and attempts to write the column will be ignored. Setting the auth function to NULL disables this security checking. In the default operating mode, the auth function is NULL.

SQLite supports optional proprietary encryption technology to protect information in databases. It supports four encryption schemes: RC4, AES-128 OFB, AES-128 CCM, and AES-256 OFB. The entire database file (user data plus metadata) and the journal files are encrypted with the keys supplied by applications. Right after successfully invoking the `sqlite3_open` API function, the application needs to inform SQLite the encryption key by invoking the `sqlite3_key` API function against the connection handle returned by the `sqlite3_open` API function. One can re-encrypt an already encrypted database by invoking the `sqlite3_rekey` API function with a new encryption key. Encryption though makes SQLite slow, but it is the best option for data protection in SQLite.

10.15 Unicode

Unicode is a single character set (aka, alphabet) that includes characters from European, Latin American, Asian, African, and many other major world languages. The set is so big that it cannot be enumerated in a single byte. There are two widely used encoding standards for the Unicode character set. In one standard, each Unicode character is represented by a fixed length unique 16-bits (two 8-bit bytes) integer number; it is an UTF-16 representation. (UTF stands for Universal Text Format.) The ordering of the two bytes can be either in little-endian or in big-endian format. In another standards, called UTF8, in which each Unicode character is represented by one, two, or three 8-bit byte string.

SQLite supports a different sets of API functions that accept input text as UTF-8 and UTF-16 in the native byte order of the host machine.[4] (Native byte ordering is what the platform supports to represent layout of 16-bit integers.) For example, the `sqlite3_open16` API function is equivalent

[4]A SQLite API function whose name ends with "16" is an UTF16 API; otherwise it is an UTF8 API. There are some APIs that work for both, and the suffix "16" is omitted from the API name.

to `sqlite3_open` except that it expects an input database file name in the UTF16 text string in the native byte order. SQLite uses a generic type of void* to refer to UTF-16 strings.

For each database file, SQLite manages data values with the TEXT storage type as either UTF-8, UTF-16BE (big-endian), or UTF-16LE (little-endian), and never in a mix of them. Internally (in-memory) or externally (on the disk file), the same text representation is used everywhere. This is the text encoding mode of the database file. You may recall that the encoding mode information is stored in the database file header record at offset 56. If a database file is created by the `sqlite3_open` (respectively, `sqlite3_open16`) API function, by default, the text encoding mode is UTF8 (respectively, UTF16 with the default native ordering of 16-bit integers). You may override this default text encoding mode by setting the `pragma encoding` appropriately before initializing the database file right after its creation. Once a file is initialized, the text encoding format is set forever and cannot be changed. If the text encoding of the database file does not match the input text encoding required by the interface APIs, then the input text is converted on-the-fly. For example, if a database is created for UTF8, but an application invokes an UTF16 API function to insert an UTF16 text, the text is converted into an UTF8 text before inserting it into the database. SQLite uses standard conversion algorithms for this purpose.

Advice: Constantly converting texts from one UTF representation to another can be computationally expensive, so it is advised that application developers choose a single representation and stick with it throughout their application. They have been warned! ◁

Note: All attached databases must have the same text encoding format. SQLite strictly follows the encoding of the main database. Thus, all attached databases must have the encoding of the main. Otherwise, it is an error, and the attach command is rejected. New databases created using the ATTACH command will have the same default text encoding of the main database. If the main database has not been initialized and/or created when the ATTACH command is executed, the initialization is done before the ATTACH execution. ◁

When creating a new user-defined SQL function or collating sequence, it can specify whether it works with UTF-8, UTF-16BE, or UTF-16LE texts. Separate implementations can be registered for different encoding standards. To execute a SQL statement, if an SQL function or collating sequence is required but a version for the current text encoding is not available, then the text is automatically converted before calling the function. As mentioned before, this conversion is computation intensive, so application developers are advised to pick a single encoding standard and stick with it in order to minimize the amount of unnecessary juggling.

