Inside Symbian SQL

A Mobile Developer's Guide to SQLite

Ivan Litovski
with
Richard Maynard

WILEY

SYMBIAN

Inside Symbian SQL

A Mobile Developer's Guide to SQLite

Inside Symbian SQL

A Mobile Developer's Guide to SQLite

Lead Authors
Ivan Litovski with Richard Maynard

With
James Aley, Philip Cheung, James Clarke, Lorraine Martin, Philip Neal, Mike Owens, Martin Platts

Reviewed by
Erhan Cetintas, Renzo Cherin, Alex Dimitrov, Chris Dudding, Lucian Piros, George Sewell, Andrew Thoelke

Head of Technical Communications, Symbian Foundation
Jo Stichbury

Managing Editor, Symbian Foundation
Satu Dahl

WILEY

A John Wiley and Sons, Ltd., Publication

This edition first published 2010
© 2010 Symbian Foundation

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at ***www.wiley.com***.

ISBN 978-0-470-74402-4

A catalogue record for this book is available from the British Library.

Typeset in 10/12 Optima by Laserwords Private Limited, Chennai, India
Printed in Great Britain by TJ International Ltd, Padstow, Cornwall

Contents

Foreword

RICHARD HIPP

Are you skeptical of the benefits of using the Structured Query Language (SQL) and SQLite in a mobile device? I would be if I were 25 years younger and reading this book for the first time. When I first encountered the SQL language two decades ago, I was deeply suspicious of the whole idea. SQL seemed like a complex solution to a simple problem. Wouldn't it be easier to just read and write directly to various disk files (sometimes called a 'pile-of-files' database) and use a simple key–value pair database?

In the years that followed that first encounter, I discovered that the SQL programming language does in fact make many programming tasks much, much easier. As applications grow more complex and use richer content, it becomes important to be able to organize that content in a structured way. The pile-of-files and key–value approaches quickly become unwieldy. Despite its initial complexity, SQL so greatly simplifies the management of complex data sets that it ends up being a huge net simplification. SQL frees the programmer from the drudgery of low-level I/O interactions and allows the developer to focus on user-facing aspects of the application.

A programmer familiar with SQL can often replace hundreds or thousands of lines of low-level I/O code with a single 10-line SQL statement, and in so doing simultaneously scale their productivity and code maintainability. The use of SQL also allows programmers to defer data storage design decisions until late in the design cycle or to make major changes to storage strategies late in a design with minimal impact to code. Furthermore, the transactional model supported by SQL turns out to be an exceedingly powerful abstraction that helps to make applications both more robust and easier to implement.

And so within 10 years of my first encounter with SQL, I was using the SQL language regularly in the development of custom workstation applications. One such application used a well-known commercial SQL database engine. That database worked great, but it was difficult to install and administer and was used in a setting where skilled database administrators were unavailable. Hence the idea arose for a new SQL database engine that was small, lightweight and trivial to operate – a 'zero-administration' database engine. This idea became 'SQLite'. The first source code was written on May 29, 2000.

Early versions of SQLite were used in a variety of custom projects undertaken by me and my associates over the next several years. I released the SQLite source code into the public domain so that other independent programmers might benefit from it and it quickly became popular on the Internet. Then, in October 2005, Symbian contacted me and expressed an interest in using SQLite in their mobile operating system. That contact lead to a series of meetings during which I had a crash course in the important differences between workstations and mobile devices.

Since you are reading this book, you probably already know that the run-time environment of a mobile device is very different from that of a workstation. Workstations have essentially infinite stack space, infinite power, and they never fail a memory allocation. But that is not the case with mobile devices. Sure, SQLite was compact, efficient and zero-administration, which are the qualities that made it attractive to Symbian in the first place, but the reality was that SQLite still needed important refinements prior to use in embedded devices.

With guidance and encouragement from Symbian, my team and I began to make SQLite much more mobile-friendly. We enhanced SQLite so that it would always fail gracefully on a memory allocation fault. We greatly reduced stack space requirements. We added the 'shared cache' mode which allows two or more applications to share a common database cache and hence reduce system memory requirements. We reduced the amount of raw I/O that needed to occur in order to write to the database. And we enhanced the overall reliability and performance of SQLite to meet the exacting requirements proscribed by Symbian. The end result of these efforts was a new and improved 'phone-ready' SQLite, which is the subject of this book.

But Symbian was not content to push the SQLite team for technical improvements only. They also pushed us toward a more solid business model. For its first seven years, SQLite was maintained mostly by volunteers. There would be an occasional support contract available to us maintainers, but such contracts did not provide us with a living. In order to secure a more stable future for SQLite, the management at Symbian suggested the formation of the SQLite Consortium. The SQLite Consortium is a group of companies that use SQLite extensively, including Nokia, Mozilla, Adobe, and Bloomberg, and contribute an annual fee that allows three SQLite developers (Dan Kennedy, Shane Harrelson, and myself) to continue working on SQLite full-time, while keeping the SQLite source code free, open, and accessible to all. In its two years of operation, the SQLite Consortium has made a huge difference, allowing us to add many new features to SQLite while simultaneously improving code quality, reducing defect rates, and maintaining its compact size and efficient operation.

Since SQLite is in the public domain and can be copied freely, we have no way of tracking how widely SQLite is used. But a conservative estimate, derived from counting the number of software products and gadgets that are known to use SQLite, puts the number of SQLite instances currently deployed in excess of 500 million. That means SQLite probably has more deployments than all other SQL database engines combined. We SQLite developers continued to be amazed and honored by the success of our humble little database engine. And we are very grateful to Symbian and Nokia for pushing us to make the improvements to SQLite that will help ensure its success in the future.

I sincerely hope that you find SQLite useful. If you are suspicious of the claimed benefits of using SQL and SQLite I hope that you will persevere and give them an honest try. I believe that you will find, as I did, that despite its initial complications, the SQL language really does make the development of applications with complex data requirements dramatically simpler. And I think you will also find that the tight integration between SQLite and the operating system within the Symbian platform (which is the subject of this book) will make coding to the SQLite interface one of your easier programming tasks.

D. Richard Hipp, creator of SQLite
Charlotte, NC
October 2009

Foreword

CHRIS DUDDING

My first encounter with SQLite was during the selection of a new relational database for the Symbian platform in 2005. I worked with Richard Maynard and his team to study several commercial and non-commercial databases. As a stakeholder representing application software components that needed more sophisticated database capabilities than the existing solution, I had a keen interest in choosing the right database. We studied the functionality offered by each product and created benchmarks representative of the usage planned by the Phonebook and Messaging applications. We evaluated a wide range of criteria including response time, memory consumption, disk usage and I/O operations. SQLite was the best choice – it had high performance and enough functionality for our planned usage. We were not alone in choosing SQLite: Apple, Google and Palm also made the same choice for their mobile products.

This book provides a unique insight into Symbian SQL, the client–server database incorporating SQLite on Symbian. It tells you what you need to know to take advantage of the performance and scalability offered by SQLite databases. The authors are all active contributors to the Symbian platform and have personally influenced the implementation and evolution of Symbian SQL.

The early chapters of this book contain much of what you'd expect – an introductory tutorial to demonstrate basic usage of the Symbian SQL API, an explanation of the relational model for database management, the key concepts of Symbian database programming and a class-by-class description of the Symbian SQL API. There's also an in-depth study into the architecture and internals of the SQLite library in Chapter 7, which includes details of the Symbian platform porting layer and technical information about performance optimizations identified by Symbian and the authors of SQLite. Chapter 8 covers the tricky subject of designing efficient database schemas and optimizing your SQL queries and API usage to get the most out of Symbian SQL. It contains advice from the developers of Symbian SQL, the authors of SQLite and hard-won experience from early users of Symbian SQL.

I think Chapter 8 is one of the book's greatest contributions to the Symbian community. In 2007, I was the technology architect of the team responsible for migrating the Contacts Model, an application engine used in the Phonebook application, from DBMS to Symbian SQL. To design the database schema, we benchmarked several possible options and consulted heavily with the developers of Symbian SQL. The information in Chapter 8 would have saved us a lot of effort and last-minute performance tuning.

You can read more about our experience migrating from DBMS in Chapter 9, described by one of the developers, James Clarke, together with two case studies demonstrating the potential of Symbian SQL including the Wikipedia demonstration created by Ivan Litovski and Martin Platts. The appendices provide useful advice for troubleshooting problems, information about the SDB tool for creating databases and a summary of Symbian SQL error codes.

Open Source

I have been working on the Symbian platform for a long time now. I joined Symbian Software Limited as a software engineer in September 1998, initially developing application software for the Psion Series 5mx and then working on the first Symbian devices such as the Ericsson R380. Back then all mobile phones had monochrome displays. The excitement of seeing a prototype of the color screen Nokia 9210 Communicator in a London hotel suite in 1999 compares well to the surprise and excitement I felt when I heard that there were plans to make the code available to everyone via the Symbian Foundation in June 2008.

Symbian SQL is available now as part of the Persistent Data Services package that is part of the Symbian platform. I'm the package owner for Persistent Data Services, which is the operating system layer package that provides the standard frameworks and libraries for handling persistent data storage. The source code is available to all under an Eclipse Public License, from ***developer.symbian.org***.

Persistent Data Services has a dedicated mailing list for discussion of new package features, a discussion forum for the OS Base Services technology domain, a bug database to report defects, a published backlog of features under development, and wiki articles about our technology. All can be accessed from the wiki page we have created for this book at ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***.

Our current backlog includes the introduction of the standard SQLite C API in Symbian^3 and changes to make Symbian SQL ready for symmetric multiprocessing. The SQLite C API will allow application authors already using SQLite to port their software more easily to the Symbian platform. Performance and memory usage are a constant concern for me, and as the package owner I will continue to ensure that Symbian SQL maintains and improves its current system qualities.

In addition to our package development plans, the Qt major contribution proposal from Nokia, targeted at Symbian^4, will add the QtSQL module, providing a new way to access SQLite databases for Qt-based applications.

The future of Symbian SQL is very much in your hands as users, developers, and, I hope, fellow contributors. The open development model of the Symbian platform provides opportunities at many levels to influence the development of the package. I encourage you to join our community.

And Finally . . .

I'd like to thank Ivan Litovski, Richard Maynard, and the other authors for creating an invaluable reference for anybody working with Symbian SQL. If you use Symbian SQL or want to learn how, then read this book!

Chris Dudding
Persistent Data Services Package Owner, Nokia
November 2009

Author Biographies

Ivan Litovski

Ivan joined Symbian in 2002, working first in the Java team and later with the System Libraries, Persistent Data Services and Product Creation Tools teams. Ivan has been a software professional since 1997. He is experienced with Java, embedded software, information security, networking, scalable servers and Internet protocols. He enjoys solving difficult problems including in research and development. He has authored several patents and papers in peer-reviewed, international journals. He holds a BSc in Electronics and Telecommunications from the University of Nis, Serbia and is an Accredited Symbian Developer.

Richard Maynard

Richard has worked within the Symbian Devices division of Nokia Devices R&D as an architect for persistent data storage since Nokia acquired Symbian in 2009. Richard joined Symbian in 2002 working in the System Libraries team. After leading the team's transition to platform security in Symbian OS v9, he moved to head up the newly formed Persistent Data Services team which had been created in order to introduce a new standards-compliant SQL database to the platform. Prior to working at Symbian, Richard worked in the fields of 3G telecommunications and radar design.

James Aley

James joined Symbian in 2006 for a year as the industrial placement portion of his Computer Science degree at King's College London. After graduating in 2008, he worked as a Software Engineer for Nokia until autumn 2009. He is now working at Symbian Foundation as a Technology Specialist, focussing on the productivity technology domain.

James authored a metadata-based search service for the Symbian platform based on Symbian SQL. This service made very heavy use of the technology and is a good example of how it makes demanding persistence projects simple for the developer.

Philip Cheung

Philip joined Symbian as a graduate software engineer in 2006, working in the System Libraries team. He now works as a software engineer in the Persistent Data Services team in the Symbian Devices division of Nokia Devices R&D. During his time at Symbian and Nokia, Philip has been involved in various projects on Symbian SQL. Philip received an MEng in Computer and Communications Systems Engineering from UMIST in 2004.

James Clarke

James joined Symbian in 2005, working on the contacts database engine in the Personal Information Management team. Before that he worked for 10 years in academia as a researcher and database developer. James now works as a software engineer in the Persistent Data Services team in the Symbian Devices division of Nokia Devices R&D. He has a BA in Geography and an MSc in Computer Science, both from the University of London.

Lorraine Martin

Born and raised in Glasgow, Lorraine graduated from Glasgow University with a first class Joint Honors degree in Computing Science and Mathematics in 1997. She worked for Cisco Systems before taking a year out in 2001 to indulge her passion for traveling.

Lorraine has lived in London, England since 2002 and is a senior software engineer at Quickoffice, the worldwide leading provider of office software for mobile phones, on platforms including iPhone, Symbian and BlackBerry. Lorraine previously worked for three years at the Symbian offices in London where she was Technical Lead for various SQLite projects.

Philip Neal

Philip read Classics and Modern Languages at Queen's College, Oxford and researched into computational linguistics at UMIST, Lancaster and Essex universities. He is a translator and technical author.

Michael Owens

Michael is the IT Director for CENTURY 21 Mike Bowman, Inc. An avid open source developer, Michael has written for several publications, including the *Linux Journal*, *C++ User's Journal*, and *Dr. Dobbs Journal*. He has been a speaker on SQLite at the O'Reilly Open Source Convention (OSCON).

Michael earned a bachelor's degree in chemical engineering from the University of Tennessee. He has worked at the Oak Ridge National Laboratory as a process design engineer and Nova Information Systems as a software developer. He is the original developer of PySqlite, the Python extension for SQLite.

Martin Platts

Martin joined the Symbian Databases team in 2007, where he worked as Tech Lead on the database characterization project to compare DBMS, SQL and other third-party databases. Before joining Symbian, Martin gained four years of Symbian experience, working on projects for companies such as Nokia and Sony Ericsson. These projects included, among other things, DBMS databases, messaging and multimedia. Martin is now a Project Manager in the Graphics team at Nokia. He is also a part-time DJ and club night promoter, and full-time football addict.

Author's Acknowledgments

I first thought about writing this book while walking home after a particularly satisfying day at work. At the time, I was in Symbian's Persistent Data Services team. Symbian SQL had just been delivered for the first time to our customers and we were doing a pretty serious performance analysis. Outside the team, not many people knew much about Symbian SQL. We had accumulated considerable domain knowledge and it was time to share it by all possible means – through API documentation, engineering documents and examples. I thought that a book would be perfect medium for this and started slowly developing the idea.

Serendipitously, within days I was invited to a meeting with George Sewell, Jo Stichbury, Satu Dahl, Richard Maynard and several team members – to discuss writing a book about Symbian SQL! Clearly, I wasn't the only person thinking about this. George Sewell and Lorraine Martin had produced a proposal for the contents, not much different from what is in this book. A team was assembled, consisting of expert engineers who had built Symbian SQL and several that had used it within Symbian.

It is these people I would like to thank first. George Sewell's foresight on Symbian SQL, the SQLite Consortium and this book has contributed immensely to the success of Symbian SQL. I'd also like to thank him for his friendship and the opportunity to work in a fantastic team.

We wouldn't have gotten far without Jo Stichbury and Satu Dahl. Thanks to their publishing experience, we had precious guidance throughout the process. I am in debt for their patience, expertise and hard work, without which this book would never have seen the light of day.

I am especially thankful to the wonderful people who have contributed expert content to this book: Richard Maynard, Lorraine Martin, Mike

Owens, Martin Platts, Philip Cheung, James Clarke, James Aley and Philip Neal. Special mentions are due to Richard, Lorraine and Mike. Richard has led the team through the development of Symbian SQL and has contributed immensely to the positive and constructive atmosphere enjoyed by the team. Lorraine's unique knowledge of implementation details, her dedication and hard work have made a huge impact on this book. Mike Owens has contributed valuable content from his seminal book *The Definitive Guide to SQLite* that has been updated and put into the Symbian SQL context.

I would also like to thank Dr Richard Hipp for making SQLite and giving it to the world for free. Dr Hipp has supported our work from the outset and has provided a brilliant foreword to this book.

Thanks also to top Symbian engineers who have reviewed drafts for chapters: Alex Dimitrov, Andrew Thoelke, Renzo Cherin, Lucian Piros and Erhan Cetintas. A very special thanks goes to Chris Dudding who has been an invaluable ally in finalizing the text and has reviewed the whole book with a keen technical eye.

Thanks also go to the good people at John Wiley & Sons who have done a great job transforming the manuscript into a book and getting it to readers. Thanks also to our development editor, Kenyon Brown, for pulling the text together so smoothly and keeping us all on track.

Most of all, I'd like to thank my dear wife Aleksandra and my wonderful son Boris who have kept me well-fed and cheerful during long writing and editing sessions.

In the end, this was a great experience, and one from which I've learnt a lot. I hope you enjoy this book and put it to good use.

Ivan Litovski
London, October 2009

Symbian Acknowledgments

Satu and I would like to thank Ivan wholeheartedly for the time, care and attention he's put into this book. When we started this book project, before Nokia's acquisition of Symbian and open-sourcing of the operating system was announced, we had no idea of how prescient the Persistent Data Services team were in their choice of an open code base. However, we quickly understood that we were working with a remarkable team, who took to writing a book as enthusiastically and skillfully as they tackled their 'day jobs'.

We'd like to thank everyone who authored, reviewed or otherwise gave their support to the book project. The result is a solid, one-of-a-kind book that complements the Symbian SQL source code, which you can now access at ***developer.symbian.org***.

Apart from the book, another bonus of the work on this project has been that Ivan has now joined Satu and me in the Symbian Foundation. We are pleased to be able to welcome him to the team, as an author as well as a technical expert. I have no doubt that – once he's caught up on the sleep he sacrificed on this book – he'll be hard at work on another project to share his expertise with fellow Symbian developers. Thanks, Ivan!

Jo Stichbury
Technical Communications, Symbian

Publisher's Acknowledgements

Some of the people who helped bring this book to market include:

- Editorial and Production:
    - VP Consumer and Technology Publishing Director: Michelle Leete
    - Associate Director, Book Content Management: Martin Tribe
    - Associate Publisher: Chris Webb
    - Executive Commissioning Editor: Birgit Gruber
    - Assistant Editor: Colleen Goldring
    - Publishing Assistant: Ellie Scott
    - Content Editor: Claire Spinks
    - Project Editor: Juliet Booker
    - Development Editor: Kenyon Brown
    - Copy Editor: Shena Deuchars
- Marketing:
    - Senior Marketing Manager: Louise Breinholt
    - Marketing Executive: Kate Batchelor
- Composition Services:
    - Compositor: Laserwords Private Limited
    - Proofreader: Shena Deuchars
    - Indexer: Terry Halliday

1
Introduction

This book has been long in the making and written by a fairly large and diverse group of authors. The common cause is supporting Symbian developers in creating new phones, database-driven applications, or porting DBMS-based application code to use Symbian SQL.

In order to understand just how important this cause is, we only need to consider the massive growth of file storage available on today's mobile devices. Just a few years ago, storage space was measured in megabytes and even desktops were shipping with 20 GB hard disks. Modern media devices are unlikely to succeed without gigabytes of storage. The expansion shows no signs of slowing as users become content creators as much as content consumers. With the ability to store thousands of songs, photos or videos on phones and the corresponding need to catalog them, it is clear that a relational database is a necessary component of a future-proof, scalable, mobile operating system.

Applications such as media management, as essential as they are, only scratch the surface of what mobile databases are about. Relational databases are an enabling technology, making possible things that we cannot always anticipate. A great example is the Colombo Search service discussed in Section 9.2 – a 'desktop' search API for Symbian that allows searching contacts, calendar, messaging, bookmarks, notes, and more, through a single, simple API.

1.1 Where Should You Start?

In writing this book, we envisage two major groups of readers – those familiar with relational databases who want to re-acquire the skill on Symbian and those familiar with Symbian C++ who want to acquire database skills or to move from DBMS to Symbian SQL.

For both our major target groups, after this introduction, a good starting point is Chapter 2 which offers a quick HelloWorld example using both

Symbian C++ and Symbian SQL database API. Get your hands dirty and run it on the phone. With your environment set up and a working example, it will be considerably easier to follow the remaining text of the book.

Prior knowledge of databases is not required – this book will get you up to speed in the first few chapters. However, beginners in Symbian C++ are strongly advised to read more about it before taking on this book. There are several good books on Symbian C++ and we can warmly recommend *Symbian OS C++ for Mobile Phones Volume 3* by Richard Harrison and Mark Shackman and *Quick Recipes on Symbian OS* by Michael Aubert, the encyclopedic how-to book that no developer should be without. Details of all the books available for Symbian developers can be found at ***developer.symbian.org/wiki/index.php/Category:Book***.

1.2 Symbian Terminology and Version Numbering

In June 2008, Nokia announced its intention to acquire the shares in Symbian Ltd that it didn't already own and create a nonprofit organization, called the Symbian Foundation. By donating the software assets (formerly known as Symbian OS) of Symbian Ltd, as well as its own S60 platform, to this nonprofit entity, an open source and royalty-free mobile software platform was created for wider industry and community collaboration and development. The acquisition of Symbian Ltd was approved in December 2008 and the first release of the Symbian platform occurred in June 2009.

Most of the material in this book applies to versions of Symbian code released before the creation of the Symbian Foundation. Table 1.1 maps the naming and numbering of the releases of the Symbian platform, Symbian OS and S60 covered in this book. Symbian^1 is a renaming of S60 5th Edition and Symbian OS v9.4 to denote the starting point for the Symbian platform releases. The first version of the platform released independently by the Symbian Foundation is Symbian^2 (pronounced 'Symbian two').

1.3 The Relational Model and the Structured Query Language (SQL)

For most Symbian C++ engineers looking to develop database-backed applications, details of the relational model may be helpful but are by no means necessary. By analogy, it is not necessary to grasp all the inner workings of the internal combustion engine in order to be a good driver. However, for deeper understanding of the underlying theory common

Table 1.1 Versions of the Symbian platform

Symbian platform	S60	Symbian OS
n/a	3rd Edition	v9.1
n/a	3rd Edition, Feature Pack 1	v9.2
n/a	3rd Edition, Feature Pack 2	v9.3
Symbian^1	5th Edition	v9.4
Symbian^2	5th Edition, Feature Pack 1	v9.4 (and some features back-ported from v9.5)
Symbian^3	n/a[1]	v9.5 (and community contributions)
Symbian^4	n/a	n/a[2]

[1]When the Symbian Foundation is fully operational, Nokia is expected to stop marketing S60 as an independent brand.
[2]Although Symbian Ltd had a roadmap beyond v9.5, the content of future releases is unlikely to resemble the previous plans very closely.

to all databases, Chapter 3 is an essential read. Apart from gaining a respectable level of geekiness, concepts such as keys, views, referential integrity, normal forms and normalization are explained there. Reading this chapter will help you know what you are doing.

Some readers may not know about SQL. It is the language we use to interact with databases. The Symbian platform uses an application programming interface (API) to pass SQL queries to the database and retrieve results. You can learn SQL in Chapter 4, which describes it in great detail and is specifically targeted at SQLite.

Database engineers may find Chapters 3 and 4 useful for reference. Chapter 4 in particular may be helpful for switching from other relational database management systems to SQLite.

1.4 What Is Symbian SQL?

From its earliest days, Symbian OS was intended to support a variety of personal information management tools. Its designers foresaw that applications such as contact management, agendas, to-do lists, and email would be central to the mobile user experience of the future, an idea that is borne out today.

These applications share a common theme: they all have to store and retrieve structured data and, perhaps more importantly, this data is often central to the user's everyday life. It is therefore essential that the data is protected from loss or corruption. Who would trust a phone if it lost all its contact information when the battery ran out?

These facts led the designers of Symbian OS to include a robust set of storage services as part of the original operating system. These included a lightweight database called DBMS which had a slew of features and served as a major enabler. It was used for the contacts, messaging and multimedia applications shipping with the operating system in several generations of mobile phones.

DBMS is famed for an extremely small footprint, especially its ability to run in only a few kilobytes of memory. Its main advantage was that it was built from the ground up for Symbian OS. In several scenarios – usually with a low number of rows – DBMS produces stellar performance.

In recent years, however, shortcomings in DBMS have begun to become apparent. Very limited support for SQL and relational operations make porting database applications from different environments more difficult and require native applications to include complex code to perform operations which are supported by more capable relational databases. Several low-level operations, such as compaction and index-validity checking, are exposed and require development attention. Although detailed and functional, the DBMS API is somewhat different to most modern database APIs, resulting in difficulties in porting and a skewed learning curve. Although built with some scalability in mind, DBMS was designed primarily for low resource usage. Thousands of records in a database were extremely rare while memory prices were a major driving force in device design. Today, when applications may need to store several thousand records or more, DBMS performance does not scale well enough to meet performance requirements.

This state of affairs triggered the search for a new database. Several candidates were considered and analyzed, and a new engine was selected. The choice was SQLite, a database which offers many advantages: excellent SQL support, good performance, a small footprint, good portability and unencumbered licensing – it is in the public domain. Moreover, the SQLite project has a group of brilliant engineers and a great community which keeps it sharp and up to date. In order to support further development of SQLite and safeguard its future, Symbian Software Ltd and Mozilla collaborated with the SQLite developers to found the SQLite Consortium.[1] Today, the consortium has expanded to include Adobe and Bloomberg.

Two significant Symbian requirements could not be accommodated by SQLite alone. First, Symbian needs the ability to securely share a database between clients: only clients that satisfy a per-database security policy can be allowed access. Second, in standard SQLite, if multiple

[1] ***www.sqlite.org/consortium.html***

clients share a single database then each of the clients has a private cache of the database in its address space. This is not appropriate for a resource-constrained mobile device; Symbian needs a model where multiple clients share a single cache.

These needs led to the design and implementation of Symbian SQL, which integrated SQLite within a Symbian client–server framework as shown in Figure 1.1. Client–server IPC is the basis for the Symbian platform security framework and a server process is the ideal mechanism for securely sharing access to a single cache. For the design to become a reality, Symbian worked with the SQLite developers to evolve the database to meet the requirements. Changes were needed to enable SQLite to share a cache between multiple connections and to improve operational characteristics under low-memory conditions.

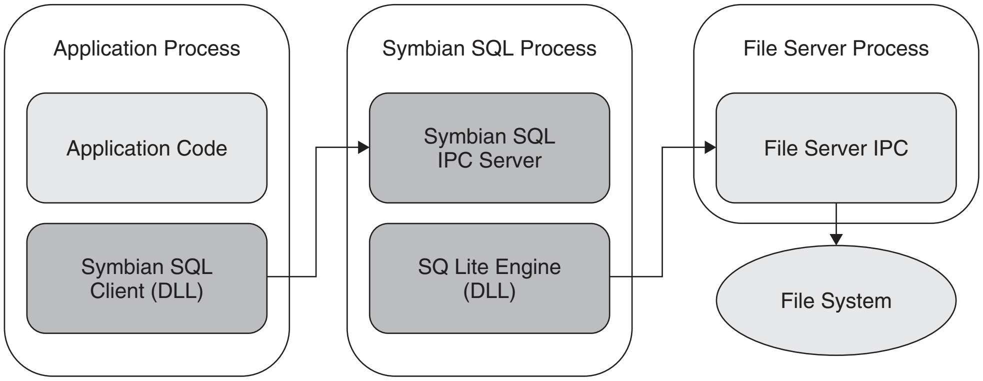


Figure 1.1 Symbian SQL architecture

A number of client APIs were considered, including the native SQLite C API. The final choice was a native Symbian C++ API which supported Symbian idioms, such as descriptors and asynchronous requests. Engineers familiar with databases will immediately recognize the connection and statement classes and find that they have the expected functionality. We focus on the API in Chapter 6 where all is explained in detail.

Symbian SQL was first delivered on devices based on Symbian OS v9.3. Demand was sufficient for it to be offered as an installable version backported to Symbian OS v9.2. The C++ API was ideal for early Symbian clients but it was recognized that the standard SQLite C API would be a valuable addition to facilitate porting SQLite-based, open source applications. The C API was introduced in a later release. It is covered only briefly in Chapter 7 because it is well documented online, especially at ***www.sqlite.org***.

After the initial release of Symbian SQL, collaboration with the SQLite developers continued to further enhance performance and functionality. Chapter 7 explains some of these enhancements in detail.

1.5 A Well-Oiled Machine

One of the main goals of this book is to help readers to get the most out of Symbian SQL. Symbian SQL can offer excellent performance, however at times this requires a level of expertise and some work on optimizing the application or database schema, or even on tuning system parameters. In order to help in the process, Chapter 8 is entirely dedicated to performance tuning.

Finally, we wanted to share some success stories. We have recruited three happy engineers that have earned bragging rights in getting their applications to fly with Symbian SQL. Chapter 9 tells their stories – if you use a Symbian phone, you have benefited from their work too.

1.6 Tools and Troubleshooting

SQLite is great for learning and experimenting. Essentially, all that is necessary to start experimenting is one executable (`sqlite3.exe`) that is available for a variety of platforms. Using this tool, developers can create databases, experiment with queries and analyze performance. This is very beneficial because it eliminates the need for installing and maintaining a separate database server or running in an OS emulator.

Once prototyping is done, we are faced with a variety of questions on specifics, such as the compaction (auto-vacuum) policy, security options, and page size.

A collection of common problems, including programming and execution errors, is included in Appendix A.

Further to this, Symbian SQL databases contain a metadata table. To cater for all these requirements, Symbian provides a tool, called SDB, that offers a systematic way of configuring all database options. Appendix B covers the SDB tool in more detail.

1.7 Further Reading and Resources

We originally started writing this book as part of the Symbian Press Technology series back in 2008. Since that time, a lot has changed in the mobile software world, for us in Symbian (now Nokia) and for those working with Symbian devices.

Determining the content of a technical book in a fast-moving industry is never easy but when organizations and strategies are changing, it becomes extremely difficult. For that reason, several sections of this book have been revised to record the most relevant and up-to-date information. Nevertheless, we anticipate further changes, if only to some of the URLs we reference. For that reason, you can find a wiki page for this book at ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***, which also hosts any example code accompanying the book and records errata and updates to the information presented in this version of the text.

A variety of additional resources about Symbian SQL can be found on the Symbian developer site, ***developer.symbian.org***, such as the API reference for the Symbian platform and support from peer developers via a number of discussion forums.

The official SQLite website at ***www.sqlite.org*** provides various articles detailing the behavior of the underlying SQLite engine. These include:

- SQL syntax
- atomic commit
- locking and concurrency
- an overview of the optimizer.

The official SQLite website also offers support for the underlying SQLite engine through mailing lists that announce future releases and enable user discussions.

2
Getting Started

A journey of a thousand miles starts with a single step.
Lao-tzu

Symbian SQL is easy to learn and straightforward to use. This chapter aims to provide everything needed to start using it. The first section in this chapter provides details of where and how to get Symbian SQL. Basic database operations are then demonstrated in a console application example. Finally, the location of additional documentation is listed.

2.1 Where to Get Symbian SQL

Symbian SQL is included with SDKs based on Symbian OS v9.3 onwards, including all Symbian Foundation releases. From the device perspective, Symbian SQL is included in S60 3rd Edition Feature Pack 2 onwards. For users of S60 3rd Edition Feature Pack 1, Symbian SQL can be downloaded in a SIS file from the Symbian Foundation developer website at ***developer.symbian.org***. Additional releases may be supported in the future. Please check the wiki page for this book (***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***) for more details.

2.2 Overview of Symbian SQL APIs

Symbian SQL consists of two main classes along with other classes used for streaming binary large objects (BLOBs), binding columns, defining security polices and providing utility functions.

The two most important classes are as follows:

- `RSqlDatabase` enables a client to create, open and attach to a database and to execute queries which do not require data to be returned (e.g. INSERT queries).
- `RSqlStatement` enables a client to execute queries, including ones which return data (e.g. SELECT queries).

The other classes are:

- `RSqlSecurityPolicy` defines a security policy for a shared secure database.
- `TSqlScalarFullSelectQuery` is used to execute queries that return a single row consisting of a single column value.
- `RSqlColumnReadStream` is used to read the content of a column containing either binary or text data.
- `RSqlParamWriteStream` is used to set binary or text data into a parameter.
- `RSqlBlobReadStream` provides a more memory-efficient method to read binary or text data column.
- `RSqlBlobWriteStream` provides a more memory-efficient method to set binary or text data column.
- `TSqlBlob` is used to read or write the entire content of a binary or text data column in a single call.

These classes are discussed in more detail in Chapter 6. The BLOB classes are available from Symbian^3 onwards.

2.3 First Database Example

In order to introduce Symbian SQL, we write a simple music library database application. It illustrates key database operations using a simple, console-based interface. The console interface has been chosen to minimize complexity by avoiding unnecessary GUI code. The Symbian SQL APIs work in both GUI and non-GUI applications, so the code can be easily incorporated into a GUI application. The example code archive can be found on this book's wiki page at ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***.

The application uses various Symbian SQL operations to insert song entries into the database, and modify and retrieve the entries. We do these operations in the following sequence:

1. Create the empty music database.
2. Create a table of songs in the database.
3. Create an index.
4. Insert a number of song entries into the table.
5. Select all song entries and display them in the console.
6. Update a song entry within the table.
7. Update song entries using column binding.
8. Delete a song entry from the table.
9. Close the connection to the database file.

2.3.1 Project Setup

In order to use Symbian SQL, your source files should include `sqldb.h`. The Symbian SQL library, `sqldb.lib`, must be included from your project MMP file.

2.3.2 Creating a Database

The first task is to open a connection to a database file; this can be done in two ways:

- Create (and connect to) an empty database file
- Open (and connect to) an existing database file

In this example, we start from scratch and create a new database file. This is done by using `RSqlDatabase::Create()`, which creates a database file at the specified location. Subsequently, a connection is established between the client and the Symbian SQL server.

```
RSqlDatabase database;
CleanupClosePushL(database);
_LIT(KDbName, "C:\\MusicDb.db");
TInt error = database.Create(KDbName);
//Handle any errors and ignore KErrAlreadyExists
```

The file name passed to the `Create()` function is the full path of the database. This may be up to a maximum of 256 characters in length (the Symbian standard).

An overloaded version of this function is used to create a shared secure database. Shared secure databases allow different applications to access a database stored in a secure manner. Access to these databases is granted based on the capabilities defined by the Symbian platform security model. Shared secure databases are discussed in more detail in Chapter 6.

`RSqlDatabase::Open()` can be used to open an existing database file. The full file path is passed to the function, which is used to open both secure and non-secure databases.

2.3.3 Creating a Table

After the database file is created, the next step is to create a table so that data can be stored. The layout of tables is known as the 'schema'. The design of a database table schema is a complicated subject and is discussed in more detail in Chapter 5.

For this example, a single table is added to the database file. The song itself is stored in the table in its raw binary format as a BLOB. The songs table layout is described in Table 2.1.

Table 2.1 Layout of the songs table

Name	**Data type**
id	INTEGER
title	TEXT
artist	TEXT
song	BLOB

The `RSqlDatabase::Exec()` function is used to execute SQL queries which do not require data to be returned (e.g., Insert, Update or Delete queries). The function is used to execute the SQL CREATE TABLE statement:

```
_LIT(KCreateTable, "CREATE TABLE songsTbl(id INTEGER, title TEXT,
                                          artist TEXT, song BLOB)");
// The database object is already connected to a database file
// and placed on the cleanup stack
User::LeaveIfError(database.Exec(KCreateTable));
```

2.3.4 Creating an Index

Indexes are created to speed up search operations. In this example, the index is used to retrieve entries from the songs table. Further details of indexes, including recommendations on their use, are given in later chapters.

An index on columns is created by executing the SQL CREATE INDEX statement using RSqlDatabase::Exec():

```
_LIT(KCreateIndex, "CREATE INDEX TitleIdx on songsTbl(title)");
// The database object is already connected to a database file
// and placed on the cleanup stack
User::LeaveIfError(database.Exec(KCreateIndex));
```

2.3.5 Inserting Records

Three records are added to the table using SQL INSERT statements. Again, as the INSERT query does not require any data to be returned, the function RSqlDatabase::Exec() is used.

```
_LIT(KBegin, "BEGIN");
_LIT(KCommit, "COMMIT");
_LIT(KInsertRec1,"INSERT INTO songsTbl VALUES(1,
   'Like a Rolling Stone', 'Bob Dylan', NULL)");
_LIT(KInsertRec2,"INSERT INTO songsTbl VALUES(2, 'Satisfaction',
                                  'The Rolling Stones', NULL)");
_LIT(KInsertRec3,"INSERT INTO songsTbl VALUES(3, 'Imagine',
                                    'John Lennon', NULL)");

// The database object is already connected to a database file
// and placed on the cleanup stack
User::LeaveIfError(database.Exec(KBegin));
User::LeaveIfError(database.Exec(KInsertRec1));
User::LeaveIfError(database.Exec(KInsertRec2));
User::LeaveIfError(database.Exec(KInsertRec3));
User::LeaveIfError(database.Exec(KCommit);
```

Note that the INSERT operations are placed in a transaction which consists of a BEGIN at the start and a COMMIT at the end. Transactions are recommended when executing multiple SQL statements consecutively. This is because a transaction delays the persisting of the changed data to the database file until the COMMIT operation, which improves performance for bulk insertions. Transactions are discussed in more detail in Chapter 4. After the COMMIT operation is complete, the data is persisted to the database file.

2.3.6 Selecting Records

An RSqlStatement object allows data to be returned to the client. Firstly the SQL statement is passed into RSqlStatement::PrepareL(). The function compiles the statement into SQLite bytecodes, so that it can be evaluated (bytecodes are discussed in more detail in Chapter 7). After preparing the SQL statement, RSqlStatement::Next() is called to evaluate the pre-compiled statement. If the function returns KSqlAtRow, then a record has been successfully retrieved from the query and the column data can be accessed using the appropriate RSqlStatement member function.

```
_LIT(KSelectRec, "SELECT id, artist, title FROM songsTbl
                                       ORDER BY title");
_LIT(KId, "id");
_LIT(KTitle, "title");
_LIT(KArtist, "artist");
TInt id;
TPtrC title;
TPtrC artist;
RSqlStatement stmt;
CleanupClosePushL(stmt);

// The database object is already connected to a database file
// and placed on the cleanup stack
stmt.PrepareL(database, KSelectRec);
TInt idCol = stmt.ColumnIndex(KId);
TInt titleCol = stmt.ColumnIndex(KTitle);
TInt artistCol = stmt.ColumnIndex(KArtist);
User::LeaveIfError(idCol);
User::LeaveIfError(titleCol);
User::LeaveIfError(artistCol);

TInt err = KErrNone;
//Iterate the result set and retrieve each record
while((err = stmt.Next()) == KSqlAtRow)
{
  id = stmt.ColumnInt(idCol);
  stmt.ColumnTextL(titleColIdx, title);
  stmt.ColumnTextL(artistColIdx, artist);
  //<Display the data on screen>
}
//Check for errors
if (err != KSqlAtEnd)
{
  //In this case an error has occurred with Next()
  //therefore the error is handled here
}
CleanupStack::PopAndDestroy(&stmt);
```

Note that the SELECT statement uses an ORDER BY clause to sort the records returned using the title column. However, the index on the title column significantly speeds up this ORDER BY operation; because the index already contains references to the table sorted in order of the title column, no further sorting is necessary.

The RSqlStatement function ColumnIndex() is used to identify the ordinal number of each column from the column name. This function can return a negative value if an error occurs, so where applicable it is good practice to call User::LeaveIfError(), or handle the error in some other way. This index value is then passed to the RSqlStatement functions `ColumnInt()` and `ColumnTextL()` to retrieve the data for the current row and return them as integer and descriptor data types respectively.

Finally the `RSqlStatement` object needs to be closed after it is used by calling the `RSqlStatement::Close()` function (or `CleanupStack::PopAndDestroy()` if it is on the cleanup stack, as in the example). This frees any resources allocated to the object, otherwise memory leaks occur in the application and in the Symbian SQL server.

2.3.7 Updating a Record

Next, a single record is updated in the table. Since the update operation requires no data to be returned, the `RSqlDatabase::Exec()` function is used. Here the title of the first record is changed to 'Blowin' in the Wind'.

```
_LIT(KUpdateRec1, "UPDATE songsTbl SET title = 'Blowin'' in the Wind'
                              WHERE title ='Like a Rolling Stone'");
// The database object is already connected to a database file
// and placed on the cleanup stack
User::LeaveIfError(database.Exec(KUpdateRec1));
```

2.3.8 Updating Records Using Column Binding and Streaming

Next, the song is included in each record by streaming the data into the song column. In order to achieve this, the example introduces two new concepts:

- column binding
- write streaming.

`RSqlParamWriteStream::BindBinaryL()` is used to bind to the song column. After this the binary content of the song can be streamed into each record:

```
_LIT(KBulkUpdate, "UPDATE songTbl(song)  SET song=:Prm1
                                     WHERE id=:Prm2");
_LIT(KSongVal, ":Prm1");
_LIT(KIdVal, ":Prm2");
const TInt numOfRecs = 3;
RSqlStatement stmt;
CleanupClosePushL(stmt);
// The database object is already connected to a database file
// and placed on the cleanup stack
stmt.PrepareL(database, KBulkUpdate);
TInt songPrmIndex = stmt.ParameterIndex(KSongVal);
TInt idPrmIndex = stmt.ParameterIndex(KIdVal);
User::LeaveIfError(songPrmIndex);
User::LeaveIfError(idPrmIndex);
for(TInt i=1; i=<numOfRecs; i++)
{
  User::LeaveIfError(stmt.BindInt(idPrmIndex, i));
  RSqlParamWriteStream writeStrm;
  CleanupClosePushL(writeStrm);
  writeStrm.BindBinaryL(stmt, paramIndex);
  //<Stream the data>
  writeStrm.CommitL();
  User::LeaveIfError(stmt.Exec());
  User::LeaveIfError(stmt.Reset());
 CleanupStack::PopAndDestroy(&writeStrm);
}
CleanupStack::PopAndDestroy(&stmt);
```

In binding, an SQL statement is compiled (using `RSqlStatement::Prepare()`) with a placeholder parameter in the statement instead of the actual value. In the above example, `:Prm1` is a placeholder parameter. Once a column is bound, the same statement can be executed many times with different values without recompiling the SQL statement. All data types can be bound using `RSqlStatement` functions. However, the above example used `RSqlParamWriteStream::BindBinaryL()` to bind the column because additional write streaming functionality is provided by this API.

Write streaming allows the user to insert data into a text or a binary column using incremental writes. In this example, `RSqlParamWriteStream` is used to stream data into the bound column. One method for doing this is the `WriteL()` function.

After the SQL statement has been executed, the `RSqlStatement` object is reset to its initial state using the `Reset()` function. This allows the prepared statement to be executed multiple times.

Note that there is a more efficient API that provides write streaming, called `RSqlBlobWriteStream`. However, this new class is only available in Symbian^3 onwards. More details of these APIs are given in Chapter 6.

2.3.9 Deleting a Record

Records are deleted in a similar way to the update operation using the `RSqlDatabase::Exec()` function. Here the record for the song 'Imagine' is deleted from the database.

```
_LIT(KDeleteRec, "DELETE FROM songsTbl WHERE title ='Imagine'");
// The database object is already connected to a database file
// and placed on the cleanup stack
User::LeaveIfError(database.Exec(KDeleteRec));
```

2.3.10 Closing a Database

The connection to the database file is terminated using `RSqlDatabase::Close()`. Alternatively, `CleanupStack::PopAndDestroy()` can be used if the database object is on the cleanup stack.

```
// The database object is already connected to a database file
database.close();
```

2.4 Summary

This chapter was about setting up the environment, getting examples and running some code. We have gone slightly beyond this call and covered the main database operations: initialization and database creation, issuing statements such as INSERT and UPDATE, retrieving information using SELECT and, finally, closing the database. At this point, most developers should be able to use this code to write their own database application – perhaps using the Symbian Developer Library API reference at ***developer.symbian.org***.

Now that we have a working application and an understanding of basic database operations, we can dig deeper and learn more about databases. The next two chapters provide the theoretical basis for databases and a detailed SQL tutorial. This will not just help in writing applications and system software – gaining this knowledge elevates a person's 'cool'.

3
The Relational Model

This chapter is an essential read if you are looking for a deeper understanding of common database theory, and explains concepts such as keys, views, referential integrity, normal forms and normalization. You don't have to read this chapter in order to understand SQL; it merely provides a theoretical and historical backdrop and is an appetizer to Chapter 4, which covers SQL in great detail and is specifically targeted at SQLite. This chapter presents only the minimal theoretical basis needed to get a good understanding and appreciation of the origin, aims, and means of SQL. It presents the theoretical roots of SQL, and examines its power and elegance. The aim is to prepare you to deal with SQL, not in isolation, but in the context of the relational model. If nothing else, it should give you an appreciation for how elegant, powerful, and complex a beast a relational database really is. You may be surprised by what even the most rudimentary relational databases are capable of.

SQL has a very practical exterior but a very theoretical interior. That interior is the relational model. The relational model came before SQL and created a need for SQL. The power of SQL lies not in the language itself but in the concepts set forth in the relational model. These concepts form the basis of SQL's design and operation.

The relational model is a powerful and elegant idea that has pervaded not only computer science but our daily lives as well, whether we know it or not. Like the automobile, or penicillin, it is one of the many great examples of human thought and discovery that has fundamentally impacted the way the world works. The relational model has spawned a billion-dollar industry and has become an integral part of almost every other industry. You'll see the relational model at work nearly everywhere that deals with information of any kind – in Fortune 500 companies, universities, hospitals, grocery stores, websites, routers, MP3 players, cell phones, and even smart cards. It is truly pervasive.

3.1 Background

The relational model was born in 1969, inside IBM. A researcher named E. F. Codd distributed an internal paper titled 'Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks,' which defined the basic theory of the relational model. This paper was circulated internally and was not widely distributed. In 1970, Codd published a more refined version of this paper, which is more widely recognized as the seminal work on relational theory. This is the paper that changed the industry.

Despite the enormous influence of this paper, the relational model in its entirety did not appear overnight or within the scope of a single historic paper. It evolved, grew, and expanded over time and was the product of many minds, not just Codd alone. For example, the term 'relational model' wasn't coined until 1979, nine years after publication of the paper proposing it (Date 1999). Codd's 12 rules, now famous as the working definition of the relational model, weren't posited until 1985, appearing in a two-part series published by *ComputerWorld* magazine (Codd 1985a; 1985b). Thus, while there was a seminal paper proposing the general concept, much of the relational model as we know it today is actually the result of a series of papers, articles, and debates by many people over a 30-year period and, indeed, continues to develop to this very day. That is also why in some of the various quotes included in this chapter you may see terms such as 'data base,' which may initially appear to be an error. They aren't. At the time of these quotes, the terminology still had not congealed and all sorts of terms were being thrown around.

3.1.1 The Three Components

By 1980, enough was known about the relational model for (Codd 1980) to identify three principal components:

- **The structural component**: This component defines how information is structured or represented. Specifically, all information is represented as *relations*, which are composed of *tuples*, which in turn are composed of *attribute* and *value* components. The relation is the sole data structure used to represent all information in the database. Relations, as defined in the relational model, are derived from relations in set theory, a branch of mathematics, and share many of their properties. They are formally defined in (Codd 1970).
- **The integrity component**: This component defines methods that enforce relationships within and between relations (or tables) in the structural component. These methods are called *constraints* and are expressed in the form of rules. There are three main types of integrity: domain integrity, governing values in columns; entity integrity, governing rows in tables; and referential integrity, governing how tables

relate to one another. Integrity has no analog in set theory – it is unique to relational theory. It was initially addressed in (Codd 1970) and greatly expanded upon in the 1980s by Codd and others.

- **The manipulative component**: This aspect defines the methods with which to operate on or manipulate information. Like relations, these operations also have their roots in mathematics. They are formalized in *relational algebra* and *relational calculus* as originally presented in (Codd 1972).

SQL is structured similarly along these lines – so similarly, in fact, that it is hard to talk about SQL without addressing the relational model to some degree, directly or indirectly. It is true that SQL is, for the most part, a straightforward language, which to many people appears as an island unto itself. But in reality, it is the offspring of the relational model and, in many ways, is clearly a reflection of it.

3.1.2 SQL and the Relational Model

In this chapter, I present the theoretical aspects in terms of several influential papers and articles written by Codd between 1970 and 1985 that define the essence of the relational model, along with the ideas and work of other contributors to the field.

In the following sections, I illustrate the relational model in terms of its three components in general and in the context of Codd's 12 rules in particular. Together, these rules lay the groundwork for much of what you will see in SQL, casting a great deal of light on why it is the way it is as well as what it means to be relational. The rules as they are presented here are taken directly from (Codd 1985a).

3.2 The Structural Component

The structural component of the relational model lays the foundation upon which the other components build. It defines the form in which information is represented. It is defined by the first of Codd's 12 rules, which is the cornerstone of the relational model.

3.2.1 The Information Principle

The first rule, called the *Information Rule*, is also known as the *Information Principle*. It is defined as follows:

> ***Rule 1. The Information Rule.*** *All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.*
>
> Codd (1985a)

Date (2003) summarizes this rule as follows:

> *The entire information content of the database is represented in one and only one way, namely as explicit values in column positions in rows in tables.*

There are two important expressions in the information principle: 'logical level' and 'values in tables.' The logical level, or logical representation, refers to the way that the user sees the database and information within it. It is a kind of ideal worldview for information. The logical level is a consistent, uniform depiction of data, which has two important properties:

- The view presented in the logical level consists of tables, made up of rows, which in turn are made up of values.
- The view is completely independent of the database system – the technology (software or hardware) that enables it.

The logical representation is a world unto itself, which is completely distinct from how the database is implemented, how it stores data physically, and how it operates internally. These latter components – software, operating system, and hardware – are referred to as the *physical representation*, the technology of the system. If the database vendor decides to store database tables in a different way on disk, the information principle mandates that such a change in physical representation can in no way affect or change the logical representation of that data – the way in which the user (or programs) see that data. The logical representation is independent of the physical representation. A result of the information principle is that it is possible to represent the same information in the same way across multiple database implementations on multiple operating systems on different hardware, as shown in Figure 3.1.

You can create a relation (or table) in Oracle on Solaris that is represented in exactly the same way as a table in PostgreSQL on Linux or SQLite on Mac OS X.[1] The information principle guarantees you a consistent logical view of information regardless of how the database software is implemented, or the operating system or hardware it runs on.

[1] In reality, this is not 100 per cent true (it's more like 95 per cent true). There are slight differences in database implementations so that relations in one database may contain features not present in other databases. Nevertheless, they still all adhere to the same general structure: relations made of tuples made of values.

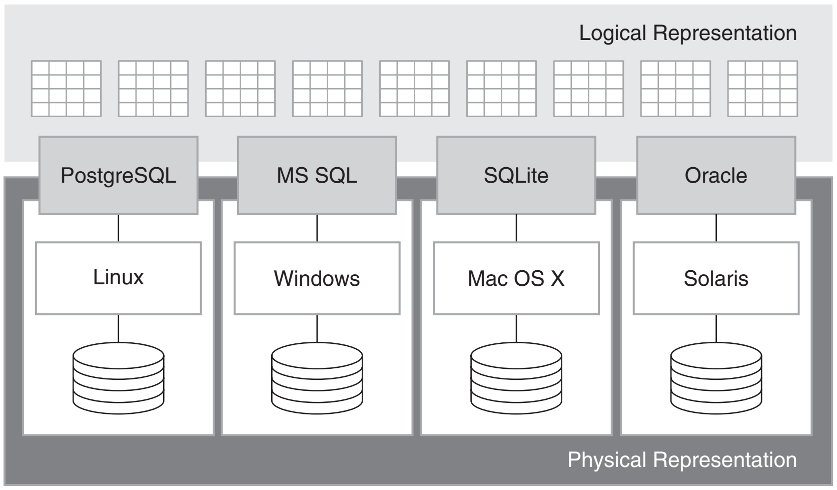


Figure 3.1 Logical and physical representations of data

3.2.2 The Sanctity of the Logical Level

So important are these two constraints in the relational model that they are expanded upon and reinforced in several other rules (8, 9, 11, and 12) to eliminate any possible ambiguity. In short, Codd (1985a) says:

> ***Rule 8. Physical Data Independence.*** *The logical view can in no way be impaired by the underlying software or hardware.*
> ***Rule 9. Logical Data Independence.*** *Changes even to the logical level do not impact applications using the database.*
> ***Rule 11. Distribution Independence.*** *Even if the database is spread across various locations, it cannot impact the logical view of data.*
> ***Rule 12. Nonsubversion Rule.*** *The database software may not provide any facility which can subvert the integrity constraints of the logical view.*

The separation of the logical level from physical was very important to Codd from the outset. In the opening of his original paper, he had strong words concerning this separation:

> *Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)... Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed.*
>
> Codd (1970)

The relational model was in part a reaction to the database systems of the day, which closely tied applications to both database implementation and data format on disk. Codd's relational model challenged this:

> *It provides a means of describing data with its natural structure only – that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.*
>
> Codd (1970)

So, in the relational model, the information principle provides a level of abstraction through which information can be represented in a consistent way. The user sees and works with data exclusively in terms of this logical representation, which is completely insulated from the underlying technology. It cannot be undermined, influenced, or affected by it in any way. As stated before, the information principle is the foundation of the relational model.

3.2.3 The Anatomy of the Logical Level

The logical level is made up of more than relations. It is made up of *tables*, *rows*, *columns*, and *types*. In relational parlance, these are often referred to more formally as *relation variables*, *tuples*, *values*, and *domains*, respectively. SQL, for example, uses many of the former terms and relational theory tends to use many of the latter terms. Throughout the literature, however, these terms are used almost interchangeably. Even in Codd's papers, both sets of terms are used. Interestingly, there is one term that both lexicons have in common: *relations*. (In case you're wondering, tables and relations are not the same thing; we'll address the difference later in this chapter.)

This big soup of terminology comprises the anatomy of the relational body, which is explained in detail over the next few sections. The relationships between the terms are illustrated in Figure 3.2.

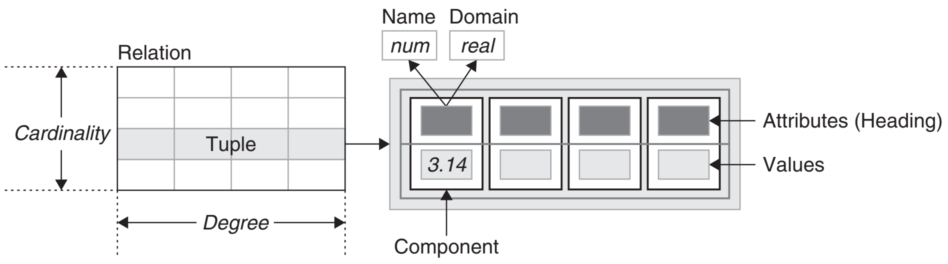


Figure 3.2 The logical representation of data

At the center of everything is the relation. It is the central object around which the structural, integrity, and manipulative components of the relational model are built. All of the fundamental operations in the relational model are expressed in terms of relations and all integrity constraints are defined within relations. Your understanding of the relational model is only as good as your understanding of relations themselves. To have a good grasp of relations, however, you must first understand tuples.

3.2.4 Tuples

A tuple is a set of *values*, each of which has an associated *attribute*. An attribute defines a value's *name* and *domain*. The attribute's name is used to identify the value in the tuple and its domain defines the kind of information stored within it. The combination of attribute and value is called a *component*. In Figure 3.2, the first component is named *num* and it has a domain of *real* (as in real numbers) and a value of 3.14.

A component is a tidy, self-contained unit of data. It has three essential ingredients: a name, a domain, and a value. Its name gives it identity, its domain a description of its content, and its value the content itself. Together, components aggregate into larger structures such as tables and relations, and their qualities propagate into those structures, imparting to them identity, description, and content as well.

The name and value parts of a component are easy enough to understand, but what exactly is a domain? The word *domain*, like the words *relation* and *tuple*, comes from mathematics. In mathematics, the domain of a function is the set of all values for which the function is defined. Some familiar domains in mathematics are the sets of all integers, rational numbers, real numbers, and complex numbers. Generally, the term *domain* corresponds to a set of permissible values. A domain is sometimes referred to as a *type* and is synonymous with data types in programming languages. A domain, especially in the relational sense, also implies a set of operators that can be used to operate on its associated values. For example, common operators associated with integers are addition, subtraction, multiplication, and division.

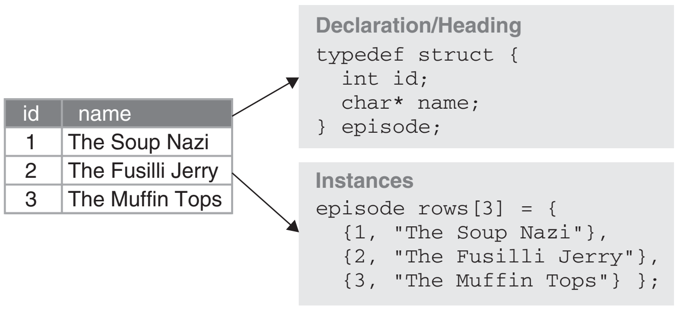
So the job of a domain is to define a finite or infinite set of permissible values along with ways of operating on them. In a way, the domain controls or restricts the value of an attribute, but it does so only by providing information. The database system restricts an attribute's value so that it conforms with its associated domain. As you will see later, this restriction is defined in the integrity component of the relational model and is called *domain integrity*.

The group of collective attributes in a tuple is called its *heading*. Just as an attribute defines the properties of its associated value, the heading defines the properties of the tuple.

3.2.5 Relations

A relation, simply enough, is a set of one or more tuples that share the same heading. Just as domains have analogs in programming languages, so do tuples and relations. And even if you are not a programmer, the analogy is quite helpful in illustrating the relationship between tuples, relations, and headings.

For example, consider the relation and its C equivalent shown in Figure 3.3. You could say that a tuple is similar to a C `struct`. They are both made up of attributes that have a name and a type. As shown in the figure, a C `struct`'s attributes are defined in its declaration, just as a tuple's attributes are defined in its heading. The declaration and heading both provide information about the contents of their respective data structures.



id	name
1	The Soup Nazi
2	The Fusilli Jerry
3	The Muffin Tops

```
typedef struct {
  int id;
  char* name;
} episode;
```

```
episode rows[3] = {
  {1, "The Soup Nazi"},
  {2, "The Fusilli Jerry"},
  {3, "The Muffin Tops"} };
```

Figure 3.3 Header as declaration and tuple as instance

Similarly, a tuple's values are analogous to an *instance* of a C `struct`. Its values form a composite data structure that corresponds to the attributes defined in its heading. Each value in the structure is identifiable by an attribute name and each value is restricted by an attribute domain.

Thus, tuples are more than just rows of amorphous values such as you might find in a spreadsheet. They are well-defined data structures with a high degree of specificity over the information within them. They have more in common with constructs in a programming language than with rows in a spreadsheet.

But the analogy doesn't stop there. As also shown in Figure 3.3, a relation is the equivalent of an array of `structs` in C. Each structure in the array along with the array itself share the same type, just as relations and their tuples share the same heading.

The bottom line is that relations and tuples are highly structured. Furthermore, this structure is defined by their common heading.

Degree and Cardinality

Associated with tuples and relations are the notions of *degree* and *cardinality*. These are just fancy words for width and height, respectively. You could say a relation's width is the number of attributes in its heading. This is called the degree. Its height is the number of tuples it contains; this is called cardinality. These terms are illustrated in Figure 3.2. A relation with four attributes and five tuples would be said to be of degree four and cardinality five.

While tuples don't have cardinality, they nonetheless have their own fancy terms. A tuple of degree one is said to be a *unary* tuple. Tuples of degree two, three, and four are said to be *binary*, *ternary*, and *quaternary*, respectively. Generally, a tuple of degree *N* is said to be an *N*-ary tuple. In fact, the word *tuple* is taken from the *N*-ary form. In the strictest sense, a unary tuple is called a *monad*, a binary tuple a *pair*, a ternary tuple a *triple*, a quaternary tuple a *tetrad*, and so on, as shown in Table 3.1.

Table 3.1 Tuple terminology

Degree	Qualification	Designation	Example
1	Unary	Monad	1
2	Binary	Pair, twin	(0, 1)
3	Ternary	Triple, triad	(0, 1, 2)
4	Quaternary	Quadruple, tetrad	(0, 1, 2, 3)
N	*N*-ary	*N*-tuple (tuple)	(0, 1, 2, . . . , *N*)

Typically, the term *tuple* is used as a generic catchall for tuples of all degrees, and using terms such as monad and triad is often more confusing than it is precise. As with tuples, a relation composed of binary tuples is a binary relation. A relation composed of *N*-ary tuples is an *N*-ary relation, and so forth.

Mathematical Relations

The notion of a relation in relational theory is taken directly from the relation in set theory, with a few modifications. To really understand relations, as defined in relational theory, you must understand their predecessors in mathematics.

Just as in relational theory, mathematical relations are sets of mathematical tuples. Mathematical relations and mathematical tuples differ somewhat from their relational namesakes. In mathematics, tuples are ordered sequences and relations are unordered sets. Sets by their very nature are indifferent to order. That is, the order of elements in a set does not affect the fundamental identity of that set, whereas the order of items in a sequence does affect its identity.

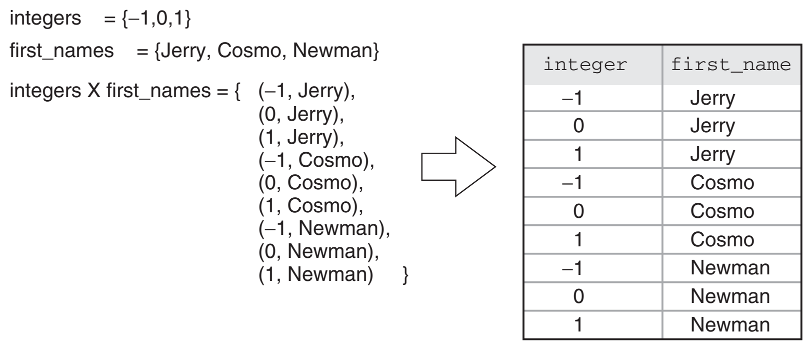
To begin with, every value in a tuple has a specific domain associated with it. Suppose we have a tuple composed of the following two domains:

- The domain of all integers {. . . , −1, 0, 1, . . . }, denoted by **I**
- The domain of the first names of all *Seinfeld* characters[2] {'Jerry', 'Cosmo', 'Newman', . . . }, denoted by **F**

Now, suppose we have a relation composed of such tuples. The relation then is also composed of the domains **I** and **F** (in that order). The first column of the relation must consist of integer values, and the second column must consist of the first name of a *Seinfeld* character. Using this as an example, the formal definition of a relation can be expressed as follows:

> A relation over **I** and **F** is any subset of the cross product of **I** and **F**, represented by **I**~TMS**F**.

This is a somewhat annoying definition because it defines a relation in terms of another perhaps unfamiliar concept: the *cross product*. The cross product, also called the *Cartesian product*, is quite simple, however. It is the combination of every value in a domain with every value in every other domain. To compute **I**~TMS**F**, for example, you take each integer *i* in **I** and pair it with each name *f* in **F**. This yields an infinitely large set of (*i*,*f*) tuples (binary tuples), as illustrated in Figure 3.4. Note that the figure is somewhat simplistic and depicts the set of all integers as the values {−1, 0, 1}.



```
integers    = {–1,0,1}
first_names    = {Jerry, Cosmo, Newman}
integers X first_names = {  (–1, Jerry),
                            (0, Jerry),
                            (1, Jerry),
                            (–1, Cosmo),
                            (0, Cosmo),
                            (1, Cosmo),
                            (–1, Newman),
                            (0, Newman),
                            (1, Newman)    }
```

integer	first_name
–1	Jerry
0	Jerry
1	Jerry
–1	Cosmo
0	Cosmo
1	Cosmo
–1	Newman
0	Newman
1	Newman

Figure 3.4 The cross product of integers and *Seinfeld* character first names

What does the cross product do here? Why is it used? The cross product of domains **I** and **F** is a set containing every tuple that could ever exist

[2] The use of *Seinfeld* characters in this example is explained further in the next chapter. Please see Section 4.2 or consult ***en.wikipedia.org/wiki/Seinfeld*** for more information.

by the combination of these two domains. That is, every tuple that has an integer as its first value and the first name of a *Seinfeld* character as its second value is contained in the cross product **I**~TMS**F**. The cross product is itself just a set, albeit a very big one. It is a set defining the limit of every tuple that could ever be formed from the domains **I** and **F**. That being said, any subset of this cross product is a relation, specifically a 'relation over **I** and **F**.' That is a mathematical relation: any subset of a cross product.

Put more simply, a relation is defined in terms of its constituent domains – it is a portion (or subset) of their cross product. This cross product is a superset containing every possible tuple that could ever appear in the relation. (I know this seems circuitous.) The proper way of expressing a relation is to speak of a *relation R over the domains X, Y, and Z*. If we were to expand our relation to include the domain of all *Seinfeld* episodes, denoted by **E**, it would then be 'a relation *R* over **I**, **F**, and **E**' – represented by **I**~TMS**F**~TMS**E**, which is an even larger set than **I**~TMS**F**.

Relational Relations

There is one fundamental difference between tuples in mathematics and tuples in relational theory. In mathematics, the order of values in a tuple is significant; in relational theory it isn't. This is because members (attributes) of relational tuples have names, which are used to identify them. Mathematical ones do not. Therefore, mathematical tuples in relations over **I**~TMS**F** are always of the form (*i*,*f*) because this is the only way to identify them – by ordinal positions. This convention is not needed in relational tuples because the attributes have names to identify them, rather than ordinal positions. Order, then, in a relational tuple offers no real advantage. The same applies to relational relations, as they share the same attributes through their common heading.

In his original paper, Codd differentiated mathematical relations (which he called domain-ordered relations) from relational relations (called domain-unordered relations) by referring to the latter as 'relationships':

> *Accordingly, we propose that users deal, not with relations which are domain-ordered, but with relationships which are their domain-unordered counterparts. To accomplish this, domains must be uniquely identifiable at least within any given relation, without using their position.*
>
> Codd (1970)

Domains (or attributes) are uniquely identifiable through attribute names. Therefore, neither column nor row order matter in the relational model. This is the principal way in which the relations of set theory and the relations of relational theory differ. In all other respects, tuples and relations in relational theory share the properties of their counterparts in set theory.

And that is a relation. It is the fundamental object upon which relational theory is built. If you understand it, you are well on your way to understanding the core mechanics of relational theory. Relational theory is fitted so as to mirror the true mathematical relation as closely as possible: the closer the relational model fits the mathematical model, the more it benefits from other facets of mathematics already well established. A prime example is relational algebra, which is part of the manipulative component of the relational model. Many of the operations defined for relations in set theory carry over to relational algebra because relational relations are sufficiently similar. The same is true for relational calculus, which employs methods of formal logic. As Codd put it:

> *Moreover, the (relational) approach has a close tie to first-order predicate logic – a logic on which most of mathematics is based, hence a logic which can be expected to have strength, endurance, and many applications.*
>
> Codd (1970)

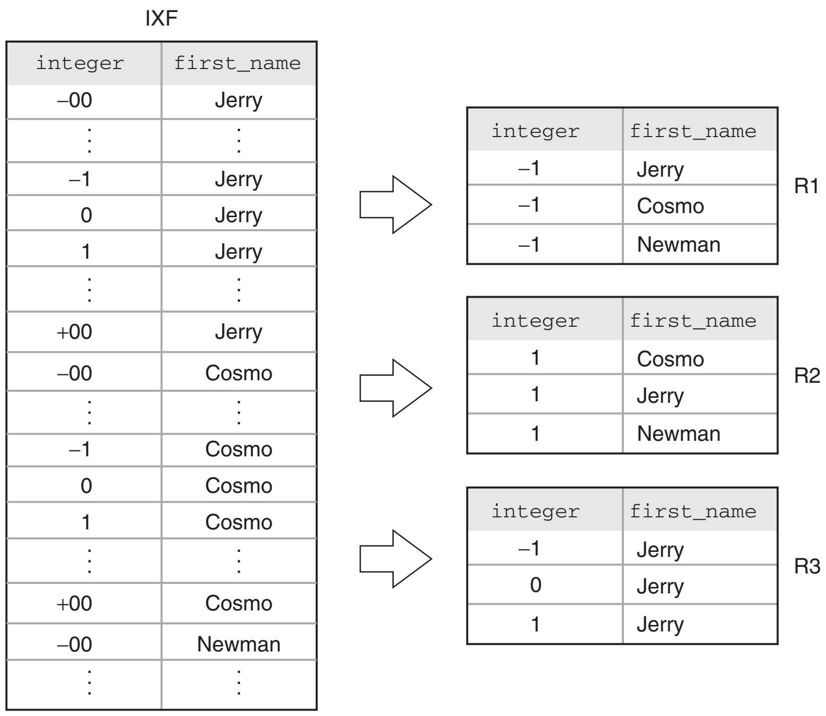
Codd recognized not only that the elegance of mathematics is beneficial as a basis to build upon, but that it also offers a vast reservoir of existing knowledge that can be harnessed.

3.2.6 Tables: Relation Variables

A relation, though it contains values, is itself just a value, like an integer or a string. The value of a relation is given by the particular set of tuples it contains. Likewise, the value of each tuple in turn is given by the specific values within it. Thus, the value of a relation is determined by the sum of its parts. Figure 3.5 depicts *R1*, *R2*, and *R3*, taken over **I**~TMS**F**. Each represents a different value or relation.

R1, *R2*, and *R3* are not relations but rather *relation variables*. They *represent* relations. Codd referred to them as *named relations*. Date (2003) calls them *relvars*. SQL calls them *tables*. They are also known as *base tables*. The bottom line is that they are variables whose values are relations. I will simply refer to them here as tables, as that is what you will be dealing with in SQL.

In practice, the precise meaning of ‘relation’ can sometimes be a bit murky. Relations are often referred to in the same context as tables. However, they are not the same thing. One is a value; the other is a variable. A relation is a *value*, just as an integer value is 1, 2, or 3. A table is a variable to which relations are assigned. Tables, like variables, have both a name and a value. The name is a symbol. The value is a relation. They are no different to variables in algebra, such as *x* and *y* in the equation of a line. Tables share all the properties of relations (heading, degree, cardinality, etc.) just as integer variables share the properties of integers.



IXF

integer	first_name
–00	Jerry
⋮	⋮
–1	Jerry
0	Jerry
1	Jerry
⋮	⋮
+00	Jerry
–00	Cosmo
⋮	⋮
–1	Cosmo
0	Cosmo
1	Cosmo
⋮	⋮
+00	Cosmo
–00	Newman
⋮	⋮

R1

integer	first_name
–1	Jerry
–1	Cosmo
–1	Newman

R2

integer	first_name
1	Cosmo
1	Jerry
1	Newman

R3

integer	first_name
–1	Jerry
0	Jerry
1	Jerry

Figure 3.5 Relations over **I**~TMS**F**

With this in mind, what does it mean to update or modify a table? If you are familiar with SQL, you probably think of it in terms of modifying a single value or a few rows. You see it as the same as changing cells in a spreadsheet. You modify *parts* of the spreadsheet. You see those parts as individual values. The relational view is different. In the relational view, you don't modify parts of a table. A table is a variable holding an entire relation as its value. If you change anything, you change *the entire relation* – no matter how small the difference might be. You aren't changing a row or column; you are actually swapping one relation with an entirely new one, where the new relation contains the rows or columns you want changed.

I know that may seem like frivolous semantics. But, just as you don't modify part of an integer variable, you don't modify part of a relation variable. There is nothing wrong with thinking in terms of the SQL view – where you are changing values in a row. It is logically equivalent: in an SQL update, you articulate the change you want to make in a statement and that statement produces a new relation. But that is where you should make the distinction: the change produces a *new* relation, not a patched-up version of the old one. That new relation becomes a new value that the table holds.

Be aware that when you talk about tables in a database, you are referring to variables, specifically relation variables – not relations. If a database were composed strictly of relations, then its contents would be fixed values, not subject to change. But databases are dynamic. And the reason is because their contents – tables – are relation variables, which are subject to change. They change by *assigning* those relation variables new relation values, not by adding, subtracting, or changing tuples.

3.2.7 Views: Virtual Tables

Views are virtual tables. They look like tables and act like tables, but they're not. Views are relational expressions that yield relations. It is like saying that the algebraic expression $x+y$ is not a number per se, though it can yield a number if the expression is evaluated, provided we have specific values for x and y. The same is true for views. Codd (1980) described them as follows:

> *A view is a virtual relation (table) defined by means of an expression or sequence of commands. Although not directly supported by actual data, a view appears to a user as if were an additional base table kept up-to-date and in the state of integrity with other base tables. Views are useful for permitting application programs and users at terminals to interact with constant view structures, even when the base tables themselves are undergoing structural changes at the logical level.*

Logical Data Independence

In one sense, views are simply a matter of convenience. They let you assign a name to a set of operations and treat the result like a relation. They are a kind of shorthand. This is perhaps the most common use of views. Another use was for security. In some systems, views can be used to selectively present parts of tables, while excluding other sensitive or restricted parts. But the original intent of views was much more than these two applications. The main application for views was facilitating what is called *logical data independence*, which is defined as follows:

> ***Rule 9. Logical Data Independence.*** *Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.*
>
> Codd (1985a)

Logical data independence means that applications and users should theoretically be able to see the same logical structure (e.g., the same attributes in a relation), even if it is changed (e.g., an attribute in that relation is moved to another relation). An example is when a relation is decomposed into two relations for the sake of normalization, as described in Section 3.4.

More precisely, logical data independence means that users and applications should be insulated from changes at the logical level of the database. That doesn't mean the database is supposed to automatically shield users and programs from the database administrator (DBA) doing something really bone-headed, such as dropping a table and going home for the day. Rather, it means that the relational model provides a way for the DBA to bridge the gap if she wants to make substantive changes at the logical level that would impact the logical view seen by users and applications. For instance, if the DBA decomposes a table into two smaller ones, she can create a view that looks like the original table, though it is in reality a relational expression that combines (joins) the two new tables. The users and applications know no different.

Updatable Views

But logical data independence entails more than just viewing data. Many people who are familiar with views understand them as read-only. However, the relational model clearly states that you should be able to write to views as well, just as if they were ordinary tables. While logical data independence would seem to imply this, Rule 6 makes it explicit:

> ***Rule 6. View Updating Rule.*** *All views that are theoretically updatable are also updatable by the system.*
>
> Codd (1985a)

Rule 6 is essential for logical data independence. 'Theoretically updatable' means the view is constructed in such a way that it is theoretically possible to map changes made on it to its respective base tables. That is, if it can be done in theory, the system should be able to do it.

That said, Rule 6 and updatable views (sometimes referred to as 'materialized views') are not fully supported in all relational databases, simply because they are not easy to implement. Programming a system to know what is 'theoretically updatable' in relational algebra or calculus is not exactly a weekend project. On the other hand, read-only views are extremely common. In fact, it is almost hard to find a relational database today that doesn't support them.

3.2.8 The System Catalog

As a database is composed of tables and views, you might at some point wonder how you can find out exactly what is in a given database. As it turns out, a relational database contains tables and views describing its tables and views – information about the information. That is, all tables, views, constraints, and other database objects in a database are registered in what is referred to as the *system catalog*:

> ***Rule 4. Dynamic On-Line Catalog Based on the Relational Model.*** *The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.*
>
> Codd (1985a)

The system catalog is subject to the information principle: it is required to be represented in a relational format that can be queried in the same way as other relations in the system. This means that even metadata (information about information) in a database has to be represented relationally. In many database products, many of the tables in the system catalog are implemented as views. The beauty of views and the system catalog is that together they enable you to extend or enhance the catalog itself. You can create your own catalog views that contain information you find useful or informative. Those views in turn may even use catalog tables as a basis.

3.3 The Integrity Component

While the structural aspect of the model relates to the structure of information, the integrity aspect relates to the information within the structure. Information can be arranged in relations in a way that gives rise to various relationships, both within columns of a single relation and between columns of different relations. These relationships provide additional structure and indeed add even more information to what is already present.

The integrity aspect of the relational model provides a way to explicitly define and protect such relationships. Although their use in a database is entirely optional, the degree to which they are employed can ultimately determine the consistency of the data.

3.3.1 Primary Keys

Codd's second rule is the starting point of data integrity; it deals with the nature of data within relations. This rule is called the *Guaranteed Access Rule* and is defined as follows:

> ***Rule 2. Guaranteed Access Rule.*** *Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.*
>
> Codd (1985a)

The relational model requires that every relation have a primary key. The primary key is the set of attributes in a relation that uniquely identifies each tuple within it. This rule carries with it an important corollary: relations may not contain duplicate tuples.

The guaranteed access rule states that every field (or value in a tuple) in a relational database must be addressable. To be addressable, it must be identifiable. To be identifiable, each tuple must be distinguishable in some way from all other tuples in the relation, which is to say unique (thus, duplicate tuples would violate this constraint).

Uniqueness is the business of the primary key. A key is a designated attribute (or group of attributes) in a relation such that:

1. The value (or combined values) of that attribute (or attributes) is unique for every tuple in the relation.
2. If the key is composed of more than one attribute, all of the attributes that define the key must be necessary to ensure uniqueness. That is, every attribute in the key is sufficient to ensure uniqueness, but also necessary as well – if one were absent, then the uniqueness condition would not hold.

If both conditions are met, then the resulting attribute or group of attributes is a key (also called a candidate key). If condition 1 is met but not condition 2, then the attribute (or group of attributes) is called a *superkey*. It is a key that could stand to lose some weight. That is, it has more attributes than necessary to ensure uniqueness: a smaller key containing fewer attributes could be defined that still guarantees uniqueness.

A relation may have one or more candidate keys. If it does, then which one of them is defined as the primary key is arbitrary:

> *Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called the primary key of the relation.*
>
> Codd (1970)

The primary key is just a rule that requires that every relation have at least one candidate key. The definition of a primary key serves more as an affirmation of the guaranteed access rule than it does in defining a new relational concept.

3.3.2 Foreign Keys

With keys comes more than simply identification. While keys define relationships between tuples within a single relation (you might say vertically), they can also define relationships between tuples in different relations (horizontally). A key's identification property allows a tuple in one relation to identify (or reference) a specific tuple in another relation by way of a common key value. This is called a *foreign key relationship*. Specifically, a key in one table corresponds to, or references, the primary key in another table (the foreign key), thereby relating the tuples in each table. Take, for example, the `foods` and `food_types` tables shown in Figure 3.6. Each row in `foods` corresponds to a distinct item of food (e.g. Junior Mints, Macinaw Peaches, etc.). Each row in `food_types` stores a food classification (e.g., Junk Food, Fruit, etc.). Each row in `foods` references a row in `food_types` by using a common key. `foods` contains a key called `type_id`, every value of which corresponds to a value in the primary key (`id`) of `food_types`, as illustrated by the arrow in Figure 3.6.

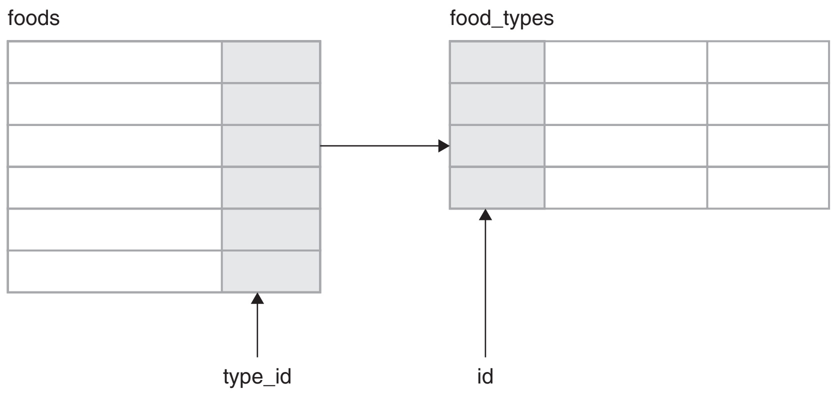


Figure 3.6 Foreign key relationship between `foods` and `food_types`

From a relational standpoint, this is merely a foreign key relationship. But in reality it is more than that. This relationship models a relationship in the real world. It has a basis in reality, has real meaning, and adds information above and beyond the information contained in the individual tables. Therefore, it is a critical part of the information itself.

It is easy to see that such relationships, if not properly maintained, can be precarious. If an application comes to assume this relationship and its normal operation depends on the fact that a tuple in `foods` always has a corresponding tuple in `food_types`, what happens if someone deletes all the tuples from `food_types`? Where do the `type_id` values in the `foods` tuples point? What has become of this relationship?

For that matter, what is to stop a user from just ignoring the primary key rule and jamming a thousand identical `food_types` tuples back into the database? If there is nothing in place to protect and ensure these relationships, they can be as destructive as they are beneficial. Such rules, like laws, are worthless without enforcement.

3.3.3 Constraints

Enter the *constraint*. Constraints are relational cops. They enforce database rules and relationships and preserve order. They bring consistency and uniformity to information within a database. With constraints, you can rest assured that tuples in the `foods` table reference legitimate tuples in `food_types`. Primary keys always exist in tables and their values are always unique. This kind of uniformity and consistency is called *data integrity*. It is the integrity component of the relational model.

Constraints work by governing database operations. Like the 'precogs' in the film *Minority Report*, they stop bad things before they happen. If a user or application issues a request that would result in an inconsistent relationship or data, the database refuses to carry out the operation and issues an error, called a *constraint violation*. In the relational model, constraints fall into four general classes of integrity:

- **Domain integrity** is the relationship between attribute values and their associated domains. In the relational model, domain integrity is instituted through *domain constraints*. A domain constraint requires that each attribute value in a tuple exists within its associated domain. For example, if the `type_id` attribute in the `foods` table is declared as an integer, then the corresponding values of `type_id` in all tuples in the `foods` table must be integer values – they cannot be floating-point numbers or strings. Domain integrity is also referred to as *attribute integrity* – it pertains to the attributes of a relation. Domain integrity is not limited to just checking that a given value resides in a given domain. It can include additional constraints, such as `CHECK` constraints (covered in Chapter 4) that can define complex rules on what constitutes a permissible value for a given attribute.
- **Entity integrity** is mandated by the guaranteed access rule: each tuple in a relation must be uniquely identifiable. Whereas domain integrity is concerned with a relation's attribute values, entity integrity is concerned with its tuples. The term 'entity' here is a rather loose term for table. It originates from database modeling (i.e., entity–relationship diagrams). In this particular context, an entity simply refers to anything in the real world that is represented in a database.
- **Referential integrity** pertains to relationships between tables, specifically the preservation of foreign key relationships. Whereas entity integrity pertains to tuples in a relation, referential integrity pertains to tuples between relations.
- **User-defined integrity** encompasses integrity not defined by the other forms. Many relational databases offer facilities that go beyond the normal constraint mechanisms. One such example is triggers, which are covered in Chapter 4.

The relational model requires that databases and query languages support integrity constraints. Furthermore, these constraints, like all other data and metadata in the database, must also be defined directly in the database, specifically in the system catalog:

> ***Rule 10. Integrity Independence.*** *Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs.*
>
> Codd (1985a)

3.3.4 Null Values

Closely associated with data integrity in the relational model is a special value (or lack thereof), which exists both inside and outside every domain. This special value denotes the *absence* of a value and is called a *null value*, or *null* for short. Nulls are prescribed in Codd's third rule:

> ***Rule 3. Systematic Treatment of Null Values.*** *Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.*
>
> Codd (1985a)

There are multiple interpretations for what null values mean. The prevailing view of null seems to be 'unknown' but there are others. For example, a tuple containing employee information may have an attribute for the employee's middle name. But not everyone has a middle name. A tuple for a person without a middle name might use a null value for that particular attribute. In this case, the null value doesn't necessarily mean 'unknown' but rather 'not applicable.' Thus, a value may be null because it is missing (a value exists but was not input), uncertain (it is not known whether the value exists), or simply not applicable for the tuple in question.

The inclusion of nulls in the relational model has been a source of controversy for many years and there are people on both sides of the debate who feel very strongly for their positions. Codd, for example felt the need for nulls:

> *In general, controversy still surrounds the problem of missing and inapplicable information in data bases. It seems to me that those who complain loudly about the complexities of manipulating nulls are overlooking the fact that handling missing information and inapplicable information is inherently complicated.*
>
> Codd (1985a)

Date is opposed to them:

> *. . . we should make it very clear that in our opinion (and in that of many other writers too, we hasten to add), nulls and 3VL are a serious mistake and have no place in a clean formal system like the relational model.*
> Date (2003)

In the quote, '3VL' stands for 'three-valued logic,' which corresponds to how nulls are evaluated in logical expressions. This is addressed in Chapter 4.

3.4 Normalization

The implication of no duplicate tuples provided by the guaranteed access rule gives rise to an important concept in database design called *normalization*. Normalization concerns itself with the organization of attributes within relations so as to minimize duplication of data. Data duplication, as you will see, has more deleterious effects than just taking up unnecessary space. It increases the opportunity for database inconsistencies to arise. Normalization is about designing your database to be more resistant to the ill effects of thoughtless users and buggy programs.

While based on principles of the relational model, normalization is a subject (some even say an art) in itself. It can become quite complicated, introducing a considerable number of new concepts and terminology. A proper treatment of the subject is beyond the scope of this book. What follows is a very brief introduction.

3.4.1 Normal Forms

As stated, the chief aim of normalization is to eradicate duplication. Relations that have had duplication removed are said to be *normalized*. However, there are degrees of normalization. These degrees are called *normal forms*. The first degree is called *first normal form*, followed by *second normal form*, and so on. They are abbreviated 1NF, 2NF, 3NF, and so on. Each normal form defines specific conditions a relation must meet in order to be so classified. A relation that meets the conditions of first normal form is said to be 'in first normal form.' Normal forms build on each other so that higher normal forms require all the conditions of lower forms as prerequisites. For data to be in 2NF, it must also be in 1NF. Essentially, the higher the normal form of a relation, the less duplication it has, and the more resistant it is to inconsistencies. The most common and, perhaps, most widely used normal form is 3NF, although there are even more advanced normal forms such as Boyce–Codd normal form (BCNF), 4NF, 5NF, and higher. The first three normal forms are easy enough to describe.

3.4.2 First Normal Form

First normal form simply states that all attributes in a relation use domains that are made up of atomic values. The working definition of 'atomic' is the same as in other disciplines, meaning simply 'that which cannot be broken down further.' For example, integer values would seem to be atomic, as you can't break them down further. But you could argue that integer values could be decomposed into their prime factors. Atomicity, then, is determined by the method of decomposition, which can be a subjective matter. For all practical purposes, it is the database management system that decides what is atomic. And the domains provided by the system can safely be considered as such.

That said, first normal form basically means that a single attribute cannot hold more than a single value. For example, take the `episodes` table (Table 3.2). 1NF states that you couldn't store both the values 1992 and 1993 (two integer values) in a `year` attribute of a single tuple. It sounds so silly that it can be kind of hard to imagine. But it's that simple. It is so simple, in fact, that you have to work at violating 1NF. Most databases won't even give you the means to.

Table 3.2 The `episodes` table without normalization

season	week	year	name
4	1	1992	The Junior Mint
4	2	1992	The Smelly Car
5	1	1993	The Mango
5	2	1993	The Puffy Shirt
6	21	1994	The Fusilli Jerry
6	25	1994	The Understudy

3.4.3 Functional Dependencies

To understand second and third normal forms, you have to first understand *functional dependencies*. The simple definition of a functional dependency is a correlation between columns in a table. If the values of one column (or set of columns) correlate to the values of another, then they have a functional dependency. More precisely, a functional dependency describes a relationship between two or more attributes in a relation such that the value of one attribute (or set of attributes) can be inferred from the value of another attribute (or attributes) for every tuple in the relation. This is more easily illustrated by example.

Consider the relation, called `episodes`, shown in Table 3.2. There is a functional dependency between `season` and `year`. For any given value of `season`, you find the same value of `year`. If you ever come across a tuple with a value of 4 for `season`, you know the value for `year` is 1992. If you know the former, you can determine the latter.

Functional dependencies are often a warning sign that duplication lurks within a table. And already you can see in this example how inconsistencies can crop up. If there is in fact a correlation between `year` and `season`, what happens to that relationship if someone modifies the first row so that its value for `year` is 1999 (but fails to do so for the second row)? That relationship has been compromised. It is inconsistent. One tuple with `season`=4 has `year`=1992, and another with `season`=4 has `year`=1999. How can season four have happened in both 1992 and 1999? This is logically inconsistent and the functional dependency (or lack of sufficient normalization) is what made it possible to introduce this inconsistency. What is even more interesting is that there is no standard integrity constraint designed to guard against this problem. It is purely the result of bad design.

Functional dependencies always involve exactly two sets of attributes: the determinant set determines (or relates to) the value of the dependent set. Call the attributes making up the determinant *A* and the attributes making up the dependent *B*. With this in place, you can speak learnedly on the topic using the following (equivalent) statements:

- *B* is functionally dependent upon *A*.
- *A* functionally determines *B*.

There is a notation for this: *A* ~TRA *B*. The bottom line is that for any value of *A*, the value of *B* has some kind of correlation.

3.4.4 Second Normal Form

Second normal form is defined in terms of functional dependencies. It requires that a relation be in first normal form *and* that all non-key attributes be functionally dependent on *all* attributes of the primary key (not just part of them). Remember that normalization is about cutting out duplication, so while this sounds like an arbitrary rule, it is in fact aimed at weeding out duplication.

You have already seen what duplication looks like when 2NF is not followed, using `episodes` as an example. The primary key in `episodes` is composed of both `season` and `week`. You know that year is functionally dependent on season, which in turn is only part of the primary key (this is a strong warning). Specifically, for every season=*x* (e.g., 4) you have the same year=*y* (e.g., 1992). There is no reason to include the year attribute in the relation if its value is correlated with season. That's duplication. It must go.

So what do you do? You decompose the table into two tables. Cut out year and move it to its own table; call it the seasons table. Now episodes is decomposed into two tables with a foreign key relationship where episodes.season (Table 3.3) references seasons.season (Table 3.4).

Table 3.3 The episodes table in second normal form

season	week	name
4	1	The Junior Mint
4	2	The Smelly Car
5	1	The Mango
5	2	The Puffy Shirt
6	21	The Fusilli Jerry
6	25	The Understudy

Table 3.4 The seasons table created by normalizing episodes

season	year
4	1992
5	1993
6	1994

This is like factoring out a common variable in an algebraic expression. Now episodes is in 2NF. Furthermore, no information is lost because year is functionally determined by season. But look what else you get: the new design allows referential integrity to back you up: you have a foreign key constraint guarding this relationship (the correlation between season and year), which you couldn't get in the previous form. What was previously only implied is now both explicit and enforceable.

Now consider the specific case mentioned earlier where someone modifies the first row. The logical inconsistency is no longer possible: it is impossible to mess up the year, because it is not in episodes. If someone changes year=1992 to 1999 in seasons, it may be inaccurate, but it is still logically consistent. That is, both rows that refer to it in episodes (season=4) will still be in agreement about the year in which season 4 took place. Normalization cannot make you get your facts right but it can ensure that your data is consistent about them.

Second normal form works by tightening the specificity between the primary key and the non-key attributes. If the values of a non-key attribute correlate to only *part* of the primary key, then logically that attribute can introduce duplication. Why? Because only part of a primary key is not unique (*all* of the primary key is required for uniqueness, as mentioned earlier); correspondence between two columns, neither of which is unique, opens the possibility of duplication; and duplication invites inconsistency.

3.4.5 Third Normal Form

Third normal form shifts attention from functional dependencies on the primary key to dependencies on non-key attributes in a relation. It roots out a special class of functional dependencies called *transitive dependencies*. A transitive dependency is a chain of two or more functional dependencies spanning two or more attribute groups (two or more correlations). For example, say you have a relation with three sets of attributes, *A*, *B* and *C*, where *A* is the primary key, and *B* is a candidate key. If *A* ∼TRA *B* and *B* ∼TRA *C*, then *C* is transitively dependent on *A*: it depends on *A* indirectly through its relationship with *B*. Transitive dependencies harbor duplication. Third normal form dictates that a relation must be in second normal form and have no transitive dependencies.

To picture this, let's add an integer primary key called id to the original episodes table (see Table 3.5). The candidate key is now the combination of season and week.

Table 3.5 The unnormalized episodes table with an integer primary key

id	season	week	year	name
1	4	1	1992	The Junior Mint
2	4	2	1992	The Smelly Car
3	5	1	1993	The Mango
4	5	2	1993	The Puffy Shirt
5	6	21	1994	The Fusilli Jerry
6	6	25	1994	The Understudy

In this version of episodes, year is functionally dependent on the season, which in turn is functionally dependent on id. So we have `id` ~TRA `season` ~TRA `year`. How do you know this? Work backwards. Ask yourself if you can determine `season` from `id`. Yes. Then can you determine `year` from `season`? Yes. Therefore, you have a transitive dependency: `id` functionally determines `season` and `season` functionally determines `year`. To be in third normal form, `year`, which is functionally dependent on a non-key attribute, must go. As in the previous example, you relegate `year` into its own table by splitting `episodes` into two tables (Tables 3.6 and 3.7). Just as in the 2NF example, decomposition has made an implicit relationship explicit.

Table 3.6 The `episodes` table in third normal form

id	season	week	name
1	4	1	The Junior Mint
2	4	2	The Smelly Car
3	5	1	The Mango
4	5	2	The Puffy Shirt
5	6	21	The Fusilli Jerry
6	6	25	The Understudy

Table 3.7 The `seasons` table created by normalizing `episodes`

Season	Year
4	1992
5	1993
6	1994

Since functional dependencies between non-key attributes are not allowed in a table, you may wonder why it is okay for functional dependencies to exist between keys. The answer is simple: uniqueness. Even if there is a correlation between one key and another, the fact that they are unique guarantees that their correlation is also unique; therefore, no duplication exists.

In the end, 2NF and 3NF aim not to introduce confusing rules, but to seek out and destroy duplication and the inconsistencies that can arise from it. It is essential for proper database design. The more important your data, the more attention you should pay to ensure that your database is properly designed and normalized.

3.5 The Manipulative Component

The manipulative component of the relational model defines the ways in which information can be manipulated and changed. It is the dynamic part of the model that connects the data in the logical view to the outside world.

3.5.1 Relational Algebra and Calculus

In his original paper, Codd described a language that could be used to operate on data:

> *The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus . . . . Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented).*
>
> Codd (1970)

This 'universal data sublanguage' would have a sound mathematical basis. Codd defined this basis in the form of the relational algebra and relational calculus. As described in the abstract of (Codd 1972):

> *In the near future, we can expect a great variety of languages to be proposed for interrogating and updating data bases. This paper attempts to provide a theoretical basis which may be used to determine how complete a selection capability is provided in a proposed data sublanguage independently of any host language in which the sublanguage may be embedded.*
> *A relational algebra and relational calculus are defined. Then, an algorithm is presented for reducing an arbitrary relation-defining expression (based on the calculus) into a semantically equivalent expression of the relational algebra.*
> *Finally, some opinions are stated regarding the relative merits of calculus-oriented versus algebra-oriented sublanguages from the standpoint of optimal search and highly discriminating authorization schemes.*

These two 'pure' languages – algebra and calculus – focused on mathematical theory rather than a particular language syntax. The latter is the job of the 'data sublanguage,' or 'query language,' as we know it today. Like the structural part of the relational model, the manipulative part drew heavily from mathematics. Relational algebra has its basis in set theory. Likewise, relational calculus has its basis in predicate calculus. From a computer science perspective, relational algebra can be considered more of a *procedural language* while relational calculus is more of a *declarative language.* The meanings of these terms are explained in more detail in Chapter 4.

As Codd states, while the particular forms of expression in the two languages are different, they are nevertheless logically equivalent. That is, any operation in the algebra can also be expressed in terms of the calculus, and vice versa. Another way of saying this is that the two languages have the same *expressive power* – the same fundamental operations can be performed or expressed in either system.

3.5.2 The Relational Query Language

Together, relational algebra and calculus serve as a guideline, or a yardstick as Codd describes it, for query languages implemented in relational databases. Any query language that can express all of the fundamental operations set forth in relational algebra or calculus is said to be *relationally complete.*

The query language must also address the other aspects (structural and integrity) of the relational model. This is summed up in Codd's fifth rule, defined as follows:

> ***Rule 5. Comprehensive Data Sublanguage Rule.*** *A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items: Data Definition, View Definition, Data Manipulation (interactive and by program), Integrity Constraints, and Authorization, Transaction Boundaries (begin, commit, and rollback).*
>
> Codd (1985a)

Additionally, Codd's seventh rule requires that the database (and, by extension, the query language) not only use relations for storage, manipulation, and retrieval, but also as operands for the purpose of modifying the database:

> ***Rule 7. High Level Insert, Update, and Delete.*** *The system must support set at a time insert, update, and delete operators.*
>
> Codd (1985a)

A user then should be able to insert tuples by providing a relation composed of the tuples to be inserted, and similarly for updating and deleting information within the system. While this rule keeps things consistent – using relations – its primary intent at the time was actually in optimizing database performance. The idea was that, in some cases, the database could make better optimizations by seeing modifications together as a set than it could by processing them individually.

So, first, a relational database must provide at least one query language that addresses the structural, integrity, and manipulative aspects of the relational model. With respect to the manipulative aspect, it must be capable of expressing the mathematical concepts set forth in algebra or calculus. Furthermore, it must accept relations as a means of modifying data in the system. Note that Codd never mandated a particular query language. He only mandated that a relational database provide one and what it must do.

3.5.3 The Advent of SQL

Over the years, there have been multiple competing query languages. However, the most popular and widely adopted of these languages today is undoubtedly SQL. SQL is a relationally complete query language that exhibits aspects of both relational algebra and relational calculus. That is, it has both declarative features (calculus) and procedural features (algebra). In fact, as you will see in Chapter 4, SQL includes almost all of the operators defined in relational algebra.

SQL also reflects each aspect of the relational model. Part of its language is dedicated to working with the structural aspect of the model, specifically to creating, altering, and destroying relations. This part of the language is called 'data definition language' (DDL). Within DDL lies also the integrity aspect, allowing the creation of keys and various database constraints. Likewise, part of the language, called the 'data manipulation language' (DML), is dedicated to the operational aspect. It includes ideas from both algebra and calculus.

Ironically, despite the clear influence of the relational model on SQL, the current SQL standard does not mention the relational model or use relational terminology.[3] And while SQL is relationally complete, in many ways it falls short of the true power of the relational model. Although it was primarily inspired by relational calculus, some have claimed that there are ways in which SQL also violates it. Furthermore, SQL lacks relational operations that some people consider important, such as relational assignment, and has a number of redundant features. Part of the reason for this was that the organizations responsible for creating the SQL standard felt that it was more important to release a standard as early as possible in order to establish a base upon which database implementations could build. Thus, when the initial standard was released in 1987, though practical, it was found wanting in many ways by researchers involved with the relational model (Codd included). Among other things, it omitted some relational operations and included no mention of referential integrity constraints. Subsequent standards filled

[3] See ***en.wikipedia.org/wiki/Relational_model***.

in some of the gaps here and there, but this seems to have done little to appease its detractors or deter database vendors from 'extending' their SQL dialects to include proprietary features.

Entire books and websites are devoted to SQL's inadequacies and to what an ideal query language should be. Furthermore, alternative query languages have been proposed and implemented (both before and after SQL) that, in the minds of their creators, are more expressive and better reflect the principles and intent of the relational model.[4]

Despite its criticisms and shortcomings, however, SQL is what we have to work with. It is undoubtedly the most popular and widely adopted query language in the industry and, given its longstanding dominance in the marketplace, it is unlikely that its position will change any time in the near future.

3.6 The Meaning of *Relational*

Given all this information, what exactly is 'relational'? It is a common misconception that relational databases derive the name 'relational' from their ability to 'relate' a column in one table to a column in another through a foreign key relationship. The true meaning of relational, however, stems from the central structural component of the relational model: the relation, which itself is based on the mathematical concept. First and foremost, a relational database is one that uses relations as the sole structural unit in which to represent information, as mandated by the information principle.

A more specific definition of 'relational', however, is provided by Codd's Rule Zero:

> ***Rule Zero.*** *For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.*
>
> Codd (1985a)

For a database to be called relational, it must provide all of the facilities required by the relational model. That is, it must conform to all of Codd's rules. And, believe it or not, this chapter has covered every one of them. You therefore should have a good idea at this point what it means to be relational. Don't get too cocky, though, as Codd later expanded the 12 rules to over 300.

[4] Tutorial D is one such example. See ***www.thethirdmanifesto.com*** for more information.

3.7 Summary

This was a introductory tour through the relational model. While the discussion isn't definitive, my goal was to give you enough history to make the topic enjoyable and enough theory for you to understand some of the thinking behind SQL – why it works the way it does. The relational model has clearly influenced its design and has provided it with a solid theoretical foundation.

The relational model has proven itself over its 30-year history. It has had an enormous impact not only on computing but on the way we do business. Today, it can be found in a wide array of electronic machinery, ranging from mainframes to cell phones.

The relational model was created to provide a logical, consistent representation of data that is independent of hardware and software. The model was built on well-founded theory set forth in mathematics. This model provides database users with a consistent, unchanging view of information, powerful methods to operate on it, and mechanisms to protect and ensure its consistency and integrity.

There are many more aspects to the relational model and much that builds on it. Although this is a large subject, its core concepts are logical and straightforward. These concepts ground, frame, and form the basis of the subject covered in the next chapter: SQL.

If you are new to SQL and you've patiently endured this chapter, you should have a much easier time grasping the concepts in the next chapter. You will see SQL, not as an arbitrary language, but as a gateway into a powerful database management system. You will find that its syntax is heavily geared to what you've learned in this chapter.

4

Everything You Ever Wanted to Know about SQL but Were Afraid to Ask

This chapter is a complete introduction – to SQL, in general, and SQLite's implementation of it, in particular. It assumes no previous experience with either SQL or the relational model. If you are new to SQL, SQLite should serve as an excellent springboard to the world of relational databases, since it will give you a good grounding in the fundamentals.

While the previous chapter on the relational model was a little theoretical and stuffy, this chapter is much more relaxed and practical. We're not here to prove theories, but to get things done. So if you didn't read the previous chapter, that's perfectly fine. You should still have no trouble with the material covered in this chapter. An understanding of the relational model is edifying, but is not necessary to learn SQL.

SQL is the sole (and almost universal) means by which to communicate with a relational database. It is a language exclusively devoted to information processing. It is designed for structuring, reading, writing, sorting, filtering, protecting, calculating, generating, grouping, aggregating, and, in general, managing information.

SQL is an intuitive, user-friendly language. It can be fun to use and is quite powerful. One of the fun things about SQL is that, regardless of whether you are an expert or a novice, you can continue to learn new ways of doing things (for better or worse). There are often many ways to tackle a given problem, and you may find yourself taking delight in trying to find more efficient ways to get what you need, either through more compact expressions or more elegant approaches, as if solving a puzzle. And you can continue to explore dusty corners of the language you never took notice of before with which to further hone your skills, no matter your experience.

The goal of this chapter is to teach you to use SQL *well* – to expose you to good techniques and, perhaps, to show off a few tricks along the way.

As you can already tell, there are many different aspects to SQL. This chapter breaks them down into discrete parts and presents them in a logical order that should be relatively easy to follow. Nevertheless, SQL is a big subject and there is a lot of ground to cover. It will take some time and more than one cup of coffee to get through this chapter. But by the time you are done, you should be well equipped to put a dent in a database.

4.1 The Relational Model

As you'll remember if you worked through Chapter 3, SQL is a consequence of the relational model, which was originally proposed by E. F. Codd in 1969. The relational model requires that relational databases provide a query language, and over the years SQL has risen to become the lingua franca.

The relational model consists of three essential parts: form, function, and consistency. Form refers to the structure of information. There is but a single data structure used to represent all information. This structure is called a *relation* (known in SQL as a *table*), which is made up of *tuples* (known in SQL as *rows*), which in turn are made up of *attributes* (known in SQL as *columns*).

The relational model's form is a *logical representation* of information. This logical representation is a pristine, abstract view of information unaffected by anything outside it. It is like a mathematical concept: clean and consistent, governed by a well-defined set of deterministic rules that are not subject to change. The logical representation is completely independent of the *physical representation*, which refers to how database software stores this information on the physical level (e.g., disk). The two representations are thus distinct: nothing that occurs in the physical level can change or affect anything at the logical level. The logical level is uninhibited by hardware, software, vendor or technology.

The second essential part of the model – the functional part – is also called the manipulative component. It defines ways of operating on information at the logical level. This was formally introduced in Codd (1972). It added the functional part to the relational model by defining *relational algebra* and *relational calculus*. These are two formal, or 'pure,' query languages with a heavy basis in mathematics. Relations, as described in the data model, are mathematical sets (with a few additional properties). Relational algebra and calculus, in turn, build on this model by adding operations from set theory, thus forming the functional component of the relational model. So both the form and function prescribed in the relational model come directly from concepts in mathematics. Each derivative, however, adds a little something to better adapt it to computers and information processing.

4.1.1 Query Languages

A query language connects the outside world with the abstract logical representation and allows the two to interact. It provides a way to retrieve and modify information. It is the dynamic part of the relational model.

Codd intended relational algebra and calculus to serve as a baseline for other query languages. Relational algebra and calculus employ a highly mathematical notation that, while good for defining theory, is not terribly user-friendly in practice. Their purpose was thus to define the mathematical or relational requirements that a more user-friendly query language should support. Query languages that met these minimum requirements were called *relationally complete*. This user-friendly query language then provides a more tractable and intuitive way for a person to work with relational data, while at the same time adhering to a sound theoretical basis.

4.1.2 Growth of SQL

Perhaps the first such query language was IBM's *System R*, a relational database research project that was a direct outgrowth of Codd's work. It was originally called SEQUEL, which stands for 'Structured English Query Language.' It was later shortened to SQL, or 'Structured Query Language.'

Other companies, such as Oracle, followed suit (in fact, Oracle beat IBM to market with the first SQL product), and pretty soon SQL was the de facto standard. There were other query languages, such as Ingres's QUEL, but in time SQL won out. The reasons SQL emerged as the standard may have had more to do with the dynamics of the marketplace than anything else. But the reasons why it is special today are perhaps a little clearer: standardization, wide adoption, and general ease of use (among others).

SQL has been accepted as the standard language for relational databases by the American National Standards Institute (ANSI), the International Standards Organization (ISO), and the International Electrotechnical Commission (IEC). It has a well-defined standard that specifies what SQL is and does. To date, five versions of the standard have been published (1986, 1989, 1992, 1999, and 2003). The standards are cumulative. Each version of the standard has built on the previous one, adding new features. So in effect, there is only one standard that has continued to grow and evolve over time. The first versions – SQL86 and SQL89 – are collectively referred to as SQL1; SQL92 is commonly referred to as SQL2, and SQL99 as SQL3. The ANSI standard, as a whole, is very large. The SQL92 standard alone is over 600 pages. No one database product conforms to the entire standard. Nevertheless, the standard goes a long way in bringing a great deal of uniformity to relational databases.

SQL's wide adoption today is obvious: Oracle, Microsoft SQL Server, DB2, Informix, Sybase, PostgreSQL, MySQL, Firebird, Teradata, Intersystems Caché, and SQLite is but an incomplete list of relational databases that use SQL as their query language.

4.2 The Example Database

Before diving into syntax, let's get situated with the obligatory example database. If you've ever watched *Seinfeld* ***(en.wikipedia.org/wiki/Seinfeld)***, you can't help but notice a slight preoccupation with food. There are more than 412 different foods mentioned in the 180 episodes of its history (according to data I found on the Internet). That's over two new foods every show and virtually a new food introduced every 10 minutes.

As it turns out, this preoccupation with food works out nicely as it makes for a database that illustrates all the requisite concepts. The database used in this chapter consists of the foods mentioned in episodes of *Seinfeld*. The database tables are shown in Figure 4.1.

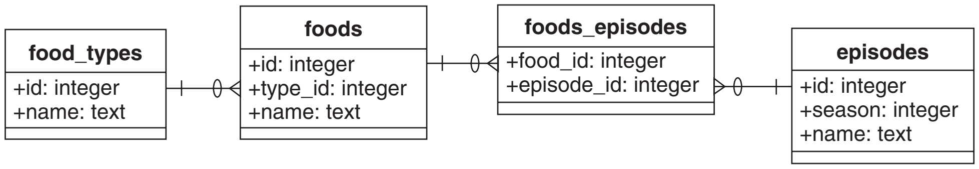


Figure 4.1 The *Seinfeld* food database

The database schema is defined in SQLite as follows:

```
create table episodes (
  id integer primary key,
  season int,
  name text );

create table foods(
  id integer primary key,
  type_id integer,
  name text );

create table food_types(
  id integer primary key,
  name text );

create table foods_episodes(
  food_id integer,
  episode_id integer );
```

The main table is foods. Each row in foods corresponds to a distinct food item, the name of which is stored in the name attribute. The type_id attribute references the food_types table, which stores the various food classifications (e.g., baked goods, drinks, or junk food). Finally, the foods_episodes table links foods in the foods table with episodes in the episodes table.

4.2.1 Installation

The food database is located in the examples zip file accompanying this book. It is available at ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***.

To create the database, locate the foods.sql file in the root directory of the unpacked zip file. To create a new database from scratch from the command line, navigate to the examples directory and run the following command:

```
sqlite3 foods.db < foods.sql
```

This creates a database file called foods.db.

4.2.2 Running the Examples

A convenient way to run the longer queries in this chapter is to copy them into your favorite editor and save them in a separate file, which you can run from the command line. For example, copy a long query into test.sql to try it out. You simply use the same method to run it as you did to create your database earlier:

```
sqlite3 foods.db < test.sql
```

The results are printed to the screen. This also makes it easier to experiment with these queries without having to retype them or edit them from inside the SQLite shell. You make your changes in the editor, save the file, and rerun the command line.

For maximum readability of the output, you may want to put the following commands at the beginning of the file:

```
.echo on
.mode col
.headers on
.nullvalue NULL
```

These commands cause the command-line program to:

- echo the SQL as it is executed
- print results in column mode
- include the column headers
- print null values as `NULL`, rather than blank.

The output of all examples in this chapter is formatted with these settings. Another option you may want to set for various examples is the `.width` option, which sets the respective column widths of the output. These vary from example to example.

For better readability, the examples are presented in two formats. For short queries, I show the SQL and output as it would be shown from within the SQLite shell. For example:

```
sqlite> SELECT * FROM foods WHERE name='JujyFruit' AND type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

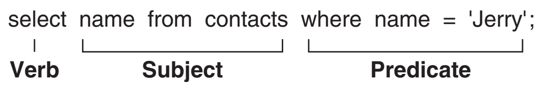
As in the previous example, I take the liberty of adding an extra line between the command and its output (in the shell, the output immediately follows the command). For longer queries, I show the SQL followed by the results in bold, as in the following example:

```
SELECT f.name name, types.name type FROM foods f
INNER JOIN (SELECT * FROM food_types WHERE id=6) types
ON f.type_id=types.id;

name                      type
------------------------- -----
Generic (as a meal)       Dip
Good Dip                  Dip
Guacamole Dip             Dip
Hummus                    Dip
```

4.3 Syntax

SQL's declarative syntax reads a lot like a natural language. Statements are expressed in the imperative mood, beginning with the verb describing the action. The verb is followed by the subject and the predicate, as illustrated in Figure 4.2.



```
select name from contacts where name = 'Jerry';
```

Figure 4.2 General structure of SQL syntax

As you can see, it reads like a normal sentence. SQL was designed specifically with nontechnical people in mind and was meant to be very simple and easy to understand.

Part of SQL's ease of use comes from its being (for the most part) a *declarative* language, as opposed to an imperative language such as C or Perl. A declarative language is one in which you describe *what* you want whereas an imperative language is one in which you specify *how* to get it. For example, consider the process of ordering a cheeseburger. As a customer, you use a declarative language to articulate your order. That is, you simply declare to the person behind the counter what you want: 'Give me a double meat Whataburger with jalapeños and cheese, but no mayo.'

The order is passed back to a chef who fulfills the order using a program written in an imperative language – the recipe. He follows a series of well-defined steps that must be executed in a specific order to create the cheeseburger to your (declarative) specifications:

1. Get ground beef from the third refrigerator on the left.
2. Make a patty.
3. Cook for three minutes.
4. Flip.
5. Cook three more minutes.
6. Repeat steps 1–5 for second patty.
7. Add mustard to top bun.
8. Add patties to bottom bun.
9. Add cheese, lettuce, tomatoes, onions, and jalapeños to burger, but not mayo.
10. Combine top and bottom buns, and wrap in yellow paper.

As you can see, declarative languages tend to be more succinct than imperative ones. In this example, it took the declarative burger language (DBL) one step to materialize the cheeseburger, while it took the imperative chef language (ICL) 10 steps. Declarative languages do more with less. In fact, SQL's ease of use is not far from this example.

A suitable SQL equivalent to the DBL statement above might be something along the lines of

```
SELECT burger FROM kitchen WHERE patties=2 AND toppings='jalopenos'
AND condiment != 'mayo' LIMIT 1;
```

Pretty simple. As we've mentioned, SQL was designed to be a user-friendly language. In the early days, SQL was targeted specifically for end users for tasks such as ad hoc queries and report generation (unlike today where it is almost exclusively the domain of developers and database administrators).

4.3.1 Commands

SQL is made up of *commands*. Commands are typically terminated by a semicolon. For example, the following are three distinct commands:

```
SELECT id, name FROM foods;
INSERT INTO foods VALUES (NULL, 'Whataburger');
DELETE FROM foods WHERE id=413;
```

The semicolon, or command terminator, is associated primarily with interactive programs designed for letting users execute queries against a database. While the command terminator is commonly a semicolon, it varies by both database system and query program. Some systems, for example, use `\g` or even the word `go`; SQLite, however, uses the semicolon as a command terminator in both the command-line program and the C API.

Commands, in turn, are composed of a series of *tokens*. Tokens can be literals, keywords, identifiers, expressions, or special characters. Tokens are separated by white space, such as spaces, tabs and new-line characters.

4.3.2 Literals

Literals, also called *constants*, denote explicit values. There are three kinds: string constants, numeric constants, and binary constants. A string constant is one or more alphanumeric characters surrounded by single quotes. Examples include:

```
'Jerry'
'Newman'
'JujyFruit'
```

String values are delimited by single or double quotes. If the string value contains an apostrophe, it must be represented as two successive single quotes. For example, ‘Kenny’s chicken’ would be expressed as:

```
'Kenny''s chicken'
```

Numeric constants are represented in integer, decimal, or scientific notation. Examples include

```
-1
3.142
6.0221415E23
```

Binary values are represented using the notation `x'0000'`, where each digit is a hexadecimal value. Binary values must be expressed in multiples of two hexadecimal values (8 bits). Here are some examples:

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

4.3.3 Keywords and Identifiers

Keywords are words that have a specific meaning in SQL. These include `SELECT`, `UPDATE`, `INSERT`, `CREATE`, `DROP`, and `BEGIN`. *Identifiers* refer to specific objects within the database, such as tables or indexes. Keywords are reserved words and may not be used as identifiers. SQL is case insensitive with respect to keywords and identifiers. The following are equivalent statements:

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

Throughout this chapter, all SQL keywords are represented in upper case and all identifiers in lower case, for clarity. By default, SQLite is case sensitive with respect to string values, so the value `'Mike'` is not the same as the value `'mike'`.

4.3.4 Comments

Comments in SQL are denoted by two consecutive hyphens (`--`), which comment the remainder of the line, or by the multiline C-style notation (`/* */`), which can span multiple lines. For example:

```
-- This is a comment on one line
/* This is a comment spanning
   two lines */
```

4.4 Creating a Database

Before you can do anything, you have to understand tables. If you don't have a table, you have nothing to work on. The table is the standard unit of information in a relational database. Everything revolves around tables. Tables are composed of rows and columns. And while that sounds simple, the sad truth is that tables are not simple. Tables bring along with them all kinds of other concepts, concepts that can't be nicely summarized in a few tidy paragraphs. In fact, it takes almost the whole chapter. So what we are going to do here is the two-minute overview of tables – just enough for you to create a simple table and get rid of it if you want to. And once we have that out of the way, all of the other parts of this chapter have something to build on.

4.4.1 Creating a Table

SQL is made up of several parts. The structural part is designed to create and destroy database objects. This part of the language is generally referred to as the *data definition language* (DDL). The functional part is designed to perform operations on those objects (e.g., retrieving and manipulating data). This part of the language is referred to as the *data manipulation language* (DML). Creating tables falls into DDL, the structural part.

You create a table with the `CREATE TABLE` command, which is defined as follows:

```
CREATE [TEMP] TABLE table_name (column_definitions [, constraints]);
```

The `TEMP` or `TEMPORARY` keyword creates a *temporary table*. This kind of table is, well, temporary – it only lasts as long the database session. As soon as you disconnect, it is destroyed (if you haven't already destroyed it manually). The brackets around `TEMP` denote that it is an optional part of the command. Whenever you see syntax in brackets, it means that the content is optional. Furthermore, the pipe symbol (|) denotes an alternative (think of the word *or*). Take, for example, the following syntax:

```
CREATE [TEMP|TEMPORARY] TABLE . . . ;
```

This means that either the `TEMP` *or* the `TEMPORARY` keyword may be optionally used. You can say `CREATE TEMP table foo` or `CREATE TEMPORARY TABLE foo`. In this case, they mean the same thing.

If you don't create a temporary table, then `CREATE TABLE` creates a *base table*, a named, persistent table in the database. This is the most common kind of table. There are other kinds of table, such as *system tables* and *views*, that can exist in the database but they aren't important right now. In general, the term 'base table' is used to differentiate tables created by `CREATE TABLE` from the other kinds.

The minimum required information for `CREATE TABLE` is a table name and a column name. The name of the table, given by `table_name`, must be unique. In the body, `column_definitions` consists of a comma-separated list of column definitions composed of a name, a *domain*, and a comma-separated list of *column constraints*. The domain, sometimes referred to as a *type*, is synonymous with a data type in a programming language. It denotes the type of information that is stored in the column. There are five native types in SQLite: `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL`. All of these domains are covered in Section 4.7.3. *Constraints* are constructs that control the kinds of value that can be placed in the table or in individual columns. For instance, you can ensure that only unique values are placed in a column by using a `UNIQUE` constraint. Constraints are covered in Section 4.7. Consider the following example:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone) );
```

Column `id` is declared to have type `INTEGER` and constraint `PRIMARY KEY`. The combination of this type and constraint has a special meaning in SQLite. `INTEGER PRIMARY KEY` basically turns the column into a column that increments automatically (see Section 4.7.1). Column `name` is declared to be of type `TEXT` and has two constraints: `NOT NULL` and `COLLATE NOCASE`. Column `phone` is of type `TEXT` and also has two constraints. After that, there is a table-level constraint of `UNIQUE`, which is defined for columns `name` and `phone` together.

This is a lot of information to absorb all at once, but it is all explained in due course. I warned you that tables bring a lot of baggage with them. The important thing here, however, is that you understand the general format of the `CREATE TABLE` statement.

Tables can also be created from `SELECT` statements, allowing you to create not only the structure but also the data at the same time. This particular use of the `CREATE TABLE` statement is covered in Section 4.6.1.

4.4.2 Altering a Table

You can change parts of a table with the `ALTER TABLE` command. SQLite's version of `ALTER TABLE` can either rename a table or add columns. The general form of the command is

```
ALTER TABLE table { RENAME TO name | ADD COLUMN column_def }
```

Note that there is some new notation here: `{}`. Braces enclose a list of options, where one option is required. In this case, we have to use either `ALTER TABLE table RENAME` or `ALTER TABLE table ADD COLUMN`. That is, you can either rename the table using the `RENAME` clause or add a column with the `ADD COLUMN` clause. To rename a table, you simply provide the new name given by `name`.

If you add a column, the column definition, denoted by `column_def`, follows the form in the `CREATE TABLE` statement. It is a name, followed by an optional domain and list of constraints. For example:

```
sqlite> ALTER TABLE contacts
        ADD COLUMN email TEXT NOT NULL DEFAULT '' COLLATE NOCASE;
sqlite> .schema contacts

CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        email TEXT NOT NULL DEFAULT '' COLLATE NOCASE,
                        UNIQUE (name,phone) );
```

To view a table definition with the SQLite command-line program, use the `.schema` shell command followed by the table name. It prints the current table definition. If you don't provide a table name, then `.schema` prints the entire database schema.

4.5 Querying a Database

As mentioned in Section 4.4, the manipulative component of SQL is called *data manipulation language* (DML). DML has two basic parts: data retrieval and data modification. Data retrieval is the domain of the `SELECT` command. It is the sole command for querying a database. `SELECT` is by far the largest and most complex command in SQL. `SELECT` derives many of its operations from relational algebra and encompasses a large portion of it.

4.5.1 Relational Operations

There are 13 relational operations used in `SELECT`, which are divided into three categories:

- fundamental operations:
  - restriction
  - projection
  - Cartesian product
  - union
  - difference
  - rename
- additional operations:
  - intersection
  - natural join
  - assign
- extended operations:
  - generalized projection
  - left outer join
  - right outer join
  - full outer join.

The fundamental operations are just that: fundamental. They define the basic relational operations and all of them (with the exception of *rename*) have their basis in set theory. The additional operations are for convenience. They can be expressed (or performed) in terms of the fundamental operations. They simply offer a shorthand way of performing frequently used combinations of the fundamental operations. Finally, the extended operations add features to the fundamental and additional operations. The generalized projection operation adds arithmetic expressions, aggregates, and grouping features to the fundamental projection operations. Outer joins extend the join operations and allow additional or incomplete information to be retrieved from the database.

In ANSI SQL, SELECT can perform every one of these relational operations. These operations make up all of the original relational operators defined by Codd (and then some) with one exception – *divide*. SQLite supports all of the relational operations in ANSI SQL with the exception of right and full outer joins (although these operations can be performed by other indirect means).

All of these operations are defined in terms of relations. They take one or more relations as their input and produce a relation as their output. It means that the fundamental structure of information never changes as a result of the operations performed upon it. Relational operations take only relations and produce only relations. This property is called *closure*. Closure makes possible *relational expressions*. The output of one operation, being a relation, can therefore serve as the input to another operation. Thus, operations can take as their arguments not only relations but also other operations (as they produce relations). This allows operations to be strung together into relational expressions. Relational expressions can therefore be created to an arbitrary degree of complexity. For example, the output of a SELECT operation (a relation) can be fed as the input to another, as follows:

```
SELECT name FROM (SELECT name, type_id FROM (SELECT * FROM foods));
```

The output of the innermost select is fed to the next SELECT, whose output is in turn is fed to the outermost SELECT. It is all a single relational expression.

The end result is that relational operations are not only powerful in their own right but can be combined to make even more powerful and elaborate expressions. The additional operators are, in fact, relational expressions composed of fundamental operations. For example, the intersection operation is defined in terms of two *difference* operations.

4.5.2 The Operational Pipeline

Syntactically, the SELECT command incorporates many of the relational operations through a series of *clauses*. Each clause corresponds to a specific relational operation. Almost all of the clauses are optional, so you can selectively employ only the operations you need to obtain the data you are looking for.

SELECT is a very large command. A very general form of SELECT (without too much distracting syntax) can be represented as follows:

```
SELECT DISTINCT heading FROM tables WHERE predicate
       GROUP BY columns HAVING predicate
       ORDER BY columns LIMIT count,offset;
```

Each keyword – such as DISTINCT, FROM, WHERE, HAVING – is a separate clause. Each clause is made up of the keyword followed by arguments. The syntax of the arguments varies according to the clause. The clauses, their corresponding relational operations, and their various arguments are listed in Table 4.1. The ORDER BY clause does not correspond to a formal relational operation as it only reorders rows. From here on, I mostly refer to these clauses simply by name. That is, rather than 'the WHERE clause,' I simply use 'WHERE'. There is both a SELECT command and a SELECT clause; I refer to the command as 'the SELECT command.'

Table 4.1 Clauses of the SELECT statement

Order	Clause	Operation	Input
1	FROM	Join	List of tables
2	WHERE	Restriction	Logical predicate
3	GROUP BY	Restriction	List of columns
4	HAVING	Restriction	Logical predicate
5	SELECT	Projection	List of columns or expressions
6	ORDER BY	–	List of columns
7	DISTINCT	Restriction	List of columns
8	LIMIT	Restriction	Integer value
9	OFFSET	Restriction	Integer value

Operationally, the SELECT command is like a pipeline that processes relations. This pipeline has optional processes that you can plug into it as you need them. Regardless of whether you include or exclude a particular operation, the overall order of operations is always the same. This order is shown in Figure 4.3.

The SELECT command starts with FROM, which takes one or more input relations, combines them into a single composite relation, and passes it through the subsequent chain of operations. Each subsequent operation takes exactly one relation as input and produces exactly one relation as output.

All operations are optional with the exception of SELECT. You must always provide at least this clause to make a valid SELECT command. By far the most common invocation of the SELECT command consists of three clauses: SELECT, FROM, and WHERE. This basic syntax and its associated clauses are shown as follows:

```
SELECT heading FROM tables WHERE predicate;
```

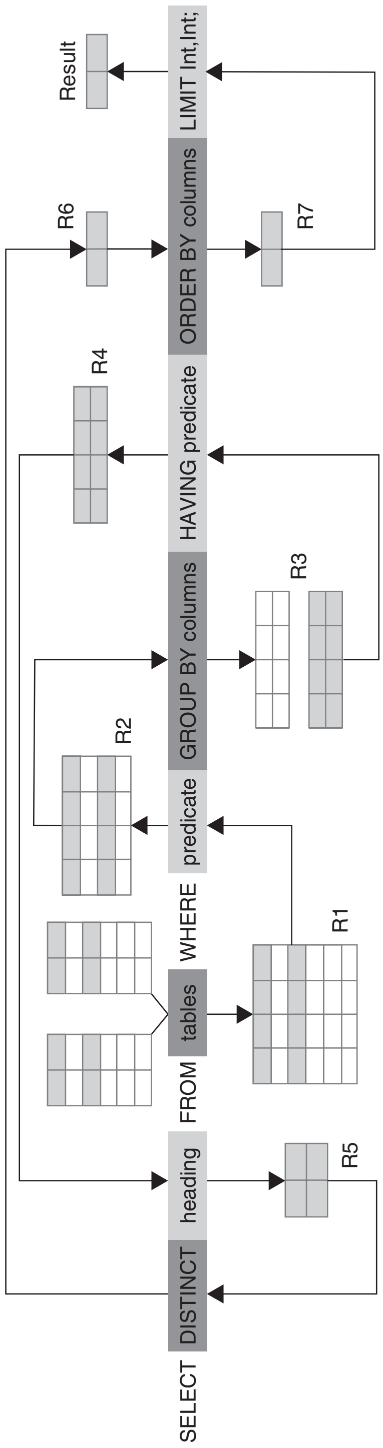


Figure 4.3 SELECT phases

The FROM clause is a comma-separated list of one or more tables (represented by the variable tables in Figure 4.3). If more than one table is specified, they are combined to form a single relation (represented by R1 in Figure 4.3). This is done by one of the join operations. The resulting relation produced by FROM serves as the initial material. All subsequent operations work directly from it or from derivatives of it.

The WHERE clause filters rows in R1. In a way, it determines the number of rows (or *cardinality*) of the result. WHERE is a restriction operation, which (somewhat confusingly) is also known as *selection*. Restriction takes a relation and produces a row-wise subset of that relation. The argument of WHERE is a *predicate*, or logical expression, that defines the selection criteria by which rows in R1 are included in (or excluded from) the result. The selected rows from the WHERE clause form a new relation R2, a restriction of R1.