SQLite is not very picky about the text it receives from users and it does process text strings that are not normalized or even well-formed UTF-8 or UTF-16. Thus, you can store ISO8859 texts using the UTF-8 API functions. As long as you do not attempt to use an UTF-16 collating

sequence or SQL function, the byte sequence of the text will not be modified in any way, and you are safe in such situations.

Parsing: In the current implementation of SQLite, the SQL parser only works with UTF-8 text. So if you give UTF-16 text, it will be converted into UTF-8 text before SQLite parses the text. This is how SQLite implements its parser now. In future, SQLite can parse UTF-16 encoded SQL as it comes. ◁

10.16 Collation

A *collating sequence* is merely a way of defining an order on two text strings. When SQLite sorts (or uses a comparison operator such as '<' or '<=' on) data values, the sort order is determined by the storage types of data values. The default ordering rules are given in Section 7.5.3.2 on page 196. As mentioned there, collating sequences are used only for ordering strings of the TEXT type. Collating sequences do not change the ordering of NULLs, numbers, or BLOBs; only text.

A collating sequence operator is implemented as a C function that takes two text strings being compared as inputs and returns a negative, zero, or positive integer number depending on the first string is less than, equal to, or greater than, respectively, the second one in the collation order. It is a callback function that resides in the application space. SQLite implements three built-in collating sequences, namely 'BINARY', 'RTRIM', and 'NOCASE'. BINARY is the default collating sequence. It is implemented by using the `memcmp()` function from the standard C library. (It works well for English texts.) NOCASE is same as binary, except that the 26 upper case characters used by the English language are folded to their respective lower case counter parts before the comparison is performed. RTRIM is same as the binary except that trailing space characters are ignored.

10.16.1 Collation examples

SQLite allows you to define arbitrary text comparison functions, known as user-defined collating sequences, that it can use instead of the default BINARY collation. You need to register the new collating sequence operator with SQLite (see Section 10.16.3). The decision of which collating sequence to use is controlled by the COLLATE clause in SQL statements. Each column of each table has a default collation type. If a collation type other than the default is required, a COLLATE clause must be specified as a part of the column definition. A COLLATE clause can also occur on a column of an index, or in the ORDER BY clause of a SELECT statement.

A collate clause in a permissible SQL statement is applicable on a single column only. The collation is used only for ordering text entries in that column. SQLite uses the corresponding collating sequence operator to determine the order of two text values from the column; the other

non text values are ordered by the rule stated in Section 7.5.3.2 on page 196. Different columns in the same table can have different collations.

The SQL statement `create table table1(col1 text, col2 text collate Russian, col3 text)` creates a table with three text columns. The COLLATE clause on col2 column specifies that the user defined Russian collating sequence operator will be used when comparing text entries from the column. It is the default collation for the column, but can be overridden by another collation in index creation and order by statements. The SQL statement `create index idx on table1(col1 collate Spanish)` creates an index idx1 on col1 from table1. The text entries in that column of the index are arranged in ascending order of col1 values by user defined Spanish collation function. The SQL statement `create index idx2 on table1(col2)` creates an index on col2. The text entries in that column are arranged in ascending order by user defined collation function for Russian. `Select * from table1 order by col3 collate Bengali` produces text output ordered by the value of `col3` as sorted by Bengali collation function.

10.16.2 Collation resolution

For a binary comparison (=, <, ≤, >, ≥, ! =, IS, and IS NOT) of two text operands, if only one operand is a column, then the default collation type of the column determines the collating sequence operator to be used for the comparison. If both operands are columns, then the collation type for the left operand is used in the comparison. If neither operand is a column, then the BINARY collation sequence is used.

The expression "x BETWEEN y and z" is equivalent to "x >= y AND x <= z". The expression "x IN (SELECT y from ...)" is handled in the same way as the expression "x = y" for the purposes of determining the collation sequence to use. The collating sequence used for expressions of the form "x IN (y, z ...)" is the default collation type of x if x is a column, or BINARY otherwise.

An ORDER BY clause that is part of a SELECT statement may be assigned a collating sequence to be used for the sort operation explicitly. In this case the explicit collating sequence supersedes others and is always used. Otherwise, if the expression sorted by an ORDER BY clause is a column, then the default collation type of the column is used to determine the sort order. If the expression is not a column, then the BINARY collation sequence is used.

10.16.3 Collation registration

A collation function can be registered by executing the `sqlite3_create_collation` (or `sqlite3_create_collation16`) API function. The signature of the function is given below.

```
int sqlite3_create_collation(
```

```
    sqlite3* db,
    const char* zName,
    int pref16,
    void*,
    int(*xCompare)(void*, int, const void*, int, const void*)
);
```

The function is used to add a new (or replace an existing) collation function to the `sqlite3` handle via the first argument. The name of the new collation sequence is specified as an UTF-8 string for `sqlite3_create_collation` and an UTF-16 string for `sqlite3_create_collation16`. In both cases the collation name is passed as the second argument (zName). The third argument must be one of the constants SQLITE_UTF8, SQLITE_UTF16, SQLITE_UTF16LE, or SQLITE_UTF16BE, indicating that the user-defined collation routine (xCompare) is expected to be passed pointers to strings encoded using UTF-8, UTF-16 with native byte order, UTF-16 little-endian, or UTF-16 big-endian, respectively. A pointer to the user defined collation comparator function must be passed as the fifth argument. If it is NULL, this is the same as deleting the collation sequence (so that SQLite cannot call it anymore).

Each time a user-defined collating sequence comparator is invoked it is given a copy of the void* argument (passed as the fourth argument to the collation registration function) as the first argument to the comparator. The remaining arguments to the comparator are two strings, each represented by a ⟨length, data⟩ pair, and are encoded in the encoding format that was passed as the third argument when the collation sequence was registered. The comparator must return a negative, zero, or positive number depending on the first string is less than, equal to, or greater than, respectively, the second string.

Registering all collation functions at the beginning of application initialization might be a big burden on application developers. SQLite supports a helper API function, namely `sqlite3_collation_needed` (or `sqlite3_collation_needed16`) to register a generic user defined function that SQLite invokes when it needs a comparator for an unknown collating sequence. When the generic function is called, the application can register a comparator for the unknown collation.

10.17 WAL Journaling

SQLite development team introduces a new journaling mechanism in release 3.7.0, called WAL journaling. This is an (exclusive) alternative to the legacy rollback journal discussed earlier. You can turn on this mode of journaling by `pragma journal_mode=WAL`. When the database is in the WAL mode (i.e., the file format value is 2 at file offset 18), SQLite uses wal journaling instead of

rollback journaling. In wal journaling, the journal file name is the same as the database file name with '-wal' appended, and it resides in the same directory as the original (main, temp, or attached) database file does.

The journaling activities are pretty much the same as the legacy journaling. A write-transaction adds page images (in log records) before modifying them. (The page image however is the new version of the page being updated or added; popularly known as the *redo* image.) When the write-transaction commits, a commit record is appended at the end of the wal journal file. The transaction may not acquire an exclusive lock on the database file and may not write the file. This avoids the transaction to block for concurrent read-transactions to complete. Neither the read-transactions block a write-transaction. That is, the read-transactions behave as if no write-transactions are present. Newer write-transactions keep appending their logs to the wal journal file.

The first 32 bytes of a wal journal file describes the format of the file. Figure 10.4 shows the structure of the wal journal header. There are eight 4-byte unsigned integers, each stored as a big-endian number. The two magic numbers identify which alternative checksum algorithm is used to compute the checksums for log frames (see the next paragraph).

Offset	Size	Description
0	4	Magic number: 0x377f0682 or 0x377f0683
4	4	File format version (currently 3007000)
8	4	Database page size
12	4	Checkpoint sequence number
16	4	Salt-1: a random integer, incremented with each checkpoint
20	4	Salt-2: a different random integer for each checkpoint
24	4	Checksum-1: first part of a checksum on the first 24 bytes of header
28	4	Checksum-2: second part of the checksum on the first 24 bytes of header

Figure 10.4: Structure of wal journal file header.