In this particular example, R2 passes through the other operations unscathed until it reaches the SELECT clause. The SELECT clause filters the columns in R2. Its argument consists of a comma-separated list of columns or expressions that define the result. This is called the *projection list*, or *heading*, of the result. The number of columns in the heading is called the *degree* of the result. The SELECT clause is a projection operation, which is the process of producing a column-wise subset of a relation. Figure 4.4 illustrates a projection operation. The input is a relation composed of seven columns, including columns 1, 2 and 3. The projection operation represented by the arrow produces a new relation by extracting these columns and discarding the others.

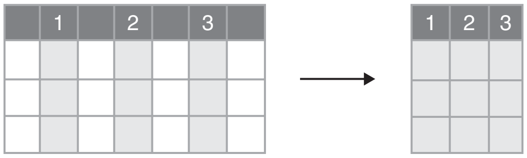


Figure 4.4 Projection

Consider the following example:

```
sqlite> SELECT id, name FROM food_types;

id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fowl
4           Condiments
```

```
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
```

There is no WHERE clause to filter rows. The SELECT clause specifies all of the columns in food_types and the FROM clause does not join tables. So there really isn't much taking place. The result is an exact copy of food_types. The input relation and the result relation are the same. If you want to include all possible columns in the result, rather than listing them one by one, you can use a shorthand notation – an asterisk (*). Thus, the previous example can just as easily be expressed as:

```
SELECT * FROM food_types;
```

This is almost the simplest SELECT command. In reality, the simplest SELECT command requires only a SELECT clause and the SELECT clause requires only a single argument. Therefore, the simplest SELECT command is:

```
sqlite> SELECT NULL;

NULL
----
NULL
```

The result is a relation composed of one row and one column, with an unknown value – NULL (see Section 4.5.13). Now let's look at an example that actually does something operationally. Consider the following query:

```
SELECT name, type_id FROM foods;
```

This performs a projection on foods, selecting two of its three columns – name and type_id. The id column is thrown out, as shown in Figure 4.5.

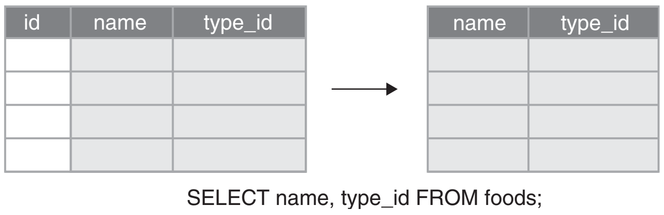


Figure 4.5 A projection of foods

Let's summarize: the FROM clause takes the input relations and performs a join, which combines them into a single relation R1. The WHERE clause takes R1 and filters it via a restriction, producing a new relation R2. The SELECT clause takes R2 and performs a projection, producing the final result. This process is shown in Figure 4.6.

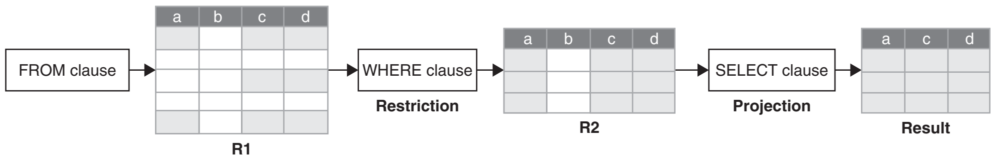


Figure 4.6 Restriction and projection in a SELECT command

With this simple example, you can begin to see how a query language in general, and SQL in particular, ultimately operates in terms of relational operations. There is real math under the hood.

4.5.3 Filtering Rows

If the SELECT command is the most complex command in SQL, then the WHERE clause is the most complex clause in SELECT. And, just as the SELECT command pulls in aspects of set theory, the WHERE clause also pulls in aspects of formal logic. By and large, the WHERE clause is usually the part of the SELECT command that harbors the most complexity. But it also does most of the work. Having a solid understanding of its mechanics will most likely bring the best overall returns in your day-to-day use of SQL.

The database applies the WHERE clause to each row of the relation produced by the FROM clause (R1). As stated earlier, WHERE – a restriction – is a filter. The argument to WHERE is a logical predicate. A predicate, in its simplest sense, is an assertion about something. Consider the following statement:

The dog (subject) is purple and has a toothy grin (predicate).

The dog is the subject and the predicate consists of two assertions (color is purple and grin is toothy). This statement may be true or false, depending on the dog to which the predicate is applied. In the terminology of formal logic, a statement consisting of subject and a predicate is called a *proposition*. All propositions are either true or false.

A predicate then says something about a subject. The subject in the WHERE clause is a row. The row is the logical subject. The WHERE clause is the logical predicate. Together (as in a grammatical sentence), they form a logical proposition, which evaluates to true or false. This proposition is formulated and evaluated for every row in R1. Each row in which the proposition evaluates to true is included (or selected) as part of the result (R2). Each row in which it is false is excluded. So the dog proposition translated into a relational equivalent would look something like this:

```
SELECT * FROM dogs WHERE color='purple' AND grin='toothy';
```

The database will take each row in relation dogs (the subject) and apply the WHERE clause (the predicate) to form the logical proposition:

```
This row has color='purple' AND grin='toothy'.
```

This is either true of the given row or it is false – nothing more. If it is true, then the row (or dog) is indeed purple and toothy, and is included in the result. If the proposition is false, then the row may be purple but not toothy, or toothy but not purple, or neither purple nor toothy. In any case, the proposition is false and the row is therefore excluded.

WHERE is a powerful filter. It provides you with a great degree of control over the conditions with which to include (or exclude) rows in (or from) the result. As a logical predicate, it is also a logical expression. A logical expression is an expression that evaluates to exactly one of two possible logical outcomes: *true or false*. A logical predicate then is just a logical expression that is used in a specific way – to qualify a subject and form a proposition.

At their simplest, logical expressions consist of two values or value expressions compared by *relational operators*. *Value expressions* are built from *values* and *operators*. Here, 'relational operator' refers to the mathematical sense, not the relational sense, as defined in relational algebra. A relational operator in mathematics is an operator that relates two values and evaluates to true or false (e.g., $x > y$, $x < =y$). A relational operator in relational algebra is an operator that takes two or more relations and produces a new relation. To minimize confusion, the scope of this discussion is limited to one relational operation – restriction – as implemented by the `WHERE` clause in SQL. The `WHERE` clause is expressed in terms of logical propositions, which use relational (in the mathematical sense) operators. For the remainder of this section, 'relational operator' refers specifically to the mathematical sense.

Values

Everything begins with *values*, which represent some kind of data in the real world. Values can be classified by their domain (or type), such as a numerical value (1, 2, 3, etc.) or string value ('JujyFruit'). Values can be expressed as *literal values* (explicit quantities, such as 1 or 'JujyFruit'), *variables* (often in the form of column names, such as `foods.name`), expressions (such as 3 + 2/5), or the results of functions (such as `COUNT(foods.name)`, see Section 4.5.5).

Operators

An operator takes one or more values as input and produces a value as output. An operator is so named because it performs some kind of operation, producing some kind of result. *Binary operators* are operators that take two input values (or operands). *Ternary operators* take three operands; *unary operators* take just one, and so on.

Many operators produce the same kind of information that they consume (for example, operators that operate on numbers produce numbers). Such operators can be strung together, feeding the output of one operator into the input of another (Figure 4.7), forming value expressions.

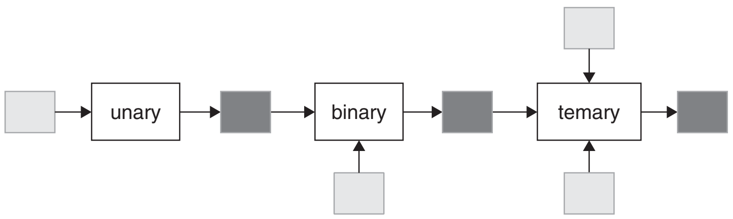


Figure 4.7 Unary, binary, and ternary operators

By stringing operators together, you can create value expressions that are expressed in terms of other value expressions, to an arbitrary degree of complexity. For example:

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

Unary Operators

The unary operators supported by SQLite are shown in Table 4.2, in order of precedence.

Table 4.2 Unary operators

Operator	Type	Action
`COLLATE`	Postfix	Collation
-	Arithmetic	Invert
+	Noop	Noop
~	Botwise	Invert
`NOT`	Logical	Invert

The COLLATE operator can be thought of as a unary postfix operator. The COLLATE operator has the highest precedence. It always binds more tightly than any prefix unary operator or any binary operator.

The unary operator + is a no-op. It can be applied to strings, numbers, or BLOBs and it always gives as its result the value of the operand.

Binary Operators

Binary operators are by far the most common operators in SQL. Table 4.3 lists the binary operators supported in SQLite by *precedence*, from highest to lowest. Operators in each group have equal precedence. Precedence determines the default order of evaluation in an expression with multiple operators. For example, take the expression $4+3*7$. It evaluates to 25. The multiplication operator has higher precedence than addition and is, therefore, evaluated first. So the expression is computed as $(3*7)+4$. Precedence can be overridden using parentheses. The expression $(4+3)*7$ evaluates to 49. The parentheses declare an explicit order of operation.

Arithmetic and Relational Operators

Arithmetic operators (e.g., addition, subtraction, division) are binary operators that take numeric values and produce a numeric value.

Table 4.3 Binary operators

Operator	Type	Action
`||`	String	Concatenation
`*`	Arithmetic	Multiply
`/`	Arithmetic	Divide
`%`	Arithmetic	Modulus
`+`	Arithmetic	Add
`-`	Arithmetic	Subtract
`<<`	Bitwise	Right shift
`>>`	Bitwise	Left shift
`&`	Logical	And
`|`	Logical	Or
`<`	Relational	Less than
`<=`	Relational	Less than or equal to
`>`	Relational	Greater than
`>=`	Relational	Greater than or equal to
`=`	Relational	Equal to
`==`	Relational	Equal to
`<>`	Relational	Not equal to
`!=`	Relational	Not equal to
`IN`	Logical	In
`AND`	Logical	And
`OR`	Logical	Or
`LIKE`	Relational	String matching
`GLOB`	Relational	Filename matching

Relational operators (e.g., >, <, =) are binary operators that compare values and value expressions and return a *logical value* (also called a *truth* value), which is either true or false. Relational operators form *logical expressions*, for example:

```
x > 5
1 < 2
```

A logical expression is any expression that returns a truth value. In SQLite, false is represented by the number 0, while true is represented by anything else. For example:

```
sqlite> SELECT 1 > 2;

1 > 2
----------
0

sqlite> SELECT 1 < 2;

1 < 2
----------
1

sqlite> SELECT 1 = 2;

1 = 2
----------
0

sqlite> SELECT -1 AND 1;

-1 AND 1
----------
1
```

Logical Operators

Logical operators (`AND`, `OR`, `NOT`, `IN`) are operators that operate on truth values or logical expressions. They produce a specific truth value depending on their inputs. They are used to build more complex logical expressions from simpler expressions, such as

```
(x > 5) AND (x != 3)
(y < 2) OR (y > 4) AND NOT (y = 0)
(color='purple') AND (grin='toothy')
```

The truth value produced by a logical operator for a given pair of arguments depends on the operator. For example, logical `AND` requires that both input values evaluate to true in order for it to return *true*. Logical `OR`, on the other hand, only requires that one input value evaluate to *true* in order for it to return *true*. All possible outcomes for a given logical operator are defined in what is known as a *truth table*. The truth tables for `AND` and `OR` are shown in Tables 4.4 and 4.5, respectively.

This is the stuff of which the `WHERE` clause is made. Using logical operators, you can create a complex logical predicate.

Table 4.4 Truth table for logical AND

Argument 1	Argument 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

Table 4.5 Truth table for logical OR

Argument 1	Argument 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

The predicate defines how `WHERE`'s relational operation applies a restriction. For example:

```
sqlite> SELECT * FROM foods WHERE name='JujyFruit' AND type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

The restriction here works according to the expression `(name='JujyFruit') AND (type_id=9)`, which consists of two logical expressions joined by logical `AND`. Both of these conditions must be true for any record in `foods` to be included in the result.

The `LIKE` Operator

A particularly useful relational operator is `LIKE`. `LIKE` is similar to equals (=), but is used to match string values against patterns. For example, to select all rows in `foods` whose names begin with the letter 'J,' you could do the following:

```
sqlite> SELECT id, name FROM foods WHERE name LIKE 'J%';

id     name
-----  --------------------
156    Juice box
```

```
236    Juicy Fruit Gum
243    Jello with Bananas
244    JujyFruit
245    Junior Mints
370    Jambalaya
```

A percent symbol (`%`) in the pattern matches any sequence of zero or more characters in the string. An underscore (_) in the pattern matches any single character in the string. The percent symbol is greedy. It 'eats' everything between two characters except those characters. If it is on the extreme left or right of a pattern, it consumes everything on the respective side. Consider the following examples:

```
sqlite> SELECT id, name FROM foods WHERE name LIKE '%ac%P%';

id     name
-----  --------------------
 38    Pie (Blackberry) Pie
127    Guacamole Dip
168    Peach Schnapps
198    Macinaw Peaches
```

Another useful trick is to use `NOT` to negate a pattern:

```
sqlite> SELECT id, name FROM foods
        WHERE name like '%ac%P%' AND name NOT LIKE '%Sch%'

id     name
-----  --------------------
 38    Pie (Blackberry) Pie
127    Guacamole Dip
198    Macinaw Peaches
```

4.5.4 Limiting and Ordering

You can limit the size and range of the result using the `LIMIT` and `OFFSET` keywords. `LIMIT` specifies the maximum number of records to return. `OFFSET` specifies the number of records to skip. For example, the following statement obtains the second record in `food_types` using `LIMIT` and `OFFSET`:

```
SELECT * FROM food_types LIMIT 1 OFFSET 1 ORDER BY id;
```

The `OFFSET` clause skips one row (the Bakery row) and the `LIMIT` clause returns a maximum of one row (the Cereal row).

But there is something else here as well: ORDER BY. This clause sorts the result by a column or columns before it is returned. The reason it is important is because the rows returned from SELECT are not guaranteed to be in any specific order – the SQL standard declares this. Thus, the ORDER BY clause is essential if you need to count on the result being in a specific order. The syntax of the ORDER BY clause is similar to the SELECT clause: it is a comma-separated list of columns. Each entry may be qualified with a sort order – ASC (ascending, the default) or DESC (descending). For example:

```
sqlite> SELECT * FROM foods WHERE name LIKE 'B%'
        ORDER BY type_id DESC, name LIMIT 10;

id     type_id   name
-----  --------  --------------------
382    15        Baked Beans
383    15        Baked Potato w/Sour
384    15        Big Salad
385    15        Broccoli
362    14        Bouillabaisse
328    12        BLT
327    12        Bacon Club (no turke
326    12        Bologna
329    12        Brisket Sandwich
274    10        Bacon
```

Typically you only need to order by a second (and third, etc.) column when there are duplicate values in the first (and second, etc.) ordered columns. In this example, there were many duplicate type_ids. I wanted to group them together and then arrange the foods alphabetically within these groups.

LIMIT and OFFSET are not standard SQL keywords as defined in the ANSI standard. Nevertheless, they are found in several other databases, such as MySQL and PostgreSQL. Oracle, MS SQL, and Firebird also have functional equivalents, although they use different syntax.

If you use both LIMIT and OFFSET together, you can use a comma notation in place of the OFFSET keyword. For example, the following SQL:

```
SELECT * FROM foods WHERE name LIKE 'B%'
ORDER BY type_id DESC, name LIMIT 1 OFFSET 2;
```

can be expressed equivalently as:

```
sqlite> SELECT * FROM foods WHERE name LIKE 'B%'
        ORDER BY type_id DESC, name LIMIT 1,2;
```

```
id     type_id   name
-----  --------  --------------------
384    15        Big Salad
```

Here the comma following `LIMIT 1` adds the `OFFSET` of 2 to the clause. Also, note that `OFFSET` depends on `LIMIT`. That is, you can use `LIMIT` without using `OFFSET` but not the other way around.

Notice that `LIMIT` and `OFFSET` are last in the operational pipeline. One common misconception of `LIMIT` and `OFFSET` is that they speed up a query by limiting the number of rows that must be collected by the `WHERE` clause. This is not true. If it were, then `ORDER BY` would not work properly. For `ORDER BY` to do its job, it must have the entire result in hand to provide the correct order.

`ORDER BY`, on the other hand, works after `WHERE` but before `SELECT`. How do I know this? The following statement works:

```
SELECT name FROM foods ORDER BY id;
```

I am asking SQLite to order by a column that is not in the result. The only way this could happen is if the ordering takes place before projection (while the `id` column is still in the set). While this works in SQLite, is also specified in SQL2003.

4.5.5 Functions and Aggregates

Relational algebra supports the notion of functions and aggregates through the extended operation known as *generalized projection*. The `SELECT` clause is a generalized projection rather than a fundamental projection. The fundamental projection operation only accepts column names in the projection list as a means of producing a column-wise subset. Generalized projection also accepts arithmetic expressions, functions, and aggregates in the projection list, in addition to other features such as `GROUP BY` and `HAVING`.

SQLite comes with various built-in functions and aggregates that can be used within various clauses. Function types range from mathematical functions, such as `ABS()`, which computes the absolute value, to string formatting functions, such as `UPPER()` and `LOWER()`, which convert text to upper and lower case, respectively. For example:

```
sqlite> SELECT UPPER('hello newman'), LENGTH('hello newman'), ABS(-12);

UPPER('hello newman')  LENGTH('hello newman') ABS(-12)
---------------------  --------------------   ----------
HELLO NEWMAN           12                     12
```

Notice that the function names are case insensitive (i.e., `upper()` and `UPPER()` refer to the same function). Functions can accept column values as their arguments:

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
        WHERE type_id=1 LIMIT 10;

id     UPPER(name)                LENGTH(name)
-----  -------------------------  ------------
1      BAGELS                     6
2      BAGELS, RAISIN             14
3      BAVARIAN CREAM PIE         18
4      BEAR CLAWS                 10
5      BLACK AND WHITE COOKIES    23
6      BREAD (WITH NUTS)          17
7      BUTTERFINGERS              13
8      CARROT CAKE                11
9      CHIPS AHOY COOKIES         18
10     CHOCOLATE BOBKA            15
```

Since functions can be part of any expression, they can also be used in the `WHERE` clause:

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
        WHERE LENGTH(name) < 5 LIMIT 5;

id     upper(name)           length(name)
-----  --------------------  --------------------
36     PIE                   3
48     BRAN                  4
56     KIX                   3
57     LIFE                  4
80     DUCK                  4
```

Just for reinforcement, let's go through the relational operations performed to carry out the preceding statement:

1. `FROM` clause (join)
   The `FROM` clause in this case does not join tables; it only produces a relation R1 containing all the rows in `foods`.
2. `WHERE` clause (restriction)
   For each row in R1:
   a. Apply the predicate `LENGTH(name) < 5` to the row. That is, evaluate the proposition 'row has LENGTH(name) < 5.'
   b. If true, add the row to R2.

3. SELECT clause (projection)
   For each row in R2:
   a. Create a new row *r* in R3.
   b. Copy the value of the id field in R2 into the first column of *r*.
   c. Copy the result of the expression UPPER(row.name) to the second column of *r*.
   d. Copy the result of the expression LENGTH(row.name) to the third column of *r*.
4. LIMIT clause (restriction)
   Restrict R3 to just the first five records.

Aggregates are a special class of functions that calculate a composite (or aggregate) value over a group of rows (a relation). The dictionary defines an aggregate as a value 'formed by the collection of units or particles into a body, mass, or amount.' The particles here are rows in a table. Standard aggregate functions include SUM(), AVG(), COUNT(), MIN(), and MAX(). For example, to find the number of foods that are baked goods (type_id=1), we can use the COUNT aggregate as follows:

```
sqlite> SELECT COUNT(*) FROM foods WHERE type_id=1;

count
-----
47
```

The COUNT aggregate returns a count of every row in the relation. Whenever you see an aggregate, you should automatically think, 'For each row in a table, do something.' It is the computed value obtained from doing something with each row in the table. For example, COUNT might be expressed in terms of the following pseudocode:

```
int COUNT():
  count = 0;
  for each row in Relation:
    count = count + 1
  return count;
```

Aggregates can aggregate not only column values, but any expression – including functions. For example, to get the average length of all food names, you can apply the AVG aggregate to the LENGTH(name) expression as follows:

```
sqlite> SELECT AVG(LENGTH(name)) FROM foods;

AVG(LENGTH(name))
-----------------
12.58
```

Aggregates operate within the SELECT clause. They compute their values on the rows selected by the WHERE clause, not from all rows selected by the FROM clause. The SELECT command filters first, then aggregates.

4.5.6 Grouping

An essential part of aggregation is grouping. That is, in addition to computing aggregates over an entire result, you can also split that result into groups of rows with like values, and compute aggregates on each group – all in one step. This is the job of the GROUP BY clause. For example:

```
sqlite> SELECT type_id FROM foods GROUP BY type_id;

type_id
----------
1
2
3
.
.
.
15
```

GROUP BY is a bit different from other parts of SELECT, so you need to use your imagination a little to wrap your head around it. Operationally, GROUP BY sits between the WHERE clause and the SELECT clause. GROUP BY takes the output of WHERE and splits it into groups of rows which share a common value (or values) for a specific column (or columns). These groups are then passed to the SELECT clause. In Figure 4.8, there are 15 different food types (type_id ranges from 1 to 15) and GROUP BY organizes all rows in foods into 15 groups varying by type_id. SELECT takes each group and extracts its common type_id value and puts it into a separate row. Thus, there are 15 rows in the result, one for each group.

When GROUP BY is used, the SELECT clause applies aggregates to each group separately, rather than to the entire relation as a whole. Since

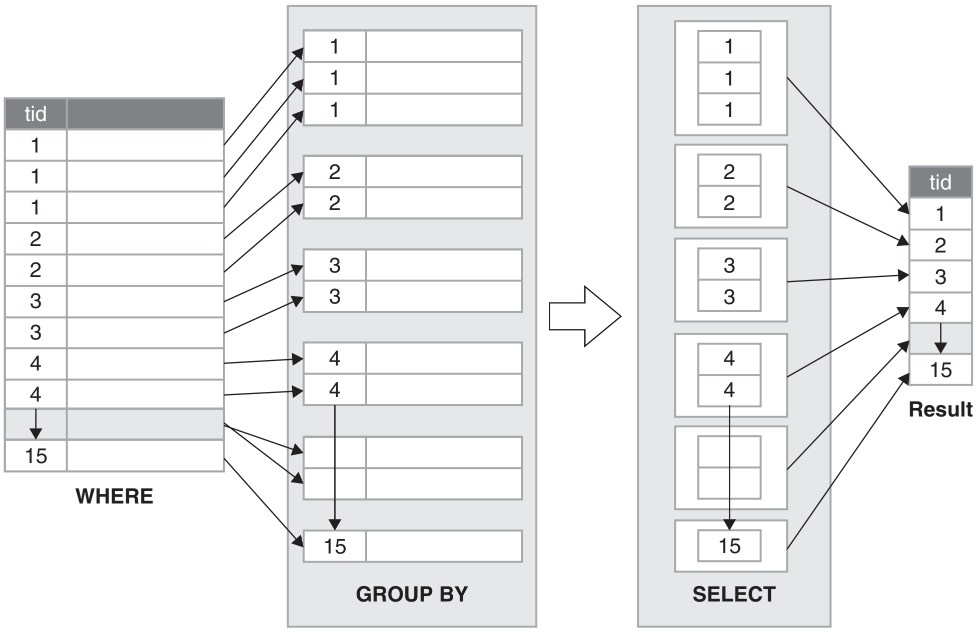


Figure 4.8 GROUP BY process

aggregates produce a single value from a group of values, they collapse these groups of rows into single rows. For example, consider applying the `COUNT` aggregate to get the number of records in each `type_id` group:

```
sqlite> SELECT type_id, COUNT(*) FROM foods GROUP BY type_id;

type_id     COUNT(*)
----------  ----------
1           47
2           15
3           23
4           22
5           17
6           4
7           60
8           23
9           61
10          36
11          16
12          23
13          14
14          19
15          32
```

Here, `COUNT()` is applied 15 times – once for each group, as illustrated in Figure 4.9. (Note that the diagram does not show the actual number

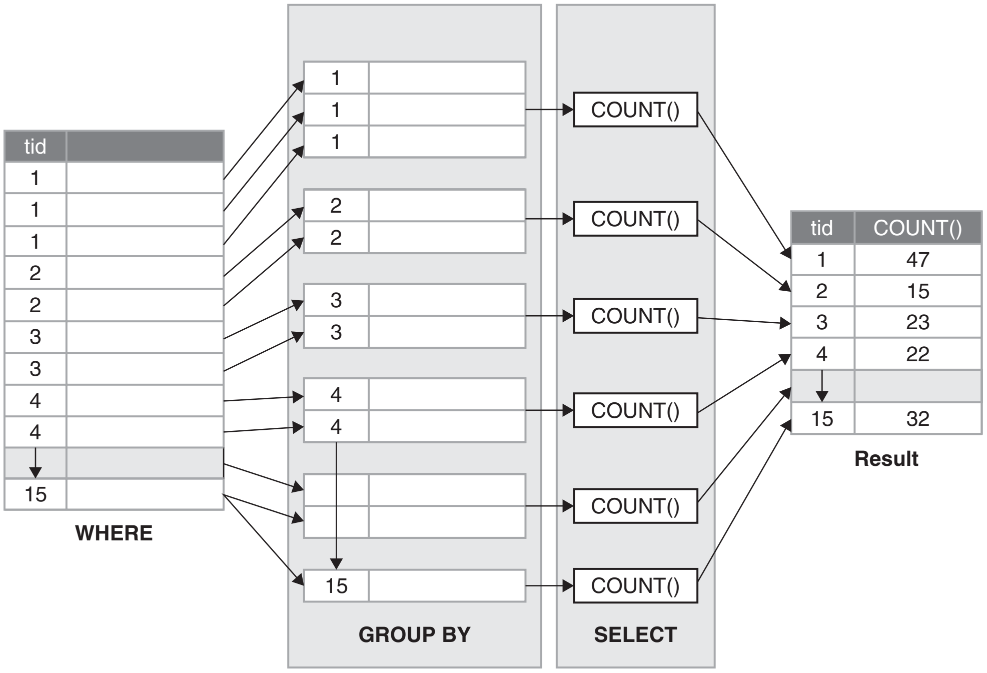


Figure 4.9 GROUP BY and aggregation

of records in each group i.e., it doesn't show 47 records in the group for `type_id=1`.)

The number of records with `type_id=1` (Baked Goods) is 47. The number with `type_id=2` (Cereal) is 15. The number with `type_id=3` (Chicken/Fowl) is 23, and so forth. So, to get this information, you could run 15 queries as follows:

```
select count(*) from foods where type_id=1;
select count(*) from foods where type_id=2;
select count(*) from foods where type_id=3;
.
.
.
select count(*) from foods where type_id=15;
```

Or, you can get the results using a single `SELECT` command with a `GROUP BY` as follows:

```
SELECT type_id, COUNT(*) FROM foods GROUP BY type_id;
```

But there is more. Since `GROUP BY` has to do all this work to create groups with like values, it seems a pity not to let you filter these groups before handing them off to the SELECT clause. That is the purpose of HAVING, a predicate that you apply to the result of GROUP BY. It filters the groups from GROUP BY in the same way that the WHERE clause filters rows from the FROM clause. The only difference is that WHERE's predicate is expressed in terms of individual row values and HAVING's predicate is expressed in terms of aggregate values.

Take the previous example, but this time say you are only interested in looking at the food groups that have fewer than 20 foods in them:

```
sqlite> SELECT type_id, COUNT(*) FROM foods
        GROUP BY type_id HAVING COUNT(*) < 20;

type_id     COUNT(*)
----------  ----------
2           15
5           17
6           4
11          16
13          14
14          19
```

Here, HAVING applies the predicate COUNT(*) < 20 to all of the groups. Any group that does not satisfy this condition (i.e., groups with 20 or more foods) is not passed on to the SELECT clause. Figure 4.10 illustrates this restriction.

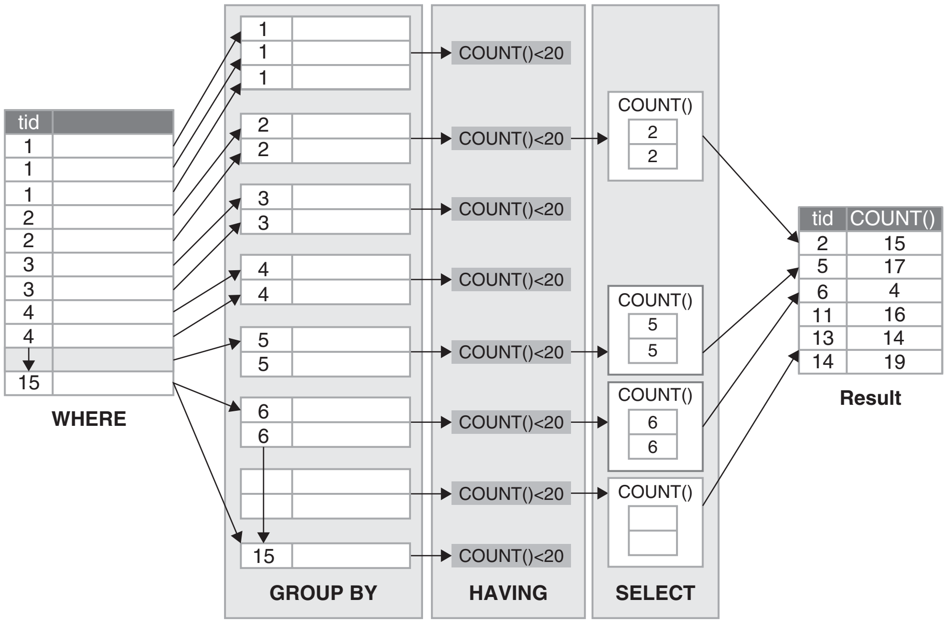


Figure 4.10 HAVING as group restriction

So GROUP BY and HAVING work as additional restriction phases. GROUP BY takes the restriction produced by the WHERE clause and breaks it into groups of rows that share a common value for a given column. HAVING then applies a filter to each of these groups. The groups that make it through are passed on to the SELECT clause for aggregation and projection.

4.5.7 Removing Duplicates

The next operation in the pipeline is another restriction: DISTINCT takes the result of the SELECT clause and filters out duplicate rows. For example, you'd use this to get all the distinct values of type_id from foods:

```
sqlite> SELECT DISTINCT type_id FROM foods;

type_id
----------
1
2
3
.
.
.
15
```

This statement works as follows: the WHERE clause returns the entire foods table (all 412 records). The SELECT clause pulls out just the type_id column and DISTINCT removes duplicate rows, reducing the number from 412 rows to 15 rows, all unique. This particular example produces the same result as the GROUP BY example, but it goes about it in a completely different manner. DISTINCT simply compares all columns of all rows listed in the SELECT clause and removes duplicates. There is no grouping on a particular column or columns nor do you specify predicates. DISTINCT is just a uniqueness filter.

4.5.8 Joining Tables

Your current knowledge of SELECT so far is based entirely on its filtering capabilities: start with something, remove rows, remove columns, aggregate, remove duplicates, and perhaps limit the number of rows even more. The SELECT, WHERE, GROUP BY, HAVING, DISTINCT, and LIMIT clauses are all filters of some sort.

Filtering, however, is only half of the picture. SELECT has two essential parts: collect and refine. Up until now you've only seen the refining, or filtering, part. The source of all this refinement has been only a single table. But SELECT is not limited to filtering just a single table; it can link tables. SELECT can construct a larger and more detailed picture made of different parts of different tables and treat that composite as your input table. Then it applies the various filters to whittle it down and isolate the parts you're interested in. This process of linking tables together is called *joining*. Joining is the work of the FROM clause.

Joins are the first operations of the SELECT command. They produce the initial information to be filtered and processed by the remaining parts of the statement. The result of a join is a *composite relation* (or table), which I will refer to as the *input relation*. It is the relation that is provided as the input or starting point for all subsequent (filtering) operations in the SELECT command.

It is perhaps easiest to start with an example. The foods table has a column type_id. As it turns out, the values in this column correspond to values in the id column in the food_types table. A relationship exists between the two tables. Any value in the foods.type_id column must correspond to a value in the food_types.id column, and the id column is the *primary key* (described later) of food_types. The foods.type_id column, by virtue of this relationship, is called a *foreign key*: it contains (or references) values in the primary key of another table. This relationship is called a *foreign key relationship*.

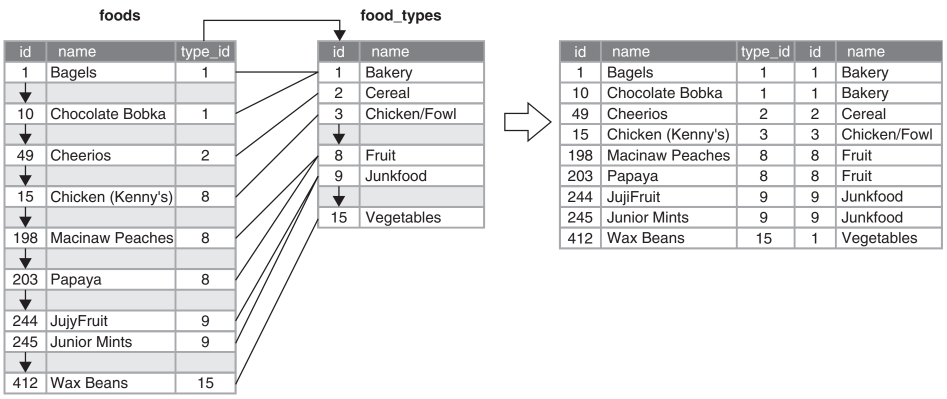
Using this relationship, it is possible join the foods and food_type tables on these two columns to make a new relation, which provides more detailed information, namely the food_types.name for each food in the foods table. This is done with the following SQL:

```
sqlite> SELECT foods.name, food_types.name
        FROM foods, food_types
        WHERE foods.type_id=food_types.id LIMIT 10;

name                         name
-------------------------    ---------------
Bagels                       Bakery
Bagels, raisin               Bakery
Bavarian Cream Pie           Bakery
Bear Claws                   Bakery
Black and White cookies      Bakery
Bread (with nuts)            Bakery
Butterfingers                Bakery
Carrot Cake                  Bakery
Chips Ahoy Cookies           Bakery
Chocolate Bobka              Bakery
```

You can see foods.name in the first column of the result, followed by food_types.name in the second. Each row in foods is linked to its associated row in food_types using the foods.type_id ~TSRA food_type.id relationship (Figure 4.11).

I am using a new notation in this example to specify the columns in the SELECT clause. Rather than specifying just the column names, I am



id	name	type_id
1	Bagels	1
↓		
10	Chocolate Bobka	1
↓		
49	Cheerios	2
↓		
15	Chicken (Kenny's)	8
↓		
198	Macinaw Peaches	8
↓		
203	Papaya	8
↓		
244	JujyFruit	9
245	Junior Mints	9
↓		
412	Wax Beans	15

id	name
1	Bakery
2	Cereal
3	Chicken/Fowl
↓	
8	Fruit
9	Junkfood
↓	
15	Vegetables

id	name	type_id	id	name
1	Bagels	1	1	Bakery
10	Chocolate Bobka	1	1	Bakery
49	Cheerios	2	2	Cereal
15	Chicken (Kenny's)	3	3	Chicken/Fowl
198	Macinaw Peaches	8	8	Fruit
203	Papaya	8	8	Fruit
244	JujiFruit	9	9	Junkfood
245	Junior Mints	9	9	Junkfood
412	Wax Beans	15	1	Vegetables

Figure 4.11 Joining `foods` and `food_types`

using the notation `table_name.column_name`. The reason is because I have multiple tables in the `SELECT` statement. The database is smart enough to figure out which table a column belongs to – as long as that column name is unique among all tables. If you use a column whose name is also defined in other tables of the join, the database cannot figure out which of the columns you are referring to and returns an error. In practice, when you are joining tables, it is always a good idea to use the `table_name.column_name` notation to avoid any possible ambiguity. This is explained in detail in Section 4.5.9.

To carry out the join, the database finds these matched rows. For each row in the first table, the database finds all rows in the second table that have the same value for the joined columns and includes them in the input relation. In this example, the `FROM` clause builds a composite relation by joining the rows of `foods` and `food_types`.

The subsequent operations (`WHERE`, `GROUP BY`, etc.) work exactly the same. It is only the input that has changed through joining tables. The predicate in the `WHERE` clause (`foods.type_id=food_types.id`) controls the records returned from the join. This happens in restriction. You might be wondering how this can be if restriction takes place after joining. Well, the short answer is that the database picks up on this by seeing two tables specified in the `FROM` clause. But this still doesn't explain how anything in the `WHERE` clause can have any effect on anything in the `FROM` clause, which is performed before it. You'll find out shortly.

As it turns out, there are six kinds of joins. The one just described, called an *inner join*, is the most common.

Inner Joins

An inner join is where two tables are joined by a relationship between two columns in the tables. It is the most common (and, perhaps, the most generally useful) type of join.

An inner join uses another set operation in relational algebra, called an *intersection*. An intersection of two sets produces a set containing elements that exist in both sets. Figure 4.12 illustrates this. The intersection of the set {1, 2, 8, 9} and the set {1, 3, 5, 8} is the set {1, 8}. The intersection operation is represented by a Venn diagram showing the common elements of both sets.

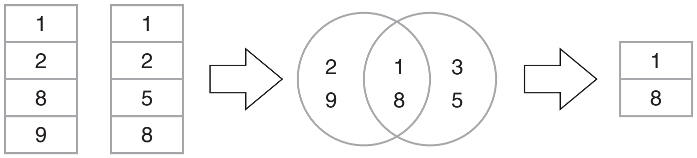
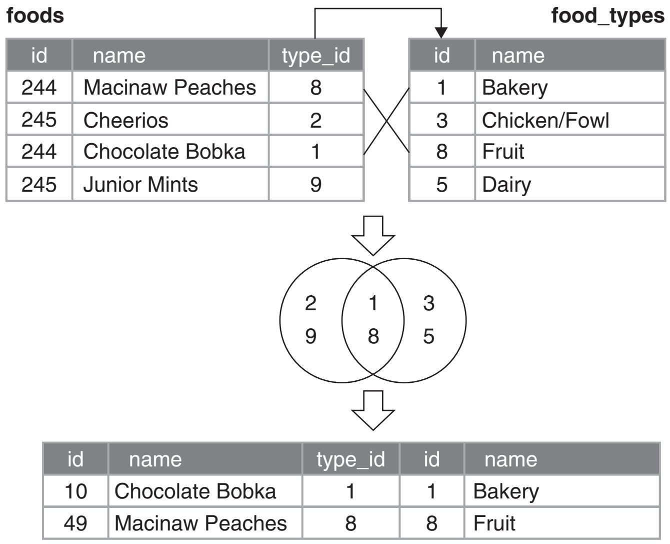


Figure 4.12 Set intersection

This is precisely how an inner join works, but the sets in a join are common elements of the related columns. Assume that the left-hand set in Figure 4.12 represents values in the `foods.type_id` column and the right set represents values of the `food_types.id` column. Given the matching columns, an inner join finds the rows from both sides that contain like values and combines them to form the rows of the result (Figure 4.13). (Note that this example assumes that the records shown are the only records in `foods` and `food_types`.)



foods

id	name	type_id
244	Macinaw Peaches	8
245	Cheerios	2
244	Chocolate Bobka	1
245	Junior Mints	9

food_types

id	name
1	Bakery
3	Chicken/Fowl
8	Fruit
5	Dairy

id	name	type_id	id	name
10	Chocolate Bobka	1	1	Bakery
49	Macinaw Peaches	8	8	Fruit

Figure 4.13 Join between `foods` and `food_types` as intersection

An inner join returns only rows that satisfy the given column relationship, also called the *join condition*. They answer the question 'Which rows in *B* match rows in *A* given the following relationship?' Consider the tables shown in Figure 4.14. They both have three columns, with one in common (named `a`).

a	b	c
1	4	7
2	5	8
3	6	9

A

a	d	e
1	4	7
2	5	8
4	6	9

B

Figure 4.14 Tables A and B

You can see that two rows in B match two rows in A with respect to column `a`. This is the inner join:

```
SELECT * FROM A, B where A.a=B.a;
```

The result is shown in Figure 4.15.

a	b	c	a	d	e
1	4	7	1	4	7
2	5	8	2	5	8

Figure 4.15 Inner join of tables A and B

Notice that the third row of B does not match any row in A for this condition (and vice versa). Therefore only two rows are included in the result. The join condition (in this case, `A.a=B.a`) is what makes this an inner join opposed to another type of join.

Cross Joins

Imagine for a moment that there were no join condition. What would you get? If the two tables were not related in any way, `SELECT` would produce a more fundamental kind of join (the *most* fundamental), which is called a *cross join* or *Cartesian join* (if you read Chapter 3, you know this as the *cross product* or *Cartesian product*). The Cartesian join is one of the fundamental relational operations. It is a brute-force, almost nonsensical join that results in the combination of all rows from the first table with all rows in the second. A cross join of tables A and B can be expressed with the following pseudocode:

```
for each record a in A
  for each record b in B
    make record a + b
```

In SQL, the cross join of `A` and `B` is expressed as follows:

```
SELECT * FROM A,B;
```

`FROM`, in the absence of anything else, produces a cross join. The result is shown in Figure 4.16. Every row in A is combined with every row in B. In a cross join, no relationship exists between the rows; there is no join condition – they are simply jammed together.

a	b	c	a	d	e
1	4	7	1	4	7
1	4	7	2	5	8
1	4	7	4	6	9
2	5	8	1	4	7
2	5	8	2	5	8
2	5	8	4	6	9
3	6	9	1	4	7
3	6	9	2	5	8
3	6	9	4	6	9

Figure 4.16 Cross join of tables A and B

This, then, is what the `WHERE` clause was filtering with the (inner) join condition in the preceding example. It was removing all those rows in the cross join that had no sensible relationship. An inner join is a subset of a cross join. A cross join contains every possible combination of rows in two tables, whereas an inner join contains only those rows that satisfy a specific relationship between columns in the two tables.

In a purely relational sense, a join is composed of the following set operations, in order:

1. **Cross join**: Take the cross product of all tables in the source list.
2. **Restrict**: Apply the join condition (and any other restrictions) to the cross product to narrow it down. An inner join takes the intersection

of related columns to select matching rows. Other joins use other criteria.

3. **Project**: Select the desired columns from the restriction.

In this sense, all joins begin as a cross join of all tables in the `FROM` clause, producing a set of every combination of rows therein. This is the mathematical process. It is most certainly *not* the process used by relational databases. It would be wasteful, to say the least, for databases to blindly start every join by computing the cross product of all the tables listed in the `FROM` clause. There are many ways that databases optimize this process to avoid combining rows that will simply be thrown out by the `WHERE` clause. Thus, the mathematical concept and the database implementation are completely different. However, the end results are logically equivalent.

Outer Joins

Three of the remaining four joins are called *outer joins*. An inner join selects rows across tables according to a given relationship. An outer join selects all of the rows of an inner join plus some rows outside of the relationship. The three outer join types are called *left*, *right*, and *full*. A left join operates with respect to the 'leftmost table' in the SQL command. For example, in the following command, table A is the left table:

```
SELECT * FROM A LEFT JOIN B ON A.a=B.a;
```

The left join favors it. It is the table of significance in a left join. The left join tries to match every row of A with every row in B with respect to the join condition (`A.a=B.a`). All matching rows are included in the result and the remaining rows of A that don't match B are also included in the result. In this case, these rows have empty values for the B columns. The result is shown in Figure 4.17.

The third row of A has no matching row in B with respect to column `a`. Despite this, the left join includes all rows of A.

a	b	c	a	d	e
1	4	7	1	4	7
2	5	8	2	5	8
3	6	9			

Figure 4.17 Left join of tables A and B

The right join works similarly, except the right table is the one whose rows are included, matching or not. Operationally, left and right joins are identical; they do the same thing. They differ only in order and syntax. You could argue that there is never any need to do a right join as the only thing it does is swap the arguments in a left join.

A full outer join is the combination of a left and right outer join. It includes all matching records, followed by unmatched records in the right and left tables. Currently, neither right nor full outer joins are supported in SQLite. However, as mentioned earlier, a right join can be replaced with a left join and a full outer join can be performed using compound queries (see Section 4.5.11).

Natural Joins

The last join on the list is called a *natural join*. It is actually an inner join in disguise, but with a little syntax and convention thrown in. A natural join joins two tables by their common column names. Thus, using the natural join you can get the inner join of `A` and `B` without having to add the join condition `A.a=B.a`:

```
sqlite> SELECT * FROM A NATURAL JOIN B;

a           b           c           d           e
----------  ----------  ----------  ----------  ----------
1           4           7           4           7
2           5           8           5           8
```

Here, the natural join automatically detects the common column names in `A` and `B` (in this case, column `a`) and links them. In practice, it is a good idea to avoid natural joins in your applications. Natural joins join *all* columns of the same names. Just the process of adding a column to or removing a column from a table can drastically change the results of a natural join query. Say a program uses a natural join query on A and B. Then suppose someone comes along and adds a new column `e` to A. That causes the natural join (and thus the program) to produce completely different results. It's always better to explicitly define the join conditions of your queries than to rely on the semantics of the table schema.

Preferred Join Syntax

Syntactically, there are various ways of specifying a join. The inner join example of A and B illustrates performing a join implicitly in the `WHERE` clause:

```
SELECT * FROM A,B WHERE A.a=B.a;
```

When the database sees more than one table listed, it knows there is a join – at the very least a cross join. The `WHERE` clause here calls for an inner join.

This implicit form, while rather clean, is an older form of syntax that you should avoid. The correct way to express a join in SQL (as defined in SQL92) is to use the `JOIN` keyword. The general form is

```
SELECT heading FROM LEFT_TABLE join_type RIGHT_TABLE ON join_condition;
```

This explicit form can be used for all join types. For example:

```
SELECT * FROM A INNER JOIN B ON A.a=B.a;
SELECT * FROM A LEFT JOIN B ON A.a=B.a;
SELECT * FROM A NATURAL JOIN B ON A;
SELECT * FROM A CROSS JOIN B ON A;
```

Finally, when the join condition is based on columns that share the same name, it can be simplified with the `USING` keyword. `USING` simply names the common column (or columns) to include in the join condition:

```
SELECT * FROM A INNER JOIN B USING (a);
```

The argument to `USING` is a comma-separated list of column names within parentheses.

Joins are processed from left to right. Consider the multiway join shown in Figure 4.18.

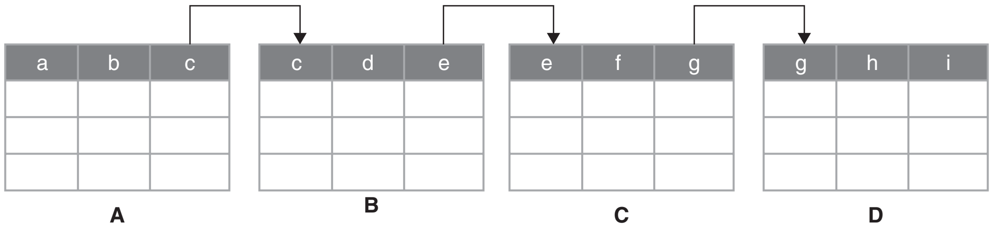


Figure 4.18 Multiway join

Using the preferred form, you simply keep tacking on additional tables by adding more join expressions to the end of the statement:

```
SELECT * FROM A JOIN B USING (c) JOIN C USING (e) JOIN D USING (g);
```

The first relation, R1, is from `A join B on c`. The second relation, R2, is `R1 join C on e`. The final relation, R3, is `R2 join D on g`. This is shown in Figure 4.19.

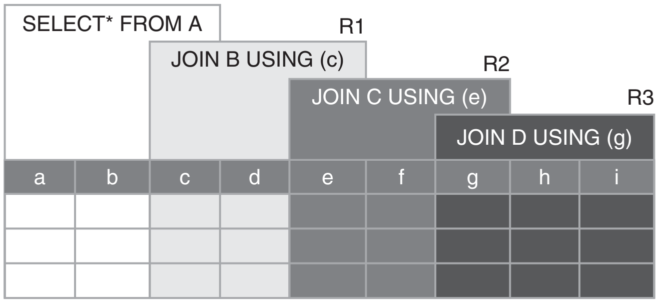


Figure 4.19 Correct syntax for a multiway join construction

4.5.9 Column Names and Aliases

When joining tables, column names can become ambiguous. For example, tables A and B, as defined in Figure 4.14, both have a column `a`. What if they were to have another column, named `name`, and you want to select the `name` column of table B but not table A? To help with this type of task, you can qualify column names with their table names to remove any ambiguity. So in this example, you could write

```
SELECT B.name FROM A JOIN B USING (a);
```

Another useful feature is *aliases*. If your table name is particularly long, and you don't want to have to use its name every time you qualify a column, you can use an alias. Aliasing is a fundamental relational operation called *rename*. The rename operation simply assigns a new name to a relation. For example, consider the statement:

```
SELECT foods.name, food_types.name FROM foods, food_types
WHERE foods.type_id=food_types.id LIMIT 10;
```

There is a good bit of typing there. You can rename the tables in the source clause by simply including the new name directly after the table name, as in the following example:

```
SELECT f.name, t.name FROM foods f, food_types t
WHERE f.type_id=t.id LIMIT 10;
```

Here, the foods table is assigned the alias f and the food_types table is assigned the alias t. Now, every other reference to foods or food_types in the statement must use the alias f or t, respectively. Aliases make it possible to join a table to itself (a *self-join*). For example, say you want to know which foods mentioned in season 4 are mentioned in other seasons. You would first need to get a list of episodes and foods in season 4, which you would obtain by joining episodes and episodes_foods. But then you would need a similar list for foods outside of season 4. Finally, you would combine the two lists based on their common foods. The following query uses self-joins to do the trick:

```
SELECT f.name as food, e1.name, e1.season, e2.name, e2.season
FROM episodes e1, foods_episodes fe1, foods f,
     episodes e2, foods_episodes fe2
WHERE
  -- Get foods in season 4
  (e1.id = fe1.episode_id AND e1.season = 4) AND fe1.food_id = f.id
  -- Link foods with all other epsisodes
  AND (fe1.food_id = fe2.food_id)
  -- Link with their respective episodes and filter out e1's season
  AND (fe2.episode_id = e2.id AND e2.season != e1.season)
ORDER BY f.name;

food               name            season  name                   season
----------------   --------------  ------  ---------------------  ------
Bouillabaisse      The Shoes       4       The Stake Out          1
Decaf Cappuccino   The Pitch       4       The Good Samaritan     3
Decaf Cappuccino   The Ticket      4       The Good Samaritan     3
Egg Salad          The Trip 1      4       Male Unbonding         1
Egg Salad          The Trip 1      4       The Stock Tip          1
Mints              The Trip 1      4       The Cartoon            9
Snapple            The Virgin      4       The Abstinence         8
Tic Tacs           The Trip 1      4       The Merv Griffin Show  9
Tic Tacs           The Contest     4       The Merv Griffin Show  9
Tuna               The Trip 1      4       The Stall              5
Turkey Club        The Bubble Boy  4       The Soup               6
Turkey Club        The Bubble Boy  4       The Wizard             9
```

I have put comments in the SQL to better explain what is going on. This example uses two self-joins. There are two instances of each of episodes and foods_episodes, but they are treated as if they are independent tables. The query joins foods_episodes back on itself to link the two instances of episodes. Furthermore, the two episodes instances are related to each other by an inequality condition to ensure that they are in different seasons.

You can alias column names and expressions in the same way. For example, to get the top ten episodes with the most foods, nicely labeled, you'd use this query:

```
SELECT e.name AS Episode, COUNT(f.id) AS Foods
FROM foods f
  JOIN foods_episodes fe on f.id=fe.food_id
  JOIN episodes e on fe.episode_id=e.id
GROUP BY e.id
ORDER BY Foods DESC
LIMIT 10;

Episode                Foods
--------------------   -----
The Soup               23
The Fatigues           14
The Bubble Boy         12
The Finale 1           10
The Merv Griffin Show  9
The Soup Nazi          9
The Wink               9
The Dinner Party       9
The Glasses            9
The Mango              9
```

Note that the AS keyword is optional. I just use it because it seems more legible that way to me. Column aliases change the column headers returned in the result set. You may refer to columns or expressions by their aliases elsewhere in the statement if you wish (as in the ORDER BY clause in the example above), but you are not required to do so (unlike with table aliases).

4.5.10 Subqueries

Subqueries are SELECTs within SELECTs. They are also called *subselects*. Subqueries are useful in many ways: they work anywhere that normal expressions work and they can, therefore, be used in a variety of places in a SELECT statement. Subqueries are useful in other commands as well.

Perhaps the most common use of a subquery is in the WHERE clause, specifically using the IN operator. The IN operator is a binary operator that takes an input value and a list of values and returns true if the input value exists in the list. Here are examples:

```
sqlite> SELECT 1 IN (1,2,3);
1

sqlite> SELECT 2 IN (3,4,5);
0

sqlite> SELECT COUNT(*) FROM foods WHERE type_id IN (1,2);
62
```

Using a subquery, you can rewrite the last statement in terms of names from the food_types table:

```
SELECT COUNT(*) FROM foods
WHERE type_id IN (SELECT id FROM food_types
                  WHERE name='Bakery' OR name='Cereal');
62
```

Subqueries in the SELECT clause can be used to add data from other tables to the result set. For example, to get the number of episodes in which each food appears, the count from foods_episodes can be performed in a subquery in the SELECT clause:

```
SELECT name,
       (SELECT COUNT(id) FROM foods_episodes WHERE food_id=f.id) count
FROM foods f ORDER BY count DESC LIMIT 10;

name            count
---------- ----------
Hot Dog         5
Pizza           4
Ketchup         4
Kasha           4
Shrimp          3
Lobster         3
Turkey Sandwich 3
Turkey Club     3
Egg Salad       3
Tic Tacs        3
```

The ORDER BY and LIMIT clauses here serve to create a top-ten list. Notice that the subquery's predicate references a table in the enclosing SELECT command: food_id=f.id. The variable f.id exists in the outer query. The subquery in this example is called a *correlated subquery* because it references, or correlates with, a variable in the outer (enclosing) query.

Subqueries in the SELECT clause are also helpful with computing percentages of aggregates. For example, the following SQL command breaks foods down by types and their respective percentages:

```
SELECT (SELECT name FROM food_types WHERE id=f.type_id) Type,
  COUNT(type_id) Items,
  COUNT(type_id)*100.0/(SELECT COUNT(*) FROM foods) as Percentage
FROM foods f GROUP BY type_id ORDER BY Percentage DESC;
```

```
Type           Items    Percentage
-------------- ------- --------------
Junkfood        61       14.76997578692
Drinks          60       14.52784503631
Bakery          48       11.62227602905
Meat            36       8.716707021791
Vegetables      32       7.748184019370
Chicken/Fowl    23       5.569007263922
Fruit           23       5.569007263922
Sandwiches      23       5.569007263922
Condiments      22       5.326876513317
Soup            19       4.600484261501
Dairy           17       4.116222760290
Rice/Pasta      16       3.874092009685
Cereal          15       3.631961259079
Seafood         14       3.389830508474
Dip             4        0.968523002421
```

A subquery must be used as the divisor, rather than COUNT(), because the statement uses a GROUP BY. Remember that aggregates in GROUP BY are applied to groups, not the entire result set; therefore the subquery's COUNT() must be used to get the total rows in foods.

Subqueries can also be used in the ORDER BY clause. The following SQL statement groups foods by the size of their respective food groups, from greatest to least:

```
SELECT * FROM foods f
ORDER BY
  (SELECT COUNT(type_id)
   FROM foods WHERE type_id=f.type_id) DESC;
```

ORDER BY, in this case, does not refer to any specific column in the result. How does this work? The ORDER BY subquery is run for each row and the result is associated with the given row. You can think of it as an invisible column in the result set, which is used to order rows.

Finally, we have the FROM clause. There may be times when you want to join only part of a table rather than all of it. Yet another job for a subquery:

```
SELECT f.name, types.name FROM foods f
INNER JOIN (SELECT * FROM food_types WHERE id=6) types
ON f.type_id=types.id;

name                       name
-------------------------- -----
Generic (as a meal)        Dip
Good Dip                   Dip
Guacamole Dip              Dip
Hummus                     Dip
```

Notice that the use of a subquery in the FROM clause requires a rename operation. In this case, the subquery was named types. This query could have been written using a full join of food_types, of course, but it may have incurred more overhead as there would have been more records to match.

Another use of subqueries is to reduce the number of rows in an aggregating join. Consider the following query:

```
SELECT e.name name, COUNT(fe.food_id) foods FROM episodes e
INNER JOIN foods_episodes fe ON e.id=fe.episode_id
GROUP BY e.id
ORDER BY foods DESC
LIMIT 10;

name            foods
-------------- -------
The Soup        23
The Fatigues    14
The Bubble Boy  12
The Finale 1    10
The Mango       9
The Glasses     9
The Dinner Par  9
The Wink        9
The Soup Nazi   9
The Merv Griff  9
```

This query lists the top ten shows with the most food references. The join here matches the 181 rows in episodes with 502 rows in foods_episodes (181/502). Then it computes the aggregate. The fewer rows it has to match, the less work it has to do and the more efficient (and faster) the query becomes. Aggregation collapses rows, right? Then a great way to reduce the number of rows the join must match is to aggregate *before* the join. The only way to do this is with a subquery:

```
SELECT e.name, agg.foods FROM episodes e
INNER JOIN
(SELECT fe.episode_id as eid, count(food_id) as foods
   FROM foods_episodes fe
   GROUP BY episode_id ) agg
ON e.id=agg.eid
ORDER BY agg.foods DESC
LIMIT 10;
```

This query moves aggregation into a subquery that is run before the join, the results of which are joined to the episodes table for the final result. The subquery produces one aggregate (the number of foods) per episode. There are 181 episodes. Thus, this query reduces the join size from 181/502 to 181/181. However, since we are only looking for the top ten food-referencing episodes, we can move the LIMIT 10 clause into the subquery as well and reduce the number further, to 181/10:

```
SELECT e.name, agg.foods FROM episodes e
INNER JOIN
(SELECT fe.episode_id as eid, count(food_id) as foods
   FROM foods_episodes fe
   GROUP BY episode_id
   ORDER BY foods DESC LIMIT 10) agg
ON e.id=agg.eid
ORDER BY agg.foods DESC;
```

This subquery returns only ten rows, which correspond to the top ten food shows. The join disrupts the descending order of the subquery so another `ORDER BY` is required in the main query to reestablish it.

The thing to remember about subqueries is that they can be used *anywhere* a relational expression can be used. A good way to learn how, where, and when to use them is to play around with them and see what you can get away with. There is often more than one way to achieve something in SQL. When you understand the big picture, you can make more informed decisions on when a query might be rewritten to run more efficiently.

4.5.11 Compound Queries

Compound queries are the inverse of subqueries. A compound query is a query that processes the results of other queries using three specific relational operations: union, intersection, and difference. In SQL, these are defined using the `UNION`, `INTERSECT`, and `EXCEPT` keywords, respectively.

Compound query operations require a few things of their arguments:

- The relations involved must have the same number of columns (must be of the same *degree*).
- There can only be one `ORDER BY` clause, which is at the end of the compound query and applies to the combined result.

Furthermore, relations in compound queries are processed from left to right.

The `INTERSECT` operation takes two relations *A* and *B* and selects all rows in *A* that also exist in *B*. The following SQL command uses `INTERSECT` to find which of the top ten foods appear in seasons 3 through 5:

```
SELECT f.* FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) as count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 10) top_foods
  ON f.id=top_foods.food_id
INTERSECT
SELECT f.* FROM foods f
  INNER JOIN foods_episodes fe ON f.id = fe.food_id
  INNER JOIN episodes e ON fe.episode_id = e.id
  WHERE e.season BETWEEN 3 and 5
ORDER BY f.name;

id     type_id  name
----- ------- --------------------
4      1        Bear Claws
146    7        Decaf Cappuccino
153    7        Hennigen's
55     2        Kasha
94     4        Ketchup
164    7        Naya Water
317    11       Pizza
```

To produce the top ten foods, I needed an `ORDER BY` in the first `SELECT` statement. Since compound queries only allow one `ORDER BY` at the end of the statement, I got around this by performing an inner join on a subquery in which I computed the top ten most common foods. Subqueries can have an `ORDER BY` clause because they run independently of the compound query. The inner join produces a relation containing the top ten foods. The second query returns a relation containing all foods in episodes 3 through 5. The `INTERSECT` operation then finds all matching rows.

The `EXCEPT` operation takes two relations *A* and *B* and finds all rows in *A* that are not in *B*. By changing the `INTERSECT` in the previous example to `EXCEPT`, you can find which top ten foods are *not* in seasons 3 through 5:

```
SELECT f.* FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 10) top_foods
  ON f.id=top_foods.food_id
EXCEPT
SELECT f.* FROM foods f
  INNER JOIN foods_episodes fe ON f.id = fe.food_id
  INNER JOIN episodes e ON fe.episode_id = e.id
  WHERE e.season BETWEEN 3 and 5
ORDER BY f.name;

id     type_id  name
----- ------- --------
192    8        Banana
133    7        Bosco
288    10       Hot Dog
```

As mentioned earlier, the EXCEPT operation in SQL is referred to as the *difference* operation in relational algebra.

The UNION operation takes two relations, *A* and *B*, and combines them into a single relation containing all distinct rows of *A* and *B*. In SQL, UNION combines the results of two SELECT statements. By default, UNION eliminates duplicates. If you want duplicates included in the result, then use UNION ALL. For example, the following SQL command finds the single most and single least frequently mentioned foods:

```
SELECT f.*, top_foods.count FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 1) top_foods
  ON f.id=top_foods.food_id
UNION
SELECT f.*, bottom_foods.count FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) LIMIT 1) bottom_foods
  ON f.id=bottom_foods.food_id
ORDER BY top_foods.count DESC;

id     type_id  name       top_foods.count
----- ------- --------- ---------------
288    10       Hot Dog    5
1      1        Bagels     1
```

Both queries return only one row. The only difference in the two is which way they sort their results. The UNION simply combines the two rows into a single relation.

UNION ALL also provides a way to implement a full outer join. You simply perform a left join in the first SELECT, a right join (written in the form of a left join) in the second, and the result is a full outer join.

Compound queries are useful when you need to process similar data sets that are materialized in different ways. Basically, if you cannot express everything you want in a single SELECT statement, you can use a compound query to get part of what you want in one SELECT statement and part in another (and perhaps more), and process the sets accordingly. For example, the earlier INTERSECT example compared two lists of foods that could not be materialized with a single SELECT statement. One list contained the top ten foods and the second contained all foods that appeared in specific seasons. I wanted a list of the top ten foods that appeared in the specific seasons. A compound query was the only way to get this information. The same was true in the UNION example. It is not possible to select just the first and last records of a result set. The UNION took two slight variations of a query to get the first and last records and combined the results. Also, as with joins, you can string as many SELECT commands together as you like by using any of these compound operations.

4.5.12 Conditional Results

The CASE expression allows you handle various conditions within a SELECT statement. There are two forms. The first and simplest form takes a static value and lists various case values linked to return values:

```
CASE value
  WHEN x THEN value_x
  WHEN y THEN value_y
  WHEN z THEN value_z
  ELSE default_value
END
```

Here's a simple example:

```
SELECT name || CASE type_id
                 WHEN 7  THEN ' is a drink'
                 WHEN 8  THEN ' is a fruit'
                 WHEN 9  THEN ' is junk food'
                 WHEN 13 THEN ' is seafood'
                 ELSE NULL
               END description
FROM foods
WHERE description IS NOT NULL
ORDER BY name
LIMIT 10;

description
---------------------------------------------
All Day Sucker is junk food
Almond Joy is junk food
Apple is a fruit
Apple Cider is a drink
Apple Pie is a fruit
Arabian Mocha Java (beans) is a drink
Avocado is a fruit
Banana is a fruit
Beaujolais is a drink
Beer is a drink
```

The CASE expression in this example handles a few different type_id values, returning a string appropriate for each one. The returned value is called description, as qualified after the END keyword. This string is concatenated to name by the string concatenation operator (||), making a complete sentence. For all type_ids not specified in a WHEN condition, CASE returns NULL. The SELECT statement filters out such NULL values in the WHERE clause, so all that is returned are rows that the CASE expression handles.

The second form of CASE allows for expressions in the WHEN condition. It has the following form:

```
CASE
  WHEN condition1 THEN value1
  WHEN condition2 THEN value2
  WHEN condition3 THEN value3
  ELSE default_value
END
```

CASE works equally well in subselects comparing aggregates. The following SQL command picks out frequently mentioned foods:

```
SELECT name,(SELECT
               CASE
                 WHEN count(*) > 4
                   THEN 'Very High'
                 WHEN count(*) = 4
                   THEN 'High'
                 WHEN count(*) IN (2,3)
                   THEN 'Moderate'
                 ELSE 'Low'
               END
             FROM foods_episodes
             WHERE food_id=f.id) frequency
FROM foods f
WHERE frequency LIKE '%High';

name       frequency
--------- ----------
Kasha      High
Ketchup    High
Hot Dog    Very High
Pizza      High
```

This query runs a subquery for each row in foods that classifies the food by the number of episodes it appears in. The result of this subquery is included as column called frequency. The WHERE predicate filters frequency values that include the word 'High.'

Only one condition is executed in a CASE expression. If more than one condition is satisfied, only the first of them is executed. If no conditions are satisfied and no ELSE condition is defined, CASE returns NULL.

4.5.13 The Thing Called NULL

Most relational databases support a special value called NULL. NULL is a placeholder for missing information. NULL is not a value per se. Rather, NULL is the absence of a value. Better yet, it is a value denoting the absence of a value. Some say it stands for 'unknown,' 'not applicable,' or 'not known.' But truth be told, NULL is still rather vague and mysterious. Try as you may to nail it down, NULL can still play with your mind: NULL is not nothing; NULL is not something; NULL is not true; NULL is not false; NULL is not zero. Simply put, NULL is resolutely what it is: NULL. And not everyone can agree on what that means. To some, NULL is a four-letter word. NULL rides a Harley, sports racy tattoos, and refuses to conform. To others, it is a necessary evil and serves an important role in society. You may love it. You may hate it. But it's here to stay. And if you are going to keep company with NULL, you'd better know what you are getting yourself into.

Based on what you already know, it should come as no surprise to learn that even the SQL standard is not completely clear on how to deal with NULL in all cases. Regardless, the SQLite community came to a consensus by evaluating how a number of other major relational databases handled NULL in particular ways. The result was that there was some but not total consistency in how they worked. Oracle, PostgreSQL, and DB2 were almost identical with respect to NULL handling, so SQLite's approach was to be compatible with them.

Working with NULL is appreciably different than working with any other kind of value. For example, if you are looking for rows in foods whose name is NULL, the following SQL is useless:

```
SELECT * FROM foods WHERE foods.name=NULL;
```

It won't return any rows, period. The problem here is that the expression *anything*=NULL evaluates to NULL (even the expression NULL=NULL is NULL). And NULL is not true (nor is it false), so the WHERE clause will never evaluate to true, and therefore no rows are selected by the query in its current form. In order to get it to work as intended, you must recast the query to use the IS operator:

```
SELECT * FROM foods WHERE foods.name IS NULL;
```

The IS operator properly checks for a NULL and returns true if it finds one. If you want values that are not NULL, then use IS NOT NULL.

But this is just the beginning of our NULL fun. NULL has a kind of Midas-like quality in that everything it touches turns to NULL. For example:

```
sqlite> SELECT NULL=NULL;
NULL
```

```
sqlite> SELECT NULL OR NULL;
NULL

sqlite> SELECT NULL AND NULL;
NULL

sqlite> SELECT NOT NULL;
NULL

sqlite> SELECT 9E9 - 1E-9*NULL;
NULL
```

Additionally, it is important to note that `COUNT(*)` and `COUNT(column)` are distinctly different with respect to how they handle `NULL`. `COUNT(*)` counts rows, regardless of any particular column value, so `NULL` has no effect on it. `COUNT(column)`, on the other hand, only counts the rows with non-`NULL` values in `column`; rows where `column` is `NULL` are ignored.

In order to accommodate `NULL` in logical expressions, SQL uses something called *three-value* (or *tristate*) logic, where `NULL` is one of the truth values. The truth table for logical `AND` and logical `OR` with `NULL` thrown into the mix is shown in Table 4.6.

Table 4.6 AND and OR with `NULL`

x	y	x AND y	x OR y
True	True	True	True
True	False	False	True
True	NULL	NULL	True
False	True	False	True
False	False	False	False
False	NULL	False	NULL
NULL	NULL	NULL	NULL

Since a single `NULL` can nullify an entire expression, it seems that there should be some way to specify a suitable default in case a `NULL` crops up. To that end, SQLite provides a function for dealing with `NULL`, called `COALESCE`, which is part of the SQL99 standard. `COALESCE` takes a list of values and returns the first non-`NULL` in the list. Take the following example:

```
SELECT item.price-(SELECT SUM(amount) FROM discounts WHERE id=item.id)
FROM products WHERE . . .
```

This is a hypothetical example of a query that calculates the price of an item with any discounts applied. But what if the subquery returns NULL? Then the whole calculation and, therefore, the price becomes NULL. That's not good. COALESCE provides a barrier through which NULL may not pass:

```
SELECT item.price-(SELECT COALESCE(SUM(amount),0)
                   FROM discounts WHERE id=item.id)
FROM products WHERE . . .
```

In this case if SUM(amount) turns out to be NULL, COALESCE returns 0 instead, taking the teeth out of NULL.

Conversely, the NULLIF function takes two arguments and returns NULL if they have the same values; otherwise, it returns the first argument:

```
sqlite> SELECT NULLIF(1,1);
NULL

sqlite> SELECT NULLIF(1,2);
1
```

If you use NULL, you need to take special care in queries that refer to columns that may contain NULL in their predicates and aggregates. NULL can do quite a number on aggregates if you are not careful. Consider the following example:

```
sqlite> CREATE TABLE sales (product_id int, amount real,
                                         discount real);
sqlite> INSERT INTO sales VALUES (1, 10.00, 1.00);
sqlite> INSERT INTO sales VALUES (2, 10000.00, NULL);
sqlite> SELECT * FROM sales;

product_id  amount      discount
----------  ----------  ----------
1           10          1
2           10000       NULL
```

You have a sales table that contains the products sold throughout the day. It holds the product ID, the price, and the total discounts that were included in the sale. There are two sales in the table: one is a $10 purchase with a $1 discount. Another is a $10,000 purchase with no discount, represented by a NULL, as in 'not applicable.' At the end of the day you want to tabulate net sales after discounts. You try the following:

```
sqlite> SELECT SUM(amount-discount) FROM sales;

SUM(amount-discount)
--------------------
9.0
```

Where did the $10,000 sale go? Well, 10,000 – NULL is NULL. NULLs are weeded out by the WHERE clause, and therefore contribute nothing to an aggregate, so it simply disappears. Your calculation is off by 99.9 percent. So, knowing better, you investigate and specifically look for the missing sale using the following SQL:

```
sqlite> SELECT SUM(amount) from sales WHERE amount-discount > 100.00;

NULL
```

What's the problem here? Well, when the database evaluates the WHERE clause for the record of interest, the expression becomes 10,000 – NULL > 100.00. Breaking this down, the predicate evaluates to NULL:

```
(10000 – NULL > 100.00)  ~TSRA  (NULL > 100.00)  ~TSRA  NULL
```

Again, NULLs don't pass through WHERE, so the $10,000 row seems invisible.

If you are going to stay in the black, you need to handle NULL better. If NULL is allowed in the discount column, then queries that use that column have to be NULL-aware:

```
sqlite> SELECT SUM(amount-COALESCE(discount,0)) FROM sales;
10009

sqlite> SELECT SUM(amount) from sales
        WHERE amount-COALESCE(discount,0) > 100.00;
10000.0
```

So, NULL can be useful, and can indeed have a very specific meaning, but using it without understanding the full implications can lead to unpleasant surprises.

4.5.14 Set Operations

Congratulations, you have learned the SELECT command. Not only have you learned how the command works, but you've covered a large part of relational algebra in the process. SELECT contains 12 of the 14 operations defined in relational algebra. Here is a list of all of these operations, along with the parts of SELECT that employ them:

- **Restriction**: Restriction takes a single relation and produces a row-wise subset of it. Restriction is performed by the WHERE, HAVING, DISTINCT, and LIMIT clauses.

- **Projection**: Projection produces a column-wise subset of its input relation. Projection is performed by the `SELECT` clause.
- **Cartesian product**: The Cartesian product takes two relations, *A* and *B*, and produces a relation whose rows are the composite of the two relations by combining every row of *A* with every row in *B*. `SELECT` performs the Cartesian product when multiple tables are listed in the `FROM` clause and no join condition is provided.
- **Union**: The union operation takes two relations, *A* and *B*, and creates a new relation containing all distinct rows from both *A* and *B*. *A* and *B* must have the same degree (number of columns). `SELECT` performs union operations in compound queries, which employ the `UNION` keyword.
- **Difference**: The difference operation takes two relations, *A* and *B*, and produces a relation whose rows consist of the rows in *A* that are not in *B*. `SELECT` performs difference in compound queries that employ the `EXCEPT` operator.
- **Rename**: The rename operation takes a single relation and assigns it a new name. Rename is used in the `FROM` clause.
- **Intersection**: The intersection operation takes two relations, *A* and *B*, and produces a relation whose rows are contained in both *A* and *B*. `SELECT` performs the intersection operation in compound queries that employ the `INTERSECT` operator.
- **Natural join**: A natural join takes two relations and performs an inner join on them by equating the commonly named columns as the join condition. `SELECT` performs a natural join with joins that use the `NATURAL JOIN` clause.
- **Generalized projection**: The generalized projection operation is an extension of the projection operation, which allows the use of arithmetic expressions, functions, and aggregates in the projection list. Generalized projection is used in the `SELECT` clause. Associated with aggregates is the concept of grouping, whereby rows with similar column values can be separated into individual groups. This is expressed in the `SELECT` command using the `GROUP BY` clause. Furthermore, groups can be filtered using the `HAVING` clause, which consists of a predicate similar to that of the `WHERE` clause. The predicate in `HAVING` is expressed in terms of aggregate values.
- **Left outer join**: The left outer join operation takes two relations, *A* and *B*, and returns the inner join of *A* and *B* along with the unmatched rows of *A*. The first relation defined in the `FROM` clause is the left

relation. The left join includes the unmatched rows of the left relation along with the matched columns in the result.

- **Right and full outer join**: The right outer join operation is similar to the left join, only it includes the unmatched rows of the right relation. The full outer join includes unmatched rows from both relations.

4.6 Modifying Data

Compared to the SELECT command, the statements used to modify data are quite easy to use and understand. There are three DML statements for modifying data – INSERT, UPDATE, and DELETE – and they do pretty much what their names imply.

4.6.1 Inserting Records

You insert records into a table using the INSERT command. INSERT works on a single table, and can insert one row at a time or many rows at once using a SELECT command. The general form of the INSERT command is as follows:

```
INSERT INTO table (column_list) VALUES (value_list);
```

The variable table specifies into which table – the target table – to insert. The variable column_list is a comma-separated list of column names, all of which must exist in the target table. The variable value_list is a comma-separated list of values that correspond to the names given in column_list. The order of values in value_list must correspond to the order of columns in column_list. For example, you'd use this command to insert a row into foods:

```
sqlite> INSERT INTO foods (name, type_id) VALUES ('Cinnamon Bobka', 1);
```

This statement inserts one row, specifying two column values. The first value in the value list – 'Cinnamon Bobka' – corresponds to the first column in the column list – name. Similarly, the value 1 corresponds to type_id, which is listed second. Notice that id was not mentioned. In this case, the database uses the default value. Since id is declared as INTEGER PRIMARY KEY, it is automatically generated and associated with the record (as explained in Section 4.7.1). The inserted record can be verified with a simple SELECT:

```
sqlite> SELECT * FROM foods WHERE name='Cinnamon Bobka';

id          type_id     name
----------  ----------  --------------
413         1           Cinnamon Bobka

sqlite> SELECT MAX(id) from foods;

MAX(id)
----------
413
```

Notice that the value 413 was automatically generated for id, which is the largest value in the column. Thus, SQLite provided a monotonically increasing value. You can confirm this with the built-in SQL function last_insert_rowid(), which returns the last automatically generated key value:

```
sqlite> SELECT last_insert_rowid();

last_insert_rowid()
-------------------
413
```

If you provide a value for every column of a table in INSERT, then the column list can be omitted. In this case, the database assumes that the order of values provided in the value list correspond to the order of columns as declared in the CREATE TABLE statement. For example:

```
sqlite> INSERT INTO foods VALUES(NULL, 1, 'Blueberry Bobka');
sqlite> SELECT * FROM foods WHERE name LIKE '%Bobka';

id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
```

Notice here the order of arguments. 'Blueberry Bobka' came after 1 in the value list. This is because of the way the table was declared. To view the table's schema, type .schema foods at the shell prompt:

```
sqlite> .schema foods

CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

The first column is `id`, followed by `type_id`, followed by `name`. This, therefore, is the order in which you must list values in `INSERT` statements on `foods`. Why did I use a `NULL` value for `id` in the preceding `INSERT` statement? Because SQLite knows that `id` in `foods` is an autoincrement column and specifying `NULL` is the equivalent of not providing a value. This triggers the automatic key generation. It's just a convenient trick. There is no deeper meaning or theoretical basis behind it. We look at the subtleties of autoincrement columns in Section 4.7.1.

Subqueries can be used in `INSERT` statements, both as components of the value list and as a complete replacement of the value list. Here's an example:

```
INSERT INTO foods
VALUES (NULL,
       (SELECT id FROM food_types WHERE name='Bakery'),
       'Blackberry Bobka');
SELECT * FROM foods WHERE name LIKE '%Bobka';

id          type_id     name
---------- ---------- ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobka
```

Here, rather than hard-coding the `type_id` value, I had SQLite look it up for me. Here's another example:

```
INSERT INTO foods
SELECT last_insert_rowid()+1, type_id, name FROM foods
WHERE name='Chocolate Bobka';
SELECT * FROM foods WHERE name LIKE '%Bobka';

id          type_id     name
---------- ---------- ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobks
416         1           Chocolate Bobka
```

This query completely replaces the value list with a `SELECT` statement. As long as the number of columns in the `SELECT` clause matches the number of columns in the table (or the number of columns in the columns list, if provided), INSERT will work just fine. Here I added another entry for 'Chocolate Bobka' and used the expression last_insert_rowid()+1 as the id value. I could just as easily have used NULL. In fact, I probably should have used NULL rather than last_insert_rowid(), as last_insert_rowid() returns 0 if you have not already inserted a row in the current session. I knew that this would work properly for this example, but it would not be a good idea to make this assumption in a program.

There is nothing to stop you from inserting multiple rows at a time using the SELECT form of INSERT. As long as the number of columns matches, INSERT inserts every row in the result. For example:

```
sqlite> CREATE TABLE foods2 (id int, type_id int, name text);
sqlite> INSERT INTO foods2 SELECT * FROM foods;
sqlite> SELECT COUNT(*) FROM foods2;

COUNT(*)
--------------------
418
```

This creates a new table foods2 and inserts into it all of the records from foods.

However, there is an easier way to do this. The CREATE TABLE statement has a special syntax for creating tables from SELECT statements. The previous example could have been performed in one step using this syntax:

```
sqlite> CREATE TABLE foods2 AS SELECT * from foods;
sqlite> SELECT COUNT(*) FROM list;

COUNT(*)
--------
418
```

CREATE TABLE does both steps in one fell swoop. This can be especially useful for creating temporary tables:

```
CREATE TEMP TABLE list AS
SELECT f.name Food, t.name Name,
       (SELECT COUNT(episode_id)
        FROM foods_episodes WHERE food_id=f.id) Episodes
FROM foods f, food_types t
WHERE f.type_id=t.id;
SELECT * FROM list;
```

```
Food                 Name       Episodes
-------------------- ---------- ----------
Bagels               Bakery     1
Bagels, raisin       Bakery     2
Bavarian Cream Pie   Bakery     1
Bear Claws           Bakery     3
Black and White cook Bakery     2
Bread (with nuts)    Bakery     1
Butterfingers        Bakery     1
Carrot Cake          Bakery     1
Chips Ahoy Cookies   Bakery     1
Chocolate Bobka      Bakery     1
```

When using this form of `CREATE TABLE`, be aware that any constraints defined in the source table are not created in the new table. Specifically, the autoincrement columns are not created in the new table; nor are indexes, `UNIQUE` constraints, and so forth.

It is also worth mentioning here that you have to be aware of `UNIQUE` constraints when inserting rows. If you add duplicate values into columns that are declared as `UNIQUE`, SQLite stops you in your tracks:

```
sqlite> SELECT MAX(id) from foods;

MAX(id)
-------
416

sqlite> INSERT INTO foods VALUES (416, 1, 'Chocolate Bobka');
SQL error: PRIMARY KEY must be unique
```

4.6.2 Updating Records

You update records in a table using the `UPDATE` command. The `UPDATE` command modifies one or more columns within one or more rows in a table. `UPDATE` has the general form:

```
UPDATE table SET update_list WHERE predicate;
```

The `update_list` is a list of one or more column assignments of the form `column_name=value`. The `WHERE` clause works exactly as in `SELECT`. Half of `UPDATE` is really a `SELECT` statement. The `WHERE` clause identifies rows to be modified using a predicate. Those rows then have the update list applied to them. For example:

```
UPDATE foods SET name='CHOCOLATE BOBKA'
WHERE name='Chocolate Bobka';
```

```
SELECT * FROM foods WHERE name LIKE 'CHOCOLATE%';

id     type_  name
----- ----- ------------------------------
10     1      CHOCOLATE BOBKA
11     1      Chocolate Eclairs
12     1      Chocolate Cream Pie
222    9      Chocolates, box of
223    9      Chocolate Chip Mint
224    9      Chocolate Covered Cherries
```

UPDATE is a very simple and direct command, and this is pretty much the extent of its use. As in INSERT, you must be aware of any UNIQUE constraints, as they stop UPDATE every bit as much as INSERT:

```
sqlite> UPDATE foods SET id=11 where name='CHOCOLATE BOBKA';
SQL error: PRIMARY KEY must be unique
```

This is true for any constraint, however.

4.6.3 Deleting Records

The DELETE command deletes rows from a table. DELETE has the general form:

```
DELETE FROM table WHERE predicate;
```

Syntactically, DELETE is a watered-down UPDATE statement. Remove the SET clause from UPDATE and you have DELETE. The WHERE clause works exactly as in SELECT, except that it identifies rows to be deleted. For example:

```
DELETE FROM foods WHERE name='CHOCOLATE BOBKA';
```

4.7 Data Integrity

Data integrity is concerned with defining and protecting relationships within and between tables. There are four general types: *domain integrity*, *entity integrity*, *referential integrity*, and *user-defined integrity*. Domain integrity involves controlling values within columns. Entity integrity involves controlling rows in tables. Referential integrity involves controlling rows between tables – specifically foreign key relationships. And user-defined integrity is a catchall for everything else.

Data integrity is implemented using *constraints*. A constraint is a control measure used to restrict the values that can be stored in a column or columns. Going by just the values in columns, the database can enforce all four types of integrity constraint. In SQLite, constraints also include support for *conflict resolution*. Conflict resolution is covered in detail in Section 4.8.2.

This section is a logical continuation of Section 4.4 at the beginning of this chapter, so the examples in this section use the `contacts` table defined there. It is listed again here for convenience:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone) );
```

As you know by now, constraints are part of a table's definition. They can be associated with a column definition or defined independently in the body of the table definition. Column-level constraints include `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `CHECK`, and `COLLATE`. Table-level constraints include `PRIMARY KEY`, `UNIQUE`, and `CHECK`. All of these constraints are covered in the following sections according to their integrity type.

The reason this material was not addressed earlier is because it requires familiarity with the `UPDATE`, `INSERT`, and `DELETE` commands. Just as these commands operate on data, constraints operate on them, making sure that they work within the guidelines defined in the tables they modify.

4.7.1 Entity Integrity

Entity integrity stems from the guaranteed access rule, as explained in Section 3.3.1. It requires that every field in every table be addressable. That is, every field in the database must be uniquely identifiable and capable of being located. In order for a field to be addressable, its corresponding row must also be addressable. And for that, the row must be unique in some way. This is the job of the primary key.

The primary key consists of least one column, or group of columns with a `UNIQUE` constraint. The `UNIQUE` constraint, as you will soon see, simply requires that every value in a column (or group of columns) be distinct. Therefore, the primary key ensures that each row is somehow distinct from all other rows in a table, ultimately ensuring that every field is also addressable. Entity integrity basically keeps data organized in a table. After all, what good is a field if you can't find it?

UNIQUE Constraints

Since primary keys are based on UNIQUE constraints, we'll start with them. A UNIQUE constraint simply requires that all values in a column or group of columns are distinct from one another, or unique. If you attempt to insert a duplicate value, or update a value to a value that already exists in the column, the database issues a constraint violation and aborts the operation. UNIQUE constraints can be defined at the column or table level. When defined at the table level, UNIQUE constraints can be applied across multiple columns. In this case, the combined value of the columns must be unique. In contacts, there is a unique constraint on name and phone together. See what happens if I attempt to insert another 'Jerry' record with a phone value 'UNKNOWN':

```
sqlite> INSERT INTO contacts (name,phone) VALUES ('Jerry','UNKNOWN');
SQL error: columns name, phone are not unique

sqlite> INSERT INTO contacts (name) VALUES ('Jerry');
SQL error: columns name, phone are not unique

sqlite> INSERT INTO contacts (name,phone) VALUES ('Jerry', '555-1212');
```

In the first case, I explicitly specified name and phone. This matched the values of the existing record and the UNIQUE constraint kicked in and did not let me do it. The third INSERT illustrates that the UNIQUE constraint applies to name and phone combined, not individually. It inserted another row with 'Jerry' as the value for name, which did not cause an error, because name by itself it not unique – only name and phone together.

NULL and UNIQUE

Pop quiz: Based on what you know about NULL and UNIQUE, how many NULL values can you put in a column that is declared UNIQUE? Answer: It depends on which database you are using. PostgreSQL and Oracle say as many as you want. Informix and Microsoft SQL Server say only one. DB2, SQL Anywhere, and Borland InterBase say none at all.

SQLite follows Oracle and PostgreSQL – you can put as many NULLs as you want in a unique column. This is another classic case of NULL befuddling everyone. On one side, you can argue that one NULL value is never equal to another NULL value because you don't have enough information about either to know if they are equal. On the other side, you don't really have enough information to prove that they are different either. The consensus in SQLite is to assume that they are all different, so you can have a `UNIQUE` column stuffed full of `NULL`s if you like.

`PRIMARY KEY` *Constraints*

In SQLite, a primary key column is always defined when you create a table, whether you define one or not. This column is a 64-bit integer value called `ROWID`. It has two aliases, `_ROWID_` and `OID`, which can be used to refer to it. Default values are automatically generated for it in monotonically increasing order.

SQLite provides an autoincrement feature for primary keys, should you want to define your own. If you define a column's type as `INTEGER PRIMARY KEY`, SQLite creates a `DEFAULT` constraint on it that provides a monotonically increasing integer value that is guaranteed to be unique in that column. In reality, however, this column is simply an alias for `ROWID`. They all refer to the same value. Since SQLite uses a 64-bit number for the primary key, the maximum value for this column is 9,223,372,036,854,775,807.

You may be wondering where the maximum value of a key value comes from. It is based on the limits of a 64-bit integer. A 64-bit integer has 64 bits of storage that can be used to represent a number. That is, 2^{64} (or 18,446,744,073,709,551,616) possible values can be represented in 64 bits. Think of this as a range of values. An *unsigned* integer defines this range starting at 0. The range is exclusively positive values and, therefore, needs no sign to designate it – hence 'unsigned.' The maximum value of an unsigned 64-bit integer is therefore 18,446,744,073,709,551,615 (one less because the range starts at 0, not 1). However, SQLite uses *signed* integers. A signed integer splits the range so that half of it is less than zero and half is greater than zero. The range of a signed 64-bit integer is −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (which is a range of 18,446,744,073,709,551,616 values). Key values in SQLite are signed 64-bit integers. Therefore, the maximum value of a key in SQLite is +9,223,372,036,854,775,807.

Even if you manage to reach this limit, SQLite simply starts searching for unique values that are not in the column for subsequent `INSERT`s. When you delete rows from the table, `ROWID`s may be recycled and reused on subsequent `INSERT`s. As a result, newly created `ROWID`s might not always be in strictly ascending order.

In the examples so far, I have managed to insert two records into `contacts`. Not once did I specify a value for `id`. As mentioned before, this is because `id` is declared as `INTEGER PRIMARY KEY`. Therefore, SQLite supplied an incremental integer value for each INSERT automatically, as you can see here:

```
sqlite> SELECT * FROM contacts;

id   name   phone
---  -----  --------
1    Jerry  UNKNOWN
2    Jerry  555-1212
```

Notice that the primary key is accessible from all the aforementioned aliases, in addition to id:

```
sqlite> SELECT ROWID, OID, _ROWID_, id, name, phone FROM CONTACTS;

id  id  id  id  name   phone
--  --  --  --  ----   -----
1   1   1   1   Jerry  UNKNOWN
2   2   2   2   Jerry  555-1212
```

If you include the keyword AUTOINCREMENT after INTEGER PRIMARY KEY, SQLite uses a different key generation algorithm for the column. This algorithm prevents ROWIDs from being recycled. It guarantees that only new (not recycled) ROWIDs are provided for every INSERT. When a table is created with a column containing the AUTOINCREMENT constraint, SQLite keeps track of that column's maximum ROWID in a system table called sqlite_sequence. It uses a value greater than that maximum on all subsequent INSERTs. If you ever reach the absolute maximum, then SQLite returns an SQLITE_FULL error on subsequent INSERTs. For example:

```
sqlite> CREATE TABLE maxed_out(id INTEGER PRIMARY KEY AUTOINCREMENT,
                                                        x text);
sqlite> INSERT INTO maxed_out VALUES (9223372036854775807, 'last one');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  -------------------
maxed_out   9223372036854775807

sqlite> INSERT INTO maxed_out VALUES (NULL, 'wont work');
SQL error: database is full
```

Here, I provided the primary key value. SQLite then stored this value as the maximum for maxed_out.id in sqlite_sequence. I supplied the very last (maximum) 64-bit value. In the next INSERT, I used the generated default value, which must be a monotonically increasing value. This wrapped around to 0 and SQLite issued an SQLITE_FULL error.

While SQLite tracks the maximum value for an AUTOINCREMENT column in the sqlite_sequence table, it does not prevent you from providing your own values for it in the INSERT command. The only requirement is that the value you provide must be unique within the column. For example:

```
sqlite> DROP TABLE maxed_out;
sqlite> CREATE TABLE maxed_out(id INTEGER PRIMARY KEY AUTOINCREMENT,
                                                            x text);
sqlite> INSERT INTO maxed_out values(10, 'works');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   10

sqlite> INSERT INTO maxed_out values(9, 'works');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   10

sqlite> INSERT INTO maxed_out VALUES (9, 'fails');
SQL error: PRIMARY KEY must be unique

sqlite> INSERT INTO maxed_out VALUES (NULL, 'should be 11');
sqlite> SELECT * FROM maxed_out;

id          x
----------  ------------
9           works
10          works
11          should be 11

sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   11
```

Here I dropped and re-created the maxed_out table, and inserted a record with an explicitly defined ROWID of 10. Then I inserted a record with a ROWID less than 10, which worked. I tried it again with the same value and it failed, due to the UNIQUE constraint. Finally, I inserted another record using the default key value and SQLite provided the next monotonically increasing value – 10+1.

In summary, AUTOINCREMENT prevents SQLite from recycling primary key values (ROWIDs) and stops when the ROWID reaches the maximum (signed) 64-bit integer value. This feature was added for specific applications that required this behavior. Unless you have such a specific need in your application, it is perhaps best to use `INTEGER PRIMARY KEY` for autoincrement columns.

Like `UNIQUE` constraints, `PRIMARY KEY` constraints can be defined over multiple columns. You don't have to use an integer value for your primary key. If you choose to use another value, SQLite still maintains the `ROWID` column internally, but it also places a `UNIQUE` constraint on your declared primary key. For example:

```
sqlite> CREATE TABLE pkey(x text, y text, PRIMARY KEY(x,y));
sqlite> INSERT INTO pkey VALUES ('x','y');
sqlite> INSERT INTO pkey VALUES ('x','x');
sqlite> SELECT ROWID, x, y FROM pkey;

rowid       x           y
----------  ----------  ----------
1           x           y
2           x           x

sqlite> INSERT INTO pkey VALUES ('x','x');
SQL error: columns x, y are not unique
```

The primary key here is technically just a `UNIQUE` constraint across two columns, nothing more. As stated before, the concept of a primary key is more or less just lip service to the relational model – SQLite always provides one whether you do or not. If you do define a primary key, it is in reality just another `UNIQUE` constraint, nothing more.

4.7.2 Domain Integrity

The simplest definition of domain integrity is the conformance of a column's values to its assigned domain. That is, every value in a column should exist within that column's defined domain. However, the term *domain* is a little vague. Domains are often compared to types in programming languages, such as strings or floats. And while that is not a bad analogy, domain integrity is actually much broader than that.

Domain constraints make it possible for you to start with a simple type – such as an integer – and add additional constraints to create a more restricted set of acceptable values for a column. For example, you can create a column with an integer type and add the constraint that only three such values are allowed: {−1, 0. 1}. In this case, you have modified the range of acceptable values (from the domain of all integers to just three integers) but not the data type itself. You are dealing with two things: a type and a range.

Consider another example: the `name` column in the `contacts` table. It is declared as follows:

```
name TEXT NOT NULL COLLATE NOCASE
```

The domain TEXT defines the type and initial range of acceptable values. Everything following it serves to restrict and qualify that range further. The name column is then the domain of all TEXT values that do not include NULL values where uppercase letters and lowercase letters have equal value. It is still TEXT, and operates as TEXT, but its range of acceptable values is restricted from that of TEXT.

You might say that a column's domain is not the same thing as its type. Rather, its domain is a combination of two things: a type and a range. The column's type defines the representation and operators of its values – how they are stored and how you can operate on them – sort, search, add, subtract, and so forth. A column's range is the set of acceptable values you can store in it, which is not necessarily the same as its declared type. The type's range represents a maximum range of values. The column's range – as you have seen – can be restricted through constraints. So for all practical purposes, you can think of a column's domain as a type with constraints tacked on.

Similarly, there are essentially two components to domain integrity: type checking and range checking. While SQLite supports many of the standard domain constraints for range checking (NOT NULL, CHECK, etc.), its approach to type checking is where things diverge from other databases. In fact, SQLite's approach to types and type checking is one of its most controversial, misunderstood, and disputed features.

But before we get into how SQLite handles types, let's cover the easy stuff first: default values, NOT NULL constraints, CHECK constraints, and collations.

Default Values

The DEFAULT keyword provides a default value for a column if one is not provided in an INSERT command. DEFAULT is not a constraint, because it doesn't enforce anything. It simply steps in when needed. However, it does fall within domain integrity because it provides a policy for handling NULL values in a column. If a column doesn't have a default value and you don't provide a value for it in an INSERT statement, then SQLite inserts NULL for that column. For example, contacts.phone has a default value of 'UNKNOWN'. With this in mind, consider the following example:

```
sqlite> INSERT INTO contacts (name) VALUES ('Jerry');
sqlite> SELECT * FROM contacts;

id          name        phone
----------  ----------  ----------
1           Jerry       UNKNOWN
```

The INSERT command inserted a row, specifying a value for name but not phone. As you can see from the resulting row, the default value for phone kicked in and provided the string 'UNKNOWN'. If phone did not have a default value, then the value for phone in this row would have been NULL instead.

DEFAULT also accepts three predefined ANSI/ISO reserved words for generating default dates and times. CURRENT_TIME generates the current local time in ANSI/ISO time format (HH:MM:SS). CURRENT_DATE generates the current date (in YYYY-MM-DD format). CURRENT_TIMESTAMP produces a combination of these two (in YYYY-MM-DD HH:MM:SS format). For example:

```
CREATE TABLE times ( id int,
  date NOT NULL DEFAULT CURRENT_DATE,
  time NOT NULL DEFAULT CURRENT_TIME,
  timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP );
INSERT INTO times(id) values (1);
INSERT INTO times(id) values (2);
SELECT * FROM times;

id          date        time        timestamp
----------  ----------  ----------  -------------------
1           2009-10-13  17:08:18    2009-10-13 17:08:18
2           2009-10-13  17:08:18    2009-10-13 17:08:18
```

These defaults come in handy for tables that need to log or timestamp events.

NOT NULL Constraints

If you are one of those people who are not fond of NULL, then the NOT NULL constraint is for you. NOT NULL ensures that values in a column may never be NULL. INSERT commands may not add NULL in the column and UPDATE commands may not change existing values to NULL. Often, you will see NOT NULL raise its ugly head in INSERT statements. Specifically, a NOT NULL constraint without a DEFAULT constraint prevents any unspecified values from being used in the INSERT (because the default values provided in this case are NULL). In the contacts table, the NOT NULL constraint on name requires that an INSERT command always provide a value for that column. For example:

```
sqlite> INSERT INTO contacts (phone) VALUES ('555-1212');
SQL error: contacts.name may not be NULL
```

This INSERT command specified a phone value but not a name. The NOT NULL constraint on name kicked in and forbade the operation.

The way to deal with NOT NULL is to include a DEFAULT constraint on the column. This is the case for phone, which has a NOT NULL constraint and a DEFAULT constraint as well. If an INSERT command does not specify a value for phone, the DEFAULT constraint steps in and provides the value 'UNKNOWN', thus satisfying the NOT NULL constraint. To this end, people often use DEFAULT constraints in conjunction with NOT NULL constraints so that INSERT commands can safely use default values while at the same time keeping NULL out of the column.

CHECK Constraints

CHECK constraints allow you to define expressions to test values whenever they are inserted into or updated within a column. If the values do not meet the criteria set forth in the expression, the database issues a constraint violation. Thus, it allows you to define additional data integrity checks beyond UNIQUE and NOT NULL to suit your specific application. An example of a CHECK constraint might be to ensure that the value of a phone number field is at least seven characters long. To do this, you can either add the constraint to the column definition of phone or as a standalone constraint in the table definition as follows:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone),
                        CHECK(LENGTH(phone)>=7) );
```

Here, any attempt to insert or update a value for phone with fewer than seven characters results in a constraint violation. You can use any expression in a CHECK constraint that you can use in a WHERE clause. For example, say you have the table foo defined as follows:

```
CREATE TABLE foo( x integer,
                  y integer CHECK(y>x),
                  z integer CHECK(z>ABS(y)) );
```

In this table, every value of z must always be greater than y, which in turn must be greater than x. To illustrate this, try the following:

```
INSERT into foo values (-2, -1, 2);
INSERT into foo values (-2, -1, 1);
SQL error: constraint failed

UPDATE foo SET y=-3 WHERE x=-3;
SQL error: constraint failed
```

The CHECK constraints for all columns are evaluated before any modification is made. For the modification to succeed, the expressions for all constraints must evaluate to true.

Functionally, triggers can be used just as effectively as check constraints for data integrity. In fact, triggers can do much more. If you find that you can't quite express what you need in a CHECK constraint, then triggers are a good alternative. Triggers are covered in Section 4.9.3.

Collations

Collation is related to domain integrity in that it defines what constitutes unique text values. Collation specifically refers to how text values are compared. Different collations employ different comparison methods. For example, one collation might be case insensitive, so the strings 'JujyFruit' and 'JUJYFRUIT' are considered to be the same. Another collation might be case sensitive, in which case the strings would be considered different.

SQLite has three built-in collations. The default is BINARY, which compares text values byte by byte using a specific C function called memcmp(). This happens to work nicely for many Western languages such as English. NOCASE is basically a case-insensitive collation for the 26 ASCII characters used in English. Finally there is REVERSE, which is the reverse of the BINARY collation. REVERSE is more for testing (and perhaps illustration) than anything else.

The SQLite C API provides a way to create custom collations. This feature allows developers to support languages or locales that are not well served by the BINARY collation. See Chapter 7 for more information.

The COLLATE keyword defines the collation for a column. For example, the collation for contacts.name is defined as NOCASE, which means that it is case insensitive. Thus, if I try to insert a row with a name value of 'JERRY' and a phone value of '555-1212' it should fail:

```
sqlite> INSERT INTO contacts (name,phone) VALUES ('JERRY','555-1212');
SQL error: columns name, phone are not unique
```

According to name's collation, 'JERRY' is the same as 'Jerry', and there is already a row with that value. Therefore, a new row with name='JERRY' would be a duplicate value. By default, collation in SQLite is case sensitive. The previous example would have worked had I not defined NOCASE on name.

4.7.3 Storage Classes

As mentioned earlier, SQLite does not work like other databases when it comes to handling data types. It differs in the types it supports and in how they are stored, compared, enforced, and assigned. The following sections explore SQLite's radically different but surprisingly flexible approach to data types and its relation to domain integrity.

With respect to types, SQLite's domain integrity is better described as *domain affinity*. In SQLite, it is referred to as *type affinity*. To understand type affinity, however, you must first understand *storage classes* and *manifest typing*.

Internally, SQLite has five primitive data types, which are referred to as *storage classes*. The term storage class refers to the format in which a value is stored on disk. Regardless, it is still synonymous with 'type' or 'data type.' The five storage classes are described in Table 4.7.

Table 4.7 SQLite storage classes

Name	**Description**
`INTEGER`	Values are whole numbers (positive and negative). They can vary in size: 1, 2, 3, 4, 6, or 8 bytes. The maximum integer range (8 bytes) is {−9223372036854775808, −1, 0, 1, +9223372036854775807}. SQLite automatically handles the integer sizes based on the numeric value.
`REAL`	Values are real numbers with decimal values. SQLite uses 8-byte floats to store real numbers.
`TEXT`	Values are character data. SQLite supports various character encodings, which include UTF-8 and UTF-16 (big and little endian). The maximum string value in SQLite is unlimited.
`BLOB`	Binary large object (`BLOB`) data is any kind of data. The maximum size for a `BLOB` in SQLite is unlimited.
`NULL`	Values represent missing information. SQLite has full support for `NULL` values.

SQLite infers a value's type from its representation. The following inference rules are used to do this:

- A value specified as a literal in an SQL statement is assigned the class `TEXT` if it is enclosed by single or double quotes.
- A value is assigned the class `INTEGER` if the literal is specified as an unquoted number with no decimal point or exponent.
- A value is assigned the class `REAL` if the literal is an unquoted number with a decimal point or exponent.
- A value is assigned the class `NULL` if its value is `NULL`.
- A value is assigned the class `BLOB` if it is of the format `X'ABCD'`, where `ABCD` are hexadecimal numbers. The `X` prefix and values can be either uppercase or lowercase.

The `typeof()` SQL function returns the storage class of a value based on its representation. Using this function, the following SQL command illustrates type inference in action:

```
sqlite> select typeof(3.14), typeof('3.14'),
        typeof(314), typeof(x'3142'), typeof(NULL);

typeof(3.14)  typeof('3.14')  typeof(314) typeof(x'3142')  typeof(NULL)
------------  --------------  ----------- ---------------  ------------
real          text            integer     blob             null
```

Here are all of the five internal storage classes invoked by specific representations of data. The value 3.14 looks like a `REAL` and therefore is inferred to be a `REAL`. The value `'3.14'` looks like `TEXT` and therefore is inferred to be a `TEXT`, and so on.

A single column in SQLite may contain values of *different storage classes*. Consider the following example:

```
sqlite> DROP TABLE domain;
sqlite> CREATE TABLE domain(x);
sqlite> INSERT INTO domain VALUES (3.142);
sqlite> INSERT INTO domain VALUES ('3.142');
sqlite> INSERT INTO domain VALUES (3142);
sqlite> INSERT INTO domain VALUES (x'3142');
sqlite> INSERT INTO domain VALUES (NULL);
sqlite> SELECT ROWID, x, typeof(x) FROM domain;

rowid       x           typeof(x)
----------  ----------  ----------
1           3.142       real
2           3.142       text
3           3142        integer
4           1B          blob
5           NULL        null
```

This raises a few questions. How are the values in a column sorted or compared? How do you sort a column with `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL` values? How do you compare an `INTEGER` with a `BLOB`? Which is greater? Can they ever be equal?

As it turns out, values in a column with different storages classes can be sorted. And they can be sorted because they can be compared. There are well-defined rules for doing so. Storage classes are sorted by using their respective *class values*, which are defined as follows:

1. The `NULL` storage class has the lowest class value. A value with a `NULL` storage class is considered less than any other value (including another value with storage class `NULL`). Within `NULL` values, there is no specific sort order.

2. INTEGER or REAL storage classes have higher values than NULL and share equal class value. INTEGER and REAL values are compared numerically.
3. The TEXT storage class has a higher value than INTEGER or REAL. A value with an INTEGER or a REAL storage class is always less than a value with a TEXT storage class no matter its value. When two TEXT values are compared, the comparison is determined by the collation defined for the values.
4. The BLOB storage class has the highest value. Any value that is not of class BLOB is always less than the value of class BLOB. BLOB values are compared using the C function memcmp().

So when SQLite sorts a column, it first groups values according to storage class – first NULLs, then INTEGERs and REALs, TEXTs, and finally BLOBs. It then sorts the values within each group. NULLs are not ordered at all, INTEGERs and REALs are compared numerically, TEXTs are arranged by the appropriate collation, and BLOBs are sorted using memcmp(). Figure 4.20 illustrates a column sorted in ascending order.

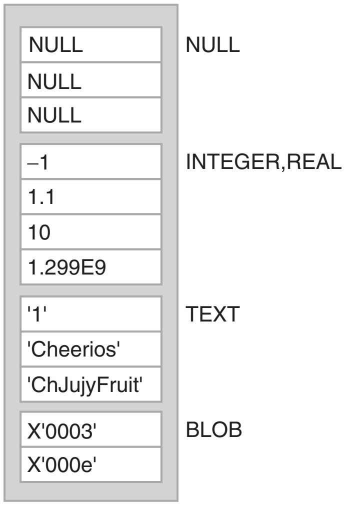


Figure 4.20 Storage class sort order

The following SQL illustrates the differences between storage class values:

```
sqlite> SELECT 3 < 3.142, 3.142 < '3.142', '3.142' < x'3000',
               x'3000' < x'3001';

3 < 3.142  3.142 < '3.142' '3.142' < x'3000' x'3000' < x'3001'
---------  --------------- ----------------- ------------------
1          1               1                 1
```

INTEGERs and REALs are compared numerically and are both less than TEXTs and TEXTs are less than BLOBs.

4.7.4 Manifest Typing

SQLite uses manifest typing. If you do a little research, you will find that the term *manifest typing* is subject to multiple interpretations. In programming languages, manifest typing refers to how the type of a variable or value is defined or determined. There are two main interpretations:

- *Manifest typing means that a variable's type must be explicitly declared in the code.* By this definition, languages such as C/C++, Pascal, and Java would be said to use manifest typing. Dynamically typed languages, such as Perl, Python, and Ruby, are the direct opposite as they do not require that a variable's type be declared.
- *Manifest typing means that variables don't have types at all. Rather, only values have types.* This seems to be in line with dynamically typed languages. Basically, a variable can hold any value and the type of that variable at any point in time is determined by its value at that moment. Thus if you set variable x=1, then x at that moment is of type INTEGER. If you set x='JujyFruit', it is then of type TEXT. That is, if it looks like an INTEGER and acts like an INTEGER, it is an INTEGER.

For the sake of brevity, I refer to the first interpretation as MT 1 and the second as MT 2. At first glance, it may not be readily apparent as to which interpretation best fits SQLite. For example, consider the following table:

```
CREATE TABLE foo( x integer,
                  y text,
                  z real );
```

Say we now insert a record into this table as follows:

```
INSERT INTO foo VALUES ('1', '1', '1');
```

When SQLite creates the record, what type is stored internally for x, y, and z? The answer: INTEGER, TEXT, and REAL. Thus it seems that SQLite uses MT 1: variables have declared types. But wait a second; column types in SQLite are optional, so we could have just as easily defined foo as follows:

```
CREATE TABLE foo(x, y, z);
```

Now let's do the same insert:

```
INSERT INTO foo VALUES ('1', '1', '1');
```

What types are `x`, `y`, and `z` now? The answer: `TEXT`, `TEXT`, and `TEXT`. Well, maybe SQLite is just setting columns to `TEXT` by default. If you think that, then consider the following `INSERT` statement on the same table:

```
INSERT INTO foo VALUES (1, 1.0, x'10');
```

What are `x`, `y`, and `z` in this row? `INTEGER`, `REAL`, and `BLOB`. This looks like MT 2, where the value itself determines its type.

So which one is it? The short answer: neither and both. The long answer is a little more involved. With respect to MT 1, SQLite lets you declare columns to have types if you want to. This looks and feels like what other databases do. But you don't *have to*, thereby violating this interpretation as well. This is because in all situations SQLite can take any value and infer a type from it. It doesn't need the type declared in the column to help it out. With respect to MT 2, SQLite allows the type of the value to 'influence' (maybe not completely determine) the type that gets stored in the column. But you can still declare the column with a type and that type will exert some influence, thereby violating this interpretation as well – that types come from values only. What we really have here is MT 3 – the SQLite interpretation. It borrows from both MT 1 and MT 2.

Interestingly enough, manifest typing does not address the whole issue with respect to types. It seems to be concerned only with declaring and resolving types. What about type checking? That is, if you declare a column to be type `INTEGER`, what exactly does that mean?

First let's consider what most other relational databases do. They enforce *strict type checking* as part of standard domain integrity: you declare a column's type and only values of that type can go in it, end of story. You can use additional domain constraints if you want, but under no conditions can you ever insert values of other types. Consider the following example with Oracle:

```
SQL> create table domain(x int, y varchar(2));
Table created.

SQL> INSERT INTO domain VALUES ('pi', 3.14);
INSERT INTO domain VALUES ('pi', 3.14)
                          *
ERROR at line 1:
ORA-01722: invalid number
```

The value 'pi' is not an integer value. And column x was declared to be of type int. I don't even get to hear about the error in column y because the whole INSERT is aborted due to the integrity violation on x. When I try this in SQLite, there's no problem:

```
sqlite> CREATE TABLE domain (x int, y varchar(2));
sqlite> INSERT INTO domain VALUES ('pi', 3.14);
sqlite> SELECT * FROM domain;

x     y
----  -----
pi    3.14
```

I said one thing and did another, and SQLite didn't stop me. *SQLite's domain integrity does not include strict type checking*. So what is going on? Does a column's declared type count for anything? Yes. Then how is it used? It is all done with something called *type affinity*.

In short, SQLite's manifest typing states that columns can have types and types can be inferred from values. Type affinity addresses how these two things relate to one another. Type affinity is a delicate balancing act that sits between strict typing and dynamic typing.

4.7.5 Type Affinity

In SQLite, columns don't have types or domains. While a column can have a declared type, internally it only has a type affinity. Declared type and type affinity are two different things. Type affinity determines the storage class SQLite uses to store values within a column. The actual storage class a column uses to store a given value is a function of both the value's storage class and the column's affinity. Before getting into how this is done, however, let's first talk about how a column gets its affinity.

Column Types and Affinities

To begin with, every column has an affinity. There are four different kinds: NUMERIC, INTEGER, TEXT, and NONE. A column's affinity is determined directly from its declared type (or lack thereof). Therefore, when you declare a column in a table, the type you choose to declare it as ultimately determines that column's affinity. SQLite assigns a column's affinity according to the following rules:

- By default, a column's affinity is NUMERIC. That is, if a column is not INTEGER, TEXT, or NONE, then it is automatically assigned NUMERIC affinity.

- If a column's declared type contains the string `'INT'` (in uppercase or lowercase), then the column is assigned `INTEGER` affinity.
- If a column's declared type contains any of the strings `'CHAR'`, `'CLOB'`, or `'TEXT'` (in uppercase or lowercase), then that column is assigned `TEXT` affinity. Notice that `'VARCHAR'` contains the string `'CHAR'` and thus confers `TEXT` affinity.
- If a column's declared type contains the string `'BLOB'` (in uppercase or lowercase), *or if it has no declared type*, then it is assigned `NONE` affinity.

Pay attention to defaults. If you don't declare a column's type, then its affinity is `NONE`, in which case all values are stored using their given storage class (or inferred from their representation). If you are not sure what you want to put in a column or you want to leave it open to change, this is the best affinity to use.

However, be careful of the scenario where you declare a type that does not match any of the rules for `NONE`, `TEXT`, or `INTEGER`. While you might intuitively think the default should be `NONE`, it is actually `NUMERIC`. For example, if you declare a column of type `JUJIFRUIT`, it does *not* have affinity `NONE` just because SQLite doesn't recognize it. Rather, it has affinity `NUMERIC`. (Interestingly, the scenario also happens when you declare a column's type to be `numeric` for the same reason.) Rather than using an unrecognized type that ends up as numeric, you may prefer to leave the column's declared type out altogether, which ensures it has affinity `NONE`.

Affinities and Storage

Each affinity influences how values are stored in its associated column. The rules governing storage are as follows:

- A `NUMERIC` column may contain all five storage classes. A `NUMERIC` column has a bias toward numeric storage classes (`INTEGER` and `REAL`). When a `TEXT` value is inserted into a `NUMERIC` column, it attempts to convert it to an `INTEGER` storage class. If this fails, it attempts to convert it to a `REAL` storage class. Failing that, it stores the value using the `TEXT` storage class.
- An `INTEGER` column tries to be as much like a `NUMERIC` column as it can. An `INTEGER` column stores a `REAL` value as `REAL`. *However*, if a `REAL` value has *no fractional component*, then it is stored using an `INTEGER` storage class. `INTEGER` columns try to store a `TEXT` value as a `REAL` if possible. If that fails, they try to store it as `INTEGER`. Failing that, `TEXT` values are stored as `TEXT`.
- A `TEXT` column converts all `INTEGER` and `REAL` values to `TEXT`.

- A NONE column does not attempt to convert any values. All values are stored using their given storage class.
- No column ever tries to convert NULL or BLOB values – regardless of affinity. NULL and BLOB values are stored as such in every column.

These rules may initially appear somewhat complex, but their overall design goal is simple: to make it possible for SQLite to mimic other relational databases if you need it to. That is, if you treat columns like a traditional database, type affinity rules store values in the way you expect. If you declare a column of type INTEGER and put integers into it, they are stored as INTEGER. If you declare a column to be of type TEXT, CHAR, or VARCHAR and put integers into it, they are stored as TEXT. However, if you don't follow these conventions, SQLite still finds a way to store the values.

Affinities in Action

Let's look at a few examples to get the hang of how affinity works. Consider the following:

```
sqlite> CREATE TABLE domain(i int, n numeric, t text, b blob);
sqlite> INSERT INTO domain VALUES (3.142,3.142,3.142,3.142);
sqlite> INSERT INTO domain VALUES ('3.142','3.142','3.142','3.142');
sqlite> INSERT INTO domain VALUES (3142,3142,3142,3142);
sqlite> INSERT INTO domain VALUES (x'3142',x'3142',x'3142',x'3142');
sqlite> INSERT INTO domain VALUES (null,null,null,null);
sqlite> SELECT ROWID,typeof(i),typeof(n),typeof(t),typeof(b) FROM domain;

rowid       typeof(i)   typeof(n)   typeof(t)   typeof(b)
----------  ----------  ----------  ----------  ----------
1           real        real        text        real
2           real        real        text        text
3           integer     integer     text        integer
4           blob        blob        blob        blob
5           null        null        null        null
```

The first INSERT inserts a REAL value. You can see this both by the format in the INSERT statement and by the resulting type shown in the typeof(b) column returned in the SELECT statement. Remember that BLOB columns have storage class NONE, which does not attempt to convert the storage class of the input value, so column b uses the storage class that was defined in the INSERT statement. Column i keeps the NUMERIC storage class, because it tries to be NUMERIC when it can. Column n doesn't have to convert anything. Column t converts it to TEXT. Column b stores it exactly as given in the context. You can see how the conversion rules were applied in each subsequent INSERT.

The following SQL illustrates storage class sort order and interclass comparison (which are governed by the same set of rules):

```
sqlite> SELECT ROWID, b, typeof(b) FROM domain ORDER BY b;

rowid  b       typeof(b)
-----  ------  ---------
5      NULL    null
1      3.142   real
3      3142    integer
2      3.142   text
4      1B      blob
```

Here you see that NULLs sort first, followed by INTEGERs and REALs, followed by TEXTs, then BLOBs. The following SQL shows how these values compare with the integer 1000:

```
sqlite> SELECT ROWID, b, typeof(b), b<1000 FROM domain ORDER BY b;

rowid  b      typeof(b)  b<1000
-----  -----  --------   ----------
5      NULL   null       NULL
1      3.142  real       1
3      3142   integer    1
2      3.142  text       0
4      1B     blob       0
```

The INTEGER and REAL values in b are less than 1000 because they are compared numerically, while TEXT and BLOB are greater than 1000 because they are in a higher storage class.

The primary difference between type affinity and strict typing is that type affinity never issues a constraint violation for incompatible data types. SQLite always finds a data type suitable for putting any value into any column. The only question is what type it uses to do so. The only role of a column's declared type in SQLite is to determine its affinity. Ultimately, it is the column's affinity that has a bearing on how values are stored inside it. However, SQLite does provide facilities for ensuring that a column may only accept a given type or range of types. You do this using CHECK constraints, explained in the 'Makeshift Strict Typing' sidebar later in this section.

Storage Classes and Type Conversions

Another thing to note about storage classes is that they can sometimes influence how values are compared. Specifically, SQLite sometimes converts values between numeric storage classes (INTEGER and REAL) and TEXT before comparing them. For binary comparisons, it uses the following rules:

- When a column value is compared to the result of an expression, the affinity of the column is applied to the result of the expression before the comparison takes place.
- When two column values are compared, if one column has INTEGER or NUMERIC affinity and the other doesn't, then NUMERIC affinity is applied to TEXT values in the non-NUMERIC column.
- When two expressions are compared, SQLite does not make any conversions. The results are compared as is. If the expressions are of like storage class, then the comparison function associated with that storage class is used to compare values. Otherwise, they are compared on the basis of their storage class.

Note that the term *expression* here refers to any scalar expression or literal *other than a column value*. To illustrate the first rule, consider the following:

```
sqlite> select ROWID, i, typeof(i), i>'2.9' from domain ORDER BY i;

rowid  i      typeof(i  i>'2.9'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.142  real      1
3      3142   integer   1
2      3.142  real      1
4      1B     blob      1
```

The expression '2.9', although TEXT, is converted to INTEGER before the comparison. So the column interprets the value in light of what it is. What if '2.9' were a non-numeric string? Then SQLite falls back to comparing storage class, in which INTEGER and NUMERIC types are always less than TEXT:

```
sqlite> SELECT ROWID,i,typeof(i),i>'text' FROM domain ORDER BY i;

rowid  i      typeof(i  i>'text'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.14   real      0
3      314    integer   0
2      3.14   real      0
4      1B     blob      1
```

The second rule simply states that, when comparing a numeric and non-numeric column, where possible, SQLite tries to convert the non-numeric column to numeric format:

```
sqlite> CREATE TABLE rule2(a int, b text);
sqlite> insert into rule2 values(2,'1');
sqlite> insert into rule2 values(2,'text');
sqlite> select a, typeof(a), b, typeof(b), a>b from rule2;

a           typeof(a)   b           typeof(b)   a>b
----------  ----------  ----------  ----------  ----------
2           integer     1           text        1
2           integer     text        text        0
```

Column `a` is `INTEGER` and `b` is `TEXT`. When evaluating the expression `a>b`, SQLite tries to coerce `b` to `INTEGER` where it can. In the first row, `b` is `'1'`, which can be coerced to `INTEGER`. SQLite makes the conversion and compares integers. In the second row, `b` is `'text'` and can't be converted. SQLite then compares storage classes `INTEGER` and `TEXT`.

The third rule just reiterates that storage classes established by context are compared at face value. If what looks like a `TEXT` type is compared with what looks like an `INTEGER` type, then `TEXT` is greater.

Additionally, you can manually convert the storage type of a column or expression using the `CAST` function. Consider the following example:

```
sqlite> SELECT typeof(3.14), typeof(CAST(3.14 as TEXT));

typeof(3.14)  typeof(CAST(3.14 as TEXT))
------------  --------------------------
real          text
```

Makeshift Strict Typing

If you need something stronger than type affinity for domain integrity, then `CHECK` constraints can help. You can implement something akin to strict typing directly using a single built-in function and a `CHECK` constraint. As mentioned earlier, SQLite has a function which returns the inferred storage class of a value – `typeof()`. You can use `typeof()` in any relational expression to test for a value's type. For example:

```
sqlite> select typeof(3.14) = 'text';
0

sqlite> select typeof(3.14) = 'integer';
0
```

```
sqlite> select typeof(3.14) = 'real';
1

sqlite> select typeof(3) = 'integer';
1

sqlite> select typeof('3') = 'text';
1
```

Therefore, you can use this function to implement a `CHECK` constraint that limits the acceptable types allowed in a column:

```
sqlite> create table domain (x integer CHECK(typeof(x)='integer'));
sqlite> INSERT INTO domain VALUES('1');
SQL error: constraint failed

sqlite> INSERT INTO domain VALUES(1.1);
SQL error: constraint failed

sqlite> INSERT INTO domain VALUES(1);
sqlite> select x, typeof(x) from domain;

x   typeof(x)
--  ----------
1   integer

sqlite> update domain set x=1.1;
SQL error: constraint failed
```

The only catch here is that you are limited to checking for SQLite's native storage classes (or what can be implemented using other built-in SQL functions). However, if you are a programmer and use the SQLite C API directly, you can implement more elaborate functions for type checking, which can be called from within `CHECK` constraints.

4.8 Transactions

A transaction defines the boundaries around a group of SQL commands such that they either all successfully execute together or fail. A classic example of the rationale behind transactions is a money transfer. Say a bank program is transferring money from one account to another. The money transfer program can do this in one of two ways: insert (credit) the funds into account 2 then delete (debit) it from account 1; or delete it from account 1 and insert it into account 2. Either way, the transfer is a two-step process: an `INSERT` followed by a `DELETE` or a `DELETE` followed by an `INSERT`.

Assume the program is in the process of making a transfer. The first SQL command completes successfully and then the database server suddenly crashes or the power goes out. Whatever the reason, the second operation does not complete. Now the money either exists in both accounts (the first scenario) or has been completely lost (the second scenario). Either way, someone's not going to be happy. And the database is in an inconsistent state.

The point here is that these two operations must either happen together or not at all. That is what transactions are for. Now let's replay the example with transactions. In the new scenario, the program first starts a transaction in the database, completes the first SQL operation, and then the power goes out. When it comes back on and the database comes back up, it sees an incomplete transaction. It undoes the changes of the first SQL operation, which brings it back into a consistent state – where it started before the transfer.

4.8.1 Transaction Scope

Transactions are issued with three commands: `BEGIN`, `COMMIT`, and `ROLLBACK`. `BEGIN` starts a transaction. Every operation following a `BEGIN` can potentially be undone and is undone if a `COMMIT` is not issued before the session terminates. The `COMMIT` command commits the work performed by all operations since the start of the transaction. Similarly, the `ROLLBACK` command undoes all of the work performed by operations since the start of the transaction. A transaction is a scope in which operations are performed together and committed or completely reversed. Consider the following example:

```
sqlite> BEGIN;
sqlite> DELETE FROM foods;
sqlite> ROLLBACK;
sqlite> SELECT COUNT(*) FROM foods;

COUNT(*)
--------
418
```

I started a transaction, deleted all the rows in `foods`, changed my mind, and reversed those changes by issuing a `ROLLBACK`. The `SELECT` statement shows that nothing has changed.

By default, every SQL command in SQLite is run in its own transaction. That is, if you do not define a transaction scope with `BEGIN` ... `COMMIT`/ `ROLLBACK`, SQLite implicitly wraps every individual SQL command with a `BEGIN` ... `COMMIT`/`ROLLBACK` and every command that completes successfully is committed. Likewise, every command that encounters an error is rolled back. This mode of operation (implicit transactions) is referred to as *autocommit mode*: SQLite automatically runs each command in its own transaction and, if the command does not fail, its changes are automatically committed.

4.8.2 Conflict Resolution

As you've seen in previous examples, constraint violations cause the command that committed the violation to terminate. What exactly happens when a command terminates in the middle of making a bunch of changes to the database? In most databases, all of the changes are undone. That is the way the database is programmed to handle a constraint violation, end of story.

SQLite, however, has a unique feature that allows you to specify different ways to handle (or recover from) constraint violations. It is called *conflict resolution*. Take, for example, the following `UPDATE`:

```
sqlite> UPDATE foods SET id=800-id;
SQL error: PRIMARY KEY must be unique
```

This results in a `UNIQUE` constraint violation because once the `UPDATE` statement reaches the 388th record, it attempts to update its `id` value to $800 - 388 = 412$. But a row with an `id` of 412 already exists, so it aborts the command. But SQLite has already updated the first 387 rows before it reached this constraint violation. What happens to them? The default behavior is to terminate the command and reverse all of the changes it made, while leaving the transaction intact.

But what if you wanted these 387 changes to stick despite the constraint violation? Well, believe it or not, you can have it that way too, if you want. You just need to use the appropriate conflict resolution mode. There are five possible resolutions, or policies, that can be applied to address a conflict (constraint violation): `REPLACE`, `IGNORE`, `FAIL`, `ABORT`, and `ROLLBACK`. These five resolutions define a spectrum of error tolerance or sensitivity. At one end of the spectrum is `REPLACE`, which effectively allows a statement to plow through almost every possible constraint violation. At the other end is `ROLLBACK`, which terminates the entire transaction upon the first violation of any kind. The resolutions are defined as follows, in increasing order of their severity:

- `REPLACE`: When a `UNIQUE` constraint violation is encountered, SQLite removes the row (or rows) that caused the violation and replaces it (them) with the new row from the `INSERT` or `UPDATE`. The SQL operation continues without error. If a `NOT NULL` constraint violation occurs, the `NULL` value is replaced by the default value for that column. If the column has no default value, then SQLite applies the `ABORT` policy. It is important to note that when this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. This behavior, however, is subject to change in a future release.

- IGNORE: When a constraint violation is encountered, SQLite allows the command to continue and leaves the row that triggered the violation unchanged. Other rows continue to be modified by the command. Thus, all rows in the operation that trigger constraint violations are simply left unchanged and the command proceeds without error.
- FAIL: When a constraint violation is encountered, SQLite terminates the command but does not restore the changes it made prior to encountering the violation. That is, all changes within the SQL command up to the violation are preserved. For example, if an UPDATE statement encountered a constraint violation on the 100th row it attempts to update, then the changes to the 99 rows already modified remain intact but changes to rows 100 and beyond never occur as the command is terminated.
- ABORT: When a constraint violation is encountered, SQLite restores all changes the command made and terminates it. ABORT is the default resolution for all operations in SQLite. It is also the behavior defined in the SQL standard. As a side note, ABORT is the most expensive conflict resolution policy, requiring extra work even if no conflicts ever occur.
- ROLLBACK: When a constraint violation is encountered, SQLite performs a ROLLBACK – aborting the current command along with the entire transaction. The net result is that all changes made by the current command *and all previous commands* in the transaction are rolled back. This is the most drastic level of conflict resolution where a single violation results in a complete reversal of everything performed in a transaction.

The type of conflict resolution can be specified within SQL commands as well as within table and index definitions. Specifically, conflict resolution can be specified in INSERT, UPDATE, CREATE TABLE, and CREATE INDEX commands. Furthermore, it has specific implications within triggers. The syntax for conflict resolution in INSERT and UPDATE is as follows:

```
INSERT OR resolution INTO table (column_list) VALUES (value_list);
UPDATE OR resolution table SET (value_list) WHERE predicate;
```

The conflict resolution policy comes right after the INSERT or UPDATE command and is prefixed with OR. The INSERT OR REPLACE expression can be abbreviated as just REPLACE.

In the above example, the updates made to 387 records were rolled back because the default resolution is ABORT. If you want the updates to stick, you could use the FAIL resolution. To illustrate this, in the following example, I copy foods into a new table called test and add an additional column called modified, the default value of which is 'no'. In the UPDATE, I change this to 'yes' to track which records are updated before the constraint violation occurs. Using the FAIL resolution, these updates remain unchanged and I can track afterward how many records were updated.

```
sqlite> CREATE TABLE test AS SELECT * FROM foods;
sqlite> CREATE UNIQUE INDEX test_idx on test(id);
sqlite> ALTER TABLE test
        ADD COLUMN modified text NOT NULL DEFAULT 'no';
sqlite> SELECT COUNT(*) FROM test WHERE modified='no';

COUNT(*)
--------------------
412

sqlite> UPDATE OR FAIL test SET id=800-id, modified='yes';
SQL error: column id is not unique

sqlite> SELECT COUNT(*) FROM test WHERE modified='yes';

COUNT(*)
--------------------
387

sqlite> DROP TABLE test;
```

There is one consideration with FAIL that you need to be aware of. The order in which records are updated is nondeterministic. That is, you cannot be certain of the order of the records in the table or the order in which SQLite processes them. You might assume that it follows the order of the ROWID column, but this is not a safe assumption to make: there is nothing in the documentation that says so. In many cases, it might be better to use IGNORE than FAIL. IGNORE finishes the job and modifies all records that can be modified rather than bailing out on the first violation.

When defined within tables, conflict resolution is specified for individual columns. For example:

```
sqlite> CREATE TEMP TABLE cast(name text UNIQUE ON CONFLICT ROLLBACK);
sqlite> INSERT INTO cast VALUES ('Jerry');
sqlite> INSERT INTO cast VALUES ('Elaine');
sqlite> INSERT INTO cast VALUES ('Kramer');
```

The cast table has a single column name with a UNIQUE constraint and conflict resolution set to ROLLBACK. Any INSERT or UPDATE that triggers a constraint violation on name is resolved by the ROLLBACK resolution rather than the default ABORT. The result aborts not only the statement but the entire transaction:

```
sqlite> BEGIN;
sqlite> INSERT INTO cast VALUES('Jerry');
SQL error: uniqueness constraint failed

sqlite> COMMIT;
SQL error: cannot commit - no transaction is active
```

COMMIT failed here because the name's conflict resolution already aborted the transaction. CREATE INDEX works the same way. Conflict resolution within tables and indices changes the default behavior of the operation from ABORT to that defined for the specific columns when those columns are the source of the constraint violation.

Conflict resolution at statement level (DML) overrides that defined at object level (DDL). Working from the previous example:

```
sqlite> BEGIN;
sqlite> INSERT OR REPLACE INTO cast VALUES('Jerry');
sqlite> COMMIT;
```

The REPLACE resolution in the INSERT overrides the ROLLBACK resolution defined on cast.name.

4.8.3 Transaction Types

SQLite provides different transaction types that start a transaction in different locking states. Locking is used as a way to manage multiple clients trying to access a database at the same time. It is discussed at length in Section 5.3.

On Symbian SQL, transactions can be started as DEFERRED or IMMEDIATE. A further SQLite type, EXCLUSIVE, is equivalent to IMMEDIATE on Symbian SQL. A transaction's type is specified in the BEGIN command:

```
BEGIN [ DEFERRED | IMMEDIATE | EXCLUSIVE ] TRANSACTION;
```

A *deferred transaction* does not acquire any locks until it has to. Thus with a deferred transaction, the BEGIN statement itself does nothing – it starts in the unlocked state and has no effect on other clients. This is the default. If you simply issue a BEGIN, then your transaction is DEFERRED and sitting in the unlocked state. Multiple sessions can start DEFERRED transactions at the same time without taking any locks.

An *immediate transaction* attempts to obtain a lock as soon as the BEGIN command is executed. If successful, BEGIN IMMEDIATE guarantees that no other session can write to the database. Other sessions can read from the database, but the lock prevents them from reading a table that is being written. Another consequence of the lock is that no other sessions can successfully issue a BEGIN IMMEDIATE command.

The bottom line is this: if you are using a database that no other connections are using, then a simple BEGIN suffices. If, however, you are using a database to which other connections are also writing, both you and they should use BEGIN IMMEDIATE to initiate transactions. It works out best that way for both of you. Transactions and locks are covered in more detail in Chapter 5.

4.9 Database Administration

Database administration is generally concerned with controlling how a database operates. From an SQL perspective, this includes various topics such as views, triggers, and indexes. Additionally, SQLite includes some unique administrative features of its own, such as the means to 'attach' multiple databases to a single session.

4.9.1 Views

Views are virtual tables. They are also known as *derived tables*, as their contents are derived from other tables. While views look and feel like base tables, they aren't. The contents of base tables are persistent whereas the contents of views are dynamically generated when they are used. Specifically, a view is composed of relational expressions that take other tables and produce a new table. The syntax to create a view is as follows:

```
CREATE VIEW name AS sql;
```

The name of the view is given by `name` and its definition by `sql`. The resulting view looks like a base table named `name`. Imagine you have a query you run all the time, so much that you get sick of writing it. Views are the cure for this particular sickness. Say your query is as follows:

```
SELECT f.name, ft.name, e.name
FROM foods f
INNER JOIN food_types ft on f.type_id=ft.id
INNER JOIN foods_episodes fe ON f.id=fe.food_id
INNER JOIN episodes e ON fe.episode_id=e.id;
```

This returns the name of every food, its type, and every episode it was in. It is one big table of 504 rows with just about every food fact. Rather than having to write out (or remember) the previous query every time you want these results, you can tidily restate it in the form of a view. Let's name it `details`:

```
CREATE VIEW details AS
  SELECT f.name AS fd, ft.name AS tp, e.name AS ep, e.season as ssn
  FROM foods f
  INNER JOIN food_types ft on f.type_id=ft.id
  INNER JOIN foods_episodes fe ON f.id=fe.food_id
  INNER JOIN episodes e ON fe.episode_id=e.id;
```

Now you can query `details` just as you would a table. For example:

```
sqlite> SELECT fd as Food, ep as Episode
        FROM details WHERE ssn=7 AND tp='Drinks';

Food                  Episode
--------------------  --------------------
Apple Cider           The Bottle Deposit 1
Bosco                 The Secret Code
Cafe Latte            The Postponement
Cafe Latte            The Maestro
Champagne Coolies     The Wig Master
Cider                 The Bottle Deposit 2
Hershey's             The Secret Code
Hot Coffee            The Maestro
Latte                 The Maestro
Mellow Yellow soda    The Bottle Deposit 1
Merlot                The Rye
Orange Juice          The Wink
Tea                   The Hot Tub
Wild Turkey           The Hot Tub
```

The contents of views are dynamically generated. Thus, every time you use `details`, its associated SQL is re-executed, producing results based on the data in the database at that moment.

Views also have other purposes, such as security, although this particular kind of security does not exist in SQLite. Some databases offer row- and column-level security in which only specific users, groups, or roles can view or modify specific parts of tables. In such databases, views can be defined on tables to exclude sensitive columns, allowing users with fewer security privileges to access parts of tables that are not secured. For example, say you have a table `secure`, defined as follows:

```
CREATE TABLE secure (id int, public_data text, restricted_data text);
```

You may want to allow users access to the first two columns but not the third. In other databases, you would limit access to `secure` to just the users who can access all columns. You would then create a view that contains just the first two columns that everyone else can look at:

```
CREATE VIEW public_secure AS SELECT id, public_data FROM secure;
```

Some of this kind of security is available if you program with SQLite, using its operational control facilities.

Finally, to drop a view, use the `DROP VIEW` command:

```
DROP VIEW name;
```

The name of the view to drop is given by `name`.

The relational model calls for updatable views, sometimes referred to as *materialized views*. These are views that you can modify. You can run `INSERT` or `UPDATE` statements on them, for example, and the respective changes are applied directly to their underlying tables. Materialized views are not supported in SQLite. However, using triggers, you can create things that looks like materialized views. These are covered in Section 4.9.3.

4.9.2 Indexes

Indexes are a construct designed to speed up queries under certain conditions. Consider the following query:

```
SELECT * FROM foods WHERE name='JujyFruit';
```

When a database searches for matching rows, the default method it uses to perform this is called a *sequential scan*. That is, it literally searches (or scans) every row in the table to see if its `name` attribute matches `'JujyFruit'`.

However, if this query is used frequently and `foods` is very large, there is another method available that can be much faster, called an *index scan*. An index is a special disk-based data structure (called a *B-tree*), which stores the values for an entire column (or columns) in a highly organized way that is optimized for searching.

The search speed of these two methods can be represented mathematically. The search speed of a sequential scan is proportional to the number of rows in the table: the more rows, the longer the scan. This relationship – bigger table, longer search – is called *linear time*, as in 'the search method operates in linear time.' It is represented mathematically using what is called the *Big O notation*, which in this case is *O(n)*, where *n* stands for the number of elements (or rows) in the set. The index scan, on the other hand, has *O(log(n))* search time, or logarithmic time. This is much faster. If you have a table of 10,000 records, a sequential scan will read all 10,000 rows to find all matches, while an index scan reads four rows (log(10,000)) to find the first match (and from that point on it would be linear time to find all subsequent matches). This is quite a speed-up.

But there is no such thing as a free lunch. Indexes also increase the size of the database. They literally keep a copy of all columns they index. If you index every column in a table, you effectively double the size of the table. Another consideration is that indexes must be maintained. When you insert, update, or delete records, in addition to modifying the table, the database must modify every index on that table as well. So indexes can slow down inserts, updates, and similar operations.

But in moderation, indexes can make a huge performance difference. Whether or not to add an index is more subjective than anything else. Different people have different criteria for what is acceptable. Indexes illustrate one of the great things about SQL: you only need to specify what to get and not how to get it. Because of this, you can often optimize your database (such as by adding or removing indexes) without having to rewrite your code. Just create an index in the right place.

The command to create an index is as follows:

```
CREATE INDEX [UNIQUE] index_name ON table_name (columns)
```

The variable `index_name` is the name of the index and must be unique. `table_name` is the name of the table containing the columns to index. The variable `columns` is either a single column or a comma-separated list of columns.

If you use the `UNIQUE` keyword, then the index has the added constraint that all values in the index must be unique. This applies to both the index and, by extension, the column or columns it indexes. The `UNIQUE` constraint covers all columns defined in the index and it is their combined values (not individual values) that must be unique. For example:

```
sqlite> CREATE TABLE foo(a text, b text);
sqlite> CREATE UNIQUE INDEX foo_idx on foo(a,b);
sqlite> INSERT INTO foo VALUES ('unique', 'value');
sqlite> INSERT INTO foo VALUES ('unique', 'value2');
sqlite> INSERT INTO foo VALUES ('unique', 'value');
SQL error: columns a, b are not unique
```

You can see here that uniqueness is defined by both columns collectively, not individually. Notice that collation plays an important role here as well.

To remove an index, use the `DROP INDEX` command, which is defined as follows:

```
DROP INDEX index_name;
```

Collation

You can associate a collation with each column in the index. For example, to create a case-insensitive index on `foods.name`, you'd use the following:

```
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

This means that values in the `name` column sort without respect to case. You can list the indexes for a table in the SQLite command-line program by using the `.indices` shell command. For example:

```
sqlite> .indices foods
foods_name_idx
```

For more information, you can use the `.schema` shell command:

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

You can also obtain this information by querying the `sqlite_master` table, described in Section 4.9.6.

Symbian SQL adds support for Symbian's native collation methods (Table 4.8).

Table 4.8 Symbian collation methods

Collation name	**Description**
`CompareC0`	16-bit string-collated comparison at level 0
`CompareC1`	16-bit string-collated comparison at level 1
`CompareC2`	16-bit string-collated comparison at level 2
`CompareC3`	16-bit string-collated comparison at level 3
`CompareF`	16-bit string-folded comparison

These collations can be used in the following cases:

- As column constraints in CREATE TABLE statements. For example:

  ```
  CREATE TABLE A (Col1 TEXT COLLATE CompareC1)
  ```

  In this case, every time `Col1` values have to be compared, the SQL server uses `CompareC1` collation.

- As a column constraint in CREATE INDEX statements. For example:

  ```
  CREATE INDEX I ON A (Col1 COLLATE CompareC2)
  ```

  In this case, the SQL server uses `CompareC2` collation to compare `Col1` values when using the index.

- In the ORDER BY clause of SELECT commands. For example:

  ```
  SELECT * FROM A ORDER BY Col1 COLLATE CompareC3
  ```

  In this case, the SQL server uses `CompareC3` collation to compare `Col1` values when building the result set.

The collation names in Table 4.8 correspond to techniques available in Symbian descriptors which are typically used to manage strings on Symbian OS. Collation is a powerful way to compare strings and produces a dictionary-like (‘lexicographic’) ordering. For languages using the Latin script, for example, collation is about deciding whether to ignore punctuation, whether to fold upper and lower case, how to treat accents, and so on. In a given locale there is usually a standard set of collation rules that can be used. Collation should always be used to compare strings in natural language.

With collation, there are many ways in which two strings can match, even when they do not have the same length:

- One string includes combining characters, but the collation level is set to 0 (which means that accents are ignored)
- One string contains ‘pre-composed’ versions of accented characters and the other contains ‘decomposed’ versions of the same character.
- One string contains a ligature that, in a collation table, matches multiple characters in the other string and the collation level is set to less than 3 (for example ‘æ’ might match ‘ae’).

- One string contains a 'surrogate pair' (a 32-bit encoded character) which happens to match a normal character at the level specified.
- The collation method does not have its 'ignore none' flag set and the collation level is set to less than 3. In this case, spaces and punctuation are ignored; this means that one string could be much longer than the other just by adding a large number of spaces.
- One string contains the Hangul representation and the other contains the Jamo representation of the same Korean phrase and the collation level is set to less than 3.

The collation level is an integer that can take one of the values: 0, 1, 2 or 3, and determines how tightly two strings should match. In Symbian SQL, this value is included in the collation name, e.g. `CompareC3`; in Symbian C++, it is passed as the second parameter to `TDes::CompareC()`. The values have the following meanings:

- 0 – only test the character identity; accents and case are ignored
- 1 – test the character identity and accents; case is ignored
- 2 – test the character identity, accents and case
- 3 – test the Unicode value as well as the character identity, accents and case.

At levels 0–2:

- ligatures (e.g. 'æ') are the same as their decomposed equivalents (e.g. 'ae')
- script variants are the same (e.g., 'R' matches the mathematical real number symbol (Unicode 211D))
- the 'micro' symbol (Unicode 00B5) matches Greek 'mu' (Unicode 03BC)).

At level 3 these are treated differently.

If the aim is to sort strings, then level 3 must be used. For any strings `a` and `b`, if `a` < `b` for some level of collation, then `a` < `b` for all higher levels of collation as well. It is impossible, therefore, to affect the order that is generated by using lower collation levels than 3. This just causes similar strings to sort in a random order. In standard English, sorting at level 3 gives the following order:

```
bat < bee < BEE < bug
```

The case of the 'B' only affects the comparison after all the letter identities have been found to be the same – this is usually what people are trying to achieve by using lower collation levels than 3 for sorting. It is never necessary. The sort order can be affected by setting flags in the `TCollationMethod` object.

Note that when strings match at level 3, they do not necessarily have the same binary representation or even the same length. Unicode contains many strings that are regarded as equivalent, even though they have different binary representations.

`CompareF` uses *folding comparison*. Folding is a relatively simple way of normalizing text for comparison by removing case distinctions, converting accented characters to characters without accents, etc. Folding is used for tolerant comparisons that are biased towards a match.

For example, the file system uses folding to decide whether two file names are identical or not. Folding is locale-independent behavior and means that the file system, for example, can be locale-independent.

It is important to note that there can be no guarantee that folding is in any way culturally appropriate and it should not be used for comparing strings in natural language; collation is the correct functionality for this.

Folding cannot remove piece accents or deal with correspondences that are not one-to-one, such as the mapping from German upper case SS to lower case ß. In addition, folding cannot ignore punctuation.

Index Utilization

It is important to understand when indexes are used and when they are not. There are very specific conditions in which SQLite decides to use an index.

SQLite uses a single-column index, if available, for the following expressions in the `WHERE` clause:

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

Multicolumn indexes have more specific conditions before they are used. This is perhaps best illustrated by example. Say you have a table defined as follows:

```
CREATE TABLE foo (a,b,c,d);
```

Furthermore, you create a multicolumn index as follows:

```
CREATE INDEX foo_idx on foo (a,b,c,d);
```

The columns of foo_idx can only be used sequentially from left to right. That is, in the query

```
SELECT * FROM foo WHERE a=1 AND b=2 AND d=3
```

only the first and second conditions use the index. The third condition is excluded because there is no condition that uses c to bridge the gap to d. Basically, when SQLite uses a multicolumn index, it works from left to right column-wise. It starts with the left column and looks for a condition using that column. It moves to the second column, and so on. It continues until either it fails to find a valid condition in the WHERE clause that uses it or there are no more columns in the index to use.

But there is one more requirement. SQLite uses a multicolumn index only if all of the conditions use either the equality (=) or IN operators for all index columns *except for the rightmost index column*. For that column, you can specify up to two inequalities to define its upper and lower bounds. Consider, for example, the following statement:

```
SELECT * FROM foo WHERE a>1 AND b=2 AND c=3 AND d=4
```

SQLite only does an index scan on column a. The a>1 expression becomes the rightmost index column because it uses the inequality operator. As a result, all columns after it are not eligible to be used in the index. Similarly, consider the following statement:

```
SELECT * FROM foo WHERE a=1 AND b>2 AND c=3 AND d=4
```

The index columns a and b are used; b>2 becomes the rightmost index term by its use of an inequality operator.

An off-the-cuff way to time a query within an SQL statement is to use a subselect in the FROM clause to return the current time, which is joined to your input relations. Then select the current time in the SELECT clause. The time in the FROM clause is computed at the start of the query. The time in SELECT clause is computed as each row is processed. Therefore, the difference between the two times in the last row of the result set is your relative query time. For example, the following SQL is a quadruple Cartesian join on food_types. It's quite slow, as it should be. The results display the last five records of the result set:

```
SELECT CAST(strftime('%s','now') as INTEGER)-
            CAST(time.start as INTEGER) time,
        ft1.id, ft2.id, ft3.id, ft4.id
FROM food_types ft1, food_types ft2, food_types ft3, food_types ft4,
     (SELECT strftime('%s','now') start) time;

.    .     .     .     .
.    .     .     .     .
.    .     .     .     .
18   15    15    15    11
18   15    15    15    12
18   15    15    15    13
18   15    15    15    14
18   15    15    15    15
```

The first column is the elapsed time in seconds. This query took 18 seconds. Although this doesn't represent the actual query time because there is timing overhead, relative query time is all you need to judge an index's worth. If this were a slow query that you were trying to optimize, you would now add the index you think might help and rerun the query. Is it significantly faster? If so, then you may have a useful index.

In short, when you create an index, have a reason for creating it. Make sure there is a specific performance gain you are getting before you take on the overhead that comes with it. Well-chosen indexes are a wonderful thing. Indexes that are thoughtlessly scattered here and there in the vain hope of improving performance are of dubious value.

4.9.3 Triggers

Triggers execute specific SQL commands when specific database events transpire on specific tables. The general syntax for creating a trigger is as follows:

```
CREATE [TEMP|TEMPORARY] TRIGGER name [BEFORE|AFTER]
  [INSERT|DELETE|UPDATE|UPDATE OF columns] ON table
  action
```

A trigger is defined by a name, an action, and a table. The action, or trigger body, consists of a series of SQL commands. Triggers are said to *fire* when such events take place. Furthermore, triggers can be made to fire before or after the event using the BEFORE or AFTER keyword, respectively. Events include DELETE, INSERT, and UPDATE commands issued on the specified table. Triggers can be used to create your own integrity constraints, log changes, update other tables, and many other things. They are limited only by what you can write in SQL.

UPDATE Triggers

UPDATE triggers, unlike INSERT and DELETE triggers, may be defined for specific columns in a table. The general form of this kind of trigger is as follows:

```
CREATE TRIGGER name [BEFORE|AFTER] UPDATE OF column ON table
action
```

The following is an SQL script that shows an UPDATE trigger in action:

```
.h on
.m col
.w 50
.echo on
CREATE TEMP TABLE log(x);

CREATE TEMP TRIGGER foods_update_log UPDATE of name ON foods
BEGIN
  INSERT INTO log VALUES('updated foods: new name=' || NEW.name);
END;

BEGIN;
  UPDATE foods set name='JUJYFRUIT' where name='JujyFruit';
  SELECT * FROM log;
ROLLBACK;
```

This script creates a temporary table called log, as well as a temporary UPDATE trigger on foods.name that inserts a message into log when it fires. The action takes place inside the transaction that follows. The first step of the transaction updates the name column of the row whose name is 'JUJYFRUIT'. This causes the UPDATE trigger to fire. When it fires, it inserts a record into the log. Next, the transaction reads log, which shows that the trigger did indeed fire. The transaction then rolls back the change; when the session ends, the log table and the UPDATE trigger are destroyed. Running the script produces the following output:

```
mike@linux tmp # sqlite3 foods.db < trigger.sql
CREATE TEMP TABLE log(x);

CREATE TEMP TRIGGER foods_update_log AFTER UPDATE of name ON foods
BEGIN
  INSERT INTO log VALUES('updated foods: new name=' || NEW.name);
END;

BEGIN;
  UPDATE foods set name='JUJYFRUIT' where name='JujyFruit';
  SELECT * FROM log;
```

```
x
--------------------------------------------------
updated foods: new name=JUJYFRUIT
ROLLBACK;
```

SQLite provides access to both the old (original) row and the new (updated) row in `UPDATE` triggers. The old row is referred to as `OLD` and the new row as `NEW`. Notice in the script how the trigger refers to `NEW.name`. All attributes of both rows are available in `OLD` and `NEW` using the dot notation. I could have just as easily recorded `NEW.type_id` or `OLD.id`.

Error Handling

Defining a trigger before an event takes place gives you the opportunity to stop the event from happening. `BEFORE` triggers enables you to implement new integrity constraints. SQLite provides a special SQL function for triggers called `RAISE()`, which allows them to raise an error within the trigger body. `RAISE` is defined as follows:

```
RAISE(resolution, error_message);
```

The first argument is a conflict resolution policy (`ABORT`, `FAIL`, `IGNORE`, `ROLLBACK`, etc.). The second argument is an error message. If you use `IGNORE`, the remainder of the current trigger along with the SQL statement that caused the trigger to fire, as well as any subsequent triggers that would have been fired, are all terminated. If the SQL statement that caused the trigger to fire is itself part of another trigger, then that trigger resumes execution at the beginning of the next SQL command in the trigger action.

Conflict Resolution

If a conflict resolution policy is defined for an SQL statement that causes a trigger to fire, then that policy supersedes the policy defined within the trigger. If, on the other hand, the SQL statement does not have a conflict resolution policy defined and the trigger does, then the trigger's policy is used.

Updatable Views

Triggers make it possible to create something like materialized views, as mentioned earlier in this chapter. In reality, they aren't true materialized views but rather more like updatable views. With true materialized views, the view is updatable all by itself – you define the view and it figures out how to map all changes to its underlying base tables. This is not a simple thing. Nor is it supported in SQLite. However, using triggers, we can create the appearance of a materialized view.

The idea here is that you create a view and then create a trigger that handles update events on that view. SQLite supports triggers on views using the `INSTEAD OF` keywords in the trigger definition. To illustrate this, let's create a view that combines `foods` with `food_types`:

```
CREATE VIEW foods_view AS
  SELECT f.id fid, f.name fname, t.id tid, t.name tname
  FROM foods f, food_types t;
```

This view joins the two tables according to their foreign key relationship. Notice that I have created aliases for all column names in the view. This allows me to differentiate the respective `id` and `name` columns in each table when I reference them from inside the trigger. Now, let's make the view updatable by creating an `UPDATE` trigger on it:

```
CREATE TRIGGER on_update_foods_view
INSTEAD OF UPDATE ON foods_view
FOR EACH ROW
BEGIN
  UPDATE foods SET name=NEW.fname WHERE id=NEW.fid;
  UPDATE food_types SET name=NEW.tname WHERE id=NEW.tid;
END;
```

Now if you try to update `foods_view`, this trigger is called. The trigger simply takes the values provided in the `UPDATE` statement and uses them to update the underlying base tables `foods` and `food_types`. Testing it out yields the following:

```
-- Update the view within a transaction
BEGIN;
  UPDATE foods_view SET fname='Whataburger', tname='Fast Food'
  WHERE fid=413;
-- Now view the underlying rows in the base tables:
  SELECT * FROM foods f, food_types t
  WHERE f.type_id=t.id AND f.id=413;

id   type_id  name            id   name
---  -------  --------------  ---  ---------
413  1        Whataburger     1    Fast Food

-- Roll it back
```

```
ROLLBACK;

-- Now look at the original record:
SELECT * FROM foods f, food_types t WHERE f.type_id=t.id AND f.id=413;

id   type_id  name            id   name
---  -------  --------------  ---  -------
413  1        Cinnamon Bobka  1    Bakery
```

You can just as easily add `INSERT` and `DELETE` triggers and have the rough equivalent of a materialized view.

Implementing Foreign Key Constraints

One of the most interesting applications of triggers in SQLite is the implementation of foreign key constraints. To further explore triggers, I use this idea to implement foreign key constraints between `foods` and `food_types`.

As stated earlier, `foods.type_id` references `food_types.id`. Therefore, every value in `foods.type_id` should correspond to some value in `food_types.id`. The first step in enforcing this constraint lies in controlling what can be inserted into `foods`. This is accomplished with the following `INSERT` trigger:

```
CREATE TRIGGER foods_insert_trg
BEFORE INSERT ON foods
BEGIN
  SELECT CASE
    WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
    THEN RAISE( ABORT,
        'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END;
```

This trigger runs a subquery that checks for the value of `NEW.type_id` in `foods_types.id`. If no match is found, the subquery returns `NULL`, which triggers the `WHEN` condition, calling the `RAISE` function.

After installing the trigger, the following SQL tries to insert a record with an invalid `type_id` (the maximum `id` value in `food_types` is 15):

```
sqlite> INSERT INTO foods VALUES (NULL, 20, 'Blue Bell Ice Cream');
SQL error: Foreign Key Violation: foods.type_id is not in food_types.id
```

Next is `UPDATE`. The only thing that matters on `UPDATE` of `foods` is the `type_id` field, so the trigger is defined on that column alone.

Aside from this and the trigger's name, the trigger is identical to `INSERT`:

```
CREATE TRIGGER foods_update_trg
BEFORE UPDATE OF type_id ON foods
BEGIN
  SELECT CASE
    WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
    THEN RAISE(ABORT,
       'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END;
```

Testing this trigger reveals the same results:

```
sqlite> UPDATE foods SET type_id=20 WHERE name='Chocolate Bobka';
SQL error: Foreign Key Violation: foods.type_id is not in food_types.id
```

The final piece of the puzzle is `DELETE`. Deleting rows in `foods` doesn't affect the relationship with `food_types`. Deleting rows in `food_types`, however, does affect the relationship with `foods`. If a row were to be deleted from `food_types`, there could be rows in `foods` that reference it, in which case the relationship is compromised. Therefore, we need a `DELETE` trigger on `food_types` that does not allow the deletion of a row if there are rows in `foods` that reference it. To that end, the `DELETE` trigger is defined as follows:

```
CREATE TRIGGER foods_delete_trg
BEFORE DELETE ON food_types
BEGIN
  SELECT CASE
    WHEN (SELECT COUNT(type_id) FROM foods WHERE type_id=OLD.id) > 0
    THEN RAISE(ABORT,
     'Foreign Key Violation: foods rows reference row to be deleted.')
  END;
END;
```

After installing this trigger, if I try to delete the `'Bakery'` row in `food_types` I get:

```
sqlite> DELETE FROM food_types WHERE name='Bakery';
SQL error: Foreign Key Violation: foods rows reference row
                                            to be deleted.
```

To make sure this works under the correct conditions:

```
sqlite> BEGIN;
sqlite> DELETE FROM foods WHERE type_id=1;
sqlite> DELETE FROM food_types WHERE name='Bakery';
sqlite> ROLLBACK;
```

The DELETE trigger allows the delete if there are no rows in foods that reference it.

So there you have it: simple, trigger-based foreign key constraints. As mentioned earlier, while SQLite does support CHECK constraints, triggers can pretty much do everything CHECK constraints can and then some.

4.9.4 Attaching Databases

SQLite allows you to 'attach' multiple databases to the current session using the ATTACH command. When you attach a database, all of its contents are accessible in the global scope of the current database file. ATTACH has the following syntax:

```
ATTACH [DATABASE] filename AS database_name;
```

Here, filename refers to the path and name of the SQLite database file, and database_name refers to the logical name with which to reference that database and its objects. The main database is automatically assigned the name main. If you create any temporary objects, then SQLite creates an attached database with the name temp. The logical name may be used to reference objects within the attached database. If there are tables or other database objects that share the same name in both databases, then the logical name is required to reference such objects in the attached database. For example, if both databases have a table called foo and the logical name of the attached database is db2, then the only way to query foo in db2 is by using the fully qualified name db2.foo, as follows:

```
sqlite> ATTACH DATABASE '/tmp/db' as db2;
sqlite> SELECT * FROM db2.foo;

x
----------
bar
```

If you really want to, you can qualify objects in the main database using the name main:

```
sqlite> SELECT * FROM main.foods LIMIT 2;

id          type_id     name
----------  ----------  --------------
1           1           Bagels
2           1           Bagels, raisin
```

The same is true with the temporary database:

```
sqlite> CREATE TEMP TABLE foo AS SELECT * FROM food_types LIMIT 3;
sqlite> SELECT * FROM temp.foo;

id   name
---  -------------
1    Bakery
2    Cereal
3    Chicken/Fowl
```

You detach databases with the `DETACH DATABASE` command, defined as follows:

```
DETACH [DATABASE] database_name;
```

This command takes the logical name of the attached database (given by `database_name`) and detaches the associated database file.

Symbian SQL also supports attached databases. More information on how this can be done using Symbian SQL can be found in Section 6.1.3.

4.9.5 Cleaning Databases

SQLite has two commands designed for cleaning – `REINDEX` and `VACUUM`. `REINDEX` is used to rebuild indexes. It has two forms:

```
REINDEX collation_name;
REINDEX table_name|index_name;
```

The first form rebuilds all indexes that use the collation name given by `collation_name`. It is only needed when you change the behavior of a user-defined collating sequence (e.g., multiple sort orders in Chinese). All indexes in a table (or a particular index given its name) can be rebuilt with the second form.

`VACUUM` cleans out any unused space in the database by rebuilding the database file. `VACUUM` will not work if there are any open transactions. An alternative to manually running `VACUUM` statements is to auto-vacuum, which may be set as part of the Symbian SQL database configuration – see Section 7.1.3 for details.

4.9.6 The System Catalog

The `sqlite_master` table is a system table that contains information about all the tables, views, indexes, and triggers in the database. For example, the current contents of the foods database are as follows:

```
sqlite> SELECT type, name, rootpage FROM sqlite_master;

type        name                         rootpage
----------  -------------------------  ----------
table       episodes                     2
table       foods                        3
table       foods_episodes               4
table       food_types                   5
index       foods_name_idx               30
table       sqlite_sequence              50
trigger     foods_update_trg             0
trigger     foods_insert_trg             0
trigger     foods_delete_trg             0
```

The `type` column refers to the type of object, `name` is of course the name of the object, and `rootpage` refers to the first B-tree page of the object in the database file. This latter column is only relevant for tables and indexes.

The `sqlite_master` table also contains another column called `sql`, which stores the DML used to create the object. For example:

```
sqlite> SELECT sql FROM sqlite_master WHERE name='foods_update_trg';

CREATE TRIGGER foods_update_trg
BEFORE UPDATE OF type_id ON foods
BEGIN
  SELECT CASE
    WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
    THEN RAISE( ABORT,
       'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END
```

4.9.7 Viewing Query Plans

You can view the way SQLite goes about executing a query by using the `EXPLAIN` command. `EXPLAIN` lists the VDBE program that would be used to carry out an SQL command. The VDBE is the virtual machine in SQLite that carries out all of its database operations. Every query in

SQLite is executed by first compiling the SQL into VDBE bytecode and then running the bytecode. For example:

```
sqlite> EXPLAIN SELECT * FROM foods;

addr  opcode           p1    p2    p3
----  ---------------  ----  ----  ----
0     Goto             0     12
1     Integer          0     0
2     OpenRead         0     3
3     SetNumColumns    0     3
4     Rewind           0     10
5     Rowid            0     0
6     Column           0     1
7     Column           0     2
8     Callback         3     0
9     Next             0     5
10    Close            0     0
11    Halt             0     0
12    Transaction      0     0
13    VerifyCookie     0     134
14    Goto             0     1
15    Noop             0     0
```

Studying these query plans is not for the faint of heart. The average person is not going to find a VDBE program very intuitive. However, for those who are willing to try, the VDBE, its op codes, and theory of operation are covered in SQLite documentation.

4.10 Summary

SQL may be a simple language to use but there is quite a bit of it. But that shouldn't be too surprising, as it is the sole interface through which to interact with a relational database. Whether you are a casual user, system administrator, or developer, you have to know SQL if you are going to work with a relational database.

SQL is a thin wrapper around the relational model. It is composed of three essential parts: form, function, and integrity. Form relates to how information is represented. In SQL, this is done with the table. The table is the sole unit of information and is composed of rows and columns. The database is made up of other objects as well, such as views, indexes, and triggers. Views provide a convenient way to represent information in tables in different forms. Indexes work to speed up queries. Triggers enable you to associate programming logic with specific database events as they transpire on tables. All of these objects are created using data definition language (DDL). DDL is the part of SQL reserved for creating and destroying database objects. Furthermore, all objects in a database are listed in the `sqlite_master` system table.

The functional part of SQL pertains to how information is manipulated. The part of SQL reserved for this is called data manipulation language (DML). DML is composed of the `SELECT`, `UPDATE`, `INSERT`, and `DELETE` commands. `SELECT` is the largest and most complicated command in the language. `SELECT` employs 12 of the 14 operations defined in relational algebra. It arranges those operations in a customizable pipeline. You can use those operations as needed to combine, filter, order, and aggregate information within tables.

The integrity part of SQL controls how data is inserted and modified. It protects information and relationships by ensuring that the values in columns conform to specific rules. It is implemented using constraints, such as `UNIQUE`, `CHECK`, and `NOT NULL`.

SQLite has a unique way of handling data types that is unlike most other relational databases. All values have a storage class. SQLite has five storage classes: `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL`. These classes specify how values are stored and manipulated in the database. All columns have an affinity, which determines how values are stored within them. SQLite has four kinds of affinity: `NUMERIC`, `INTEGER`, `TEXT`, and `NONE`. When SQLite stores a value in a column, it uses both the column's affinity and the value's storage class to determine how to store the value. For applications that need strict type checking on a column, it can be implemented using the `typeof()` function and a `CHECK` constraint.

5
Database Concepts

The previous chapter explored the subject of SQL in detail, so you should now have a good understanding of how to add data to a database, how to remove it, how to modify it and, perhaps most importantly, how to find and retrieve the particular data you want, structured in a way best suited to your needs.

SQL is only one part of the story. The relational model may appear abstract and SQL more concrete, but SQL itself is only a language. Just as communicating successfully with people involves far more than simply knowing the right language, efficient database programming requires more than knowledge of SQL alone.

This chapter introduces a number of important concepts, all of which are essential for proficient database programming. Once you've read and understood this chapter, you'll be ready to move on to Chapter 6, which describes the Symbian SQL APIs.

The chapter starts by describing the different types of database supported by Symbian SQL. Then I look at the mechanisms Symbian SQL provides to execute SQL on such databases. Finally, I move on to look at multi-client access, in particular focusing on the locking model implemented in Symbian SQL.

5.1 Database Types

A Symbian SQL database is contained within a single file – and a file can contain only one database. This convenient fact means that new databases may be brought to the device by simply copying the database file to the correct location in the file system. Existing databases are just as easily backed up or copied. A file copy operation is all that is needed. This differs from many database systems (both commercial and open source), which do not publicize the way individual databases map to physical files.

Three types of Symbian SQL database exist: public, private and secure shared. Public databases facilitate sharing data between multiple processes but provide no security. Private and secure shared databases both provide security measures.

5.1.1 Public Databases

A database located in a publicly accessible area of the file system is known as a *public database*. Such databases may be read and written by any application – thus public databases facilitate sharing. However, this means that the content of a public database may be tampered with by a malicious application or damaged accidentally.

Security should always be a prime concern for any application designer. Symbian does not recommend the use of public databases unless the database is intended only to be read and resides in a read-only file system.

5.1.2 Private Databases

A private database is one which is located in an executable's private directory. This means that only that application can access it. Using a private database is a simple way to ensure both privacy and freedom from tampering.

5.1.3 Secure Shared Databases

The choice between a private and a public database is effectively a choice between total security and none at all. For situations where a database is intended to be read and written by a single application, a private database is the perfect solution. However, when more than one application needs access to the database, public databases are likely to fall short of what is required as they provide no protection against tampering and no privacy guarantees.

Secure shared databases fall between these extremes and provide a flexible and configurable approach to security. They allow sharing but also provide security.

A secure shared database is held in the Symbian SQL server's private directory. No applications may access the database file directly – only the server may do so. The server provides a gatekeeper service to database clients; it allows only those which satisfy a pre-defined access control policy to use the database. The policy is defined by the creator of the database and is based on the Symbian platform security model. Only the creator may delete the database.

Access Control Policies

The access control policy specified for a secure shared database defines the criteria that a client must meet in order to access that database.

Many situations require write access to be granted to a different set of clients from those granted read access. Furthermore reading and writing data from and to a database is fundamentally different from changing the structure of the database. Thus an additional level of access control for administrative and structural changes is useful.

Symbian SQL access control policies support these situations by allowing three policies to be defined, one for each type of access:

- **read** – only clients that satisfy the read policy may retrieve data from the database.
- **write** – only clients that satisfy the write policy may insert, update, or delete rows from tables in the database.
- **schema** – only clients that satisfy the schema policy are able to modify the schema of the database including creating, modifying and deleting tables, indexes and triggers.

Access Control Policies in Use

Access control policies can be used in many different ways. For example:

- **multiple readers, no writers** – a configuration that only allows reading enables a tamper-proof, read-only database
- **multiple readers, single writer** – a configuration that allows live data from a single source (or aggregator) to be accessed by multiple clients
- **multiple readers and writers** – a configuration that allows multiple interested clients to update a common database and also keep abreast of changes made by other clients. This configuration is uncommon on mobile devices
- **single reader, multiple writers** – a configuration in which multiple writers may log private information and only a single privileged reader may retrieve the data.

Anti-spoofing Protection

When creating a secure shared database, the creator must specify an access control policy in order to protect the content from unauthorized manipulation. This allows clients sharing the database to trust the database content, as they know that only trusted clients are able to modify it. Equally, clients may store sensitive data in the database as they know that only authorized clients may retrieve it.

However, at the base of this trust lies the assumption that the database access control policy has been configured by a trusted entity. If a rogue application were to create the database, it could specify a deliberately loose policy that would allow the integrity of the database content to be compromised. To defeat this attack, Symbian SQL provides an anti-spoofing mechanism for secure shared databases based on trusted credentials.

When a database is created, the creator must supply a textual name for the database. For secure shared databases, the first part of the name must always be the Secure ID (SID) of the creating application. Symbian SQL only creates the database if the SID in the name matches the SID of the creator.

For example, an application with SID 0x12345678 would be able to create a database with the name '[12345678]mydatabase' but would not be able to create one called '[12121212]mydatabase.'

Thus, clients opening the database '[12345678]mydatabase' can be confident that it can only have been created by the application with the SID 0x12345678. The SID is, in turn, guaranteed by the Symbian Signed system, meaning that the expected executable is guaranteed to have created the database.

5.2 Executing SQL

Symbian SQL provides two ways for users to execute SQL statements: a simple *one-shot* approach and a more flexible mechanism called a *prepared statement*. These methods mirror the services provided by the underlying database engine SQLite.

5.2.1 How the Database Processes SQL

Understanding the difference between the two execution mechanisms is easier if I explain how the database engine processes the SQL and executes it.

Just like any other computer language, SQL must first be parsed and converted to another form before it can be executed. Within the database engine, SQL from the client is submitted to a compiler. The compiler analyzes the SQL in order to attempt to determine the optimal way of executing it against the database schema. As shown in Figure 5.1, the output of the compilation is a bytecode program which can be directly executed within the database engine.

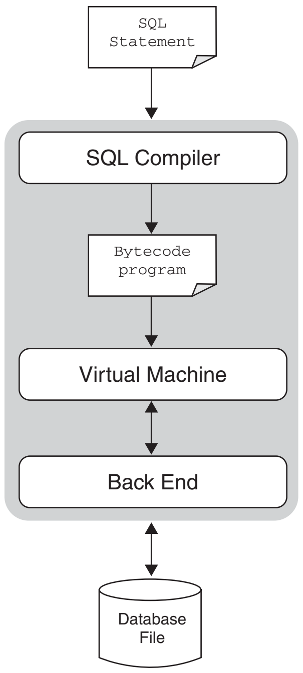


Figure 5.1

The database engine includes its own virtual machine on which the bytecode program runs. The role of the virtual machine is to drive the back-end storage according to the program in order to store and retrieve the necessary database content. It is responsible for all data processing operations including filtering, grouping and ordering.

The back-end is responsible for efficient physical data storage management and additionally provides the guarantees required to support transactions. It interfaces directly with the Symbian File Server.

For more detail on how the database engine works please refer to Chapter 7.

5.2.2 One-shot Execution

One-shot execution is designed to be the simplest mechanism to execute SQL. It hides all the complexity and effort of the SQL compilation and execution process from clients. Clients simply pass an SQL statement to an execution function on an open database connection. This causes the database engine to compile the SQL and execute the resulting bytecode. Control is only returned to the caller on completion of this chain of events, with the return code indicating success or failure. No resources allocated during the execution are left outstanding for the caller to clean up.

This approach can be ideal but it has two serious drawbacks which may render it inappropriate:

- One-shot execution is intended to hide all of the SQL execution phases and resource cleanup from the client. If Symbian SQL had included the facility to return result sets from one-shot execution, it would have had to retrieve the entire result set from the database and copy it en masse to the client. Considering the resource-constrained nature of mobile devices and also the fact that a result set can be arbitrarily large, it is not desirable or feasible to do this.
- If a client executes the same SQL statement several times then it is effectively compiling the same SQL several times. In the case of SELECT statements, the cost can be noticeable because query optimization is often computationally intensive as different access strategies need to be evaluated in order to find the fastest way to execute the query.

To overcome these limitations, clients can use a different approach called *prepared statements* which exposes the internal phases in a published API. This delivers more power and flexibility but comes at the cost of greater complexity to the application developer.

5.2.3 Prepared Statements

Prepared statement APIs allow clients to control the compilation (also called preparation), execution and cleanup phases of SQL execution individually. An SQL statement can be prepared in advance for use later and can be executed when required.

After the bytecode program has finished executing, it can be reset and kept in memory for re-use in the future. This means that if an SQL statement must be executed repeatedly, the client can prepare the statement once and run it as many times as required. When the prepared statement is no longer needed for use, it should be closed to free any associated resources. Figure 5.2 shows the flow of events.

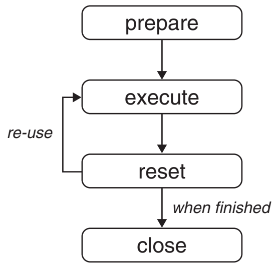


Figure 5.2

Parameterized SQL

Prepared statements can be more efficient than one-shot execution when a client wishes to repeatedly execute the same SQL statement. However, it is generally unusual to execute identical SQL statements again and again.

A more common occurrence is to execute many similar SQL statements. For example, it is rare to insert the same record multiple times, but it is likely that multiple INSERT statements are identical except for the values inserted. Similarly, SELECT statements are often customized to search for a particular record based on user input. In this case, the queries are identical except for the search parameters.

Prepared statements support a mechanism which allows an SQL statement to include parameters where actual values should be. The SQL can be prepared as usual, perhaps before the actual values are even known. When required, the values can be bound to the parameters and the prepared statement can be executed. It can then be reset, different values bound to the parameters and re-executed. For example, instead of preparing and executing two distinct SQL statements:

```
INSERT INTO locations (name) VALUES ('London');
INSERT INTO locations (name) VALUES ('New York');
```

The following single SQL statement can be prepared once:

```
INSERT INTO locations (name) VALUES (:placename);
```

It can then be executed twice with the relevant name bound to the `placename` parameter. The general flow of events is shown in Figure 5.3.

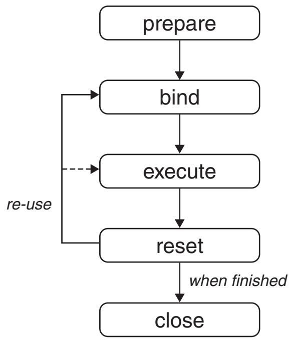


Figure 5.3 Events when executing a statement that returns no data

Imagine you have 1000 distinct records to insert into a table. Without bound parameters you must construct, compile and execute 1000 different SQL statements. However, using bound parameters you only need to compile the SQL once. The prepared statement can then be executed 1000 times with different bound values. Binding parameter values is a low-cost operation. When compared with repeated compilation of SQL, prepared statements in conjunction with bound parameters can give a valuable performance boost which can make the difference between a responsive and an unresponsive application.

An important point to note is that resetting a prepared statement does NOT clear the parameter bindings. Thus it is possible to re-execute the prepared statement directly after resetting without binding new values. This has the effect of executing exactly the same SQL statement again. This flow is represented by the dotted line in Figure 5.3. If the SQL statement includes multiple parameters then a hybrid of this is possible, where only some parameters are bound to new values. This can be very useful in situations where only some parameters need to change from one SQL statement to the next.

Working with Large Amounts of Data

Parameterized SQL is also useful when writing large amounts of text or binary data. Without using parameters, the entire text or binary value would have to be represented within the text of the SQL statement which is often impractical. Using parameters makes this situation easier as the SQL can include a parameter to which a descriptor holding the data can be bound.

When binding large amounts of binary or text data to parameters, there is a further important point to make. Binding a parameter causes an extra copy of the values to be stored in memory. If the value takes up a lot of memory then this extra copy deprives the system of memory. This does not present a problem during normal use, but the parameter should be unbound as soon as possible to release the memory back to the system.

You'll find that a common pitfall in this area is to prepare, bind and execute a statement and then reset it, assuming that this releases any resources. However, resetting a prepared statement does not unbind the parameters meaning that large parameter values continue to occupy memory unnecessarily. To rectify this, clients should re-bind applicable parameters to NULL immediately after resetting the prepared statement.

Returning Results from Queries

A prepared statement may return a result set from a query. The size of a result set can exceed the available memory on the device, so rather than assemble all result rows before passing them back to the client, the rows are returned one by one.

Just as with any other SQL statement, executing a SELECT statement involves running a bytecode program. As the program runs, it progressively produces result rows. When a row becomes available, the program breaks execution and returns control to the client. At this point, the client can use the API to extract any desired column values and take any further action. When the client has finished with the result row, it calls the API which causes the bytecode program to resume execution. This process is repeated for every result row until no more are left. The bytecode program then exits as normal, at which point the client can reset the prepared statement for later re-use. This sequence of events is shown in Figure 5.4.

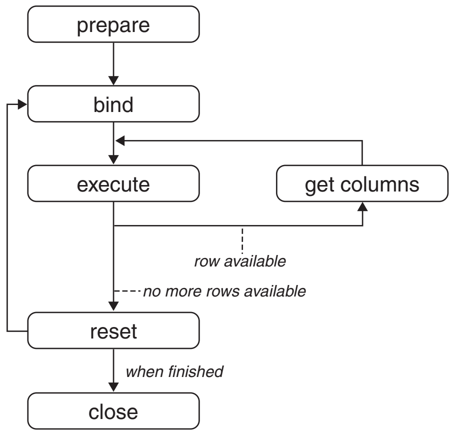


Figure 5.4 Events when executing a prepared statement that returns data

5.3 Multi-Client Access

The previous sections have explained execution of SQL from the perspective of a single client. However, in a real system, there can be more than one client actively accessing the same database. This section discusses some of the concepts that you need to be familiar with to cope with such situations.

5.3.1 Transactions

Chapter 4 introduced transactions as a mechanism for grouping related SQL statements. As a brief reminder, transactions have clear boundaries which are dictated by the database client. When a client wishes to start a transaction, it issues an explicit BEGIN statement. Every statement from that point is within the scope of the transaction until the transaction is confirmed with a COMMIT directive or abandoned with a ROLLBACK directive.

Transactions provide four important guarantees (usually called the ACID properties, from the initial letters):

- **Atomicity** – a transaction is indivisible. If any part of the transaction does not succeed then the entire transaction is abandoned. Thus if the last SQL statement in a transaction fails, the database engine ensures that the effect of any earlier statements in the transaction is rolled back so that the database content is as it was before the transaction began.
- **Consistency** – the net effect of a transaction must not transgress any of the constraints specified in the database schema. Individual component SQL statements do not fail if they contravene such constraints but the transaction commit fails. Thus, by the atomicity property, the entire transaction has no effect.
- **Isolation** – the transaction may not have any effect on the data in the database until the transaction is committed. Thus if a component SQL statement within a transaction modifies a row of a table, the changes must not be visible to other clients until the commit.
- **Durability** – once the commit has taken place any changes from the transaction are guaranteed to have been recorded in the database.

Some databases allow some of the guarantees to be relaxed. For example, isolation can sometimes be relaxed to improve concurrency and durability to improve performance. Symbian SQL offers an alternative isolation mode as discussed in Section 5.3.3.

How do these guarantees differ for SQL statements not executed within an explicit transaction? In fact, the guarantees are the same because every statement is executed within a transaction. If the client does not explicitly begin a transaction, the database runs in *autocommit* mode. In this mode, the database automatically begins a transaction when the client requests to execute an SQL statement and automatically commits the transaction before returning control to the client. Thus single statements have the same guarantees as more complicated transactions.

Initially, this may seem surprising but in fact it is necessary. Although a single SQL statement may appear simple, the underlying database schema may contain triggers which cause further operations to be carried out. If these were not wrapped within a transaction some of these operations could fail causing the database to be left in an inconsistent state. The implicit transaction means that if any triggered operations fail then all operations, including the user-specified SQL statement, are abandoned.

5.3.2 Connections

Before a client can use SQL to query or modify a database, it must first open a connection to it. A comparable state of affairs exists for a client wishing to read or write a file, where it first has to open a connection before it can perform any operations on the file. Nevertheless, we rarely think of this situation as opening a connection to the file, we think of it simply as opening the file itself, conveniently avoiding the notion of a connection. So why is it important to understand the concept of a connection in the case of a database but less important in the case of a file? The answer lies in the expectations for multi-client access.

When a client is dealing with a file, it is often assumed that the client has exclusive access to the file – meaning that no other clients are accessing it. File Server services can support multiple clients accessing the file at the same time but they provide very limited means for managing those clients, isolating them from one another and ensuring that each client has a consistent view of the data.

With databases, multi-client access is the norm and database engines are designed to mediate access by multiple clients. They incorporate strict rules which govern when a connection is allowed to access a database and what effect this has on other connections. If connections only wish to read the database, they cannot interfere with each other. However, if one connection were to be allowed to read while another is writing this could cause the reader to retrieve inconsistent data.

For example, if a client was mid-way through updating a row and the database engine allowed another client to read that same row before the update was complete then the second client would clearly be at risk of reading corrupt or incomplete data. On a larger scale, if a client were to be allowed to read updates from another client's transaction before all of the statements in that transaction had been executed then there would be a risk of the reading client getting an inconsistent view of the data.

5.3.3 Locking

Database engines use *locking* to prevent potential readers from accessing parts of the database which are being written. In practical terms, this means that a client connection trying to access part of the database must first obtain a lock pertaining to that part of the database. Predefined rules determine whether the lock can be granted or not. If the lock cannot be granted, then the client operations cannot proceed and have to wait until the lock is available. If the lock is granted, then the access is allowed to proceed but, according to predefined rules, other connections can be prevented from obtaining a lock until this client has finished its operations and the connection has relinquished the lock.

Acquiring and relinquishing locks is a process managed internally within the database engine. Clients do not have to explicitly attempt to obtain locks or give them up. The database engine automatically tries to acquire the necessary lock according to the operations being undertaken by the client. Similarly, locks are surrendered automatically when operations are concluded.

You need to understand the conditions which lead to a lock being requested as failure to acquire a lock may result in failure to execute the client's operation. Correspondingly, it is important to understand the point at which locks are released as this affects the ability of other clients to access the database.

Granularity of Locking

Locking systems can operate on the whole database, on a single table, or even on a single row – different database engines use different schemes. The least complex option for the database engine implementer is to lock at the database level. Under this system when a connection needs to write to a database and the database is not already locked, it is granted a lock which prevents other clients from either reading or writing any part of the entire database. This system is often provided by fast, lightweight database engines in environments which have low concurrency requirements. However, where multiple clients frequently compete for access to a database, this level of locking may be prohibitive as clients may have to wait a significant time to gain access. For a database supporting a multi-user interactive application, this may be unacceptable.

At the other end of the spectrum is row-level locking. A connection obtains a lock for each row to be modified. Other connections are still able to obtain locks on other rows of the database and hence can run unimpeded unless there is competition for the same row. This system allows high concurrency and is suitable in environments where many clients may need to access a database at the same time. However, it is a complex system to implement and is not commonly provided.

Between these extremes is table-level locking. Under this system, the database engine grants a client a lock on a table that it wishes to modify, but other clients may still access other tables in the database.

Symbian SQL Locking Specifics

Symbian SQL provides a hybrid of database and table locking. Write locking is at database granularity; reading is locked on a per-table basis. This means that only one client connection may be writing to the database at any one time. However, other connections may continue to read from tables that are not being modified by the writer.

When a client connection begins a transaction, no locks are initially taken. Locks are acquired based on the operations carried out within the transaction.

Every table in a Symbian SQL database has two locks – a READ lock and a WRITE lock. When a connection attempts to perform a read operation on a table, the database engine tests whether the WRITE lock for that table is active (for other connections). If this is the case, then an update on this table is in progress and reading is not permitted, so the client is denied access. If the table is not locked for writing, the connection is granted a shared READ lock and the read operation is allowed to proceed.

When a connection attempts to perform a write operation on a table the database engine tests whether there is already a write transaction active. If this is the case, then the client is denied access as only one write transaction may be active at any one time on a database. If there is no active write transaction then the database engine tests whether the READ lock for that table is active (for other connections). If it is, then the table has active readers and writing is not permitted, so the client is denied write access. If the table is not locked for reading, then the table WRITE lock is granted to the requesting connection and the write operation is allowed to proceed.

Connections can accumulate a number of different locks over the lifetime of a transaction as they may access a variety of tables. Once a connection is granted a READ or WRITE lock, it continues to hold it until the transaction is committed. For autocommit mode, this is the duration of the single SQL statement being executed. For explicit transactions, it is when a commit or rollback is performed.

Failing to Acquire a Lock

Both read and write operations can be rejected by the database due to inability to acquire the necessary lock. Read operations can be denied if the table is locked for writing by another connection. Write operations can be denied if the table is locked for reading by other connections.

If the operation is denied then a busy error is returned to the client. If the denied SQL statement is part of an explicit transaction then the transaction does not fail automatically. This means that the denied statement may be retried. Clients need to take care when retrying as continual waiting can easily lead to deadlock situations.

Read Uncommitted Mode

Isolation, described in Section 5.3.1, is an important property which ensures that changes to data made by a client in a transaction which has not yet been committed cannot be seen by other clients. However, the

Symbian SQL locking model guarantees this by preventing those other clients from reading any affected tables until the transaction commits. Guaranteeing full isolation therefore decreases potential concurrency.

Some clients may prefer to be able to see uncommitted writes in order to improve concurrency and hence gain greater application responsiveness. Symbian SQL provides an alternative isolation mode that allows clients to opt for this. The isolation mode is set on a per-connection basis which means that one client's choice of isolation mode has no effect on other clients. By default, connections are set to full isolation – a mode called *serializable*. Clients can choose *read uncommitted* mode if appropriate. However, you need to understand that the client may be reading data which has yet to be committed – and potentially may never be committed (if the transaction fails or is rolled back).

5.3.4 Connections, Transactions and Locking

Symbian SQL views each connection as the representation of a client. Thus when a client executes a BEGIN statement on a connection, it is the connection that the database engine views as owning the transaction. Similarly, locks are acquired on behalf of a transaction, hence it is the connection that owns these locks. Locks are discarded when the associated transaction is closed.

All statements executed on a connection which has an open transaction are considered part of the transaction, regardless of whether the statements read or write the database. Read statements on tables being actively modified by the transaction are allowed (as the connection holds the locks), whereas locking rules prevent other connections from reading these tables.

The connection presents a consistent view of the data. Thus reads within the transaction are affected by uncommitted writes made earlier in the transaction. This means that you need to beware how clients use data that is read during write transactions.

A potential pitfall of this is as follows. Say a client opens a transaction and makes some updates. Then, before the transaction has been committed, there is a need to read some data from the database. If the client attempts to commit the transaction and it fails, those updates are rolled back. If the client has cached the data it read during the transaction or has taken other actions based on that data, there is a danger that the application will behave inconsistently with the actual data in the database.

If a client wishes to execute two concurrent but independent sets of SQL statements – for example to build up a series of writes within a transaction, but at the same time execute independent writes which are outside the scope of the transaction – then the client needs to use two open connections. Although both connections pertain to the same client, locks are acquired on a per-connection basis, meaning that the connections are bound by the locking rules described in Section 5.3.3.

5.3.5 Prepared Statements and Locking

Pitfalls similar to those mentioned earlier can be simple to spot when dealing purely with database connections. However, more subtle problems can arise when prepared statements are brought into the picture. Nevertheless, the rules are the same. You need to remember that the prepared statement is not the transaction or locking context – the connection is.

Each prepared statement is associated with a single database connection when it is created, but a database connection may be associated with several prepared statements. When a prepared statement is executed, it acquires any locks needed to carry out the operation. However, the locks are owned by the connection, not the prepared statement. Thus locking does not prevent other actions on the same connection (including those by other prepared statements) from interfering with operations on the prepared statement. Similarly, a prepared statement is not protected from reading uncommitted writes made on the same connection (including those by other prepared statements).

In certain situations, active prepared statements on a connection can prevent a transaction commit from succeeding on that connection. As a general rule, it is best to avoid having active prepared statements when trying to commit a transaction.

The conclusion of this section is that if applications need to use multiple concurrent prepared statements on the same connection then care needs to be taken to ensure that they do not interfere with each other.

5.4 Summary

This chapter has introduced some important concepts that you need to be familiar with in order to use a database proficiently. We examined the types of database supported by Symbian SQL and explained how these provide different levels of security and privacy. We then looked at practical execution models for SQL. These models translate directly to the APIs that you meet in Chapter 6. Finally we looked at some of the complications created by multiple clients needing concurrent access to the same database. Having familiarized you with such concepts, you are now ready to explore the Symbian SQL APIs in detail.

6
Using Symbian SQL

Armed with an understanding of the key concepts described in previous chapters, you should now be ready to dive in and look at how to use Symbian SQL in your app. This chapter discusses the key classes – RSqlDatabase and RSqlStatement – and covers supporting and utility classes. All APIs discussed in this chapter are declared in the Symbian SQL header file sqldb.h.

6.1 The Database Connection Class

The RSqlDatabase class provides key APIs for managing databases. In this section, we discuss how to connect to a database, create and delete databases, attach multiple databases to a single connection, execute SQL, and set and retrieve database parameters.

6.1.1 Connecting and Disconnecting

In order to access a database, you must first establish a connection to it. When you have finished accessing the database, you must then close the connection. Symbian provides the RSqlDatabase class to represent a database connection. You open a connection to a database by calling the Open() API on the connection object. Two variants of this API are provided, one leaving and one not:

```
TInt Open(const TDesC& aDbFileName, const TDesC8* aConfig=NULL);
void OpenL(const TDesC& aDbFileName, const TDesC8* aConfig=NULL);
```

The argument aDbFileName is the name of the database. If the filename is not valid or the file does not exist, the API returns an error. It is also an error if the connection is open already. The argument aConfig is an optional text descriptor which may contain configuration options. The formats of database filenames and configuration strings are covered in the following two sections.

To close the database connection, you call its `Close()` API, which is declared as follows:

```
void Close();
```

On return, the connection has been closed but the connection object still exists. The connection object may be reused to connect to the same or any other database with a new call to `Open()`.

Database Configuration

Several APIs take a configuration string as an argument. A configuration string consists of one or more attribute–value pairs of this form:

```
attribute=value
```

Each pair should be separated by a semicolon. The recognized attributes are described in Table 6.1.

Here is an example of a valid configuration string:

```
page_size=1024;encoding=UTF-16
```

Database Filename Conventions

The format of a database filename is dependent on whether the database is of the public, private, or secure shared variety. An incorrect database filename format results in the `Open()` or `Create()` call failing with an error.

For public and private databases, clients must specify the absolute path to the database file:

```
<drive>:\<path><database-name>
```

For example, for a public database:

```
E:\myapp\mydatabase
```

Table 6.1 Database configuration attributes

Attribute	Value	Purpose	Notes
`page_size`	One of 512, 1024, 2048, 4096, 8192, 16,384, or 32,768	The physical database file is split into equal-sized pages. Individual pages are then loaded and saved by the database. This parameter sets the size of a page in bytes.	The page size can be a way of tuning database performance. This attribute may only be specified when the database is created.
`cache_size`	A positive integer; the maximum may not be reached if resources are not available	This parameter sets the maximum number of database pages that the database engine is allowed to cache.	From Symbian^3, you are recommended not to specify this parameter as Symbian SQL automatically manages caching.
`encoding`	One of 'UTF-8' or 'UTF-16'	This parameter sets the physical encoding format used to store text in a database file.	If the database will be used to store text mainly in western alphabets, you should use UTF-8; otherwise, use UTF-16. This attribute may only be specified when the database is created.
`compaction`	One of 'background', 'synchronous' or 'manual'	This parameter sets the compaction mode (see Section 6.1.6).	This attribute may only be specified when the database is created.

A private database of an application with SID 0x12345678 would be specified as:

```
E:\private\12345678\mydatabase
```

Secure shared databases are located in a private area of the file system accessible only by Symbian SQL, so a full path is not applicable for this type of database. Additionally, secure shared databases incorporate an anti-spoofing mechanism based on the creator SID (described in Chapter 5). Thus secure shared database filenames follow this pattern:

```
<drive>:[<SID>]<database-name>
```

Secure shared databases may be created on any drive, thus a drive letter must be specified. This must be followed by the SID of the process attempting to create the database. The SID must be delimited by square brackets. The SID is followed by a unique name for the database. For example, for an application with SID 0x12345678:

```
C:[12345678]mydata
```

6.1.2 Creating and Deleting Databases

We have seen how to access an existing database, but we also need to know how to create one in the first place. The `Create()` APIs of `RSqlDatabase` are declared as follows:

```
TInt Create(const TDesC& aDbFileName, const TDesC8* aConfig=NULL);
void CreateL(const TDesC& aDbFileName, const TDesC8* aConfig=NULL);

TInt Create(const TDesC& aDbFileName,
            const RSqlSecurityPolicy& aSecurityPolicy,
            const TDesC8* aConfig=NULL);
void CreateL(const TDesC& aDbFileName,
             const RSqlSecurityPolicy& aSecurityPolicy,
             const TDesC8* aConfig=NULL);
```

The first pair of APIs create public or private databases; one version may leave while the other may not. They take similar arguments to the `Open()` API. The argument `aDbFileName` is the full pathname of the database you want to create and `aConfig` is a configuration string. If the pathname is invalid or a database of that name already exists, the API call returns an error.

The second pair of APIs is used to create secure shared databases and thus take an additional argument, a security policy. Security policies (Section 6.5) specify the criteria that potential clients must meet to be allowed to access the database and are represented by the `RSqlSecurityPolicy` class.

If the API returns successfully, you have created an empty database and the connection is open and ready for use. The database doesn't contain data, but it exists in the file system.

6.1.3 Attaching Additional Databases

The Symbian SQL API allows a database connection to be associated with more than one database. The database specified during the initial `Open()` or `Create()` call is regarded as the main database, but additional auxiliary databases may then be attached. After it has been attached, the contents of a database are available across the connection in the same way as the main database. The `Attach()` API is declared as follows:

```
TInt Attach(const TDesC& aDbFileName, const TDesC& aDbName);
```

The first argument, `aDbFileName`, is the filename of the database to be attached. It follows the conventions discussed in Section 6.1.1. The second argument, `aDbName`, is a logical name for the database which may be used in SQL queries to reference objects within the attached database. This is necessary if the name of a table in an attached database is the same as one in the main database (or another attached database). For example, if a database is attached with the logical name `db2`, then a table called `tbl`, may be referenced as `db2.tbl` in SQL statements. Although unnecessary, tables in the main database may be prefixed with the logical database name `main` if desired.

`Attach()` may only be called on a database connection that is already open and fails with an error if this condition is not met. A maximum of 10 databases may be attached to a connection (excluding the main database). `Attach()` does not accept a configuration string argument.

You reverse the attachment of a database by calling the `Detach()` API:

```
TInt Detach(const TDesC& aDbName);
```

It takes one argument, `aDbName`, which is the logical name of the database specified in the call to `Attach()`. If the name specified does not match that of an attached database, `Detach()` returns `KErrNotFound`.

6.1.4 Executing SQL

Chapter 5 described the one-shot method of executing SQL statements. In the API, this purpose is realized by the `Exec()` API in the `RSqlDatabase` class:

```
TInt Exec(const TDesC& aSqlStmt);
TInt Exec(const TDesC8& aSqlStmt);
```

The argument `aSqlStmt` is a descriptor containing the SQL which you want to execute. Although SQL keywords can always be described using 8-bit characters, text column values may require 16-bit characters. For this reason both 8- and 16-bit variants of the `Exec()` API are provided. The `aSqlStmt` argument may contain any valid SQL. However, as described in Chapter 5, one-shot execution is unable to return a result set. Thus, a SELECT statement does not return any records. A mechanism for executing SQL statements that return results is explored in Section 6.2.

Some SQL statements may be long-running. Thus asynchronous variants of `Exec()` are provided in order to allow clients to continue to remain responsive while their database operations are being processed:

```
void Exec(const TDesC& aSqlStmt, TRequestStatus& aStatus);
void Exec(const TDesC8& aSqlStmt, TRequestStatus& aStatus);
```

The `Exec()` API is commonly used to insert rows into the database. From Symbian^3, `RSqlDatabase` provides the `LastInsertedRowId()` API, which returns the ROWID of the most recent successful insert into the database using this connection. A value of zero is returned if there have been no inserts on this connection. A negative value represents an error.

6.1.5 Copying and Deleting a Database File

`RSqlDatabase` provides an API to allow databases to be copied:

```
static TInt Copy(const TDesC& aSrcDbName,
                 const TDesC& aDstDbName);
```

`aSrcDbName` and `aDstDbName` are the source and destination filenames respectively and must meet the format specifications described in Section 6.1.1.

Only public and secure shared databases may be copied. Private databases may not be copied using this API. However, clients may make a copy of a private database using the standard File Server APIs, provided that there are no open connections to that database.

There are no restrictions on the copying of public databases. The copy may be placed on any publicly accessible area of the file system. If the client wishes to copy a public database to their private directory then the standard File Server APIs must be used.

The `Copy()` API is most useful when dealing with secure shared databases, as clients cannot copy them using the File Server. A secure shared database cannot be copied to a public location – the destination filename must also qualify as a secure shared database. Only the database creator (identified by the SID embedded in the database name) is allowed to copy the database. Other clients attempting this get the error `KErrPermissionDenied`.

Databases may also be deleted. This means removing the physical file from the file system, not just deleting the data it contains. The `RSqlDatabase` class provides a `Delete()` API which is declared as follows:

```
static TInt Delete(const TDesC& aDbFileName);
```

The `Copy()` and `Delete()` APIs are static member functions of the `RSqlDatabase` class. Thus, clients do not need to open a database connection in order to use the functionality.

6.1.6 Managing Disk Usage

As data is inserted throughout the lifetime of a database, the database file must expand. However, the reverse does not necessarily have to occur: if data is deleted from a database then the database file does not have to shrink in size. If the file is not made to shrink, deletion of table rows from the database leads to pockets of free space appearing within it. If more data is later inserted, the database engine re-uses this free space. However, if more deletion than insertion occurs over time, the database file accumulates free space. This section describes `RSqlDatabase` APIs that relate to this topic.

Retrieving the Database File Size and Free Space

The `Size()` API enables clients to retrieve both the size of the database on disk and the amount of free space it contains.