The wal journal header is followed by zero or more log frames. Each log frame starts with a 24-byte frame header that is followed by the content of the page being logged. New log frames are always appended at the end of the journal file. Figure 10.5 shows the structure of the wal frame header. There are six 4-byte unsigned integers, each stored as a big-endian number.

Reading a from the database is a little more involved in the wal journaling scheme. When the pager starts a new read transaction, it records the index of the last valid commit record (a log frame) in the WAL journal for the read transaction. The transaction would use this marked-frame as the sentry point for all subsequent read operations. Suppose the pager get a request to read a page p from the read-transaction. It looks at the wal journal file to see whether there is a log record for page p occurring on or before the marked-frame. In this case, the last valid instance of page p that precedes or the same as the marked-frame is used. Otherwise, the pager reads page p

Offset	Size	Description
0	4	Page number of the page being logged
4	4	For commit records, the size of the database file in pages after the commit For all other records, zero
8	4	Salt-1 (copied from the WAL header)
12	4	Salt-2 (copied from the WAL header)
16	4	Checksum-1: cumulative checksum of the first 8 bytes of this frame header and the page ima
20	4	Checksum-2: second half of the cumulative checksum

Figure 10.5: Structure of wal frame header.

from the database file. Concurrent write-transactions can and do append new log records to the wal journal, but as long as the read-transaction uses its original marked-frame value and ignores subsequently appended content, it will see a consistent snapshot of the database from a single point in time. This technique allows multiple concurrent read-transaction to view different versions of the database content simultaneously. Because log frames for the page p can appear anywhere within the wal journal, the pager needs to scan the journal from the beginning upto the sentry point looking for page p frames. If the journal is large when the transaction started, the scan will be costly. SQLite maintains a separate data structure called the *wal-index* to expedite the search. It is basically a hash index that maps a page number p to the appropriate log frame if p is in the wal journal or null otherwise.

An wal-index is implemented via shared memory or a memory mapped file. The mapped file resides in the same directory as the database file does, and has the same name as the database file with "-shm" appended. (Because of the use of the shared memory or the memory mapped file, the wal journaling cannot be used where the database file resides on a network filesystem and clients are on different machines; the distributed clients will not be able to share the same memory where the wal-index file will be mapped.) SQLite truncates or invalidates the wal-index file when the last connection to the database closes. This implies that the wal-index file is a transient file. Upon a crash recovery, the wal-index file is reconstructed from the wal journal file.

Unlike the rollback journaling scheme, the wal journaling scheme needs checkpointing to keep the journal file size in check. Applications do not need to perform checkpoints manually, but if they want to they can do so by turning off the automatic checkpointing option. SQLite automatically performs a checkpoint when the wal journal file reaches a threshold size of 1000 pages. (You can set a different threshold by setting SQLITE_DEFAULT_WAL_AUTOCHECKPOINT compile flag to a different value.) Checkpoint operations are performed sequentially, mutually exclusively. On each invocation of the checkpointing function, SQLite performs the following steps in the said sequence.

- First, it flushes the wal journal file.

- Second, it transfers some valid page contents to the database file.
- Third, it flushes the database file (only if the entire wal journal is copied into the database file).
- Fourth, the salt-1 component of the wal file header is incremented and the salt-2 is randomized (to invalidate the current page log images in the wal journal).
- Fifth, update the wal-index.

A checkpoint operation execution can run concurrently with read-transactions. In that case, the checkpoint stops on or before reading the wal-mark of any read-transaction (see below). The checkpoint remembers how far it has checkpointed; the next checkpoint restarts from there. When the entire wal journal is checkpointed, the journal is rewind to prevent the journal file to grow without bound.

There are some pros and cons for this feature. The pros are: (1) fewer flushes of the database, (2) promotes more concurrency as read-transactions do not block write-transactions and a write-transaction does not block read-transactions, (3) transaction processing is quite faster in most cases. The cons are: (1) memory map support from the operating system, (2) all applications accessing a database file must run on the same machine, (3) database files must not be NFS mounted; (4) multidatabase transactions may not be atomic across all databases though atomic in individual databases, (5) transaction rollback is slightly slower, (6) need of two additional files (-shm and -wal), (7) need of checkpointing, etc. You can find a detail description of this feature at http://www.sqlite.org/wal.html.

10.18 Compile Directives

SQLite source code has many customizable compilation flags or options, with default option values. You can build and safely use the SQLite library using their default values. However, you can use different values or omit particular SQLite features (resulting in a smaller library size). Each option can be passed down to the compiler by prefixing '-D' to the option name. For example, for NDEBUG option, use -DNDEBUG on compilation command. Some options are described below. They are categorized into a few categories. (You can find all currently supported compile options at the http://www.sqlite.org/compile.html webpage.)

- Options to set debugging flags.
    1. SQLITE_DEBUG: Many testing and debugging features are enabled.

2. SQLITE_MEMDEBUG: This option enables a modified malloc with a lot of checking such as bounds checking, using freed memory, using uninitialized memory, memory leak testing, etc.

- Options to set default parameter values.

1. SQLITE_DEFAULT_AUTOVACUUM=⟨0, 1, or 2⟩: This macro determines whether or not SQLite creates databases with the autovacuum option set by default. The default value is 0 (do not create autovacuum databases). The compilation time default may be overridden at runtime by the `PRAGMA auto_vacuum` command before creating any table in the database.
2. SQLITE_DEFAULT_CACHE_SIZE=⟨number⟩: This macro sets the default size of the page-cache for each directly opened or attached database; the number is in pages. The default value is 2000. This value can be overridden by the `PRAGMA default_cache_size` command at runtime.
3. SQLITE_DEFAULT_TEMP_CACHE_SIZE=⟨number⟩: This macro sets the default size of the page-cache for temporary files created by SQLite to store intermediate results; the number is in pages. The default value is 500. It does not affect the page-cache for the temp database.
4. SQLITE_DEFAULT_PAGE_SIZE=⟨number⟩: This macro is used to set the default page-size to be used when a database is created; the number is in bytes. The value assigned must be a power of 2, and between 512 and 65,536 (both inclusive). The default value is 1024. The compile-time default may be overridden at runtime by the `PRAGMA page_size` command.

- Options to omit features.

There are many omit options that causes specific features to be disabled from the SQLite library. Each option name is prefixed with SQLITE_OMIT_ character string. These options are primarily used to reduce the library footprint for embedded systems.

1. SQLITE_OMIT_TEMPDB: When this option is defined, SQLite does not create and open the temp database for open library connections. The sqlite_temp_master catalog table will not be created.
2. SQLITE_OMIT_ALTERTABLE: When this option is defined, the SQL `alter table` feature is not included in the library. No `alter table` statement will be recognized by the library.

3. SQLITE_OMIT_AUTHORIZATION: When this option is defined, SQLite omits the authorization callback feature from the library. The `sqlite3_set_authorizer` API function is not defined in the library.
4. SQLITE_OMIT_AUTOINCREMENT: When this option is defined, the AUTOINCREMENT functionality is not included in the SQLite library. The 'INTEGER PRIMARY KEY AUTOINCREMENT' columns will be treated as if they are declared as 'INTEGER PRIMARY KEY' when a NULL value is inserted. The library does not create nor look at the existing `sqlite_sequence` catalog. if it already exists.
5. SQLITE_OMIT_AUTOVACUUM: When this option is defined, the library cannot create or write to databases that support autovacuum. When this library opens a database that supports autovacuum, it is opened in the readonly mode; applications cannot write the file.
6. SQLITE_OMIT_PRAGMA: When this option is defined, the PRAGMA feature is excluded from the library.
7. SQLITE_OMIT_COMPOUND_SELECT: When this option is defined, the compound SELECT functionality is excluded from the library. SELECT statements that include UNION, UNION ALL, INTERSECT or EXCEPT will be rejected by the library during query parsing.
8. SQLITE_OMIT_REINDEX: When this option is defined, the REINDEX command is excluded from the library. REINDEX commands will be rejected by the library.
9. SQLITE_OMIT_SUBQUERY: When this option is defined, the support for sub-selects and the IN operator are excluded from the library.
10. SQLITE_OMIT_TRIGGER: When this option is defined, the TRIGGER feature is excluded from the library. TRIGGER commands will be rejected by the library.
11. SQLITE_OMIT_UTF16: When this option is defined,' UTF16 text encoding feature is excluded from the library. All API functions that deal with UTF16 encoded text are not available.