```
TInt Size(RSqlDatabase::TSize& aSize,
                                    const TDesC& aDbName = KNullDesC) const;
```

After a call to `Size()`, the argument `aSize` contains the database file size and the amount of free space, both in bytes. It is a structure of type `TSize` declared as follows:

```
struct TSize
  {
  /* The database size in bytes */
  TInt64  iSize;
  /* The database free space in bytes */
  TInt64  iFree;
  };
```

The second argument, aDbName, is the logical name of the database to be examined. It is optional and is only required when the client wishes to obtain information on an attached database.

On Symbian^2 and earlier, this API is not available. However, the following API may be used:

```
TInt Size() const;
```

If successful, the return value is the size of the database file. If an error occurs then a negative error code is returned.

This API is still available on Symbian^3 and above. However, clients should be aware that if the database size exceeds 2 GB, the error code KErrTooBig is returned.

Compacting a Database

Compaction is the process of shrinking the database file by removing free space. Without compaction, a database can occupy more disk space than is necessary because SQL statements such as DELETE logically remove data from the database but do not automatically free the file space which that data occupied. Prior to Symbian^3, compaction was carried out automatically at the end of each transaction. However, while this maximized disk space efficiency, it came at a performance cost. From Symbian^3 and above, a database may be created in one of three compaction modes: background, synchronous or manual.

Background is the recommended compaction mode and databases are created with this mode as a default. In background mode, no client intervention is required – a database is compacted automatically by the server. The process is triggered when Symbian SQL is not busy and it detects that free space in the file has exceeded a pre-defined threshold. Applications may call the Compact() API to force compaction at other times if desired. Background is the recommended mode as it requires no intervention by the client and is managed by the server according to load.

A database created with *manual* compaction mode is never automatically compacted. Database clients must request compaction themselves by calling the Compact() API:

```
TInt Compact(TInt64 aSize, const TDesC& aDbName = KNullDesC);
void Compact(TInt64 aSize, TRequestStatus& aStatus,
                  const TDesC& aDbName = KNullDesC);
```

The aSize argument indicates the maximum amount of space in bytes to be freed by the Compact() call. This allows manual mode clients to limit compaction to steps of a certain size to amortize the performance cost. The optional `aDbName` argument is the logical name of the database to be compacted; it defaults to the main database on the connection if the argument is not supplied. Both synchronous and asynchronous variants of the API are provided.

Clients using manual mode compaction can use the `Size()` API to measure the amount of free space in the database file in order to trigger manual compaction. Manual mode is not recommended unless there are special requirements for compaction behavior.

Synchronous mode is provided for behavioral compatibility with Symbian^2 and earlier. In this mode, a database is automatically compacted as part of every transaction commit operation. In synchronous mode, calls to the `Compact()` API have no effect. However, the API does not return an error if it is called. Databases created on Symbian^2 and earlier are automatically set to synchronous mode if transferred to a device running Symbian^3 or above.

The compaction mode of a database is set when it is created and may not be changed. Database creators may override the default setting by specifying another compaction mode in the configuration string supplied to the `Create()` API.

Managing Low Disk Space

In situations where disk space is running low, users may attempt to delete records to free up space. However, due to the transactional guarantees that the database must provide, deletion of data temporarily requires more disk space to hold data during the transaction commit. If the disk is nearly full, this additional space may not be available which can make it impossible to delete data from the database. Symbian SQL provides a set of APIs aimed at alleviating low disk space scenarios:

```
TInt ReserveDriveSpace(TInt aSize);
void FreeReservedSpace();
TInt GetReserveAccess();
void ReleaseReserveAccess();
```

Early in the lifetime of your database connection, when disk space is plentiful, you call the `ReserveDriveSpace()` API. The effect of this is to reserve some disk space from the File Server for use later.

If a time comes when access to the reserved space is needed you may seek access to it by calling `GetReserveAccess()`. If it returns success, the reserved disk space has now been made available so a DELETE statement which previously failed may now have enough disk space to succeed. This operation and a subsequent compaction should reduce the size of your database.

When you have successfully deleted data, you give up access to the reserved space by calling the `ReleaseReserveAccess()` API. The disk space remains reserved and may be re-accessed later with `GetReserveAccess()` while the database connection is still active.

When the database connection is closed the space is released automatically. If the reserved space is no longer required, but the database is to be kept open, the client can free the reserved space by calling the `FreeReservedSpace()` API.

6.1.7 Transaction APIs

The `RSqlDatabase` class provides an API to discover whether the database connection has an active transaction:

```
TBool InTransaction() const;
```

This method returns true if the database has an active transaction.

As discussed in Chapter 5, Symbian SQL provides two isolation levels – Serializable (the default) and Read Uncommitted. Isolation level is set using the `SetIsolationLevel()` API on a per-connection basis, which means that each database client can decide if they want to be able to read data before it is committed:

```
TInt SetIsolationLevel(TIsolationLevel aIsolationLevel);
```

The choices for the `aIsolationLevel` argument are:

- `EReadUncommitted`: A client can read any writes contained within an uncommitted transaction open on another connection.
- `ESerializable`: A client may not read any writes contained within an uncommitted transaction open on another connection. Thus if a client attempts to read a table that is being modified by another client, the operation fails.

A Read Uncommitted isolation level is normally used for performance reasons, since the reader does not have to wait for the transaction to complete before being able to read the data. However, clients must bear the risk that the uncommitted data could be rolled back, potentially resulting in the reader having invalid data. Additionally there is a risk that the client may read inconsistent data, as the full set of updates in the transaction may not yet have been applied.

6.1.8 Retrieving the Security Policy

The `RSqlDatabase` class includes APIs to retrieve the security policy of a secure shared database:

```
TInt GetSecurityPolicy(RSqlSecurityPolicy& aSecurityPolicy) const;
void GetSecurityPolicyL(RSqlSecurityPolicy& aSecurityPolicy) const;
```

There is no equivalent API to set the policy since it is set up when the database is created and cannot be changed.

6.2 The Prepared Statement Class

We have now covered database and connection management APIs and how to execute basic SQL statements. Our next topic is more advanced SQL, using classes written for the purpose.

We saw that the `Exec()` API of an `RSqlDatabase` object is not suitable for executing a SELECT statement which returns a data set. To process queries of that type, you need to use `RSqlStatement`.

The `RSqlStatement` class corresponds to the prepared statement concept described in Chapter 5.

6.2.1 Preparing and Executing SQL Statements

Prepared statements allow an SQL statement to be prepared for execution and then used many times. While this may also be accomplished using `RSqlDatabase::Exec()`, prepared statements provide increased efficiency. The example below shows the basic use of a prepared statement:

```
// Database connection 'db' is already open
RSqlStatement stmt;
_LIT(KSqlString,"DELETE FROM contacts");

// Get the statement ready for execution
TInt err = stmt.Prepare(db, KSqlString);
```

```
// Execute the prepared statement
if(!err)
  err = stmt.Exec();

// Reset the prepared statement so it is ready for re-execution
if(!err)
  err = stmt.Reset();

// Remainder of program including re-use of the prepared statement.

// Now we no longer need the prepared statement
stmt.Close();
```

The example begins after an `RSqlDatabase` connection object, `db`, has been opened. We declare an `RSqlStatement` object. At this point, the prepared statement object is not associated with any SQL statement so cannot be executed.

The `Prepare()` API is called, specifying the SQL to be executed. On exit, the SQL has been compiled into bytecode and the prepared statement is ready for execution. There are two groups of `Prepare()` APIs; one accepts 16-bit descriptors, the other 8-bit descriptors. Both leaving and non-leaving variants are provided:

```
TInt Prepare(RSqlDatabase& aDatabase, const TDesC& aSqlStmt);
void PrepareL(RSqlDatabase& aDatabase, const TDesC& aSqlStmt);

TInt Prepare(RSqlDatabase& aDatabase, const TDesC8& aSqlStmt);
void PrepareL(RSqlDatabase& aDatabase, const TDesC8& aSqlStmt);
```

The `Exec()` API executes the prepared statement. An asynchronous version of this API is provided to enable applications to remain responsive during long-running SQL operations:

```
TInt Exec();
void Exec(TRequestStatus& aStatus);
```

If there is a need to re-use the prepared statement it cannot be immediately re-executed – the execution context within the database must first be reset using the `Reset()` API:

```
TInt Reset();
```

If re-use is not required then there is no need to call `Reset()`. When the application no longer requires the prepared statement, it can be closed to free system resources. The `Close()` API does not return an error code:

```
void Close();
```

6.2.2 Retrieving Column Values

For database queries, we carry out similar steps but additionally we need to extract values from the result set. Here is a simple example:

```
// Database connection 'db' is already open
RSqlStatement stmt;
_LIT(KSqlString,"SELECT id FROM contacts WHERE name LIKE 'J%'");

TInt err = stmt.Prepare(db, KSqlString);
if(!err)
  while((err = stmt.Next()) == KSqlAtRow)
    {
    TInt val = stmt.ColumnInt(0); // get value of 'id'
    // Do something with the result
    }

if(err == KSqlAtEnd)
  // All result rows have been processed
else
  // Handle the error

err = stmt.Reset();  // Only required if 'stmt' will be re-used

stmt.Close();  // When statement is no longer needed for re-use
```

In common with the preceding example, we call `Prepare()` before executing the statement and `Reset()` and `Close()` after execution. However, in place of a call to `Exec()` we have a `while` loop using the `Next()` and `ColumnInt()` APIs.

The first call to `Next()` begins execution of the prepared statement. Control is returned to the caller either when a result row has been generated or when there are no more result rows. The declaration of `Next()` is:

```
TInt Next();
```

If a result row is found, then the value `KSqlAtRow` is returned. After extracting the required column values, the client should call `Next()` again to generate the next result row. In the example, the loop repeats until `Next()` fails to return `KSqlAtRow`. This happens when no more result rows are available, at which point `Next()` returns the value `KSqlAtEnd`. If an error occurs, `Next()` returns an error code.

Extracting Column Values

When `Next()` indicates that a result row has been generated, the client can use the column extraction APIs to retrieve the values:

```
TInt ColumnInt(TInt aColumnIndex) const;
TInt64 ColumnInt64(TInt aColumnIndex) const;
TReal ColumnReal(TInt aColumnIndex) const;
TPtrC ColumnTextL(TInt aColumnIndex) const;
TInt ColumnText(TInt aColumnIndex, TPtrC& aPtr) const;
TInt ColumnText(TInt aColumnIndex, TDes& aDest) const;
TPtrC8 ColumnBinaryL(TInt aColumnIndex) const;
TInt ColumnBinary(TInt aColumnIndex, TPtrC8& aPtr) const;
TInt ColumnBinary(TInt aColumnIndex, TDes8& aDest) const;
TBool IsNull(TInt aColumnIndex) const;
```

Values may be extracted as 32- or 64-bit signed integers, as floating point values, or as 16-bit or 8-bit wide descriptors. The Boolean `IsNull()` allows clients to differentiate between an intended value zero and a NULL value.

When retrieving text or binary columns multiple API variants are available. If the client has pre-allocated memory available to store the data, then the `TDes` variant may be used. If not, then either the leaving or non-leaving `TPtrC` variant may be used.

All column extraction APIs take an `aColumnIndex` argument which is the zero-based index of the result column from which to retrieve the value. For example, consider this SQL statement:

```
SELECT id, name FROM contacts
```

Here `id` has a column index of zero and `name` an index of one. When many columns are specified in the SELECT statement, it may be error-prone to hardcode column numbers. To alleviate this, the `ColumnIndex()` API enables applications to programmatically derive the column index for a named column. The reverse translation is also possible, to retrieve the name of a column from an index:

```
TInt ColumnIndex(const TDesC& aColumnName) const;
TInt ColumnName(TInt aColumnIndex, TPtrC& aNameDest);
```

Typically, the application knows the data type for a particular column value. If it does not, the data type of a column can be obtained using the `ColumnType()` API described later in this chapter. If you use

Table 6.2 Conversions attempted by column extraction APIs

Data Type	`Column-Int()`	`Column-Int64()`	`Column-Real()`	`Column-Text()`	`Column-Binary()`
`NULL`	0	0	0.0	`KNullDesC`	`KNullDesC8`
Int	**Int**	Int64	Real	`KNullDesC`	`KNullDesC8`
Int64	Clamp	**Int64**	Real	`KNullDesC`	`KNullDesC8`
Real	Round	Round	**Real**	`KNullDesC`	`KNullDesC8`
Text	0	0	0.0	**Text**	`KNullDesC8`
Binary	0	0	0.0	`KNullDesC`	**Binary**

a column extraction API that does not match the data type returned by `ColumnType()` then an automatic conversion is attempted where appropriate. Table 6.2 summarizes the conversions.

'Clamp' means that the API returns `KMinTInt` or `KMaxTInt` if the value is outside the range that can be represented by the type returned by the extraction API. 'Round' means that the floating point value is rounded to the nearest integer.

It is important to know that you may only call a column extraction API after the value returned by a call to `Next()` has indicated that a result row is available; if there is no result row, a panic is raised. This is true even when you read the first row of a result set as the virtual pointer is not initialized.

Ancillary APIs

`RSqlStatement` provides a number of APIs to return information about column values:

```
TInt ColumnCount() const;
TInt DeclaredColumnType(TInt aColumnIndex,
       TSqlColumnType& aColumnType) const;
TSqlColumnType ColumnType(TInt aColumnIndex) const;
TInt ColumnSize(TInt aColumnIndex) const;
```

`ColumnCount()` returns the number of columns generated by the statement. This number is the same for all rows and equates to the number of entries in the column list of the SELECT statement.

DeclaredColumnType() returns the column type as declared in the database schema. The result is of type TSqlColumnType which is declared as follows:

```
enum TSqlColumnType
  {
  ESqlNull,
  ESqlInt,
  ESqlInt64,
  ESqlReal,
  ESqlText,
  ESqlBinary
  };
```

The API may return an error, so the output is returned in aColumn-Type.

SQLite's manifest typing system allows data types which do not match the declared column type to be inserted. Thus the data types of column values may sometimes not match the declared type. The ColumnType() API allows clients to retrieve the data type of the column value, as opposed to the declared type.

The ColumnSize() API returns the size of a column value and corresponds to the data type (see Table 6.3).

Table 6.3 Size of columns

Column type	**Column size**
ESqlInt	4
ESqlInt64	8
ESqlReal	8
ESqlText	The number of characters in the Unicode string
ESqlBinary	The byte length of the binary data
ESqlNull	0

This API is particularly useful when retrieving from binary or text columns, as clients can determine whether the data will fit in existing buffers.

All of the above APIs require that the return value from Next() has indicated a result row is available.

6.2.3 Parameterized Queries

You may often need to execute several SQL statements that differ only in terms of the column values or search criteria specified. For instance typical queries might look like this:

```
SELECT name FROM employee WHERE salary > 10000
SELECT name FROM employee WHERE salary > 15000
```

Here the only difference between the two queries is in the search criteria: `salary > 10000` and `salary > 15000`. Symbian SQL supports a system of *parameters* to allow applications to prepare a single query to support both cases. Parameters can be identified by specifying a name for the parameter prefixed by a colon, so we write, for instance:

```
SELECT name FROM employee WHERE salary > :salarylevel
```

This query can now be passed to the `Prepare()` API. Before you can execute it, you have to assign an actual value to the parameter `salarylevel`. This is called 'binding' the value to the parameter and you do it by calling one of the `BindXXX()` APIs of the statement object, appropriate to the data type being bound. The declarations are:

```
TInt BindNull(TInt aParameterIndex);
TInt BindInt(TInt aParameterIndex, TInt aParameterValue);
TInt BindInt64(TInt aParameterIndex, TInt64 aParameterValue);
TInt BindReal(TInt aParameterIndex, TReal aParameterValue);
TInt BindText(TInt aParameterIndex, const TDesC& aParameterText);
TInt BindBinary(TInt aParameterIndex, const TDesC8& aParameterData);
TInt BindZeroBlob(TInt aParameterIndex, TInt aBlobSize);
```

The `aParameterValue` argument of these APIs is the value you are binding (not required for `BindNull()`). The `aParameterIndex` argument is the index of the parameter, the leftmost parameter having an index of 1. However there is no need to work this out manually: `RSqlStatement` supplies an API which does it for you; `ParameterIndex()` has the following declaration:

```
TInt ParameterIndex(const TDesC& aParameterName) const;
```

Putting this all together into an example, we have:

```
// Database connection 'db' is already open
RSqlStatement stmt;
_LIT(KSqlString,
     "SELECT name FROM employee WHERE salary > :salarylevel");
_LIT(KParamName, ":salarylevel");

TInt err = stmt.Prepare(db, KSqlString);
if(err==KErrNone)
```

```
  {
  const TInt KParamIndex = stmt.ParameterIndex(KParamName);
  if(KParamIndex>0)
    {
    err = stmt.BindInt(KParamIndex, 10000);
    if(err==KErrNone)
      while((err = stmt.Next()) == KSqlAtRow)
        {
        // Process data
```

6.3 Working with Variable-Length Data Objects

Text and binary column values may vary in length, from a single character (byte) to many megabytes. Working with such data presents its own set of problems. Symbian SQL provides a number of different APIs to help developers achieve their objectives.

6.3.1 Retrieving Data

It is possible to obtain binary and text column values in the same way as other types of value. The caller has to supply a variable into which the column value is copied:

```
TInt ColumnBinary(TInt aColumnIndex, TDes8& aDest) const;
TInt ColumnText(TInt aColumnIndex, TDes& aDest) const;
```

The binary and text extraction APIs are more complex to use than other types. The client must ensure that the destination descriptor is large enough to accommodate the column value. Unless the client has prior knowledge of the length of the binary or text string, this means calling the `ColumnSize()` API. If the destination descriptor is not large enough then only as much data as fits is copied and `KErrOverflow` is returned to the caller.

To help reduce this complexity, Symbian SQL supplies further variants:

```
TInt ColumnText(TInt aColumnIndex, TPtrC& aPtr) const;
TInt ColumnBinary(TInt aColumnIndex, TPtrC8& aPtr) const;

TPtrC ColumnTextL(TInt aColumnIndex) const;
TPtrC8 ColumnBinaryL(TInt aColumnIndex) const;
```

Using these APIs, Symbian SQL automatically allocates enough memory to hold the value and returns a pointer descriptor. However, the value is only available until the prepared statement is moved to the next row (or is closed or reset), at which point the memory is freed.

6.3.2 Retrieving Large Amounts of Data

The column extraction APIs provide a simple way to retrieve values. However, when dealing with large amounts of data they are memory inefficient. Symbian^3 introduced new APIs with improved efficiency:

```
class TSqlBlob
  {
public:
  static HBufC8* GetLC(RSqlDatabase& aDb,
                       const TDesC& aTableName,
                       const TDesC& aColumnName,
                       TInt64 aRowId = KSqlLastInsertedRowId,
                       const TDesC& aDbName = KNullDesC);

  static TInt Get(RSqlDatabase& aDb,
                  const TDesC& aTableName,
                  const TDesC& aColumnName,
                  TDes8& aBuffer,
                  TInt64 aRowId = KSqlLastInsertedRowId,
                  const TDesC& aDbName = KNullDesC);
  };
```

Two variants are provided. The first API allocates a descriptor to which the column value is copied. The client is responsible for freeing the descriptor when it is no longer needed. The second copies the value to an existing descriptor passed in by the client.

Usage of these APIs differs significantly from the column extractors – the `TSqlBlob` APIs are not associated with a prepared statement. Whereas the column extraction APIs may only access the current row of a result set, the `TSqlBlob` APIs use a direct addressing scheme where clients specify the table, row and column from which to retrieve the data.

The row is identified using its unique ROWID (see Section 4.7), thus clients must possess this identifier before they can retrieve the desired data. This complicates the use of the `TSqlBlob` APIs, but nevertheless their use is strongly recommended for long binary or text data as it is far more memory efficient.

Both API variants take two optional arguments. If the `aRowId` argument is not specified then it defaults to the id of the last inserted row. The `aDbName` parameter can be used if the table to be accessed lies in an attached database. If a value is not specified then the main database is assumed.

The following example code excerpts retrieve binary data. The first uses column extraction APIs to retrieve a name and thumbnail photo from a `phonebook` table:

```
// Database connection 'db' is already open
RSqlStatement stmt;
```

```
_LIT(KSqlString,"SELECT name, photo FROM phonebook
                           WHERE name LIKE 'S%'");

TInt err = stmt.Prepare(db, KSqlString);
if(!err)
  while((err = stmt.Next()) == KSqlAtRow)
    {
    TPtrC name = stmt.ColumnTextL(0);
    TPtrC8 photo = stmt.ColumnBinaryL(1);
    // Do something with the result
    }
```

The second example uses the `TSqlBlob` API to retrieve the photo. It still uses a column extractor to retrieve the name as it is unlikely to be large (perhaps a constraint on this exists in the database design):

```
// Database connection 'db' is already open
RSqlStatement stmt;
_LIT(KSqlString,"SELECT rowid, name FROM phonebook
                           WHERE name LIKE 'S%'");
_LIT(KPhonebookTable, "phonebook");
_LIT(KPhotoColumn, "photo");

TInt err = stmt.Prepare(db, KSqlString);
if(!err)
  while((err = stmt.Next()) == KSqlAtRow)
    {
    TPtrC name = stmt.ColumnTextL(1);
    TInt64 rowid = stmt.ColumnInt64(0);
    HBufC8* photo =
        TSqlBlob::GetLC(db, KPhonebookTable, KPhotoColumn, rowid);
  // Do something with the result
    CleanupStack::PopAndDestroy(photo);
    }
```

6.3.3 Streaming Data

Symbian SQL offers the ability for clients to stream binary data stored in a column, rather than retrieving the column value all at once. This could be useful in a number of situations, for example a video clip stored in a database can be streamed rather than being retrieved as a single block thereby saving RAM.

In Symbian^3, streaming APIs are provided in the `RSqlBlobRead-Stream` class:

```
class RSqlBlobReadStream : public RReadStream
  {
public:
  void OpenL(RSqlDatabase& aDb,
             const TDesC& aTableName,
             const TDesC& aColumnName,
             TInt64 aRowId = KSqlLastInsertedRowId,
             const TDesC& aDbName = KNullDesC);

  TInt SizeL();
  };
```

The class is derived from the class `RReadStream` making it compatible with the standard `Store` framework which is used for streaming on the Symbian platform.

Clients use the `OpenL()` API to open the stream based on the table, row and column provided. As with the `TSqlBlob` class discussed in the previous section, `RSqlBlobReadStream` uses a direct addressing scheme which requires the caller to know the ROWID of the row to be addressed. The `SizeL()` API returns the overall length of the binary data.

The following example omits the opening of the database and preparation of the SQL statement:

```
// Prepared statement 'stmt' has already been prepared.
// The SELECT statement has ROWID as the first result column.

while((err = stmt.Next()) == KSqlAtRow)
  {
  TInt64 rowid = stmt.ColumnInt64(0);

  RSqlBlobReadStream rdStrm;
  CleanupClosePushL(rdStrm);
  rdStrm.OpenL(db, KTableName, KColumnName, rowid);

  HBufC8* buffer = HBufC8::NewLC(KBlockSize);
  TPtr8 bufPtr(buffer->Des());
  TInt size = rdStrm.SizeL();
  while(size)
    {
    TInt bytesToRead = (size >= KBlockSize) ? KBlockSize : size ;
    rdStrm.ReadL(bufPtr, bytesToRead); // read the next block of data
    // do something with the block of data

    size =- bytesToRead;
    }
  CleanupStack::PopAndDestroy(buffer);
  }
```

In this example, the client has selected a number of rows, each of which contains binary data to be streamed. An instance of `RSqlBlob-ReadStream` is opened for the given table, column and row. The developer has decided to stream the data in blocks of `KBlockSize` bytes. The choice of `KBlockSize` would depend on the application of the data. Blocks are then streamed using the `ReadL()` API and processed until there is no more data left. As with any other `Store`-based stream, the C++ operators << and >> can also be used.

`RSqlBlobReadStream` was introduced in Symbian^3 to address the memory inefficiency of the streaming APIs provided by `RSqlColumn-ReadStream`. Use of the latter is not recommended on Symbian^3 or later. However, on Symbian^2 or earlier it is the only API that supports streaming from the database.

```
class RSqlColumnReadStream : public RReadStream
  {
public:
  TInt ColumnText(RSqlStatement& aStmt, TInt aColumnIndex);
  TInt ColumnBinary(RSqlStatement& aStmt, TInt aColumnIndex);
  void ColumnTextL(RSqlStatement& aStmt, TInt aColumnIndex);
  void ColumnBinaryL(RSqlStatement& aStmt, TInt aColumnIndex);
  };
```

Streaming using `RSqlColumnReadStream` is accomplished in a similar manner to `RSqlBlobReadStream`. However, the stream is opened in a different manner. An instance of `RSqlColumnReadStream` must be opened on an active prepared statement and, similar to a column extractor, can only access a value in the current row of the prepared statement.

6.4 The Scalar Query Class

A scalar query is a SELECT statement that returns a single result column and a single row, for instance a COUNT query on a database column. It is possible to execute queries of this kind using an `RSqlStatement` object, but Symbian provides the class `TSqlScalarFullSelectQuery` specifically to make execution of these queries easier.

The class declares a number of APIs taking a SELECT statement as an argument: which one you choose depends on the data type of the return value. The declarations are:

```
TInt SelectIntL(const TDesC& aSqlStmt);
TInt64 SelectInt64L(const TDesC& aSqlStmt);
TReal SelectRealL(const TDesC& aSqlStmt);
TInt SelectTextL(const TDesC& aSqlStmt, TDes& aDest);
TInt SelectBinaryL(const TDesC& aSqlStmt, TDes8& aDest);
TInt SelectIntL(const TDesC8& aSqlStmt);
TInt64 SelectInt64L(const TDesC8& aSqlStmt);
TReal SelectRealL(const TDesC8& aSqlStmt);
TInt SelectTextL(const TDesC8& aSqlStmt, TDes& aDest);
TInt SelectBinaryL(const TDesC8& aSqlStmt, TDes8& aDest);
```

An instance of TSqlFullSelectQuery is declared with an argument that refers to the database to be queried:

```
TSqlScalarFullSelectQuery(RSqlDatabase& aDatabase);
```

If you want to use an existing instance of the class to query a different database, you must reinitialize using the SetDatabase() API:

```
void SetDatabase(RSqlDatabase& aDatabase);
```

Here are a few examples of how TSqlScalarFullSelectQuery can be used. This first example demonstrates how a single integer can be returned:

```
// Database connection 'db' is already open
TSqlScalarFullSelectQuery fullSelectQuery(db);
_LIT(KSelectStmt, "SELECT COUNT(*) FROM table");
TInt recCnt = fullSelectQuery.SelectIntL(KSelectStmt);
```

This next example shows how to return a single descriptor:

```
// Database connection 'db' is already open
TSqlScalarFullSelectQuery fullSelectQuery(db);
HBufC* buf = HBufC::NewLC(20);
TPtr text = buf->Des();
_LIT(KSelectStmt, "SELECT value FROM table WHERE id = 1");
TInt rc = fullSelectQuery.SelectTextL(KSelectStmt, text);
```

The buffer is 20 characters long. If the column value fits in the buffer then rc contains KErrNone. However, if the value is too long to fit in the buffer then rc contains the length of the value in characters. This allows the client to re-allocate the buffer and re-execute the query:

```
if(rc > 0)
  {
  buf = buf.ReAlloc(rc);
  CleanupStack::Pop();
  CleanupStack::PushL(buf);
  text.Set(buf->Des());
  _LIT(KSelectStmt, "SELECT value FROM table WHERE id = 1");
  rc = fullSelectQuery.SelectTextL(KSelectStmt, text);
  }
```

6.5 Security Policies

We have seen that you create a secure shared database by passing a security policy as an object of the class `RSqlSecurityPolicy` to the `Create()` API. The `RSqlSecurityPolicy` class contains an enumeration of three kinds of permission:

```
enum TPolicyType
  {
  ESchemaPolicy,
  EReadPolicy,
  EWritePolicy
  };
```

The policies have the following roles:

- `ESchemaPolicy` specifies the policy to alter the structure of the database by adding, modifying or dropping tables, indexes, views or triggers.
- `EReadPolicy` specifies the policy to read from the database.
- `EWritePolicy` specifies the policy to write to the database.

A security policy on a database is any combination of these policies contained in an `RSqlSecurityPolicy` object. To create a security policy, follow these steps:

1. Declare an `RSqlSecurityPolicy` object.
2. Call its `Create()` API.
3. Call its `SetDbPolicy()` API to add the appropriate policies to that object.
4. Use the object in an `RSqlDatabase::Create()` call to create a database with those policies.
5. Call its `Close()` API to release its resources.

The `Create()` API is declared as follows:

```
TInt Create(const TSecurityPolicy& aDefaultPolicy);
void CreateL(const TSecurityPolicy& aDefaultPolicy);
```

The `aDefaultPolicy` argument specifies the policy with which the read, write and schema policies are initialized. If different policies are required for read, write and schema, you can modify them by calling the `SetDbPolicy()` API:

```
TInt SetDbPolicy(TPolicyType aPolicyType,
         const TSecurityPolicy& aPolicy);
```

Here is some example code showing how to create and apply a security policy to a database:

```
TSecurityPolicy defaultPolicy( . . . <default policy> . . . );
RSqlSecurityPolicy securityPolicy;
securityPolicy.CreateL(defaultPolicy);

TSecurityPolicy schemaPolicy(. . . <schema policy> . . . );
User::LeaveIfError(securityPolicy.SetDbPolicy(
                    RSqlSecurityPolicy::ESchemaPolicy, schemaPolicy));

RSqlDatabase db;
User::LeaveIfError(db.Create(. . . <database name> . . .,
                                          securityPolicy));

securityPolicy.Close();
```

In the above example, a default policy is set. The schema policy is then modified before creating the database. The database is therefore created with a read and write policy as specified in the default policy but with a different schema policy.

6.6 Summary

This chapter has introduced the Symbian SQL API. Using the API is straightforward and even more advanced features take little effort. Indeed, this was a major goal for the Symbian SQL API.

As a result, most developers will find that the development effort actually goes into designing and optimizing the database, as opposed to writing code. Optimization is discussed in great detail in Chapter 8 and most developers will find the Troubleshooting section in Appendix A useful. But first, Chapter 7 discusses Symbian SQL and SQLite implementation details and advances in the SQLite engine contributed by Symbian.

7
SQLite Internals on Symbian

Great oaks from little acorns grow.
14th century proverb

Chapters 5 and 6 presented a thorough overview of Symbian SQL and its core APIs. However, in order to fully understand the internal operation of Symbian SQL, it is essential to go a step deeper and examine the internals of the SQLite library. After all, the SQLite library is the engine around which the Symbian SQL architecture is built and through which all database operations are ultimately executed. The SQLite library drives the low-level file operations that create, open, delete, read from and write to database files.

The architecture of the SQLite library is explained in detail in this chapter. Section 7.1 highlights the features of the library that make it suitable for the Symbian platform. Section 7.2 gives a detailed examination of the library's core architecture and Section 7.3 discusses the configuration options that are available to users of the library. Section 7.4 then examines the various optimizations that have been applied to the SQLite library and used by Symbian SQL in order to maximize its performance on the Symbian platform. Many of these optimizations have been championed by Symbian in cooperation with the authors of SQLite (***www.sqlite.org/consortium.html***). The chapter concludes with a brief discussion on the future of SQLite, in Section 7.5.

An in-depth knowledge of SQLite will ensure that you are well prepared when it comes to customizing Symbian SQL to suit your particular database requirements. Chapter 8 provides a wealth of advice on how to apply your knowledge of SQLite and Symbian SQL to effectively analyze and tune your database application to achieve the best possible performance.

First however, let's focus on the fundamental features and architecture of the SQLite library. You should note that the information presented in this chapter is, unless otherwise stated, based on the architecture of the most recent version of SQLite that is used in Symbian^3 (SQLite v3.6.4, at the time of writing).

7.1 Why SQLite Is Right for Symbian

The SQLite project began in May 2000 when Dr. Richard Hipp, the creator and principal developer of the library, began to write the code. Although SQLite was initially intended as software for his own personal use, Dr. Hipp decided to release it as open source software in the summer of 2000. From that day to this, SQLite has remained completely free of cost and copyright – a philosophy that Dr. Hipp is passionate about maintaining.

Over nine years of dedicated development has ensured that the SQLite codebase is of a very high quality and the library is currently used in literally hundreds of software projects by companies including Adobe, Apple and Google. Symbian became another high profile user of the SQLite library after deciding that SQLite was the best solution for developing an SQL database engine on Symbian OS. This decision was made after Symbian performed strict due diligence evaluations on a range of competitive database offerings. There were a number of key factors in SQLite being chosen as the database engine for Symbian SQL:

- **ACID compliance and robustness:** The SQLite library has a well-established reputation as being very robust. In terms of maintaining database integrity, SQLite is ACID-compliant and copes gracefully with events such as memory exhaustion and power failures. The extensive test suite that comes as part of the SQLite package contains thousands of test cases which combine to provide many thousands of executable tests. The test suite provides 100% code coverage for every SQLite release.
- **Impressive performance**: The speed of SQLite is impressive, even on the resource-constrained Symbian OS platform. Symbian profiled a number of SQL database engines and found SQLite to be among the fastest in performing various database use cases. The library's careful use of file I/O makes it particularly suitable for the flash memory that is used in mobile devices.
- **Small footprint:** Physical memory is limited in the embedded world of Symbian OS. Therefore a key consideration in the design of Symbian SQL was to minimize its disk footprint. In this respect, SQLite is suitably compact, a fact that is highlighted by the name 'SQ**Lite**'. The

Symbian SQL server executable in Symbian^3, which integrates the feature-rich SQLite library, is approximately 165 KB in size. The SQLite authors aim to ensure that future versions of the library continue to be as compact as possible. This target is of importance to users, such as Symbian, who embed the library in an environment that has limited physical space and who require to be able to upgrade to newer versions of the library. In addition, SQLite is capable of operating well using a fraction of the RAM required by many competing SQL database engines.

- **Highly customizable**: The SQLite library can be configured to suit different database applications by setting various build-time and run-time options as required. The main SQLite database configuration options are discussed in Section 7.3. Furthermore, Symbian has worked closely with the authors of SQLite to commission specific changes to the library in order to optimize its performance on the Symbian OS platform. These optimizations are discussed in Section 7.4.
- **Supports a large subset of the SQL language**: At the time of writing SQLite supports the majority of the SQL92 standard (the third revision of the SQL language). Most features that are not supported are not required by clients of Symbian SQL or there are reasonable workarounds for them: for example, FOREIGN KEY constraints are not supported but can be modeled using triggers.
- **Open source software:** SQLite is open source software. The library is freely available and comes at no cost and with no licensing restrictions. It is actively developed, tested and maintained to a very high standard, as proven by its inclusion in dozens of successful commercial products including Mozilla's Firefox web browser. Symbian Software Ltd is a founding sponsor of the SQLite Consortium which, in the words of the SQLite authors, 'is a membership association dedicated to insuring the continuing vitality and independence of SQLite. SQLite is high-quality, public domain software. The goal of the SQLite Consortium is to make sure it stays that way.'

7.2 The SQLite Modules

In this section, we uncover the guts of the SQLite library and consider how it actually works. By way of an introduction, Figure 7.1 presents a high-level overview of the SQLite architecture. It illustrates how the SQLite library is effectively a layered collection of modules – or subsystems – that interact with each other. Each module is responsible for a dedicated and essential function of the database engine and it uses those modules below it in the diagram to perform its duties. Note that the SQLite library contains other modules that are not shown in Figure 7.1; they relate to test code and utility functions and do not influence the run-time operation of the SQLite library.

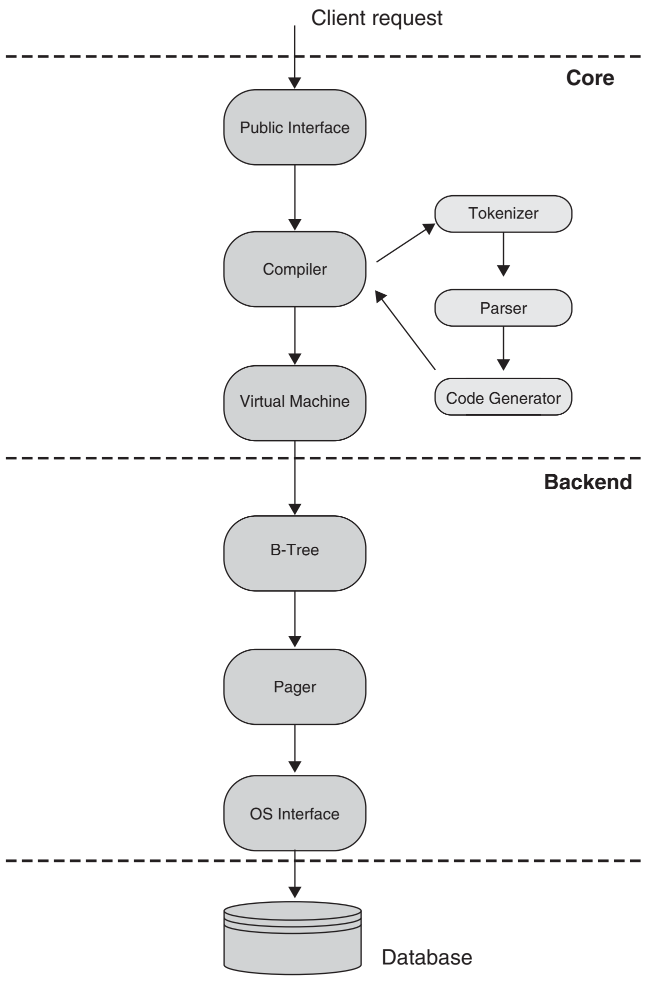


Figure 7.1 The SQLite architecture

As shown in Figure 7.1, the SQLite architecture can be viewed as a collection of 'Core' and 'Backend' modules. The Public Interface – the topmost module – is the set of public SQLite functions that can be invoked by a client and it provides the entry points to the SQLite library. The Core layer is responsible for receiving a client request and compiling it into instructions that are then executed by the Virtual Machine. The Backend layer manages the storage of database content and access to it.

It performs operations on the databases according to the instructions that are sent to it from the Virtual Machine. The Backend performs these database operations using techniques that ensure that database integrity is maintained if an unexpected event, such as a power failure, occurs during an operation.

Now let's take a look in detail at each of the individual SQLite modules that are shown in Figure 7.1.

7.2.1 Public Interface

The SQLite library is written in C and its native public C API provides over 160 functions in the latest releases of the library. The majority of these public functions are declared in the header file, `sqlite3.h`, and are implemented in a handful of source files including `main.c`, `legacy.c` and `vdbeapi.c`. All SQLite functions are prefixed with the tag `sqlite3_` to pre-empt scope resolution issues with client software. The header file, `sqlite3.h`, also defines the list of possible SQLite function result codes, such as `SQLITE_OK`, `SQLITE_ERROR` and `SQLITE_DONE`. These are referred to in the following descriptions of the public interface functions. Since the advent of SQLite, many open source language extensions for the library have been developed to allow it to be used with programming languages such as Perl, Java and Python. Symbian SQL binds to the native C API. Some of the functions in the C API provide access to specialized features and configuration options of the library and are not frequently used – for example, user-defined collation sequences can be created using the function `sqlite3_create_collation()`. In practice, there is a small subset of functions which together service the vast majority of typical database operations; these functions are outlined in the remainder of this section. For full technical details of the following functions and of the other functions that are available in the SQLite C API, the reader should refer to the dedicated SQLite documentation that can be found at ***www.sqlite.org***.

Database Connection Objects

```
typedef struct sqlite3 sqlite3;
```

To perform an operation on a database, SQLite must have an open connection to that database. An open database connection is represented by a pointer to a `sqlite3` object. For the remainder of this chapter, the term 'database connection' refers to an object of type `sqlite3*`. Many SQLite functions that perform an operation on a database take a database connection as a parameter. Database connection objects are fundamental to the operation of SQLite.

The Symbian SQL RSqlDatabase class is a wrapper around the SQLite database connection type. Every RSqlDatabase object corresponds to exactly one SQLite database connection – it is a 1:1 relation. This is true even if two RSqlDatabase objects connect to the same database – two distinct SQLite database connections are created. See Chapter 8 for more details on the RSqlDatabase class.

Creating or Opening a Database

```
int sqlite3_open(const char* filename, sqlite3** ppDb);
int sqlite3_open16(const void* filename, sqlite3** ppDb);
int sqlite3_open_v2(const char* filename, sqlite3** ppDb,
                            int flags, const char* zVfs);
```

There are three functions that can be used to open a database, as listed above. Each function takes a database filename as its first parameter. The filename argument is interpreted as a UTF-8 encoded string by sqlite3_open() and sqlite3_open_v2() and as a UTF-16 encoded string by sqlite3_open16(). The second parameter is a pointer to an uninitialized database connection and is used as an output parameter. Each function opens the specified SQLite database file or, if it does not yet exist, it is created. A database that is created using sqlite3_open() or sqlite3_open_v2() is configured to have UTF-8 database encoding and one that is created using sqlite3_open16() is configured to have UTF-16 database encoding. See Section 7.3.4 for further details on database encoding.

If the database file is created or opened successfully then the function returns SQLITE_OK and the ppDb parameter points to a new valid database connection. If the database file is not created and opened successfully then the function returns an error code. Note that if a database connection is returned in the ppDb parameter then the sqlite3_close() function must be called on the connection when it is no longer required to ensure that the resources associated with the connection are released. This is true even if the database connection was returned with an error code.

The function sqlite3_open_v2() was added to SQLite with the introduction of 'virtual file system' objects which allow SQLite to better support the different file systems of host platforms. See Section 7.2.6 for further details on virtual file system objects, which are defined by SQLite as objects of type sqlite3_vfs. The fourth parameter of sqlite3_open_v2() is the name of the sqlite3_vfs object that the new database connection is to use internally to perform file operations. The third parameter is an integer value that specifies the mode in which the new database connection is to be opened – for example, if SQLITE_OPEN_READONLY is specified then a read-only connection is opened. By default, the functions sqlite3_open() and sqlite3_open16() open a database connection in read–write mode.

The RSqlDatabase class in Symbian SQL provides C++ wrapper APIs for these functions in the form of Create(), CreateL(), Open() and OpenL(). See Section 6.1 for more details.

Closing a Database

```
int sqlite3_close(sqlite3*);
```

The sqlite3_close() function destroys the given database connection and releases all of the resources that it uses. If the database connection that is specified is successfully closed then the function returns SQLITE_OK. The function returns the error code SQLITE_BUSY if an attempt is made to close a database connection that has at least one open prepared statement – such statements must be destroyed before sqlite3_close() is called on the database connection. The rest of this section deals with prepared statements.

The RSqlDatabase class in Symbian SQL provides a C++ wrapper API for this function in the form of Close(). See Chapter 6 for more details.

Executing SQL Statements

```
int sqlite3_exec(sqlite3*,
                 const char* sql,
                 int (*callback)(void*,int,char**,char**),
                 void*,
                 char** errmsg)
```

The sqlite3_exec() function can be used to execute one or more SQL statements in a single call. Using this function can minimize the amount of code that has to be written to perform operations on a database. A database connection is passed as the first parameter to this function. The second parameter is a UTF-8 encoded string containing the SQL statement (or sequence of SQL statements) that is to be executed. The specified SQL statements are executed one at a time until an error or an interrupt occurs. The third and fourth parameters are optional (NULL should be specified if they are not required). They are used when one or more of the SQL statements to be executed returns a result set of records – for example, a SELECT statement normally results in records being returned. The third parameter defines a user-defined callback function that is called once for each record of the result set; the fourth parameter is supplied as the first argument to the callback function when it is called. The fifth parameter is a pointer that SQLite uses to indicate where the client can retrieve error messages from.

If the specified SQL statements are all successfully executed then the function returns `SQLITE_OK`; otherwise an error code is returned.

The `RSqlDatabase` class in Symbian SQL provides a C++ wrapper API for this function in the form of `Exec()`. See Chapter 6 for more details. Note that the implementation of `RSqlDatabase::Exec()` does not provide a callback facility – therefore clients of Symbian SQL typically use `RSqlDatabase::Exec()` only to perform write operations such as INSERT, UPDATE, DELETE and DROP; read operations such as SELECT *can* be executed using `RSqlDatabase::Exec()` but the result set of records cannot be accessed and so such a call would be pointless.

It is worth noting that, in order to execute an SQL statement, SQLite must process the statement to understand what is being requested by it, determine the best way to access the data, and then access the data and return any results requested. These underlying operations manipulate *prepared statements*. The `sqlite3_exec()` function internally uses the prepared statement functions.

Using Prepared Statements

The use of prepared statements is fundamental to the basic operation of SQLite. They provide the most straightforward way to programmatically access the set of records that may be returned by an SQL statement. The `sqlite3_exec()` function *can* be used with a callback function to handle each record of a result set (although, as already stated, clients of Symbian SQL do not have this choice when using `RSqlDatabase::Exec()`) but the prepared statement functions simplify the task of obtaining data from the records.

By using prepared statements, the task of understanding what is being requested by an SQL statement and how to best access the necessary data (which can be an expensive operation for SQLite to perform) can be separated from the task of actually accessing the data and returning the results. An SQL statement can be 'prepared' once and then executed multiple times, which can result in a substantial performance improvement for second and subsequent executions of the statement. There is a RAM overhead to caching and reusing a prepared statement but it can perform an operation faster than calling the `sqlite3_exec()` function, which internally compiles the supplied SQL statement and creates a prepared statement each time it is called. The use of prepared statements is particularly suited to applications that execute the same database queries repeatedly: for example, fetching the name of each employee in the marketing department from a software company's database.

Parameterized prepared statements can be used to support the common scenario of having to repeat slight variations of the same database operation again and again. For example, fetching the name of each employee in *any* given department of the software company's database: in this case, the department column would be parameterized in the prepared statement. The *structure* of a parameterized prepared statement is defined when it is created but its *data content* can be configured after the statement has been prepared, before each execution of the statement. By binding new values to the parameters of a prepared statement, the user can tweak and reuse the prepared statement and still benefit from the performance advantage of already being 'prepared'.

Prepared Statement Objects

```
typedef struct sqlite3_stmt sqlite3_stmt
```

A prepared statement is represented by a pointer to a `sqlite3_stmt` object. For the remainder of this chapter, the term 'prepared statement' refers to an object of type `sqlite3_stmt*`. Many SQLite functions that manipulate SQL statements take a prepared statement as a parameter. Prepared statement objects are fundamental to the operation of SQLite. Note that the term 'compiled statement' is sometimes used instead of 'prepared statement' in SQLite documentation.

The `RSqlStatement` class in Symbian SQL is a wrapper around the SQLite prepared statement type. Every `RSqlStatement` object corresponds to exactly one prepared statement – it is a 1:1 relation. This is true even if two `RSqlStatement` objects prepare the same SQL statement – two distinct prepared statement objects are created. See Section 6.2 for more details on the `RSqlStatement` class.

Preparing a Statement

```
int sqlite3_prepare(sqlite3* db, const char* zSql, int nByte,
                    sqlite3_stmt** ppStmt, const char** pzTail);
int sqlite3_prepare_v2(sqlite3* db, const char* zSql, int nByte,
                       sqlite3_stmt** ppStmt, const char** pzTail);
int sqlite3_prepare16(sqlite3* db, const void* zSql, int nByte,
                      sqlite3_stmt** ppStmt, void** pzTail);
int sqlite3_prepare16_v2(sqlite3* db, const void* zSql, int nByte,
                         sqlite3_stmt** ppStmt, const void** pzTail);
```

To prepare an SQL statement for execution, one of the above four functions must be called. The first parameter of each function is the database connection that is to be used when the statement is executed. The second parameter is the UTF-8 or UTF-16 encoded SQL statement string. The SQL statement string is interpreted as UTF-8 by sqlite3_prepare() and sqlite3_prepare_v2() and as UTF-16 by sqlite3_prepare16() and sqlite3_prepare16_v2(). The third parameter – nByte – indicates the number of bytes that should be read from the supplied SQL statement string. SQLite reads the string up to nBytes or up to the first zero terminator, whichever occurs first. If the value of nByte is negative then the string is read up to the first zero terminator. Each function takes the supplied SQL statement string and creates a prepared statement object to represent it.

If the prepared statement is created successfully then the function returns SQLITE_OK and the ppStmt output parameter points to the prepared statement object. The SQL statement provided by the client is then said to be 'prepared.' If the prepared statement is not created successfully then the function returns an error code and a NULL pointer is returned in ppStmt. Note that, if a prepared statement is returned in the ppStmt output parameter, then the client is responsible for calling the sqlite3_finalize() function to destroy the prepared statement when it is no longer required (details follow later in this section). On return from the function, the second output parameter pzTail points to the first byte past the end of the first SQL statement in the zSql string. This is useful because each function only processes the first SQL statement that is provided in the zSql string – pzTail points to the beginning of the section of the string that the function did not process.

The functions sqlite3_prepare_v2() and sqlite3_prepare16_v2() were added to SQLite with the introduction of virtual file system objects and it is recommended that these functions are used instead of the corresponding legacy functions. Both sqlite3_prepare_v2() and sqlite3_prepare16_v2() include a copy of the original SQL statement string inside the returned prepared statement object. This causes the sqlite3_step() function to behave slightly differently than it would otherwise (details on this function follow later in this section).

The RSqlStatement class in Symbian SQL provides C++ wrapper APIs for these functions in the form of Prepare() and PrepareL(). See Chapter 6 for more details.

To help to illustrate the use of a prepared statement, let's consider a software company's employee database. If we assume that the database contains a table called Employees with columns Employee_Id, Employee_Surname, Employee_Forename, Department_Id and JobTitle_Tag then the following SQL statement string could be passed to one of the SQLite 'prepare' functions to prepare a statement for fetching the name of each employee in a particular department:

```
SELECT Employee_Surname, Employee_Forename
FROM Employees WHERE Department_Id = :Val
```

where `:Val` is a parameter rather than a literal value.

A statement parameter can be represented in one of the following forms:

- ?
- ?NNN
- :VVV
- @VVV
- $VVV

where VVV is an alphanumeric parameter name (such as `Val`, in the example above) and NNN is an integer that defines the index of the parameter.

Note that a prepared statement doesn't have to contain parameters. It can still be beneficial to use a prepared statement even if all of the values in it are hardcoded because the time that it takes to prepare the SQL statement is avoided on second and subsequent executions of the statement.

Binding Values to a Prepared Statement

The following functions can be used to bind a value to each parameter (if there are any) in a prepared statement before the statement is executed:

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n,
                                             void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, sqlite3_int64);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n,
                                             void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int,
                                             void(*)(void*));
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
```

Each of the bind functions takes as its first parameter a prepared statement object that was returned from one of the SQLite ‘prepare’ functions.

The second parameter is the integer index of the parameter that is to be set in the prepared statement. The first parameter in the prepared statement has an index of 1. If the same named parameter is specified more than once in the prepared statement then all instances of the parameter have the same index and so binding a value to one instance binds the same value to all instances of the parameter. The index for parameters of the form '?NNN' is the value of NNN.

The third function parameter is the value that is to be bound to the specified statement parameter. For those functions that have a fourth parameter, it is used to indicate the number of bytes that are in the value to be bound (not the number of characters). For those functions that have a fifth parameter, it is used either to provide a destructor function for the value object or to specify the flags `SQLITE_STATIC` or `SQLITE_TRANSIENT`. The former flag indicates that the value object should not be destroyed by the function and the latter flag indicates that it should be copied by the function first.

The bind functions return `SQLITE_OK` if they are successful or an error code if they are not. For example, if a specified parameter index is out of range then `SQLITE_RANGE` is returned.

The `RSqlStatement` class in Symbian SQL provides C++ wrapper APIs for these functions in the form of `BindX()`, for example, `BindInt()`. See Chapter 6 for more details.

Returning to our example database, imagine that a list of the names of each employee in the Marketing department is required. If the `Department_Id` column in the `Employees` table is an integer column and the known integer value for the marketing department is 4 then the function `sqlite3_bind_int()` is called on our prepared SELECT statement, passing the value 1 as the parameter index and the value 4 as the value to be bound, as follows:

```
sqlite3_bind_int(stmt, 1, 4);
```

Stepping Through a Prepared Statement

After values have been bound to the parameters of a prepared statement, the statement is ready to be executed. Execution is performed in iterative steps by making repeated calls to the following function:

```
int sqlite3_step(sqlite3_stmt*);
```

The `sqlite3_step()` function is called on a prepared statement which is passed as the function's only parameter. The `sqlite3_step()` function must be called one or more times to fully execute the prepared statement – the number of calls required depends on whether the statement returns a record result set or not and, if so, how many records are in the result set. Note that if the prepared statement contains parameters then any parameter that is not bound to a value before the `sqlite3_step()` function is called is treated as `NULL`.

The return value of the `sqlite3_step()` function indicates whether execution of the prepared statement is now complete or the function can be called again or whether an error has occurred. Common return values include:

- `SQLITE_ROW`: This indicates that the next 'row' (i.e. record) of the result set is available to be processed by the client. After the client processes the current record, the `sqlite3_step()` function should be called again to check whether there are any more records in the result set.
- `SQLITE_DONE`: This indicates that execution of the prepared statement has finished successfully and that the `sqlite3_step()` function should not be called again (without first calling the `sqlite3_reset()` function to reset the statement). This value is returned when there are no more records to be processed in a result set.
- `SQLITE_ERROR`: This indicates that an error occurred during the current execution step and that the `sqlite3_step()` function should not be called again (until the `sqlite3_reset()` function has been called).
- `SQLITE_BUSY`: This indicates that SQLite could not obtain a lock on the database and so could not perform the current execution step (see Section 7.2.5 for more details on the SQLite file locking model). If the statement doesn't occur within a `BEGIN ... COMMIT` block or if it is a `COMMIT` statement then the user can try to execute the function `sqlite3_step()` again immediately. Otherwise, the client should execute the `ROLLBACK` command before calling the `sqlite3_step()` function again.
- `SQLITE_MISUSE`: This indicates that the `sqlite3_step()` function was called at an inappropriate point in the lifetime of the prepared statement: for example, if the previous call to the `sqlite3_step()` function returned `SQLITE_DONE`.

If the prepared statement was created using either `sqlite3_prepare_v2()` or `sqlite3_prepare16_v2()` then the `sqlite3_step()` function may return one of the other SQLite result codes or extended result codes, instead of one of the error codes listed above. If the schema of the target database changes prior to a call to the `sqlite3_step()` function then the function tries to recompile the SQL statement. If the schema change has invalidated the SQL statement then the fatal error with code `SQLITE_SCHEMA` is returned by the `sqlite3_step()` function. If the statement was prepared using one of the legacy functions, the `sqlite3_step()` function does not try to recompile the SQL statement and immediately returns the error code `SQLITE_SCHEMA`.

The `RSqlStatement` class in Symbian SQL provides a C++ wrapper API for this function in the form of the `Next()` function. See Chapter 6 for more details.

Accessing Column Values of a Record

When `SQLITE_ROW` is returned from a call to the `sqlite3_step()` function, it means that the next record of the statement's result set is available to be accessed by the client. Imagine that the `Employees` table contains three people who work in the marketing department, listed in the order shown in Table 7.1.