- Options to set size limits: The limit categories are discussed in Section 2.5 on page 69. For each category, there is a pragma that you can use to set their max values.
- Options to control operating characteristics: In this category, you have options such as thread-safe, case sensitive like, temp store.
- Options to enable features: In this category, you have options such as FTS3, ICU, RTREE, memory management.

- Options to disable features: In this category, you have choices of disabling large file support and directory sync.

You can check and get the compile time options used to building the SQLite library using the `sqlite3_compileoption_used` and `sqlite3_compileoption_get` API functions. These functions are defined in the ctime.c source file.

Summary

SQLite supports many advance features. They are optional, and you can turn them off individually by compiling the SQLite source code with appropriate compile flags set. Turning them off will reduce the SQLite library footprint. Notable of these features are pragma, subquery, view, trigger, autovacuum, shared page cache, collation.

Pragmas help you to get information about various database-metadata and to alter the behavior of the SQLite library at runtime. For example, if you execute `pragma synchronous=off`, henceforth all (system and user) transactions will become asynchronous transactions, and they will never sync/flush the log and database files.

Viwes, though not persistent tables, are often quite useful. SQLite stores a row in the master catalog for each view definition, where the create view SQL statement is stored in the `sql` column of the master catalog. When the view is used in a query, the view's select statement is executed to form a temporary table to be used in the query. SQLite does not support executing insert, delete, and update queries on views.

Trigger is an important feature. It helps you to predefine procedures containing one or more SQL statements. The procedures will be automatically executed by SQLite when some specified events occur. A trigger is defined by an event, a condition, and an action. When the event (insert, delete, or update) occurs, the trigger is said to be activated. At that point, the trigger condition is evaluated. If the condition is true, the trigger action (aka, procedure) is executed. There are two kinds of triggers: (1) for each row and (2) for each statement. SQLite supports the former one only as of today.

SQLite has an excellent support for unicode characters. It has capabilities of both UTF-8 and UTF-16 (BE and LE). A database can use only one of them. All TEXT data is translated into the chosen UTF format before using them.

Chapter 11

Further Reading

Database is a mature field today. Codd [2] is the father of the relation database; he borrowed concepts from relational algebra (a branch of set theory) to the problem of storing and organizing data. His paper changed the then landscape of the database industry forever. The entity-relationship model is introduced by Chen [1]. Teory et. al. [23] present a review of this model and its extensions. There are many books on database systems. Readers may consult text books [4, 5, 19, 20, 24] to know more about database concepts.

Gray and Reuter [9] present a thorough study of transaction processing, both concepts and implementation techniques. They present historical notes at the end of almost all chapters, which are good sources of references to the database literature. The term ACID is originally introduced by Haerder and Reuter [10]. As far as I am aware of, Eswaran et. al. [6] are the first to discuss two phase locking and transactions.

Btree algorithms are discussed by Knuth [14]. The B^+-tree organization is originally introduced by Comer [3]. Smith [21] presents a good review of many cache organizations, and placement, replacement, and fetch rules. Huffman code is developed by Huffman [13] as a Ph.D. student at MIT in 1952. Manifest typing is discussed by Graham [8].

There are a few recent books on SQLite by Newman [17], Owens [18], van der Lans [25], Kreibich [15]. They primarily talk about programming the SQLite library, and occasionally internals of SQLite. Newman [17] also talks about using SQLite in conjunction with PHP. Owens [18] also presents SQLite extensions for Java, Pearl, Python, PHP, Ruby, Tcl. The van der Lans book [25] presents hundreds of SQLite applications towards teaching how to use SQLite efficiently and effectively. I myself in [11] presented a brief overview of SQLite internals. The current book is an extension of that work, which talks less of programming but more of SQLite's internal data structures and their interrelationships. I sincerely hope that after reading this book you will get excited to write new embedded database management systems for your own suit.

SQLite webpages [22] provide a good source of information about SQLite internals, design

choices, and application development tips. The SQLite development team upgrades the information on every new release of their code base. You can search keywords in SQLite documents via the webpage and get to the pages containing information about the keywords. The SQLite source tree contains many regression tests. Those tests are also a good source of information about how to program the library. To learn minute details about the inner workings of SQLite internals, the best practice is reading thoroughly the `.h` and `.c` source files. You can also make a debugable build of the library and step through various executions to learn how SQLite components fit together. This is one of the best alternatives to learn how a software technology works.

Bibliography

[1] P.P. Chen. The entity-relationship model — toward a unified view of data. ACM Transactions on Database Systems, Vol 1(1), 1976, 9–36.

[2] E.F. Codd. A relational model of data for large shared data banks. Communications of the ACM, Vol 13(6), 1970, 377–387.

[3] D. Comer. Ubiquitous B-Tree. ACM Computing Surveys, Vol 11(2), 1979, 121–137.

[4] T.M. Connolly and C.E. Begg. Database Systems: A Practical Approach to Design, Implementation and Management. Reading, MA, Addison-Wesley, 2010.

[5] R. Elmasri and S.B. Navathe. Fundamentals of Database Systems. Reading, MA, Addison-Wesley, 2007.

[6] K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. Communications of the ACM, Vol 19(11), 1976, 624–633.

[7] E.B. Fernandez, R.C. Summers and C. Wood. Database security and integrity. Reading, MA, Addison-Wesley, 1981.

[8] P. Graham. ANSI Common Lisp. Prentice-Hall, November 1995.

[9] J. Gray and A. Reuter. Transaction processing: concepts and techniques. Morgan Kaufman Publishers, San Francisco, California, USA, 1993.

[10] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys, Vol 15(4), 1983, 287–317.

[11] S. Haldar. Inside SQLite. O'Reilly Media Inc, 2007.

[12] S. Haldar and A. Aravind. Operating Systems. First edition, Pearson Education, 2009.

[13] D.A. Huffman. A method for the construction of minimum-redundancy codes. In Proceedings of the I.R.E., 1098-1102 , 1952.

[14] D.E. Knuth. The art of computer programming, Volume 3: Sorting and searching. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973 (2nd edition 1998).

[15] J.A. Kreibich. Using SQLite. O'Reilly Media, 2010.

[16] J. Meeus. Astronomical Algorithms. 2nd Edition, Willmann-Bell, Inc, Richmond, Virginia (USA), 1998.

[17] C. Newman. SQLite: A practical guide to using, administering, and programming the database bundled with PHP 5. SAMS Publisher, November, 2004.

[18] M. Owens. The definitive guide to SQLite. Apress, May, 2006.

[19] R. Ramakrishnan and J. Gehrke. Database management systems. Third edition, 2004, McGraw-Hill.

[20] A. Silberschatz, H.F. Korth, and S. Sudarshan. Database system concepts, 6th edition, McGraw-Hill, Jan 2010.

[21] A.J. Smith. Cache memories. ACM Computing Surveys, Vol 14(3), 1982, 473–530.

[22] SQLite webpage. http://www.sqlite.org. Operational since 2000.

[23] T.J. Teorey, D. Yang and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. ACM Computing Surveys, Vol 18(2), 1986, 197–222.

[24] J.D. Ullman and J. Widom. A first course in database systems. Second edition, 2002, Prentice-Hall, Upper Saddle River, New Jersey, USA.

[25] R.F. van der Lans. The SQL guide to SQLite. LuLu.com publisher, 2009. (See www.r20.nl.)

[26] S. Verma, S. Haldar, C. Hoang, and S. Listgarten. Neighboring locking technique for increasing concurrency among transactions. United States Patent 7680794. March 2010.

Index