Table 7.1 Marketing department employees

Employee_ ID	Employee_ Surname	Employee_ Forename	DepartmentID	JobTitle_ tag
7	Bowie	Anne	4	PM
35	Williams	Daniel	4	RA
58	Davies	Michelle	4	PM

The first time that `sqlite3_step()` is called on our prepared SELECT statement, the return code is `SQLITE_ROW` and the record containing Anne Bowie's details are ready to be processed. This record in the result set contains two columns – `Employee_Surname` and `Employee_Forename` – as specified by the SELECT statement. After the call to `sqlite3_step()`, the following functions can be used to access the column values of the result record:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
sqlite3_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
```

Each of the above column accessor functions can be used to return the value of a specified column of the current result record. The first parameter to each function is the prepared statement whose result set is being processed. The second parameter is the index of the column whose value is to be retrieved from the current result record. The first column of the result record has the index 0 (note how this contrasts with the SQLite 'bind' functions where the first parameter in the prepared statement has an index of 1).

If the function call is successful then the function returns the value of the specified column, interpreted as the declared return type. SQLite attempts to perform a type conversion if the value of the specified column is not of the type that is returned by the calling function. For example, if the `sqlite3_column_double()` function is called on an integer column then SQLite converts the column's integer value to a float value before returning it from the function. If the `sqlite3_column_text()` function is called on a column of type `NULL` then a `NULL` pointer is returned. Full details of the type conversions that are performed by SQLite can be found in the dedicated SQLite documentation at ***www.sqlite.org***. The result of calling the column accessor functions is undefined if the column index is out of range or if the prepared statement does not currently point to a valid record. It only makes sense to call these functions if the last call to `sqlite3_step()` returned `SQLITE_ROW` (and no call to `sqlite3_reset()` or `sqlite3_finalize()` has been made since).

The `RSqlStatement` class in Symbian SQL provides C++ wrapper APIs for these functions in the form of `ColumnX()`, for example, `ColumnInt()`. See Chapter 6 for more details.

In our example, two calls to `sqlite3_column_text16()` (or `sqlite3_column_text()`) are made to retrieve Anne's full name. The first call specifies a column index of 0 to get Anne's surname; the second call specifies a column index of 1 to get Anne's forename.

Resetting a Prepared Statement

```
int sqlite3_reset(sqlite3_stmt* pStmt);
```

The `sqlite3_reset()` function resets the prepared statement object that is passed to it as its only parameter. In this context, 'reset' means that the prepared statement returns to the state that it was in prior to the first call to `sqlite3_step()`and the statement is ready to be executed again if required. Note that any values that were bound to the prepared statement are still bound to it after the call to `sqlite3_reset()`. If what is actually required is not that the prepared statement is reset but that all of its bound parameters are set to `NULL` then the function `sqlite3_clear_bindings()` should be used instead.

It is safe to call the `sqlite3_reset()` function if the last call to the `sqlite3_step()` function returned `SQLITE_ROW` or `SQLITE_DONE`; in these cases, the `sqlite3_reset()` function returns `SQLITE_OK`. An error code is returned by `sqlite3_reset()` if the last call to `sqlite3_step()` returned an error.

The `RSqlStatement` class in Symbian SQL provides a C++ wrapper API for this function in the form of `Reset()`. See Chapter 6 for more details.

In our example, the sqlite3_reset() function might be called after the full list of marketing employees has been returned and processed, so that the same query is ready to be executed again when it is needed.

Destroying a Prepared Statement

```
int sqlite3_finalize(sqlite3_stmt* pStmt);
```

The sqlite3_finalize() function destroys the prepared statement object that is passed to it as its only parameter and releases all resources that are associated with it.

If the function is successful then it returns SQLITE_OK. This only happens if the prepared statement has been completely executed (i.e. the last call to sqlite3_step() returned SQLITE_DONE) or has not been executed at all (i.e. there have been no calls to sqlite3_step()). The function returns the error code SQLITE_ABORT if the prepared statement has not been completely executed and other error codes are possible if the last call to sqlite3_step() returned an error. The prepared statement object is always destroyed, even if an error is returned by the sqlite3_finalize() function.

The RSqlStatement class in Symbian SQL provides a C++ wrapper API for this function in the form of Close(). See Chapter 6 for more details.

Handling Errors

```
int sqlite3_errcode(sqlite3 *db);
int sqlite3_extended_errcode(sqlite3 *db);
const char *sqlite3_errmsg(sqlite3*);
const void *sqlite3_errmsg16(sqlite3*);
```

The sqlite3_errcode() interface returns the numeric result code or extended result code for the most recent failed sqlite3_* API call associated with a database connection. If a prior API call failed but the most recent API call succeeded, the return value from sqlite3_errcode() is undefined. The sqlite3_extended_errcode() interface is the same except that it always returns the extended result code even when extended result codes are disabled.

The sqlite3_errmsg() and sqlite3_errmsg16() return English-language text that describes the error as UTF-8 or UTF-16, respectively. Memory to hold the error message string is managed internally. The application does not need to worry about freeing the result. However, the error string might be overwritten or deallocated by subsequent calls to other SQLite interface functions.

When serialized threading mode is in use, a second error may occur on a separate thread between the time of the first error and the call to these interfaces. When that happens, the second error is reported since these interfaces always report the most recent result. To avoid this, each thread can obtain exclusive use of database connection `D` by invoking `sqlite3_mutex_enter(sqlite3_db_mutex(D))` before beginning to use `D` and invoking `sqlite3_mutex_leave(sqlite3_db_mutex(D))` after all calls to the interfaces listed here are complete.

If an interface fails with `SQLITE_MISUSE`, the interface was invoked incorrectly by the application. In that case, the error code and message may or may not be set.

A full list of result codes and extended result codes (used for I/O errors) can be found on the SQLite website at ***www.sqlite.org/c3ref/c_abort.html***.

7.2.2 Compiler

The SQLite public interface provides clients with a way to execute SQL statements on their databases, to access and modify the data that is stored in them.

Take, for example, a database of music tracks that contains a table that has four columns, as defined below (for simplicity, the table does not contain a column for the binary data of a music track):

```
CREATE TABLE MusicLibrary (
  TrackName TEXT,
  AlbumName TEXT,
  Artist TEXT,
  Genre TEXT
)
```

The application that acts as a front-end to the database needs to execute various SQL statements in order to service requests from the user. For example, the user may wish to view of all of the tracks in the music collection that are tagged as being in the ‘Pop’ genre. To display this list of tracks, the application may execute the following SQL statement:

```
SELECT TrackName FROM MusicLibrary WHERE Genre="Pop" ORDER BY AlbumName
```

On the other hand, if the user wishes to add new tracks into the database then the application needs to execute a series of SQL statements of the following form:

```
INSERT INTO MusicLibrary (TrackName, AlbumName, Artist, Genre)
VALUES ("Chasing Cars", "Eyes Open", "Snow Patrol", "Alternative")
```

So what happens to SQL statements when they pass via the public interface into the SQLite library? The SQL statement embarks on a journey through the internal modules of the SQLite library, starting with the Compiler module. This module performs a syntactic evaluation of the SQL statement and generates a bytecode program that is processed by the Virtual Machine module. As with compilers for other languages, there are three key elements to the Compiler module – the Tokenizer, the Parser and the Code Generator – and we examine each of these in turn now.

Tokenizer

The Tokenizer performs the first stage of the compilation process. As its name suggests, it takes an SQL statement that has been supplied by a client and breaks it down into a sequence of valid tokens, where a token is a sequence of one or more characters that has a well-defined meaning in the context of an SQL statement. Various classes of token are defined by SQLite and each token identified by the Tokenizer is associated with a particular class. The full range of token classes is listed in `parse.h`, which defines an integer value for each token class. Some examples of token classes are:

- `TK_INSERT`: the INSERT keyword
- `TK_WHERE`: the WHERE keyword
- `TK_ID`: identifiers that are specified between SQL keywords.

The `TrackName` token in the following SQL statement would be classified as a `TK_ID` token:

```
SELECT TrackName FROM MusicLibrary WHERE Genre="Pop" ORDER BY AlbumName
```

Table 7.2 shows how the SELECT statement above would be processed by the Tokenizer.

As each token in the SQL statement is identified and classified by the Tokenizer, it is sent to the Parser immediately if appropriate. This

Table 7.2 Tokens identified by the tokenizer

Token	Token class	How the token is processed
`SELECT`	`TK_SELECT`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`TrackName`	`TK_ID`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`FROM`	`TK_FROM`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`MusicLibrary`	`TK_ID`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`WHERE`	`TK_WHERE`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`Genre`	`TK_ID`	Token forwarded to the Parser
`=`	`TK_EQ`	Token forwarded to the Parser
`Pop`	`TK_ID`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`ORDER`	`TK_ORDER`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`BY`	`TK_BY`	Token forwarded to the Parser
`" "`	`TK_SPACE`	Ignored
`AlbumName`	`TK_ID`	Token forwarded to the Parser

interaction between the Tokenizer and the Parser continues until the SQL statement has been completely tokenized.

It is not the job of the Tokenizer to spot any syntactic errors in the SQL statement that it is processing. The Tokenizer simply uses the available token classes to identify tokens and send them to the Parser.

Parser

SQLite's Parser is generated by a custom-made parser generator called Lemon which works in a similar way to the more well-known parser generators, Bison and Yacc. Lemon was written by Dr. Richard Hipp and it is open source software that is maintained as part of the SQLite project. The source file `lemon.c` contains the code for the parser generator itself and the source file `lempar.c` is the template that is used for the parser program that Lemon generates. The input to Lemon is a context-free grammar that is used to generate a program that parses and validates its input according to that grammar. The parser grammar for SQLite is defined in `parse.y`. The Parser that is generated for use by SQLite is implemented in `parse.h` and `parse.c`.

It is the Parser's job to accept tokens that are sent to it by the Tokenizer and use them to construct a parse tree, piece by piece. A parse tree is a data structure that is used to represent the tokenized SQL statement as an ordered sequence of expressions. The Parser uses the grammar rules that are defined in `parse.y` to construct the correct hierarchy in the parse tree, based on the tokens that are sent to it. Figure 7.2 illustrates the parse tree that may be constructed for our example SELECT statement.

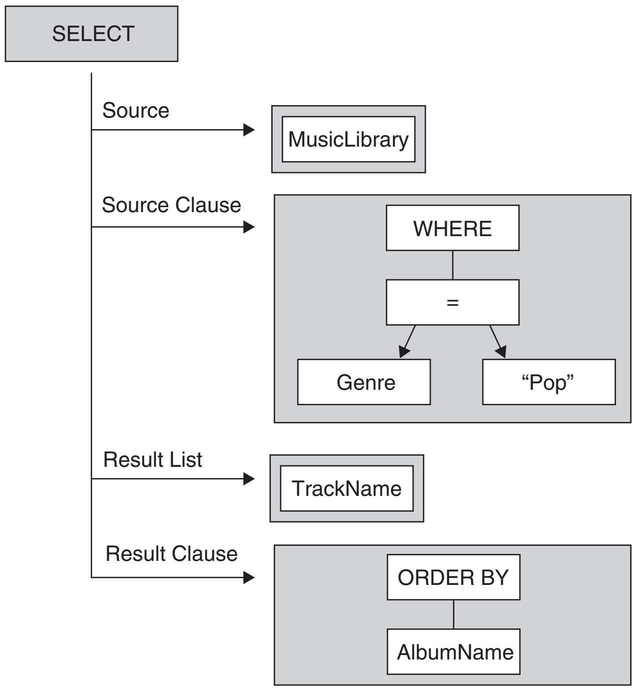


Figure 7.2 A basic parse tree

Any syntax errors that are present in the SQL statement being compiled are spotted by the Parser during its construction of the parse tree (note that the Parser does not perform semantic checks). For example, imagine that our example SELECT statement had been mistakenly specified as the following:

```
SELECT TrackName FROM  WHERE Genre="Pop" ORDER BY AlbumName
```

The Parser receives a `TK_SELECT` token, then a `TK_ID` token, then a `TK_FROM` token. When it is then sent a `TK_WHERE` token, the Parser realizes that it is an invalid sequence of tokens for a SELECT statement. At this point, the calling function (either `sqlite3_exec()` or `sqlite3_prepare()`) returns an error.

If the parse tree is successfully constructed by the Parser then it is passed to the Code Generator which is responsible for performing the final step of the compilation process.

Code Generator

The Code Generator plays a key role in the compilation process – it is responsible for translating a given parse tree into a program of bytecodes, or instructions, that SQLite's Virtual Machine can understand and execute. Put another way, inside the Code Generator is where an SQL statement ceases to exist in its human-readable form and is replaced by an executable program. This is often referred to as a *VDBE program*, where VDBE is an acronym for Virtual Database Engine.

As you might expect, the Code Generator is a complex module and it makes up a sizeable portion of the SQLite codebase. The Code Generator examines the parse tree (recursively, if needed – for example, when an UPDATE statement contains a nested SELECT statement) and generates a sequence of bytecodes to be executed for each part of the tree. The structure of the parse tree is important for the Code Generator to be able to generate the bytecodes in the correct order for execution. When the Code Generator has finished examining the parse tree, the result is a complete VDBE program that can be passed to the Virtual Machine for execution.

An important function of the Code Generator is that it tries to optimize the program that it produces by using its knowledge of the target databases to ensure that the code makes the most efficient use of the database schemas and indices. This key part of the Code Generator is referred to as the Query Optimizer. The Query Optimizer considers a range of ways in which to optimize the code including which indices, if any, can be used to speed up search operations, and whether any expressions in the SQL statement can be rewritten to be more efficient. Full details on how the Query Optimizer works can found in the dedicated SQLite documentation at ***www.sqlite.org***.

Let's examine how the Code Generator translates a parse tree into a VDBE program. For simplicity, Table 7.3 illustrates a VDBE program that is generated for the following statement (no ORDER BY clause and no indices on the table):

```
SELECT TrackName FROM MusicLibrary WHERE Genre="Pop" ORDER BY AlbumName
```

Each bytecode that is generated by the Code Generator consists of one opcode and up to five operands – P1, P2, P3, P4 and P5. You can think of an opcode as a function and an operand as a parameter to that function. At the time of writing, SQLite has 142 opcodes which are defined in the header file `opcodes.h`. Operands P1, P2 and P3 are 32-bit signed integers for every opcode. Operand P4 can be a value of various types including a 32-bit or 64-bit signed integer, a string or a pointer to a collation function. Operand P5 is an unsigned character for every opcode.

A VDBE program like the one in Table 7.3 is internally generated as a result of a call to one of the SQLite 'prepare' functions. The prepared statement that is returned from the 'prepare' function has internal access to the VDBE program. A prepared statement is sometimes called a 'compiled' statement, because on return from a call to one of the 'prepare' functions the VDBE program has been compiled. Any subsequent calls to bind values to parameters in the prepared statement cause the VDBE program to be updated to use the specified values.

Table 7.3 VDBE program

Address	Opcode	P1	P2	P3	P4	P5
0	`String8`	0	1	0	`Pop`	00
1	`Goto`	0	12	0		00
2	`SetNumColumns`	0	4	0		00
3	`OpenRead`	0	2	0		00
4	`Rewind`	0	10	0		00
5	`Column`	0	3	2		00
6	`Ne`	1	9	2	`collseq(BINARY)`	69
7	`Column`	0	0	4		00
8	`ResultRow`	4	1	0		00
9	`Next`	0	5	0		01
10	`Close`	0	0	0		00
11	`Halt`	0	0	0		00
12	`Transaction`	0	0	0		00
13	`VerifyCookie`	0	1	0		00
14	`TableLock`	0	2	0	`MusicLibrary`	00
15	`Goto`	0	2	0		00

The Code Generator uses a set of dedicated functions to generate the bytecode for individual parts of the parse tree. For example, the function `codeEqualityTerm()` in `where.c` is called to generate the bytecode for a single equality term in a WHERE clause, such as `Id = 10` or `NumChildren IN (1,2,3)`. The function `sqlite3Insert()` in `insert.c` is called to generate the bytecode for an INSERT statement. Other functions such as `sqlite3OpenTableAndIndices()` are called from within `sqlite3Insert()`. The bytecode at address 0 in Table 7.3 is generated by the function `sqlite3ExprCodeTarget()` in `expr.c`, a portion of which is shown below. This function is called from within the `codeEqualityTerm()` function in `where.c`.

```
int sqlite3ExprCodeTarget(Parse *pParse, Expr *pExpr, int target)
  {
  Vdbe *v = pParse->pVdbe;  /* The VM under construction */
  int op;                   /* The opcode being coded */
  int inReg = target;       /* Results stored in register inReg */
  int regFree1 = 0;     /* If non-zero free this temporary register */
  int regFree2 = 0;     /* If non-zero free this temporary register */
  int r1, r2, r3, r4;       /* Various register numbers */
  sqlite3 *db;
  db = pParse->db;
  assert( v!=0 || db->mallocFailed );
  assert( target>0 && target<=pParse->nMem );
  if( v==0 ) return 0;
  if( pExpr==0 ){
    op = TK_NULL;
  }else{
    op = pExpr->op;
  }
  switch( op ){
    ...
    case TK_STRING: {
      sqlite3DequoteExpr(db, pExpr);
      sqlite3VdbeAddOp4(v, OP_String8, 0, target, 0,
              (char*)pExpr->token.z, pExpr->token.n);

      break;
      }
    ...
    }
  ...
  }
```

The highlighted function call to `sqlite3VdbeAddOp4()` adds a single bytecode to the VDBE program that is passed as the first argument. The supplied opcode is `OP_String8` and there are four supplied operands: P1 is given the value 0, P2 is given the value 1 (from the `target` variable), P3 is given the value 0 and P4 is given the string value ‘Pop’ (from the `pExpr` variable). The last argument to the function indicates the type of the P4 operand value.

Once the Code Generator has called all of the necessary functions to convert the parse tree into a complete VDBE program, that’s it – the

Compiler's work is done. It has taken an SQL statement in its raw, human-readable character format and converted it into a program of bytecodes that can be understood by the Virtual Machine module. It is now up to the Virtual Machine module to accept the program and execute it.

7.2.3 Virtual Machine

Now it's crunch time! Armed with a VDBE program that has been produced by the Compiler, it falls to the Virtual Machine module to execute the program. The Virtual Machine implements an abstract computing engine called the Virtual Database Engine which executes the bytecodes in the supplied VDBE program to access and modify the target databases.

The Virtual Machine executes the VDBE program beginning with the bytecode at address 0, stopping only after it executes the final bytecode or if it encounters a fatal error or a 'halt' instruction on the way. It uses a program counter to advance through the sequence of bytecodes in the VDBE program. Cursors are used by the Virtual Machine to access database tables and indices during program execution. A cursor is a pointer to a single database table or index and it is used to access the entries within the table or index. The Virtual Machine also has a stack which is used to store intermediate values that are required by the program.

Let's consider the flow of execution when the Virtual Machine is asked to execute the VDBE program listed in Table 7.3. This program is listed again in Table 7.4 for convenience, together with commentary on the sequence of execution steps that are made by the Virtual Machine. To start with, the Virtual Machine's program counter contains address 0 and execution begins there when the first call to the `sqlite3_step()` function is made.

It is clear from this breakdown that the Virtual Machine must be able to access the contents of databases while it is executing the VDBE program. To gain database access, the Virtual Machine uses the B-tree module. This module acts only to obey instructions from the Virtual Machine to access and modify the B-tree structures on behalf of the VDBE program. You can think of the B-tree module as being a 'slave' of the Virtual Machine.

7.2.4 B-Tree Module

The B-tree module is a critical part of the SQLite library. It represents database tables and database indices internally using B-tree data structures. All of the logic for creating, accessing, updating, balancing and destroying B-trees is contained in the B-tree module. Each node of a table or index B-tree represents a page of the database and the B-tree module uses the Pager module to create, access, update and delete the pages that are represented in its B-trees (the Pager module is discussed in Section 7.2.5).

Table 7.4 Flow of execution for a VDBE program

Address	Opcode	P1	P2	P3	P4	P5	Step
0	`String8`	0	1	0	`Pop`	00	1. Store the P4 string value in the register denoted by P2. The P4 string value is a NULL-terminated, UTF-8 encoded string.
1	`Goto`	0	12	0		00	2. Jump to the address denoted by P2 and resume execution from there (i.e. set the program counter to contain address 12).
2	`SetNumColumns`	0	4	0		00	7. Set the number of columns to the value of P2 for the cursor that is opened by the next bytecode.
3	`OpenRead`	0	2	0		00	8. Open a read-only cursor on the `MusicLibrary` table. The new cursor is given an id of the value of P1. P2 denotes the number of the root page of the `MusicLibrary` table. The database file is identified by P3, where a value of 0 denotes the main database.
4	`Rewind`	0	10	0		00	9. Make the cursor point to the first record in the `MusicLibrary` table. If the table is empty then jump to the address denoted by P2 and resume execution from there (i.e. set the program counter to contain address 10).
5	`Column`	0	3	2		00	10. Access the column whose index is denoted by P2 (i.e. the `Genre` column) from the record currently pointed to by the cursor with id value P1. Store the retrieved column value in the register denoted by P3.

(*continued overleaf*)

Table 7.4 (continued)

Address	Opcode	P1	P2	P3	P4	P5	Step
6	`Ne`	1	9	2	`collseq (BINARY)`	69	11. Compare the values in the registers denoted by P1 and P3 – this compares the string ‘Pop’ to the string in the `Genre` column of the current record. The string comparison is performed using the collation function that is specified by P4. If the strings are not equal then jump to the address denoted by P2 and resume execution from there (i.e. set the program counter to contain address 9).
7	`Column`	0	0	4		00	12. Access the column whose index is denoted by P2 (i.e. the `TrackName` column) from the record currently pointed to by the cursor with id value P1. Store the retrieved column value in the register denoted by P3.
8	`ResultRow`	4	1	0		00	13. The register denoted by P1 contains the `TrackName` value of a single record of the result set of the live SELECT statement. Terminate the currently active `sqlite3_step()` call with the return value `SQLITE_ROW`. At this point, the SQLite column accessor functions can be used to access the `TrackName` value of this record. The next bytecode is executed when the next call to `sqlite3_step()` is made.
9	`Next`	0	5	0		01	14. Move the cursor to point to the next record in the `MusicLibrary` table and then jump to the address denoted by P2 and resume execution from there (i.e. set the program counter to contain address 5). If there is no next record then continue with the next bytecode instead of jumping to this address.

10	`Close`	0	0	0		00	15. Close the cursor with id value P1.
11	`Halt`	0	0	0		00	16. Exit the program. The VDBE program execution is now complete.
12	`Transaction`	0	0	0		00	3. Begin a read transaction. The transaction is started on the database file whose index is denoted by P1. If P2 were non-zero then a write transaction would have been started.
13	`VerifyCookie`	0	1	0		00	4. Verify that the schema version that is stored in the database header is equal to the value of P2. This is done to ensure that the internal cache of the schema matches the current schema of the database.
14	`TableLock`	0	2	0	`Music-Library`	00	5. Obtain a read lock on the `MusicLibrary` table. P1 denotes the index of the database and P2 denotes the number of the root page of the table to be locked (see the next section for more details on root pages). P4 contains a pointer to the name of the table that is being locked, for error reporting purposes. If the value of P3 were 1 then a write lock would have been obtained. Note that the `TableLock` opcode is only used when the shared cache mode of SQLite is enabled – see the Pager Module section for more details on shared cache mode.
15	`Goto`	0	2	0		00	6. Jump to the address denoted by P2 and resume execution from there (i.e. set the program counter to contain address 2).

Each table and index in a database is represented as a separate B-tree. B-tree structures, and the differences between table and index B-trees, are examined in this section. It should be noted that this section examines only how SQLite uses B-trees and that the general theory, characteristics and advantages of B-trees are not covered – the curious reader should refer to dedicated B-tree documentation to further explore the topic. Before we get to that, let's first understand the format of a SQLite database file.

Database File Format

SQLite has a strict mapping of one database to one database file. SQLite views a database as a collection of one or more equally sized *pages*. Historically, the page size used by SQLite is 1 KB and this is the default page size that is set by Symbian SQL. The page size of a database can be configured when the database is created and cannot be changed after a table has been created in the database. A page is the unit of currency that SQLite uses for its file I/O – every read or write operation on a database file is a read or write operation of one or more pages.

The pages in a database are numbered sequentially, beginning with page 1. Page 1 of a database is special: it contains a 100-byte database file header that identifies the file as a SQLite database and contains the version number of the SQLite library that was used to create it. The header also contains the permanent settings for the database including the page size and the database encoding (see Section 7.3 for more details on these and other database settings).

Every SQLite database contains a special table called SQLITE_MASTER which is automatically created by SQLite. This table defines the schema of the database, i.e. it contains details on all of the tables, indices, triggers and views in the database, and is sometimes referred to as the 'schema table.' SQLite keeps SQLITE_MASTER up to date as tables, indices, views and triggers are created and deleted in the database. Note that clients of SQLite can execute SELECT statements on SQLITE_MASTER to retrieve data from it, but they cannot modify the table. SQLITE_MASTER has five columns, as defined in the following code and Table 7.5:

```
CREATE TABLE sqlite_master (
  type TEXT,
  name TEXT,
  tbl_name TEXT,
  rootpage INTEGER,
  sql TEXT
)
```

Page 1 of a database acts as the root page of the B-tree structure that represents the SQLITE_MASTER table, so SQLite always knows where to find the SQLITE_MASTER table of a database.

Table 7.5 `SQLITE_MASTER` columns

Name	Type	Purpose
`type`	TEXT	The type of the schema item; the value can be 'table', 'index', 'view' or 'trigger'.
`name`	TEXT	The name of the schema item, i.e. the name of the table, index, view or trigger.
`tbl_name`	TEXT	The name of the table that this schema item is associated with if it is an index or a trigger; if it is a table or a view then `tbl_name` is the same as `name`.
`rootpage`	INTEGER	If the schema item is a table or an index then this value is the number of the root page of the B-tree that is used to represent the table or index. Otherwise the value is `NULL`.
`sql`	TEXT	The SQL statement string that was used to create the schema item: for example, the full `CREATE TABLE` statement that was used to create a table.

B-tree of a Table

The structure that is used to store the records of a table is technically a B+tree. Only the lowest nodes of the B+tree – the 'leaf' nodes – contain records of the table. The nodes that are higher up in the tree, which we refer to as 'internal' nodes, are used purely as navigation aids by the B-tree module when it is searching for a particular record or set of records in the tree. In the remainder of this section, the term 'B-tree' is used instead of 'B+tree', for simplicity.

To help to visualize how a database table is stored using a B-tree, an example is given in Figure 7.3. A key point to note is that each node of the B-tree is stored in a single page in the database. The terms 'node' and 'page' are used interchangeably in the remainder of this section.

At the top of the B-tree is the 'root' node. This is the starting point for the B-tree module when it navigates through the table. Each node in the B-tree contains one or more entries, where each entry contains a 'key' section (shown in light grey) and a 'data' section (shown in dark grey). This is true for both internal nodes and leaf nodes. The key is always the `ROWID` of a record in the table but there are some important differences between internal nodes and leaf nodes, as explained below.

Entries in an internal node are sorted by key value. The data section of an entry contains the page number of a child node in the tree – for illustration purposes this is shown as a pointer to the child node in Figure 7.3. The child node may be another internal node or it may be a leaf node. All key values of the entries in the child node are less than or

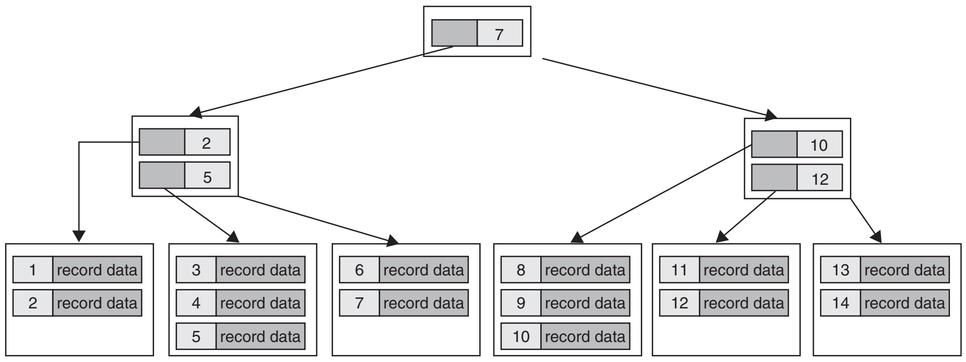


Figure 7.3 A table represented as a B-tree

equal to the key value of the entry in the parent node. Every internal node also contains a single stand-alone page number that denotes its right child node. Thus if an internal node contains N entries then it contains N+1 child node 'pointers.' For each internal node that is at the same level in the B-tree, the key values of the entries in the leftmost node are less than the key values of the entries in the next node to the right, and so on.

Entries on a leaf node are also sorted by key value. However, the data section of an entry is the actual column data of a record in the table – specifically, the record that has the `ROWID` that is specified as the entry's key. In other words, *the entries of a leaf node represent records of the table*. Again, the key values of the entries in the leftmost leaf node are less than the key values of the entries in the next leaf node to the right, and so on.

Each entry on a leaf node contains a third section that stores the total size of the data section of the entry (this value is stored before the key value). This information is used when the B-tree module is searching for a particular record in a single leaf node. If the first entry on the leaf node is not the desired record (i.e. the `ROWID` does not match the target `ROWID`) then the B-tree module can immediately skip to the start of the next entry on the leaf node (by moving forward 'total size of data section' bytes). In this way, the B-tree module can find a particular record in a leaf node without having to traverse the contents of the records before it.

Imagine that a new record is added to the table that is illustrated in Figure 7.3 and that it is allocated a ROWID of 15. SQLite always tries to fit a new record onto a leaf node such that no modifications are required for the rest of the tree. So, SQLite first tries to fit the new record with `ROWID` 15 onto the rightmost leaf node, which contains the records with `ROWID`s 13 and 14. If all of the new record cannot fit onto the target node then overflow pages are used (overflow pages are discussed later in this section). If the portion of the new record that would not be allocated to overflow pages cannot fit onto the target node then SQLite instead attempts to fit the new record onto another leaf node (or allocate a new leaf node if necessary) and rebalance the B-tree.

So how does the B-tree module know where to locate the B-tree structure for a particular table? The answer is that the Virtual Machine sends the number of the root page of the B-tree to the B-tree module. For example, in Step 5 in Table 7.4, operand P2 holds the number of the root page of the `MusicLibrary` table. The Code Generator obtains the root page number by looking up the `MusicLibrary` table's entry in the database's `SQLITE_MASTER` table and finding the number of its root page in the `rootpage` column.

Once the B-tree module has located the B-tree for the table, it can continue to follow the instructions from the Virtual Machine to locate the target records in the B-tree. For example, in Step 9 in Table 7.4, the table cursor is made to point to the first record in the `MusicLibrary` table. The B-tree module does this by searching the B-tree for the record with `ROWID` 1.

Record Data Packing

When the B-tree module has found the desired record on a leaf node of the B-tree, the entire record is accessible by the Virtual Machine (its cursor points to the record). The Virtual Machine must then manipulate the record's column data as instructed by the VDBE program. Note that the B-tree module itself does not know how to manipulate the column data of a record because, to it, the record is just a collection of bytes.

A record's column data is stored in the data section of the record's entry in the leaf node. SQLite uses a 'data packing' technique to store the column data as an ordered sequence of bytes. Information about how SQLite should interpret this sequence of bytes is provided by a header that comes before the column bytes. This packing technique is intended to use as few bytes as possible to store a record's column data and thus minimize the amount of physical space that a record requires to be stored.

Figure 7.4 illustrates the format that is used for the data section of a record's entry in a leaf node.

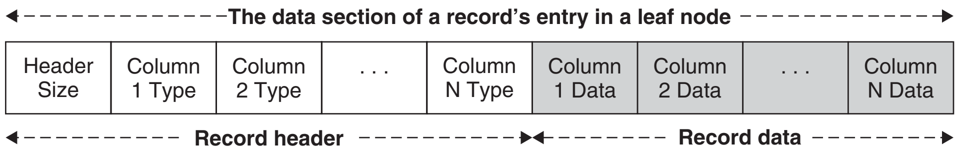


Figure 7.4 How the column data of a record is formatted

The record header begins with a variable-length integer whose value indicates the total size of the header in bytes. This is used to indicate where the record data section begins. This integer is followed by another N variable-length integers for a record that has N columns. Each integer value indicates the type and size (in bytes) of the corresponding column data. For example, if the value of 'Column 1 Type' is two then this indicates that 'Column 1 Data' is a two-byte INTEGER. If the value of 'Column N Type' is 212 then this indicates that 'Column N Data' is a BLOB that is 100 bytes in size (the size of the BLOB is calculated as half of the value of 'Column N Type' minus 12). A column type value of 0 indicates that that corresponding column value is `NULL` and takes up zero bytes of space in the record data section. The code that implements the data packing format is part of the Virtual Machine module, which understands how to pack and unpack a record. For a full list of the possible column type values and what they mean, the reader should refer to the dedicated SQLite documentation at ***www.sqlite.org***.

The cost of storing record data in the above format is that to find a column at the end of a record, the Virtual Machine must process all of the header fields in order until it finds the column type of the desired column. It can then calculate the offset of the desired column in the record data section (by adding together the size of all of the preceding columns) and access the column. This is why Symbian recommends that the most frequently accessed columns of a table be put at the start of the table, to reduce the search time for these columns. Chapter 8 discusses the effects of column ordering in more detail.

Returning to our example VDBE program, the `Genre` column of each record in the `MusicLibrary` table must be checked to see whether it contains the value 'Pop'. The Virtual Machine uses column index 3 (which is specified in Step 10 in Table 7.4) to calculate where the `Genre` column starts in the data section of the record that is currently being pointed to. The column data is then retrieved and compared to the value 'Pop.'

Overflow Pages

Figure 7.4 paints a very tidy picture of record storage in which every page contains one or more complete records. What happens if a record is too big to fit on a single page? (Remember that the page size is fixed for a database). This can happen if the size of a record is itself larger than the size of a page or if the size of a record is larger than the amount of space that is left on the page where the record is to be placed.

In both of these cases, the record must be stored using more than one page and so an *overflow page* is created by SQLite. A small portion of the start of the record is then stored on the 'base' page and the remainder of the record is stored on the overflow page. If the remainder of the record cannot fit on the overflow page then another overflow page is created and used, and so on. This chain effect is illustrated in Figure 7.5. Exactly how many overflow pages are required depends on the size of the record and the page size of the database.

Figure 7.5 illustrates the case where the record with `ROWID` 5 in the B-tree in Figure 7.3 is too large to fit on a single page. Two overflow

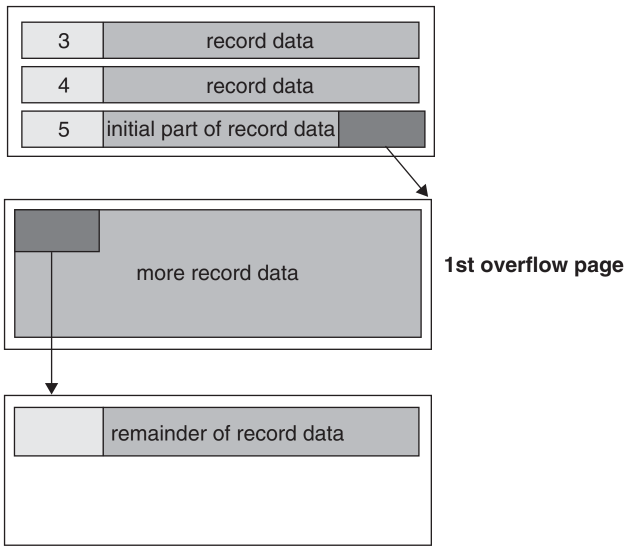


Figure 7.5 How overflow pages are used

pages are required. In the base page, the entry for the record contains the initial part of the record data and the page number of the first overflow page in the chain – for illustration purposes, this is shown as a pointer to the first overflow page in Figure 7.5. The first overflow page contains the page number of the next overflow page and as much of the remaining record data as it can. The final overflow page contains the page number 0 (to indicate that this is the final overflow page in the chain) and the remainder of the record data. This linked list of pages is essential to keep track of the overflow pages that have been created to accommodate the entire record.

A record that contains BLOBs can easily be large enough in size that it requires overflow pages to be stored. It is important to be aware that a long chain of overflow pages is inefficient to process during an operation on a table if the overflow pages have to be loaded into memory to execute the operation.

For example, consider the following table `Images`:

```
CREATE TABLE Images (Id INTEGER, Picture BLOB)
```

Imagine that each record in the table contains a BLOB that requires three overflow pages to store it. If a client executes the following SQL statement to retrieve all of the ids in the table:

```
SELECT Id FROM Images
```

then the B-tree module finds each record on a leaf node of the table B-tree. The data section of each record entry is likely to contain the `Id` column data and part of the `Picture` column data, plus a pointer to the first overflow page. The value of the `Id` column can be retrieved with no need to load any of the overflow pages. If, on the other hand, the `Picture` column was the first column in the table, then it is likely that the data section of each entry would contain part of the `Picture` column data plus a pointer to the first overflow page. Now SQLite must process the overflow pages to get to the `Id` column because it is stored on the third overflow page; to get to this page, SQLite must walk through the linked list of overflow pages from the start of the list.

Because of the overhead of using overflow pages, Symbian strongly recommends that BLOB columns should be defined as the last columns in the table. By keeping BLOB columns at the end of a table, the cost of dealing with overflow pages to get to a subsequent column can be avoided. This can greatly improve the performance of search operations that use earlier columns of the table.

B-tree of an Index

Indices are represented separately from tables and each index on a table is represented by its own B-tree. For example, imagine that the B-tree in Figure 7.3 represents a table that was created by the following SQL statement:

```
CREATE TABLE Cars (
  TotalInStock INTEGER,
  Manufacturer TEXT,
  SecondHandTotal INTEGER,
  Color TEXT
)
```

The `Cars` table contains the 14 records shown in Table 7.6. If an index was created on the table by the following SQL statement:

```
CREATE INDEX idx ON Cars(SecondHandTotal)
```

Then Figure 7.6 illustrates what the index B-tree might look like.

For illustration purposes, the example `Cars` table and `idx` index have been kept simple and the size of their B-trees has been exaggerated to emphasize the tree structures. In reality, both the `Cars` table and the `idx` index could fit on the root pages of their respective B-trees in a typical database that has a page size of 1 KB.

Just as with table B-trees, there is a ‘root’ node at the top of the index B-tree and each node of the B-tree is stored in a single page in the

Table 7.6 The `Cars` table

TotalInStock	Manufacturer	SecondHandTotal	Color	ROWID
32	Ford	18	Red	1
108	Volkswagen	3	Blue	2
46	Mini	6	Green	3
89	BMW	67	Black	4
283	Ford	15	Silver	5
123	Vauxhall	1	White	6
14	Mercedes	7	Red	7
56	Ford	13	Green	8
48	Seat	15	Red	9
263	Audi	29	Black	10
28	BMW	25	Blue	11
96	Ford	2	Black	12
67	Volkswagen	9	Silver	13
59	Vauxhall	11	Silver	14

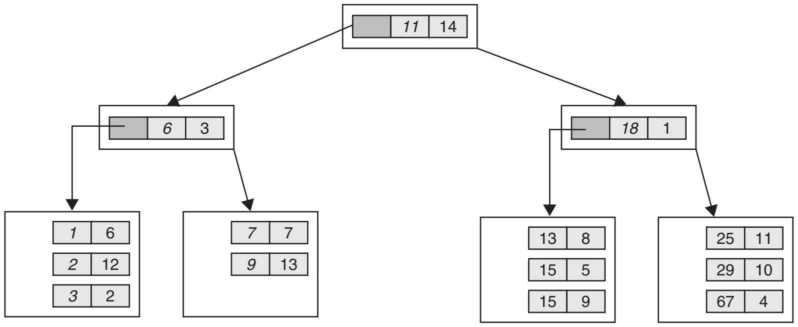


Figure 7.6 An index represented as a B-tree

database. Similarly, each node in an index B-tree contains one or more entries. However, there are some important differences between an index B-tree and a table B-tree.

In an index B-tree, each node entry has a 'key' section (shown in light grey) but this does not contain only a `ROWID`. Instead, the key is a set which contains the indexed column values of a record, in the order that they are defined in the table, plus the record's `ROWID`. Furthermore, only entries in internal nodes contain a 'data' section (shown in dark grey). The data section contains the page number of a child node in the tree – for illustration purposes, this is shown as a pointer to the child node in Figure 7.6 – and this section is not present in leaf node entries. Every internal node also contains a page number that denotes its right child node. Thus if an internal node contains N entries then it contains N+1 child node 'pointers.'

The entries in a node are sorted by key, where the order is defined by the index. The ordering is such that all key values of the entries in a child node are considered less than the key value of the entry in the parent node. Also, for each node that is at the same level in the tree, the key values of the entries in the leftmost node are considered less than the key values of the entries in the next node to the right, and so on.

An entry on a node in an index B-tree may need to use overflow pages if the indexed column values require a lot of space. This can happen if, for example, large TEXT or BLOB columns are indexed, which is generally discouraged.

Note that each entry in an index B-tree contains another section that stores the total size of the key section of the entry (this value is stored before the key section). The B-tree module uses this information to skip to the start of successive entries in a node.

So how is an index B-tree used to speed up a SELECT statement that uses the index, for example? Consider the following SELECT:

```
SELECT Manufacturer FROM Cars WHERE SecondHandTotal > 50
```

The `SecondHandTotal` column is indexed by `idx` and so SQLite first traverses the `idx` B-tree in Figure 7.6, searching for a node entry whose key set contains a value greater than 50 in its first column. Two pages need to be processed to locate the single matching entry in this case (or three if the root page is not already in the cache). When SQLite locates the entry, it retrieves `ROWID` 4 from its key set and uses this `ROWID` value to traverse the table B-tree in Figure 7.3 to find the actual record and retrieve the value of the `Manufacturer` column from it. Another two pages need to be processed to locate the actual record from the table B-tree (or three pages if the table root page is not already in the cache), so between four and six pages need to be processed to execute the above SELECT statement. However, if index `idx` did not exist then SQLite would need to traverse the *entire* table B-tree and examine every record to check whether it has a value greater than 50 in its `SecondHandTotal` column.

Note that every table in a SQLite database has an in-built index, courtesy of the fact that every record in a table is given a unique `ROWID`. This is a major advantage that SQLite has over some other database engines because it means that table searches are always based on the use of an index, even if the owner of the database hasn't explicitly created one. Note that if a column in a table is declared to be an `INTEGER PRIMARY KEY` then that column becomes an alias for `ROWID`.

7.2.5 Pager

As the B-tree module traverses a B-tree, it asks the Pager module to create, retrieve, update or delete the pages in the B-tree. The Pager is responsible for writing and reading database pages to and from disk (or the flash memory that is used in mobile devices).

The Pager has some key features and critical responsibilities, which are discussed below. In particular, two key responsibilities of the Pager are that it polices all file locking and file I/O to provide the essential support for ACID-compliant transactional behavior and that it provides page caching which allows the B-tree module to operate on database pages in memory and is essential for achieving optimal performance.

Database File Locking

The Pager uses a file-locking mechanism to implement the 'isolation' aspect of the ACID model. File locking is required to safely manage concurrent database access between different threads or processes. However, much of the supported file-locking intricacies of SQLite are not required by Symbian SQL because it has only one thread that accesses the database files. Therefore, Symbian SQL maintains an exclusive lock on every database file that it opens, which prevents the file from being modified by another process. However, in the interests of completeness in our journey through the internals of SQLite, it is worth outlining the locking philosophy of the library.

A process that wishes to read from a database file must first obtain a `SHARED` lock. The Pager attempts to obtain a shared lock on the database at the start of the read operation – for example, a SELECT statement. If a shared lock cannot be obtained (for one of the reasons outlined below) then the read operation fails with the error `SQLITE_BUSY`. If a shared lock is successfully obtained then it is held until the read operation has completed, at which point the shared lock is revoked. More than one process can hold a shared lock on the same database at the same time but no process is able to write to a database file while there is at least one other process that holds a shared lock on the database.

A process that wishes to write to a database file must first obtain the following sequence of locks. The Pager begins its attempt to obtain the locks at the start of the write operation – such as a DELETE statement, for example.

1. Obtain a **`SHARED`** lock. Failure to do so results in the write operation failing with the error `SQLITE_BUSY`.
2. Obtain a **`RESERVED`** lock. This indicates that the client process aims to write to the database during its connection to the database. A reserved lock can only be held by one process at a time; if

another process already holds a reserved lock, the write operation fails with the error `SQLITE_BUSY`. Note that a reserved lock can be obtained while other client processes hold shared locks; when a client process obtains a reserved lock, it does not prevent other clients from obtaining new shared locks or reading from the database. SQLite views a transaction as a read transaction until a reserved lock is obtained, at which point it becomes a write transaction.

3. Obtain a **`PENDING`** lock. When changes that have been made to the database in memory must be committed to the database file on disk (either because the transaction has committed or the page cache is full and some updated pages need to be swapped out of the cache), the client process needs to obtain a pending lock to indicate that it wants to write to the database as soon as all current shared locks have been revoked. When a client holds a pending lock, no new shared locks can be obtained on the database but existing shared locks are not disturbed. When all existing shared locks have been revoked, the client can attempt to 'promote' the pending lock to an exclusive lock.

4. Obtain an **`EXCLUSIVE`** lock. A client must hold an exclusive lock in order to actually write to a database file. An exclusive lock can only be obtained when there are no other shared locks on the database. Only one exclusive lock at a time is permitted on a database; while the exclusive lock is held, no other lock can be obtained on the database.

If an exclusive lock is obtained on a database then the next step is to write the updated pages in memory to the database file on disk. If the write to the database file is only needed because the page cache is full and some updated pages need to be swapped out then after the updated pages have been written to disk the exclusive lock is still held and the client can continue to make changes. If the write to the database file is needed because the transaction has been committed then, after the changes have been written to disk, all locks are revoked.

The notion of a pending lock was introduced by the authors of SQLite to lessen the likelihood of 'writer starvation' occurring for a database. By preventing any new shared locks from being obtained on a database when a pending lock is held on the database, a client who wishes to write to the database only has to wait until the existing shared locks are revoked before it can obtain an exclusive lock.

Page Cache

The Pager module uses caching to achieve the best possible performance when it is executing operations on a database. During a transaction, the pages that need to be read from or written to are retrieved from the database file and added to an in-memory page cache (if they are not already in the page cache). All necessary modifications are then made to the pages in the cache and the changes are later committed to the database file. If a page that is required by the Pager during a database operation is already present in the page cache then the Pager can avoid an expensive file I/O operation to fetch that page from disk.

In early versions of SQLite, each open connection to a database has its own page cache, even if there are two or more connections to the same database. With this caching model, the RAM usage of SQLite can quickly grow when multiple database connections are open at the same time. The notion of a 'shared' cache was introduced in SQLite v3.3.0. When SQLite is operating in shared cache mode, each *thread* within a process has its own page cache per open database to which it is connected. If the thread has more than one open connection to the same database then all of the connections share the same page cache. In SQLite v3.5.0, this was taken further: if shared cache mode is enabled then each *process* has a single page cache per open database and connections to the same database from different threads within the same process all share the same cache. Shared cache mode is disabled by default in SQLite but it is enabled in all versions of Symbian SQL to date (via a call to the function `sqlite3_enable_shared_cache()`). A page cache configuration is illustrated in Figure 7.7.

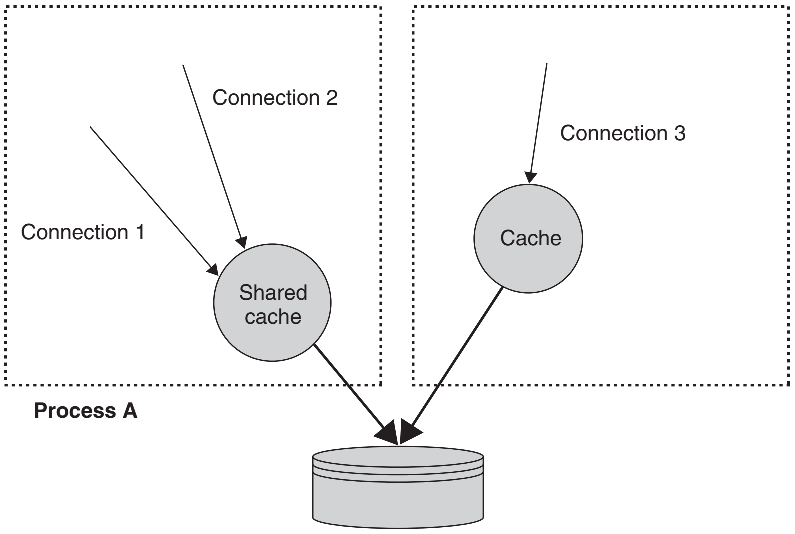


Figure 7.7 Using page caches to store database content in memory

The shared cache used by process A is treated by SQLite as a single connection to the database and the file-locking mechanism that is described above applies between the shared cache and other connections to the database. You should refer to the dedicated SQLite documentation at ***www.sqlite.org*** for information on how SQLite manages access *to* the shared cache. Suffice to say that SQLite still provides ACID guarantees to connections that use a shared cache by using table-level locks in conjunction with file-level locks.

By default, SQLite sets the maximum number of pages that a page cache can hold to be 2000 pages. This default value can be configured by a client using the `default_cache_size` PRAGMA (or the `cache_size` PRAGMA to change the maximum cache size for a specific database connection). If pages that are required by the Pager are not already in the cache and the cache is full (i.e. the page cache contains the maximum number of pages permitted) then the Pager has to swap out pages from the cache to make room for the new pages. If the pages being swapped out have been updated (i.e. the pages are 'dirty') then the Pager must write the changes to the database file as part of the swapping process. Excessive page swapping (also known as 'page thrashing') can impact the performance of database operations and so the maximum size of a page cache should be set to a reasonable size. For example, a maximum page cache size of 100 pages is likely to cause performance issues for a database that contains a table of 5000 music tracks which is frequently searched.

In all versions of SQLite up to and including SQLite v3.3.13, the page cache is flushed at the end of every transaction on the database. SQLite deems this necessary because it assumes that between transactions T1 and T3 being executed on database A by connection 1, connection 2 which uses a different page cache may also have modified database A in transaction T2. If this happens and connection 1's page cache is not flushed before T3 is executed then T3 may use pages in the cache that have since been modified on disk by connection 2. Therefore flushing the page cache at the end of every transaction is carried out as a safety precaution, but it increases the time taken to execute the next transaction because each page that the transaction requires must be retrieved from disk (or at least from the file system's cache).

This precautionary cache flushing is not necessary for Symbian SQL because the single server process uses shared cache mode and so there is only one page cache per open database. To help to optimize the operation of SQLite in such a host environment, enhancements were made in SQLite v3.3.14 to allow the library to avoid flushing the page cache in certain circumstances. These enhancements are discussed in Section 7.4.2.

Rollback Journal File

A critical responsibility of the Pager is to ensure that write operations are always atomic. Atomic writes are essential to maintain the integrity of a database – a change to the database must be made in its entirety or not at all. The Pager uses *rollback journal files* (and file locks) to implement the 'atomicity' aspect of the ACID model.

A rollback journal file is a separate file from a database file and it has the same name as the database file with which it is associated, plus the suffix '-journal'. There is only one rollback journal file per database file and it is automatically created and manipulated by SQLite as and when required. A rollback journal file is used during a write transaction to store the original state of each page that is changed by the transaction. If SQLite is operating in its default journal mode, the Pager creates a rollback journal file at the start of a write transaction (when a reserved lock is obtained, to be precise) and writes to the journal file the content of each page that is to be updated before making any change to that page in the cache. At the end of the write transaction, the updated pages are written to the database file and the journal file is deleted.

The journal mode of SQLite can be changed, using the `journal_mode` PRAGMA, to avoid the overhead of SQLite repeatedly creating and deleting journal files, which can adversely impact the performance of write transactions. Section 7.4.7 discusses the alternative journal modes of SQLite and how Symbian SQL takes advantage of them.

This journaling technique is used to allow the automatic 'roll back' of data if a transaction fails to complete, for example, because a power failure occurred midway through the transaction. In this scenario, the journal file is not deleted by SQLite (because the transaction did not complete) and so it is present when the device restarts. The journal file is referred to as a 'hot' journal file because its presence, which is detected by SQLite at the start of the next transaction on the database, signifies that a problem occurred during the last transaction and that an automatic rollback of the database is necessary because the database may now be corrupt. The process of restoring the database to the state that it was in prior to the start of the failed transaction is called a database 'rollback' and it involves updating the database with the pages that are stored in the hot journal file, which contain the original content of the pages that were partly modified during the failed transaction. Note that, when SQLite is operating in its default mode, a check is made for a hot journal file at the start of every transaction. Section 7.4.7 explains how SQLite can greatly reduce the frequency with which it checks for hot journal files when it is operating in a different mode. In the case of multiple attached databases, each database has its own journal file and SQLite uses a master journal file to keep a reference to the individual journal files.

A database rollback may also be required in less drastic circumstances. For example, a run-time error such as running out of memory or disk space could occur during the execution of a transaction. Depending on the error that occurs, SQLite may attempt to roll back the entire transaction or it may choose to 'undo' only the statement that it was processing when the error occurred and then continue with the remainder of the transaction. Clients may choose to execute the ROLLBACK command in response to any run-time errors that occur during a transaction to ensure that the entire transaction is rolled back and that the database is left in a known state. The reader should refer to the documentation at ***www.sqlite.org*** for further information on SQLite's automatic rollback behavior.

It should be noted that the automatic rollback behavior of SQLite applies to both 'implicit' and 'explicit' transactions. However, it is recommended that clients of Symbian SQL provide a rollback mechanism in their code in case a Symbian OS 'leave' occurs during an explicit transaction. This mechanism is commonly implemented by pushing onto the Symbian OS cleanup stack a function that is invoked if a leave occurs during the execution of an explicit transaction. The supplied function should execute the ROLLBACK command on the database if it is called. For more detailed information on this rollback technique, see Appendix A.

Page Reuse and Auto-vacuum

As more and more data is inserted into a database, its page population grows, which causes the size of the database file to increase. Over time, some of the data may be deleted, which can make whole pages of deleted data redundant – such pages continue to exist in the database and are called *free pages*. SQLite needs to be able to reuse such pages, otherwise database files would grow to unacceptable sizes (perhaps even to a size that the host file system cannot support).

SQLite needs a way to keep track of the free pages in a database in order to be able to reuse them. This is the purpose of the *free page list*, which is structured as a linked list of pages as shown in Figure 7.8. Each page in the linked list is known as a 'trunk' page and contains the number of the next trunk page in the list (for illustration purposes this is shown as a pointer to the next trunk page in Figure 7.8), the total number of 'leaf' pages that are associated with the trunk page and the page number of each of those leaf pages. The page number of the first trunk page in the free page list is stored in page 1 of the database file.

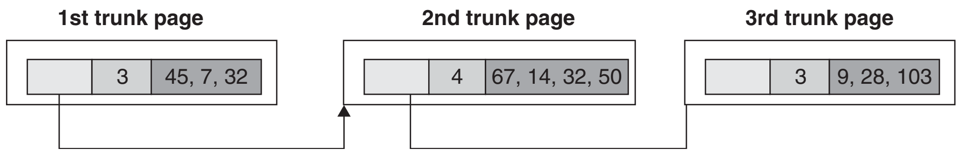


Figure 7.8 The structure of a free page list

When a write transaction is executed, details of the free pages in the database file must be journaled in case the free page list is modified during the transaction. By structuring the free page list as shown, rather than linking the trunk and leaf pages together in one large, flat linked list, SQLite only has to journal the trunk pages.

When data is deleted from a database, each whole page that becomes free (i.e. that contains only deleted data) is added to the free page list. If the database has auto-vacuum set to 'none' then the free pages remain on the free page list when the transaction completes. When a subsequent database operation – for example, an INSERT statement – requires pages to be allocated, SQLite first checks the free page list to see whether there are any free pages that can be reused (note that both the trunk and the leaf pages in the free page list are eligible for reuse). If there are free pages available then SQLite uses them and the free list page is updated to no longer include those pages. Reusing free pages in a database is clearly preferable to allocating additional disk space for new pages.

That said, the creator of a database may prefer to configure it so that when a page becomes free the space that it occupies can be reclaimed by the file system, which can result in the size of the database file being reduced. If the database has auto-vacuum set to 'full' then at the end of every transaction on the database all pages on the free page list are automatically moved to the end of the database file and the file is then truncated to remove the free pages. If the database has auto-vacuum set to 'incremental' then the free pages are still tracked on the free page list but are only moved to the end of the database file and the file truncated when an explicit request to do so is made by a client of the database using the `incremental_vacuum` PRAGMA. More details on the available SQLite auto-vacuum settings are provided in Section 7.3.1.

The pages that are used in index B-trees can also be added to the free page list of a database. For example, if deleting a number of records from a table results in all entries on a page of an index B-tree being deleted then that page is added to the free page list.

As you can see, the Pager is of critical importance to SQLite in its quest to access and modify database files in a safe and efficient manner. If you are wondering how the Pager knows which operating system functions to call to write and read pages to and from disk – indeed, how it knows which operating system it is running on in the first place (!) – the answer is, it doesn't. This is where the final module of the SQLite library steps in – the OS Interface module.

7.2.6 OS Interface

SQLite was designed from the beginning to be a portable library. The ability of SQLite to run on different platforms is due to the library's use of a low-level abstraction layer known as the OS Interface. The OS Interface is the final piece in the SQLite architectural puzzle and is the 'glue' between SQLite and the operating system that the library is running on. This abstraction layer allows the SQLite library to be ported to many platforms and the SQLite website offers precompiled binaries for Linux, Windows and Mac OS X. All that a developer requires to do to port the SQLite library to a new platform is download the SQLite source code, write some custom code to provide implementations of various OS Interface functions and create a few OS Interface objects. The functions that need to be implemented define the low-level operating system operations that are essential to execute a database request, such as file reading, file writing and file locking. The file API of every operating system is different and so by using the OS Interface module to abstract itself away from platform-specific details, SQLite is able to exist as a portable library.

In SQLite v3.6.1, used in Symbian^3, the OS Interface uses the notion of *Virtual File System* objects to support the file systems of different operating systems. The type `sqlite3_vfs` is defined as follows in `sqlite3.h`:

```
typedef struct sqlite3_vfs sqlite3_vfs
struct sqlite3_vfs
{
  int iVersion;            /* Structure version number */
  int szOsFile;            /* Size of subclassed sqlite3_file */
  int mxPathname;          /* Maximum file pathname length */
  sqlite3_vfs *pNext;      /* Next registered VFS */
  const char *zName;       /* Name of this virtual file system */
  void *pAppData;          /* Pointer to application-specific data */
  int (*xOpen)(sqlite3_vfs*, const char *zName, sqlite3_file*,
                                  int flags, int *pOutFlags);
  int (*xDelete)(sqlite3_vfs*, const char *zName, int syncDir);
  int (*xAccess)(sqlite3_vfs*, const char *zName, int flags,
                                             int *pResOut);
  int (*xFullPathname)(sqlite3_vfs*, const char *zName, int nOut,
                                                    char *zOut);
  void *(*xDlOpen)(sqlite3_vfs*, const char *zFilename);
  void (*xDlError)(sqlite3_vfs*, int nByte, char *zErrMsg);
  void *(*xDlSym)(sqlite3_vfs*,void*, const char *zSymbol);
  void (*xDlClose)(sqlite3_vfs*, void*);
  int (*xRandomness)(sqlite3_vfs*, int nByte, char *zOut);
  int (*xSleep)(sqlite3_vfs*, int microseconds);
  int (*xCurrentTime)(sqlite3_vfs*, double*);
  int (*xGetLastError)(sqlite3_vfs*, int, char *);
  /* New fields may be appended in figure versions.
  ** The iVersion value will increment whenever this happens. */
}
```

The VFS interface defines functions for operating system services including opening and deleting a file, checking the current time and a sleep function. The implementation of Symbian SQL includes a 'Symbian OS porting layer' which implements a customized version of each of these functions. The porting layer also creates a single `sqlite3_vfs` object and stores in it a pointer to each of the customized functions.

SQLite is informed of the VFS object that it is to use via a call to the `sqlite3_vfs_register()` function; in the case of Symbian

SQL, it is called by the Symbian OS porting layer's implementation of the sqlite3_os_init() function. The SQLite library expects sqlite3_os_init() to be custom-implemented by the host platform and it is called from sqlite3_initialize() which the host platform calls to initialize the SQLite library at run time. Similarly, a custom implementation of sqlite3_os_end() (which should call sqlite3_vfs_unregister() to un-register the VFS object) is necessary to deallocate operating system resources. This function is called from sqlite3_shutdown(), which the host platform must call to indicate that it has finished using the SQLite library and that all relevant library resources should be deallocated.

Another key element of the Symbian OS porting layer is that it creates a single sqlite3_io_methods object. The type sqlite3_io_methods is defined as follows in sqlite3.h:

```
typedef struct sqlite3_io_methods sqlite3_io_methods
struct sqlite3_io_methods
  {
  int iVersion;
  int (*xClose)(sqlite3_file*);
  int (*xRead)(sqlite3_file*, void*, int iAmt, sqlite3_int64 iOfst);
  int (*xWrite)(sqlite3_file*, const void*, int iAmt,
                               sqlite3_int64 iOfst);
  int (*xTruncate)(sqlite3_file*, sqlite3_int64 size);
  int (*xSync)(sqlite3_file*, int flags);
  int (*xFileSize)(sqlite3_file*, sqlite3_int64 *pSize);
  int (*xLock)(sqlite3_file*, int);
  int (*xUnlock)(sqlite3_file*, int);
  int (*xCheckReservedLock)(sqlite3_file*, int *pResOut);
  int (*xFileControl)(sqlite3_file*, int op, void *pArg);
  int (*xSectorSize)(sqlite3_file*);
  int (*xDeviceCharacteristics)(sqlite3_file*);
  }
```

The I/O methods interface defines functions for file operations such as read, write, truncate and lock. The Symbian OS porting layer provides a custom implementation of each of these functions and a pointer to each one is stored in the single sqlite3_io_methods object that is created by the porting layer. For example, the Symbian OS implementation of the xRead() function calls the Symbian OS API RFileBuf::Read().

Each time a file needs to be opened (or created) by SQLite, a call is made to the xOpen() function of the registered VFS object. This function takes an uninitialized sqlite3_file object as a parameter. The sqlite3_file type is defined as follows in sqlite3.h but can be extended by the host platform's porting layer:

```
typedef struct sqlite3_file sqlite3_file;
struct sqlite3_file
```

```
  {
  const struct sqlite3_io_methods *pMethods;// Methods for an open file
  }
```

SQLite expects the customized `xOpen()` function to initialize the supplied `sqlite3_file` object's `pMethods` data member to point to the `sqlite_io_methods` object that is created by the host platform. On return from the call to `xOpen()`, the `sqlite3_file` object can be used by SQLite in its internal calls to file operation functions, including:

```
int sqlite3OsRead(sqlite3_file *id, void *pBuf, int amt, i64 offset)
  {
  DO_OS_MALLOC_TEST;
  return id->pMethods->xRead(id, pBuf, amt, offset);
  }

int sqlite3OsWrite(sqlite3_file *id, const void *pBuf, int amt,
                                                    i64 offset)
  {
  DO_OS_MALLOC_TEST;
  return id->pMethods->xWrite(id, pBuf, amt, offset);
  }

int sqlite3OsTruncate(sqlite3_file *id, i64 size)
  {
  return id->pMethods->xTruncate(id, size);
  }

int sqlite3OsLock(sqlite3_file *id, int lockType
  {
  DO_OS_MALLOC_TEST;
  return id->pMethods->xLock(id, lockType);
  }
```

This overview of the OS Interface module and how it ties to the host platform of SQLite brings us to the bottom of the SQLite 'stack' that is illustrated in Figure 7.1. Our journey through the modules of the SQLite library is complete.

7.3 SQLite Configuration Options

Users of SQLite have the opportunity to take the library as it comes 'out of the box' and configure it to optimize its performance for their particular database application. SQLite has various default settings that can be changed and some features of the library can be excluded if they are not required. The configuration options that are available in SQLite are what give the library its versatility. This section describes the most commonly used configuration options and how they can influence the performance of a database application.

PRAGMAs are used to set many of the SQLite configuration options but the reader should note that PRAGMA execution is disabled in Symbian SQL for secure shared databases. Symbian recommends that clients of Symbian SQL do not use PRAGMAs directly and the Symbian SQL API provides alternative ways to configure SQLite settings – see Chapter 6 for more details.

7.3.1 Auto-vacuum

The auto-vacuum setting of a database determines when free space in the database file is reclaimed. There are three possible values for the auto-vacuum setting:

- None: the database file does not shrink after data is deleted from it – no free space is ever automatically reclaimed.
- Full: at the end of every transaction, the database file is automatically 'vacuumed' – all free pages in the database are reclaimed by the file system and the database file becomes smaller (if there was at least one free page to be reclaimed).
- Incremental: a client can execute the `incremental_vacuum` PRAGMA on the database to request that some vacuuming occurs immediately; the client must specify the maximum number of free pages that SQLite should reclaim; the 'incremental' auto-vacuum option was added in SQLite v3.4.0.

Databases that have auto-vacuum set to 'none' are likely to experience better performance for write transactions than databases that have auto-vacuum set to 'full'. When auto-vacuum is set to 'none', the time taken to execute a transaction does not include time taken to vacuum the database.

By default, SQLite sets the auto-vacuum setting to 'none' for every database. This default value can be changed by defining the SQLite compile-time option `SQLITE_DEFAULT_AUTOVACUUM` to have the value 1 (for 'full') or 2 (for 'incremental'). The auto-vacuum setting of a database can be overridden at run time by executing the `auto_vacuum` PRAGMA. Changing the auto-vacuum setting of a database at run time between 'full' and 'incremental' is always possible; changing it between 'none' and 'full' or 'none' and 'incremental' is only possible before tables have been created in the database.

By default, Symbian SQL sets the auto-vacuum setting to 'full' for every created database on Symbian^1 and Symbian^2, and to 'incremental' on

Symbian^3. The 'incremental' value is used on Symbian^3 to facilitate a change in how Symbian SQL automatically vacuums a database – see Section 7.4.9 for more details.

Note that the VACUUM command can be executed on any (main) database to reclaim all of its free pages immediately, regardless of its auto-vacuum setting. The VACUUM command also defragments the database, which the auto-vacuum mechanism does not.

7.3.2 Page Size

The page size of a database is set when the database is created and cannot be changed after a table has been created in the database. SQLite supports page sizes of 512 bytes, 1024 bytes, 2048 bytes, 4096 bytes, 8192 bytes, 16,384 bytes and 32,768 bytes. The maximum upper bound on the page size is 32,768 bytes.

The page size of a database can have a significant influence on the performance of operations on the database. The ratio of page size to record size of a database must be carefully considered. Too small a page size makes it likely that overflow pages will be required to store table records. Too large a page size could impact the performance of accessing a single record at a time. This is due to the fact that SQLite reads from and writes to a database file in terms of whole pages and so it is important to use a page size that ensures that as much of the space on a page is used as possible, without triggering the need for overflow pages. For example, having an average record size of 1.2 KB and a page size of 2 KB means that only one full record can fit on any one page. If two records are required then two pages must be retrieved. Using a page size of 4 KB would allow three full records to fit on every page and then potentially only a single page would need to be retrieved to access up to three records.

Historically, the default page size chosen by SQLite is 1024 bytes. In more recent releases of the library, the default page size is calculated dynamically, based on characteristics of the host platform that SQLite retrieves using OS Interface functions. The default page size can be configured by defining the SQLite compile-time option `SQLITE_DEFAULT_PAGE_SIZE` to be the desired page size in bytes. The default value can be overridden at run time by executing the `page_size` PRAGMA on a database before any tables have been created in it.

Symbian SQL uses a default page size of 1024 bytes.

7.3.3 Maximum Cache Size

The maximum size of a page cache determines the maximum number of pages that can be held in it at one time and it can have a critical impact on the performance of operations on the database. The higher this number, the less likely that file I/O is required to retrieve a target page from disk because the more likely it is to already be present in the cache. Reducing the frequency with which SQLite must perform file I/O can significantly increase the performance of database operations. Of course, too large a cache size and it could potentially starve the device of RAM. On the other hand, too small a cache size and frequent page swapping is likely, which can increase the time taken to perform database operations.

By default, SQLite uses a maximum cache size of 2000 pages. Note that defining a maximum cache size does not set the actual size of the cache. The cache is initially empty and it dynamically grows in size when pages are read into it. The default maximum cache size of 2000 pages can be changed by defining the SQLite compile-time option `SQLITE_DEFAULT_CACHE_SIZE` to be the desired maximum number of pages. The default value can be overridden at run time by executing the `default_cache_size` PRAGMA on a database connection – the change persists and applies to all new database connections. To change the maximum cache size for a single database connection, the `cache_size` PRAGMA should be executed by the connection.

Symbian SQL uses a default maximum cache size of 64 pages on Symbian OS v9.3. On Symbian^2, the maximum cache size is dynamically calculated based on the page size of the database and the configured soft heap limit of SQLite. The reader should refer to Section 7.4.4 for full details on how the maximum cache size is calculated but, as an example, the maximum cache size is set to 1024 pages for a database that has a default page size of 1024 bytes (if the default soft heap limit of 1 MB is in use).

7.3.4 Database Encoding

The database encoding of a database is set when the database is created and it cannot be changed afterwards. This setting determines which character encoding is used to store the text in a database and it can be configured to one of two possible values – UTF-8 or UTF-16. The chosen database encoding applies to *all* text content in the database – you cannot configure different encodings for different records in a table, for example.

The choice of database encoding can influence the size of a database file. It is therefore important when creating a database to consider whether it will contain text, either immediately or in the future, and from what range of alphabets the text might come. Both UTF-8 and UTF-16 are capable of storing full Unicode text, but the general recommendation is that UTF-8 encoding should be used for a database that contains Western text and UTF-16 encoding should be used for a database that contains Middle Eastern or Far Eastern text, to ensure that the text is stored as compactly as possible. UTF-8 encoding is quite capable of storing wide characters but it usually requires more bytes of storage than UTF-16 to do so. If there is a possibility that Eastern text may feature in the database then the database should probably be set to use UTF-16 encoding.

Clients of SQLite can read and write text using either UTF-8 or UTF-16 encoding by using the appropriate bind functions (`sqlite3_bind_text()` and `sqlite3_bind_text16()`) and column accessor functions (`sqlite3_column_text()` and `sqlite3_column_text16()`). SQLite performs any conversions necessary between UTF-8 and UTF-16 when moving text between the client, the database file and collation functions.

By default, SQLite uses UTF-8 encoding for every created database. The default value can be overridden at run time by executing the `encoding` PRAGMA before the database has been created. Symbian SQL uses a default database encoding of UTF-16 on the Symbian platform.

7.3.5 SQL Statement Encoding

Clients of a database have a choice of supplying SQL statement strings as either UTF-8 or UTF-16 strings. For example, the `sqlite3_prepare()` and `sqlite3_prepare_v2()` functions expect the supplied statement string to be encoded in UTF-8 and the `sqlite3_prepare16()` and `sqlite3_prepare16_v2()` functions expect the supplied statement string to be encoded in UTF-16.

As with database encoding, the client's choice of statement encoding should be based on knowledge of the type of text that is contained in the statement. The choice of database encoding and query encoding should be as compatible as possible. Internally, SQLite always operates with UTF-8 encoded statement strings. Statements that are encoded as UTF-16 are internally converted to UTF-8 as part of the statement compilation process. Therefore the advice is to only use UTF-16 statement encoding when the statement contains text that is best encoded in UTF-16. UTF-8 encoding should be used in all other cases to avoid the need for internal string conversions by SQLite.

7.4 SQLite Optimizations on Symbian OS

Thus far, we have discussed the general architecture of SQLite and examined how the library works in practice. The implementation of Symbian SQL presented the SQLite library with a new range of characteristics and requirements to consider because of the nature of the Symbian OS platform and the way in which Symbian SQL uses SQLite.

It is not surprising then that Symbian Software Ltd and the authors of SQLite have worked together since the advent of Symbian SQL to identify how it can get the best out of the SQLite library. This close collaboration, now in the formal context of the SQLite Consortium, has resulted in two things. Firstly, existing features of the SQLite library that can be used by Symbian SQL to optimize its performance have been identified and used. Secondly, new features and enhancements that would benefit Symbian SQL have been identified and implemented in SQLite. In keeping with the open source philosophy of SQLite, these changes have been committed to the public SQLite codebase and are freely available to all other users of the library.

This section focuses on the major changes that have been made to optimize the performance of Symbian SQL. For the sake of simplicity the optimizations are discussed in chronological order with regards to the version of Symbian SQL in which they were introduced. Section 7.4.1 covers Symbian SQL on Symbian OS v9.3, Sections 7.4.2 through 7.4.7 cover Symbian SQL on Symbian^2and Sections 7.4.8 and 7.4.9 cover Symbian SQL on Symbian^3. Each optimization is targeted at improving one or more aspects of Symbian SQL, such as improving the speed of particular database operations or reducing RAM consumption. These are important considerations for applications that run on Symbian OS.

Throughout this section, performance comparisons of different versions of Symbian SQL are included where necessary to illustrate the positive impact of the optimizations. The performance improvements are almost entirely a result of changes that have been made to the SQLite library. Changes that have been made in the Symbian SQL code have mostly been required to use the new SQLite features and enhancements.

You should note that the performance figures that are presented in this section are for operations performed on similar databases, stored on the internal NAND drive of a mobile device:

- Database A is the basic database. It contains a single table that has 10 columns: four INTEGER columns, one REAL column, four TEXT columns and one BLOB column, in that order. The table contains 1000 records (with no BLOBs) that are each approximately 200 bytes in size. The text in the database is Roman.
- Database B is the same as database A, with a non-unique integer index on the 1st column.
- Database C is the same as database B, with a 1000-byte BLOB in the last column of each record. Each record is approximately 1200 bytes in size.
- Database D is the same as database A but with the first 100 records deleted (therefore the database contains free pages).

All four databases are configured to have auto-vacuum set to none, a page size of 1 KB and UTF-8 database encoding.

7.4.1 Sharing the Cache

A critical consideration for an application that runs on a mobile platform, such as Symbian, is the inherent memory constraints of the platform. It is very important that Symbian SQL is as memory efficient as possible and it must strike a sensible balance between maximizing the speed of database operations and minimizing (or at least controlling) RAM usage.

SQLite caches recently used pages in memory. Since it was first released on Symbian OS v9.3, Symbian SQL has enabled the shared cache mode of the SQLite library in order to help control the amount of RAM that SQLite demands in its usage of page caches.

The benefit of operating in shared cache mode is twofold. Firstly, having fewer page caches means that less RAM is required by SQLite. Secondly, when two or more connections to the same database share a page cache means pages that are read into the cache by one connection can potentially be used by any of the connections, thus reducing the amount of file I/O that is required by each individual connection to access pages of the database.

7.4.2 Retaining Page Cache Contents

In all versions up to and including SQLite v3.3.13, the page cache is flushed at the end of every transaction on a database. This precautionary step is taken to avoid the possibility of cached pages becoming out of date when multiple connections are modifying the same database.

A mechanism was introduced in SQLite v3.3.14 to ensure that a page cache is flushed only when it is necessary, i.e. only when the contents of the cache are detected to be out of date. The mechanism uses the write counter in the header of every database file, which is incremented whenever the database is modified. SQLite v3.3.14 checks and remembers the value of the write counter at the end of a transaction on the database, instead of flushing the page cache. At the start of the next transaction on the database, SQLite compares the known value of the write counter with the current value in the database header. If the values are the same then SQLite knows that the database hasn't been modified between the two transactions and so the contents of the cache are still valid and can be reused. If the values are different then SQLite knows that the database has been modified by another transaction and therefore the cache is flushed because its contents may be out of date. By flushing a page cache only when it is required, SQLite can avoid many unnecessary reads to repopulate the cache.

Another significant enhancement that was made in SQLite v.3.3.14 was the introduction of a new operating mode for SQLite called *exclusive access mode*. When operating in exclusive access mode, SQLite assumes that it has a single client that has exclusive access to the databases. On the basis of this assumption, SQLite can ‘turn off’ the built-in checks and precautions that the library must perform to support multi-threaded database access in its normal mode of operation. For example, when operating in exclusive access mode, SQLite maintains an exclusive lock on each database file that it opens. All subsequent locking and unlocking activity can be omitted while exclusive access mode is enabled.

Exclusive access mode provides a number of advantages for single-process client environments, such as Symbian SQL, and the SQLite library is configured to operate in this mode on Symbian^2. One advantage is that the mechanism to avoid unnecessary page cache flushes is simplified. When operating in exclusive access mode, SQLite is afforded the luxury of *never* having to flush a page cache between transactions – indeed, of never having to check whether a page cache flush is required. This is possible because exclusive access mode assumes that there is only one client and so there is no need to check whether a database has been modified by another client and no need to flush the cache. A single page read operation may be avoided at the beginning of a transaction because the value of the write counter in the database header does not need to be checked.

Retaining the contents of a page cache is one of the most significant optimizations that has been made to Symbian SQL in terms of improving the speed of database operations, particularly read operations. Figure 7.9 illustrates how this change in Symbian^1 provides a valuable advantage over previous releases. The use case is ‘select all of the columns of 50 records, update a column in the 50 records and then update another column in the 50 records’, executed in three separate transactions on database B. The target records are identified using a range search on the indexed column.

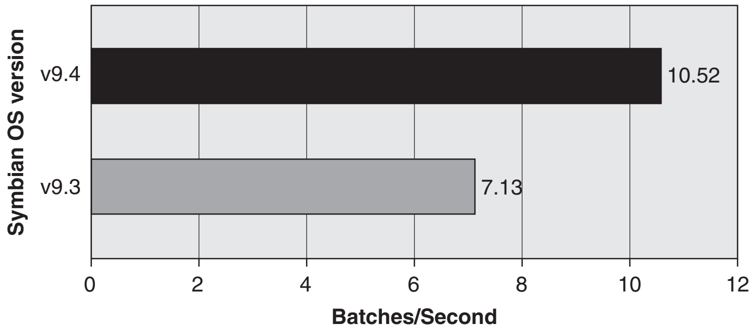


Figure 7.9 Performance comparison: updating 50 records

This use case is carried out 1.5 times faster on Symbian^1 than on Symbian OS v9.3. The first transaction (select all of the columns of 50 records) adds at least 10 pages to the page cache (there are four to five records per page). The second transaction (update a column of the 50 records) benefits from the fact that the page cache is not flushed between transactions on Symbian^1 and so the 50 records are already in the cache, which avoids having to fetch the 10 or more pages from disk again, which is required on Symbian OS v9.3. Similarly, the third transaction (update another column in the 50 records) can proceed on Symbian^1 without the need to repopulate the cache. Being able to avoid reading at least 20 pages during this sequence of operations is a key reason for the performance improvement that is experienced on Symbian^1.

7.4.3 Imposing a Soft Heap Limit

Symbian SQL's use of SQLite's shared cache mode means that there is a single page cache for each open database. However, depending on the number of databases that are open simultaneously and the maximum size of each page cache, a large amount of RAM can be required, especially as SQLite needs to allocate and use memory for other purposes too.

SQLite allows a *soft heap limit* to be set for the library via a call to the function `sqlite3_soft_heap_limit()`. This feature grants clients of the SQLite library some control over the overall amount of heap memory that the library allocates. It is a *soft* heap limit because SQLite does not guarantee that it will stay under this limit in the amount of heap memory that it allocates. The library makes every effort to stay under the limit by finding and reusing memory that is not being used for something essential – for example, memory that is being used by a page cache to store unmodified database pages – but if it cannot find any memory to reuse then it allocates new memory, which may take its total memory usage above the soft heap limit.

This approach of trying to reuse free space rather than unreservedly allocating new memory again and again is very important in memory-constrained environments. In the first release of Symbian SQL (on Symbian OS v9.3), no soft heap limit is set and so the SQLite library may allocate more and more memory until the maximum heap limit is reached. In subsequent releases of Symbian SQL – on Symbian^2 onwards – the SQLite library is configured to have a soft heap limit of 1 MB by default. SQLite therefore tries not to consume more than 1 MB of RAM on Symbian^2 onwards.

It is worth noting that device manufacturers who use Symbian SQL are able to change the soft heap limit to be a different value than the default 1 MB. It can be useful to be able to change the default soft heap limit based on knowledge of the RAM capacity of a particular phone model.

7.4.4 Increasing the Maximum Cache Size

Before Symbian SQL imposed a soft heap limit on SQLite, it had to control its total memory usage by using a modest maximum page cache size – 64 pages by default on Symbian OS v9.3. Following the implementation of a soft heap limit, which allows a page cache to reuse memory, this restriction is no longer necessary for controlling memory usage and the maximum page cache size can safely be increased.

Specifically, the maximum page cache size that is used by Symbian SQL on Symbian^2 onwards is calculated dynamically based on the soft heap limit and the page size that has been set for the database:

maximum page cache size = soft heap limit / page size

This means that a maximum page cache size of 1 MB/1 KB = 1024 pages is set for a database that has the default page size of 1 KB. For a database that has a page size of 2 KB, a maximum page cache size of 1 MB/2 KB = 512 pages is set.

The impact of this change to the maximum page cache size, from 64 pages to typically 1024 pages, is significant. A larger page cache means that more pages can be stored in the cache at the same time, which increases the likelihood of a target page already being in the cache, thus avoiding the need to perform database file I/O to retrieve it. In addition, using a larger page cache reduces the likelihood of page thrashing during database operations and so the frequency with which SQLite must write modified cache pages to disk is less. Like the optimization to retain page cache contents between transactions (refer to Section 7.4.2), the change to increase the maximum size of a page cache can significantly improve the speed of database operations, particularly read operations (which typically form the majority of use cases in a database application).

Figure 7.10 illustrates the impact of this change. The use case involves frequent reads of many records of a table – specifically, the execution of ‘select the second column of all of the records’ five times, in separate transactions, on database A. The table contains 1000 records and each transaction has to retrieve the second column from all of them.

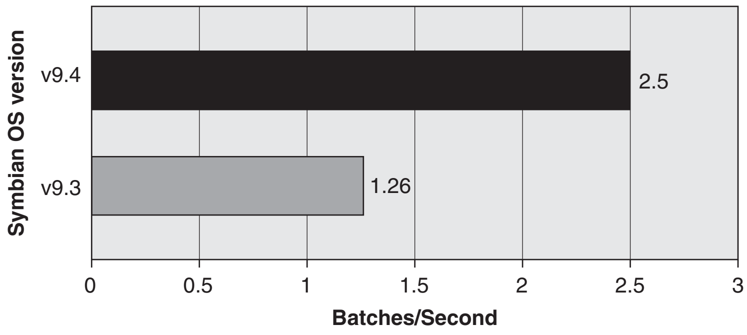


Figure 7.10 Performance comparison: read operations on many pages

This use case runs twice as fast on Symbian^2 than on Symbian OS v9.3. When the first transaction is executed, the entire table has to be read into the empty page cache on both Symbian OS v9.3 and Symbian^2. It is quicker on Symbian^1 because the cache can hold up to 1024 pages and so the entire table can fit in it at once (between 200 and 250 pages are used to store the 1000 records). On Symbian OS v9.3, only 64 pages at a time can be held in the cache and so pages have to be swapped in and out to process all 1000 records.

The other significant difference is that when the second and subsequent transactions are executed on Symbian^2, no file I/O is required because the cache is not flushed between transactions and so the entire table is already in the cache. On Symbian OS v9.3, the cache is flushed at the end of each transaction and so the pages must be read again from disk during each transaction. This again involves swapping pages in and out of the cache to process all 1000 records.

It is clear that the page cache optimizations in Symbian SQL on Symbian^2 mean that it is potentially possible (depending on the application use cases) for a cache to quickly become populated with many pages containing a wide set of records. When this happens over the course of an application's lifetime, it is increasingly likely that the pages required by any transaction are already in the cache. The benefit of the cache retention policy is then not restricted to transactions that operate on a *common* set of pages.

To illustrate this point, consider Figure 7.11. The use case is a sequence of 20 transactions, each one executing 'select the second column' on different and diverse choices of five records from the table in database B. The records are identified using the 'IN' operator on the indexed column. The table has 1000 records.

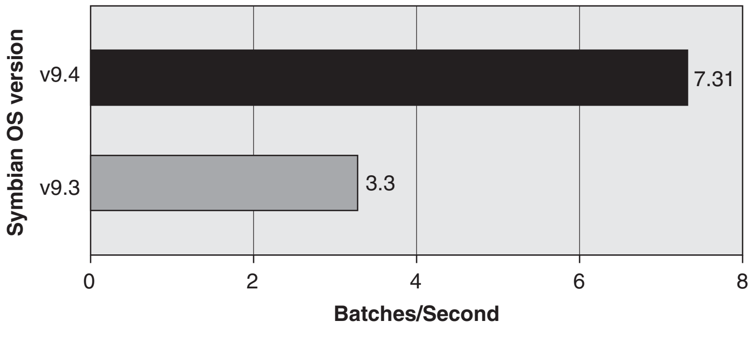
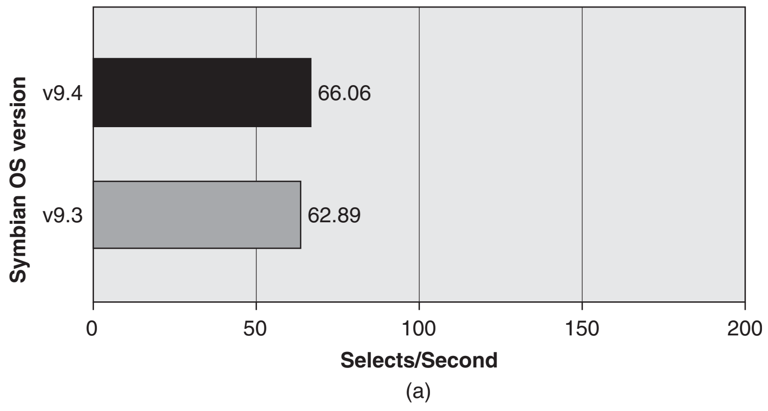
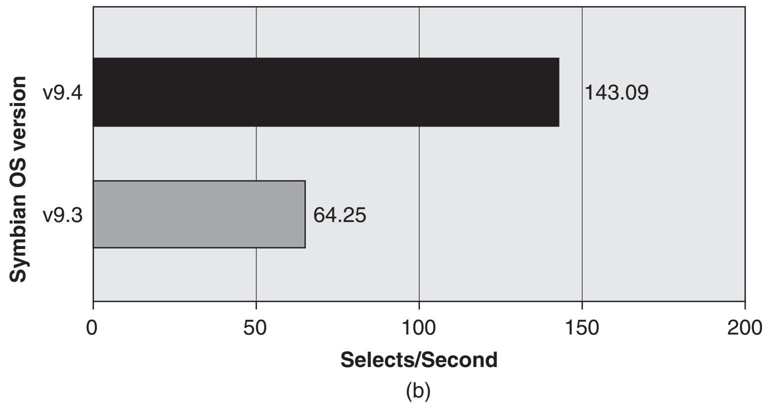


Figure 7.11 Performance comparison: accessing random pages

This use case is over twice as fast on Symbian^2 than on Symbian OS v9.3. The following graphs give an insight into the reasons behind this performance improvement. To begin with, Figure 7.12a illustrates





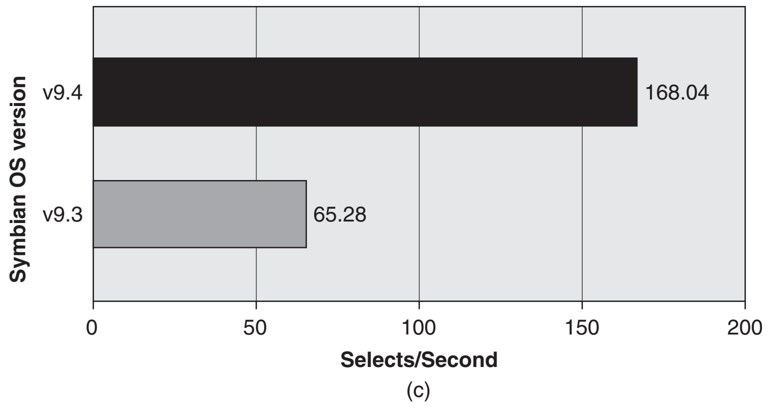


Figure 7.12 Performance comparison: selecting five records at random from 1000 a) first set, b) 10th set, and c) 20th set

the relative performance on Symbian OS v9.3 and Symbian^1 for the first transaction in the sequence of 20. Starting with an empty cache on Symbian OS v9.3 and Symbian^2, the first transaction is only slightly faster on Symbian^2. Now consider how Symbian^2 outperforms Symbian OS v9.3 for the 10th transaction in the sequence of 20, as shown in Figure 7.12b.

Notice that the execution time on Symbian OS v9.3 is more or less the same for the first and 10th transactions. However, on Symbian^2, the 10th transaction is much faster than the first transaction and the critical factor in this is the cache retention policy on Symbian^2. This gives it a huge advantage over Symbian OS v9.3; by the time the 10th transaction occurs, the page cache has become populated with various pages from throughout the table and not so many file read operations are required (if any) to access the five target records. The power of the cache retention policy is most evident in Figure 7.12c, which illustrates the performance of the final transaction in the sequence of 20.

Again, notice that the execution time stays constant on Symbian OS v9.3. However, the 20th transaction is faster than the 10th transaction on Symbian^2. This is because, by the time the 20th transaction occurs, the page cache on Symbian^2 contains even more pages than before and so it is even more likely that the five target records for this transaction are already in the cache.

7.4.5 Avoiding Unnecessary Reads

File I/O is clearly necessary in order to write changes to a database file and to read pages from a database file that are not already in the page cache. However, file I/O is responsible for much of the cost of performing database operations and many of the optimizations to SQLite are targeted at minimizing the frequency with which SQLite forces the host platform to perform file I/O. The page cache optimizations that have been discussed thus far greatly reduce the amount of file I/O that is likely to be required to perform a database operation. Symbian and the authors of SQLite identified other areas where there was potential for SQLite to reduce the amount of file I/O requests that it makes. In this section, we look at the changes that were made to avoid unnecessary read operations.

Deleting the Last Overflow Page Without Reading It

When a record that has a chain of overflow pages is deleted, each page of the overflow chain must be read in order to determine the number of the next page in the chain. However, reading the content of the last page in the overflow chain is pointless and so SQLite can avoid a single page read operation when deleting the record.

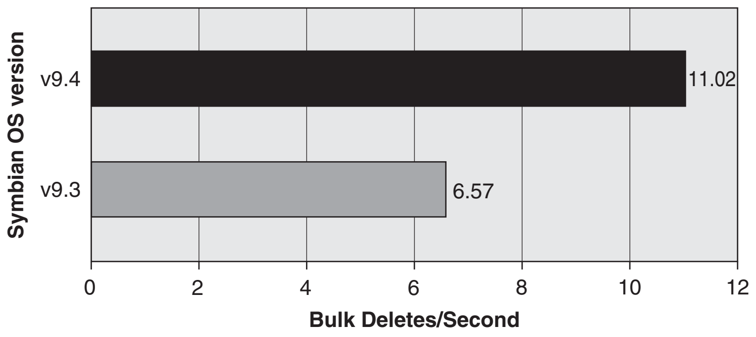


Figure 7.13 Performance comparison: deleting many records with overflow pages

As a result of this optimization, the deletion of a record that has one or more overflow pages is a faster operation on Symbian^2 than on Symbian OS v9.3. Figure 7.13 demonstrates the performance improvement that is achieved for the use case of deleting from database C in a single transaction 100 records, each with a single overflow page. The 100 records to be deleted are identified using a range search on the indexed column.

This use case is 1.7 times faster on Symbian^2 than on Symbian OS v9.3. The reading of 100 pages is avoided on Symbian^2 – the final (and only) overflow page of each of the 100 records deleted. Another non-trivial factor in the performance improvement is again the difference in the maximum page cache size on Symbian OS v9.3 and Symbian^2. On Symbian^2, the initially empty cache can easily accommodate all of the pages of the 100 deleted records. However, on Symbian OS v9.3, the 64-page limit of the cache is reached during the delete operation and it is necessary for SQLite to swap out cached pages (and write the modifications to the database file) to make room for the remaining records that are to be deleted.

Note that if each record had several overflow pages then there would be less of a measured performance improvement. Avoiding a single page read operation when there are many other overflow pages to be read does not have as much of an impact on performance.

Reusing a Free Page Without Reading It

If the auto-vacuum setting of a database is 'none' or 'incremental' and data is deleted from the database then any free pages are added to the free page list and are reused for subsequent database operations. When SQLite wishes to reuse a free page and therefore moves it from the free page list into, say, a table, the content of the free page is overwritten. Therefore is not necessary to read the content of the free page when it is moved from the free page list. By assuming that the content of the free page is all zeros, this unnecessary read can be avoided by SQLite.

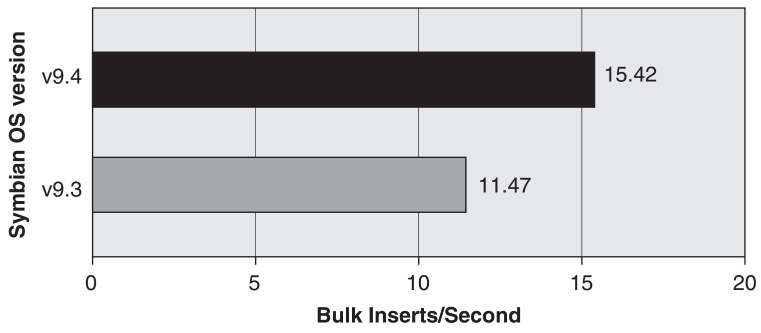


Figure 7.14 Performance comparison: reusing free pages

To illustrate the performance improvement that is gained from this optimization, consider Figure 7.14. The use case is 'insert 50 records' into the table in database D, in a single transaction. The free page list in the database contains at least 20 pages and between 10 and 15 pages are required to store the 50 new records (four to five records can be stored per page).

This use case is 1.3 times faster on Symbian^2 than on Symbian OS v9.3. A factor in this performance improvement is that SQLite is able to avoid reading at least 10 pages on the free page list on Symbian^2. The other factor is that, on Symbian^2, the journal file is truncated to zero length instead of being deleted at the end of the transaction (see Section 7.4.7).

7.4.6 Avoiding Unnecessary Writes

Symbian Software and the authors of SQLite also identified several places where SQLite was performing unnecessary write operations. This section describes the changes that were made to eliminate these unnecessary writes.

Updating the Database Write Counter

When operating in exclusive access mode, as SQLite is configured to do in Symbian^2 onwards, the write counter in a database header is only incremented for the first transaction that changes the database, not for every transaction that changes the database. This change is possible due to the fact that no other process can modify the database when SQLite is operating in exclusive access mode. The write counter must be incremented once to register that at least one change has occurred (to satisfy the semantics of the counter), but it is not necessary to increment the write counter each time the database file is changed.

As a result of this optimization, the cost of writing to page 1 of a database file can sometimes be avoided. A write to page 1 is still necessary if a transaction modifies the free page list, for example, but for the common case where the free page list is not modified, both reading and writing page 1 can be avoided.

Avoiding Page Journaling

A client can execute a statement of the following form to delete all of the records in a table:

```
DELETE FROM MusicLibrary
```

or a statement of the following form to drop a table:

```
DROP TABLE MusicLibrary
```

When SQLite executes either of these statements, it moves all of the pages of the table into the free page list. These statements do not normally involve changing the content of the table pages and so the pages do not need to be written to the journal file. SQLite can potentially avoid many file writes by not journaling the table pages during the execution of these statements.

As a result of this optimization, deleting all of the records in a table and dropping a table are now substantially faster operations on Symbian^2 than they were on Symbian OS v9.3. In effect, N page writes are avoided on Symbian^2, where N is the total number of pages that are used to store the table records. Note that it is only the *delete all* use case (with no WHERE clause in the DELETE statement) that has been optimized to such a degree.

Figure 7.15 illustrates the performance of the use case 'delete all records' from the table in database A. The table contains 1000 records that are approximately 200 bytes in size each.

This use case is 3.9 times faster on Symbian^2 than on Symbian OS v9.3. A major factor in this performance improvement is that at least 200 pages are required to store the 1000 records (there are four or five records per page) and writing these 200 pages to the journal file is avoided during the 'delete all' operation on Symbian^2. Also, all of the table pages can be read into the cache (this is part of the 'delete all' operation) on Symbian^1 without any need to swap pages in and out of the cache (a maximum of 1024 pages are permitted). On Symbian OS v9.3, cache page swapping is necessary when the 64-page cache limit is reached.

Dropping the table in database A using a DROP TABLE statement demonstrates the same level of performance improvement on Symbian^1 as deleting all records from the table, for the same reasons as above.

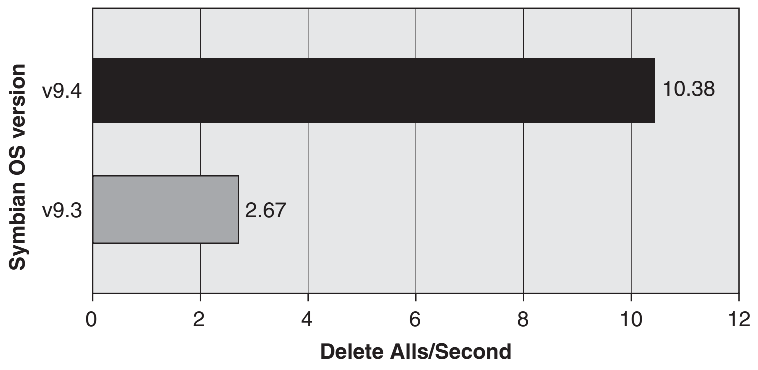


Figure 7.15 Performance comparison: deleting all records from a table

7.4.7 Minimizing the Cost of Using Journal Files

Journal files play an important role in the operation of SQLite but they impact the time taken to execute transactions. Symbian Software and the authors of SQLite identified several places where SQLite's handling of journal files could be improved. As a result, changes were made in SQLite to the way that journal files are handled, as outlined below, and Symbian^2 takes advantage of these changes.

Avoiding Journal File Deletion

Historically, SQLite creates one journal file at the beginning of a write transaction and then deletes it at the end of the transaction. Creating and deleting files can be expensive on some operating systems, including the Symbian platform, where it was identified that it is cheaper to truncate a file to zero length than to delete the file. As a result, SQLite was modified in v3.3.14 so that, when the library is operating in exclusive access mode, it truncates journal files to zero length instead of deleting them. In this mode, SQLite treats a zero-length journal file as if it does not exist, to avoid the journal file being viewed as 'hot' and thus avoiding inappropriate database rollback attempts. Symbian SQL on Symbian^2, which configures SQLite to operate in exclusive access mode, benefits from this journal file optimization.

Using zero-length journal files means that a journal file is only created by SQLite for the first transaction on the database that requires one and it does not need to be created for subsequent transactions. Many file delete operations are also avoided because the journal file is not deleted at the end of a transaction.

To illustrate the difference that avoiding journal file deletion can make, consider Figure 7.16. The use case is a transaction to 'update a text column of a single record' in the table in database B. The record

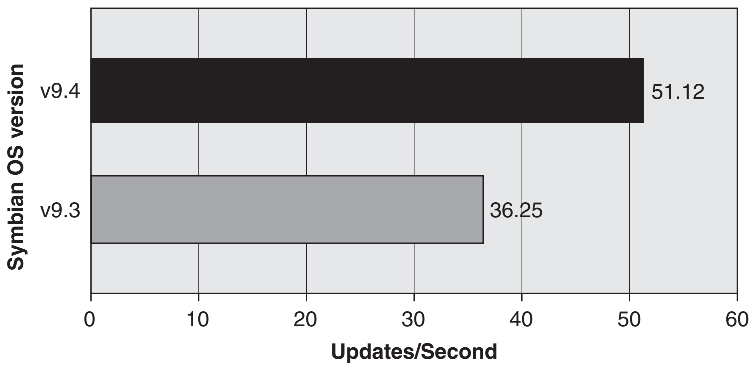


Figure 7.16 Performance comparison: updating a single record and truncating the journal file

is identified using an equality search on the indexed column and thus it takes a short amount of time to execute the transaction.

This use case is 1.4 times faster on Symbian^2 than on Symbian OS v9.3. A key factor in this performance gain is that the journal file is truncated to zero length instead of being deleted at the end of the transaction. An important point to note here is that the example transaction takes a short amount of time to execute. The measured performance improvement would be less if the transaction were more time-consuming because the time saving made by the journal file optimization would have less of an impact on the total execution time.

It was subsequently discovered by Symbian and the authors of SQLite that there is an even cheaper alternative than truncating a journal file to zero length. By writing zeros to the start of a journal file, it can be marked as ‘empty’ even though the journal file still exists and is not zero length. Writing zeros to the start of a file is a faster operation on Symbian OS than truncating the file to zero length. As a result of this observation, a new `journal_mode` PRAGMA was included in SQLite v3.6.1, which is used on Symbian^3. When SQLite is asked to operate in PERSIST journal mode, it writes zeros to the start of the journal file at the end of a write transaction, instead of truncating or deleting the journal file.

An obvious drawback of PERSIST journal mode is that a journal file can grow and grow – it is never deleted or truncated to zero length. The size of a journal file is always the size of its ‘biggest’ write transaction to date because each write transaction overwrites the existing journal file content rather than appending to it. In order to be able to keep some control over the size of a journal file, clients of SQLite can use the `journal_size_limit` PRAGMA to set an upper limit for the size of the journal file for each individual database. If a journal file expands to a size that exceeds this limit during a transaction then the journal file is truncated (back down to the set upper limit) after the transaction has completed. By default, Symbian SQL caps the size of a journal file to 2 MB on Symbian^3.

Checking for a Hot Journal File

In its default mode of operation, SQLite must check for a hot journal file at the beginning of each transaction on a database; if a hot journal file exists, then SQLite must rollback the database because a previous transaction that was executed on the database may have failed to complete.

When operating in exclusive access mode, as SQLite is configured to do from Symbian^2, things are different. Since the premise of operating in exclusive access mode is that only one process can modify the database files, there is no need to check for the presence of a hot journal file at the beginning of each transaction on a database. A check must certainly be made for the presence of a hot journal file at the beginning of the *first* transaction on a database (in case the single process itself crashed during a database transaction), but there is no need to check at the beginning of *every* transaction. This optimized behavior is enabled when SQLite is operating in exclusive access mode and it avoids the need for file system checks at the beginning of transactions (other than the first one).

7.4.8 Mass Storage and Fast Retrieval of Binary Data (BLOBs)

Mobile devices are increasingly being used as portable disk drives as the size of internal storage space that they offer continues to grow. For example, the popular Nokia 5800 device comes with an 8 GB microSD memory card for storing images and music tracks, and so on. The storage and retrieval of large amounts of binary data is becoming everyday practice for many people and Symbian devices must be able to meet the demands of mass storage and fast access to such data.

Historically, SQLite has treated binary data that is stored in BLOB columns in the same way as the other SQLite data types – namely, each column in a database record contains data that is handled in its complete form. This is not ideal when large BLOBs are being stored because the whole BLOB has to be read into memory by SQLite before it can be manipulated (or even sometimes when another column in the record needs to be manipulated). Symbian SQL provides the legacy streaming classes `RSqlColumnReadStream` and `RSqlParamWriteStream` so that at least the client-side memory usage can be controlled when handling BLOBs, but the whole BLOB must still be loaded into server-side RAM before streaming to or from the client can begin. Figure 7.17 demonstrates how a 2 MB BLOB can be read from a database using the legacy APIs of Symbian SQL.

The top two sections of Figure 7.17 illustrate retrieval in which the entire 2 MB BLOB is retrieved by the client in one call using

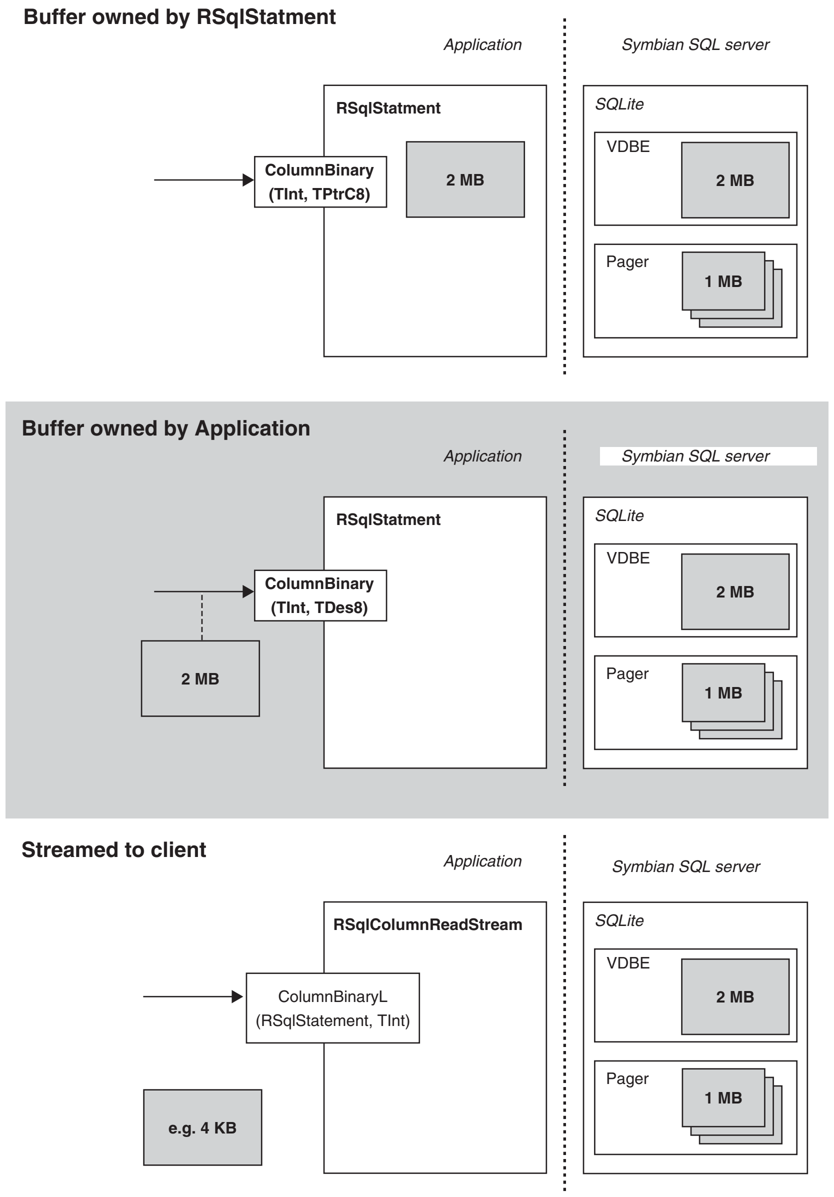


Figure 7.17 Consumption of RAM by the legacy Symbian SQL APIs for BLOB retrieval

the `RSqlStatement::ColumnBinary()` APIs. Two megabytes of client-side RAM is required to successfully use these APIs. The bottom section of the diagram illustrates how the streaming API `RSqlColumnReadStream::ColumnBinaryL()` can be used by the client to retrieve the BLOB in blocks whose size is dictated by the client. This can reduce the amount of client-side RAM that is required to retrieve the 2 MB BLOB.

The important observation to be made is that up to 3 MB of *server-side* RAM is required for both the whole BLOB retrieval and streaming use cases. As shown in Figure 7.17, the page cache size is capped at approximately 1 MB on Symbian^1 and on Symbian^2 due to the SQLite soft heap limit set by Symbian SQL, but a 2 MB buffer must be allocated to hold the entire BLOB when the VDBE program needs to retrieve it. All three of the legacy APIs that are illustrated in Figure 7.17 internally use the SQLite column accessor function `sqlite3_column_blob()` to retrieve the entire BLOB into server-side memory.

A critical side effect of having to hold the entire BLOB in server-side RAM (which would also be the case if a client required to write to the BLOB) is that there is a maximum limit on the size of BLOB that can be created, read from or written to using the legacy Symbian SQL APIs, simply because there is a limit on the amount of server-side RAM that is available. Having a direct correlation between the amount of RAM that is available and the size of BLOB that can be stored is clearly unacceptable if Symbian SQL is to be able to support increasing amounts of binary data on devices now and in the future. For this reason, Symbian engaged with the authors of SQLite to produce a new SQLite BLOB-handling API that overcomes the memory constraints of the legacy SQLite API. This new API is referred to as the *Incremental BLOB I/O* API and it is outlined in the remainder of this section. A fundamental advantage that the Incremental BLOB I/O API offers over the legacy SQLite API is that it allows a BLOB to be read from and written to incrementally, in a more RAM-efficient manner. The Symbian SQL API was extended on Symbian^2 to provide clients with a way to utilize the Incremental BLOB I/O API.

Zeroblobs

A *zeroblob* is a fundamental aspect of the Incremental BLOB I/O API. A zeroblob can be thought of as a BLOB whose content is all zeros and it is intended to act as a placeholder for a 'real' BLOB whose content is later incrementally written using the `sqlite3_blob_write()` function. Zeroblobs have the advantage that they only use a small amount of space (to store their declared size) and so SQLite is able to handle zeroblobs very efficiently when they are being manipulated.

```
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
```

The sqlite3_bind_zeroblob() function binds a zeroblob to a parameter in the specified prepared statement. The integer index of the parameter that is to be set in the prepared statement is specified by the second parameter of the function. The third parameter of the function specifies the size in bytes of the zeroblob that is to be bound to the prepared statement. If this value is less than zero then a zeroblob of zero size is bound. The function returns SQLITE_OK on success, otherwise an error code is returned.

The RSqlStatement class provides a C++ wrapper API for this function in the form of BindZeroBlob(). See Chapter 6 for more details.

The SQLite core function zeroblob(N) can be used directly in an SQL statement string to create a zeroblob of N bytes. For example:

```
INSERT INTO ImageGallery (Name, Image) VALUES ("Janet Regis",
                                               zeroblob(5000))
```

BLOB Handles

```
typedef struct sqlite3_blob sqlite3_blob;
```

A handle to a BLOB is represented by a sqlite3_blob object. For the remainder of this section, the term 'BLOB handle' is used to refer to an object of type sqlite3_blob. Many functions in the Incremental BLOB I/O API take a BLOB handle as a parameter.

The RSqlBlob class is a wrapper around the SQLite BLOB handle type. Every RSqlBlob object corresponds to exactly one SQLite BLOB handle – it is a 1:1 relation. This is true even if two RSqlBlob objects are opened on the same BLOB – two distinct handles are created. See Chapter 6 for more details on the RSqlBlob class.

Opening a Handle to a BLOB

```
int sqlite3_blob_open(sqlite3*, const char* zDb,
        const char* zTable, const char* zColumn,
                sqlite3_int64 iRow, int flags,
                        sqlite3_blob** ppBlob);
```

The sqlite3_blob_open() function opens a handle to a BLOB. The target BLOB is specified using the first five parameters of the function. These parameters specify where the BLOB is located in a database. The parameters specify, in order, the database connection that is to be used to access the target database, the name of the target database, the name of the target table, the name of the target column and the ROWID of the actual record that contains the target BLOB. Note that each of the string parameters are interpreted as UTF-8 encoded strings. The database name that is specified in the second parameter is the symbolic name of the database (not the name of the database file) – this is `main` for the main database or the symbolic name that was specified when a database was attached. The sixth parameter is a flag that specifies whether the BLOB handle is to be opened for read-only access (if the value 0 is specified) or for read and write access (if a non-zero value is specified).

If a handle to the target BLOB is successfully created, then the function returns `SQLITE_OK` and the `ppBlob` output parameter contains a pointer to the BLOB handle. In addition, a new transaction is started on the specified database connection if it is not already in a transaction. The BLOB handle returned is valid until the record that the BLOB belongs to is modified or deleted, after which the handle is said to be 'expired'. Subsequent calls to `sqlite3_blob_read()` and `sqlite3_blob_write()` then fail with the return code `SQLITE_ABORT`. If a handle is not successfully opened by the `sqlite3_blob_open()` function then an error code is returned and it is not safe for the client to use the `ppBlob` output parameter.

The `RSqlBlobReadStream` and `RSqlBlobWriteStream` classes provide C++ wrapper APIs for this function in the form of `RSqlBlobReadStream::OpenL()` and `RSqlBlobWriteStream::OpenL()`. The former provides access to a BLOB as a read-only stream of bytes and the latter provides access to a BLOB as a readable and writeable stream of bytes. See Chapter 6 for more details.

Closing a Handle to a BLOB

```
int sqlite3_blob_close(sqlite3_blob*);
```

The `sqlite3_blob_close()` function closes the specified open BLOB handle. If the BLOB handle that is specified is successfully closed then the function returns `SQLITE_OK`, otherwise an error code is returned. In either case, the BLOB handle is closed by SQLite. When the handle is closed, the current transaction commits if there are no other open BLOB handles or prepared statements on the database connection and the database connection is not in the middle of an explicit transaction. Any file I/O errors that occur during the commit (for example, as the BLOB is being updated with cached changes) are reported by an error code returned by the function.

The `RSqlBlobReadStream` and `RSqlBlobWriteStream` classes provide C++ wrapper APIs for this function in the form of `RSqlBlobReadStream::Close()` and `RSqlBlobWriteStream::Close()`.

The former closes a read-only BLOB handle opened with RSqlBlob-ReadStream::OpenL() and the latter closes a readable and writeable BLOB handle opened with RSqlBlobWriteStream::OpenL(). See Chapter 6 for more details.

Reading from a BLOB

```
int sqlite3_blob_read(sqlite3_blob*, void* Z, int N, int iOffset);
```

The sqlite3_blob_read() function is used to read data from a BLOB through an open handle to the BLOB. The handle is specified by the first parameter to the function. The second parameter refers to a client-specified buffer into which the BLOB data is to be read and the third parameter specifies how many bytes of data are to be read from the BLOB. The function reads N bytes starting at the BLOB offset that is specified by the fourth parameter. This function can be used to read BLOB data in incremental chunks, rather than retrieving the entire BLOB in one call.

If the required data is successfully read from the BLOB then the function returns SQLITE_OK, otherwise an error code or extended error code will be returned. The error code SQLITE_ERROR is returned if the specified offset is less than zero or is less than N bytes from the end of the BLOB, or if the specified number of bytes to be read is less than zero. An attempt to read from an expired BLOB handle results in SQLITE_ABORT being returned by the function.

The TSqlBlob and RSqlBlobReadStream classes provide C++ wrapper APIs for this function in the form of TSqlBlob::GetLC(), TSqlBlob::Get() and RSqlBlobReadStream::ReadL(). The Get APIs retrieve the entire content of the BLOB in one call and they require enough client-side RAM to hold the entire BLOB. The Read API allows the client to stream data from the BLOB in blocks in order to control the amount of client-side RAM that is required to be allocated. See Chapter 6 for more details.

The key difference between the above TSqlBlob and RSqlBlob-ReadStream APIs and the legacy APIs RSqlStatement::Column-Binary() and RSqlColumnReadStream::ColumnBinaryL() (which are illustrated in Figure 7.17) is that the new APIs internally use the sqlite3_blob_read() function (rather than the sqlite3_column_blob() function). This allows the Symbian SQL server process to read the target BLOB into server-side memory in incremental blocks, rather than as a whole. In Figure 7.17, if one of the new APIs was used to read the 2 MB BLOB then the need for a 2 MB server-side buffer would be eliminated. Instead, the data would be transferred to the client in server-side blocks of up to 32 KB.

Writing to a BLOB

```
int sqlite3_blob_write(sqlite3_blob*, const void* z, int n,
                                              int iOffset);
```

The `sqlite3_blob_write()` function is used to write data to a BLOB through an open handle to the BLOB. The handle is specified by the first parameter to the function. The second parameter refers to a client-specified buffer that contains the data to be written and the third parameter specifies how many bytes of the data are to be written to the BLOB. The function takes N bytes of data from the buffer and writes it to the BLOB starting at the offset that is specified by the fourth parameter. This function can be used to write BLOB data in incremental chunks, rather than creating or updating the entire BLOB in one call. Note that this function cannot be used to increase the size of a BLOB, only to modify its current content. It is also worth noting that data that is written to a BLOB using this function may be held in the page cache and only committed to disk when the `sqlite3_blob_close()` function is called.

If the required data is successfully written to the BLOB then the function returns `SQLITE_OK`, otherwise an error code or extended error code is returned. The error code `SQLITE_ERROR` is returned if the specified offset is less than zero or N bytes from the end of the BLOB, or if the specified number of bytes to be written is less than zero. An attempt to write to an expired BLOB handle results in `SQLITE_ABORT` being returned by the function. The error code `SQLITE_READONLY` is returned if the specified BLOB handle was opened in read-only mode.

The `TSqlBlob` and `RSqlBlobWriteStream` classes provide C++ wrapper APIs for this function in the form of `TSqlBlob::SetL()` and `RSqlBlobWriteStream::WriteL()`. The first API can be used to set the entire content of the BLOB in one call and it requires enough client-side RAM to provide the entire BLOB to the function. The second API allows the client to stream data to the BLOB in blocks in order to control the amount of client-side RAM that is required to be allocated. See Chapter 6 for more details.

The key difference between the `TSqlBlob` and `RSqlBlob-WriteStream` APIs and the legacy APIs `RSqlStatement::BindBinary()` and `RSqlParamWriteStream::BindBinaryL()` is that the new APIs internally use the `sqlite3_blob_write()` function (rather than the `sqlite3_bind_blob()` function). This allows the Symbian SQL server process to write to the target BLOB in incremental blocks rather than as a whole. If one of the new APIs were used to store the 2 MB BLOB in Figure 7.17 then there would be no need for a 2 MB server-side buffer. The server would use blocks of up to 32 KB to incrementally write the 2 MB of data to the BLOB.

Retrieving the Size of a BLOB

```
int sqlite3_blob_bytes(sqlite3_blob*);
```

The `sqlite3_blob_bytes()` function returns the size (in bytes) of the BLOB on which the specified handle is open.

The `RSqlBlobReadStream` and `RSqlBlobWriteStream` classes provide C++ wrapper APIs for this function in the form of `RSqlBlobReadStream::SizeL()` and `RSqlBlobWriteStream::SizeL()`. See Chapter 6 for more details.

To summarize the advantages of the Incremental BLOB I/O API and the corresponding enhancements to the Symbian SQL API, clients of Symbian SQL on Symbian^3 experience the following benefits:

- **Larger BLOBs** can be stored on Symbian^3 than was possible on Symbian OS v9.3 and Symbian^1. On Symbian^3, the size of BLOB that can be stored is not influenced by the amount of server-side RAM that is available.
- **Less read latency of BLOBs** due to the incremental approach that is used to retrieve the BLOB data in blocks. This can improve the responsiveness of client applications that require fast access to BLOB data.
- **More RAM-efficient reading and writing of BLOBs**, thus reducing the overall RAM usage of the phone.

7.4.9 Background Database Compaction

Database compaction, or 'vacuuming', is used to reclaim free space in a database file and thus reduce its size. Historically, SQLite offered two compaction options via the `auto_vacuum` PRAGMA, as described in Section 7.3.1. By default, Symbian SQL sets the auto-vacuum setting to 'full' for every database created on Symbian OS v9.3 and Symbian^1.

When a database has auto-vacuum set to 'full', the compaction step adds to the total execution time of the transaction. With this in mind, Symbian and the authors of SQLite identified a gap in SQLite's available auto-vacuum options and, as a result, a third auto-vacuum setting, 'incremental', was introduced in SQLite v3.4.0.

The aim of incremental vacuumming is to provide clients of SQLite with a compaction option that gives a better trade-off between the performance of transactions and efficient file compaction. If the auto-vacuum setting of a database is 'incremental' then no automatic compaction occurs on the database and a client must execute the `incremental_vacuum` PRAGMA to perform immediate compaction on the database. The number of pages that the client wishes to be reclaimed from the free page list must be specified in the PRAGMA call. Up to this number of pages are removed from the free page list (there may be fewer pages on the free page list than the number specified) and the database file is truncated by the amount of free space that is reclaimed.

The Symbian SQL API was enhanced on Symbian^3 to take advantage of the incremental auto-vacuum option. Clients of Symbian SQL now have a choice of three database compaction modes, as described in the following list. The compaction mode for a database can be specified when the database is created, using the configuration string parameter of the `RSqlDatabase::Create()` and `RSqlDatabase::CreateL()` APIs. For example, the configuration string `compaction=manual` should be specified to configure a database to use the manual compaction mode.

- **Manual compaction**: If a database has been configured to use manual compaction then no automatic compaction occurs on the database and it is the responsibility of clients to request compaction to occur. A compaction request is made by a call to one of the `RSqlDatabase::Compact()` APIs. There is a synchronous and an asynchronous variant of this method (see Chapter 6 for more details). The client must provide an upper limit for the amount of free space (in bytes) that is to be reclaimed in this compaction call. Care should be taken to avoid blocking the client or the Symbian SQL server by making a request to reclaim a lot of free space in a single call. The asynchronous variant of the `RSqlDatabase::Compact()` API can be used to avoid blocking the client, but the single-threaded server process needs to complete the compaction request before handling any other client requests. Symbian SQL internally sets the auto-vacuum setting to 'incremental' for a database that has been configured to use manual compaction. If no client of the database ever makes a call to `RSqlDatabase::Compact()` then the database file never shrinks, which is effectively the same behavior as a database that has been configured with auto-vacuum set to 'none'.

- **Synchronous compaction**: If a database has been configured to be synchronously compacted then the database is automatically compacted at the end of every transaction. This is the default setting for databases that are created using Symbian SQL on Symbian OS v9.3 and Symbian^1. Symbian SQL internally sets the auto-vacuum setting to 'full' for a database that has been configured to be synchronously compacted.

- **Background compaction**: If a database has been configured to use background compaction then the database is automatically compacted but the compaction step does not occur as part of a transaction and it does not necessarily occur immediately after a transaction has completed. This is the default setting for databases that are created using Symbian SQL on Symbian^3. Symbian SQL internally sets the auto-vacuum setting to 'incremental' for a database that has been configured to use background compaction. The Symbian SQL server process is then able to internally use the `incremental_vacuum` PRAGMA at its discretion to perform compaction 'in the background.' The server process attempts to perform some compaction on a database when the amount of free space in the database exceeds an internal threshold (the default threshold that is set by Symbian SQL is 50 KB). When this threshold is breached, the server process begins a background compaction cycle which uses a timer to alternate between performing some compaction on the database and checking for incoming client requests. This allows the server process to remain responsive to clients of the database.

Although a database that has been configured to use background compaction may not always be fully compacted at any point in time, it does allow automatic compaction to occur on the database without the overhead of the compaction step being included in the time taken to execute a transaction. This provides a better trade-off between the performance of transactions and efficient file compaction, while still providing the self-administering benefits of automatic compaction. Clients of a database that has been configured to use background compaction can also manually perform some compaction by calling the `RSqlDatabase::Compact()` APIs.

This optimization in Symbian^3 has a number of benefits:

- Clients of Symbian SQL have a wider choice of compaction modes when configuring their databases.
- Databases that are configured to use background compaction are likely to experience better performance for some use cases, such as DELETE operations, than databases that are configured to be synchronously compacted.
- Legacy Symbian SQL databases which were created with auto-vacuum set to 'full' are automatically migrated to use background compaction when they are first opened on Symbian^3. These legacy databases then benefit from the performance improvements that can be experienced when using background compaction.

7.5 Summary

Now that we have reached the end of this chapter, you should have a thorough understanding of the core architecture of SQLite and how it has been optimized for Symbian SQL running on the Symbian platform.

Of course, both SQLite and Symbian SQL continue to evolve and will do so for a long time to come. We cannot predict how future versions of SQLite and Symbian SQL might differ from today's versions. What is certain is the commitment of the Persistent Data Services team and the authors of SQLite to continue to work together to identify performance bottlenecks and potential improvements in SQLite and in how Symbian SQL uses SQLite. Moreover, feature enhancements and optimizations continue to be added to the SQLite and Symbian SQL codebases for release in future versions. Under the umbrella of the SQLite Consortium, this joint development will continue for the foreseeable future and will ensure that the marriage of SQLite and Symbian SQL continues to be a well-suited and harmonious one. To keep up to date, please visit the wiki page for this book, ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***, regularly; we'll add links to news and update information when significant changes occur.

8
Performance Tuning

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Donald Knuth

Getting top performance from a database management system is a fine art. Symbian SQL is no different from other environments in this respect. There are so many parameters to consider and tweak at so many levels that investigating and analyzing all possibilities is not an option.

Performance considerations are essential for applications using database systems. RDBMS is about scaling and managing large amounts of data consistently so, by definition, performance is one of the key benefits of using databases.

Ensuring good performance requires a two-pronged attack:

- design for performance
- optimization.

Design for performance can be viewed as a component of the software development process. There are some things in *the way we do business* that can ensure that late optimization can satisfy performance objectives with minimal refactoring.

Optimization is rather an opposite process. Optimization is bounded by the high cost of refactoring and, in an ideal case, should focus on peripheral, low-level tweaks. In reality, it is often the case that optimization goes deeper than this – which just highlights the importance of good design decisions.

In this chapter, we very briefly discuss design and development considerations. Then, we propose an optimization workflow for streamlining an existing application with practical details for different aspects of the application code. Finally, we present a collection of good advice from Dr Richard Hipp, author of the SQLite database engine, and from the Symbian engineers who created, or have worked with, Symbian SQL.

8.1 Design and Development Considerations

Performance problems in applications come in many varieties. Errors such as choosing the wrong low-level algorithm or not caching frequently used items are easily fixed. Deeper design problems are more difficult to fix.

One aspect of the design that can have a serious impact on performance and could be difficult to fix are internal application interfaces. Interfaces are the domain-specific language of our application. Well-conceived interfaces provide a concise and consistent language for the application. They enforce usage patterns such as the sequence of calls to perform an operation and the object lifecycle. Most importantly, an interface change could be a very expensive exercise at later stages of the development cycle.

Design flexibility – including interfaces – is essential to facilitate optimizations that may come at a late stage in the development cycle. For example, it may be a good idea to abstract some of the persistence operations (e.g. using a façade) so that changes in the persistence layer do not require refactoring the remainder of the application.

The façade design also has drawbacks. The façade may cause a performance bottleneck by adding an extra layer of code and must therefore be very carefully implemented. This is easy to avoid however by following some simple rules. For example, we should always avoid copying data in the façade layer. In practical terms, it is always a good idea to pass in data placeholders or buffers from higher level code and avoid any allocation in the façade implementation. A second potential drawback is that the façade, being an interface itself, may force some application code into inefficient implementation of a specific task. This is common because the façade is often devised before the application code that uses it. This too is avoidable – the façade interface should be open for change throughout the development cycle.

Software design and associated processes are far beyond the scope of this book. We focus next on some practical suggestions for improving the design and development procedures. Interested readers can follow up by reading about software performance engineering, which offers a number of helpful techniques for good design and performance prediction (Lavenberg 1983; Smith 1990).

It is essential to build performance considerations into the development process. Here are some key rules we should keep in mind:

- Identify key, performance-sensitive use cases.
- Given the application use cases, set performance objectives in advance to ensure a good user experience. This feeds into the performance engineering discussion – having objectives for key use cases early on can help direct the application design. Also, meeting performance objectives is a great way to know when further optimizations become optional.
- Accurately and consistently measure performance objectives.
- Develop reproducible performance tests.
- Follow up on performance test results. This is highly dependent on the stage of the development cycle. For example, early on, we may explore design alternatives or even re-examine use cases. Towards the end, we look into optimizing implementation details.

The remainder of this section discusses the above points in more detail.

8.1.1 Identify Key Use Cases

Once application use cases are defined, from the performance point of view, it is important to identify the use cases that are essential for the user experience or could pose a performance challenge.

As an example, let us consider an imaginary contacts application. In a simplified scenario, the key use cases may be:

1. Search for a contact based on a prefix.
2. Search for a contact by number.
3. List contacts sorted by name.
4. Synchronize contacts with external database.

The first use case allows the user to navigate the contacts list view by typing the first letters of the contact name. The second use case is essential when receiving calls, to display caller details given the number. The third use case displays the list view typical in most contacts applications. Finally, synchronization with an external database is a common convenience feature.

Note that we have not included adding or deleting contacts. These use cases are not so performance critical. We could specify and monitor performance objectives for these use cases, but the four listed above are sufficient examples.

8.1.2 Set Performance Objectives

Setting performance objectives can be difficult. Some thought has gone into specifying performance measures as part of UML design. Also, there are several languages for specifying the performance measures of a system, such as PMLμ and CSL. These verbose theoretical approaches are unfortunately overly complex for a typical mobile application.

In practical terms, performance objectives are typically given as the maximum absolute time allowed for an operation. For use cases that operate on sets, the performance objective may be defined in terms of a function – the number of elements against time – such as big-O notation.

Consider, for example, that we are setting performance objectives for the contacts application in Section 8.1.1. We can immediately identify that use cases 1 and 2 are carried out frequently. As these are interactive operations, they must perform well to contribute to overall device responsiveness. Therefore, we set stringent performance objectives, say 100 ms, for use cases 1 and 2.

Use case 3 is somewhat more complex and requires better definition and potentially several performance objectives. While this operation can be seen as operating on a set, it can easily be reduced to absolute limits. The key to such reduction is the fact that the listing of entries is a linear operation – the set is created once when the query is executed and subsequent work simply traverses this set. Therefore, the first objective (use case 3.1) may be the time to populate the first 'page' of the list. Given that this is a start-up operation, where we need to execute the query and retrieve several records, we can set the performance objective to 500 ms. A further objective (use case 3.2) may be a time limit for scrolling down 100 contacts.

Use case 4 is different in that it operates on a set of records which may require executing a query for each record. In terms of performance, synchronization is not critical because it is not done as frequently as other use cases. Our performance objective in this case could be that the time to process entries depends linearly on the number of records being synchronized. This can be broken down further to include an absolute objective for updating a single entry.

Clearly, none of the tasks will take exactly the time set as an objective. Most use cases will perform better if we are successful.

8.1.3 Measure Performance

Setting absolute limits may pose some difficult challenges to engineers. Performance depends on many factors such as hardware and, in a multitasking environment such as the Symbian platform, background processes. Application test suites must include repeatable tests for timing validation which may sometimes be difficult and expensive to implement.

As discussed before, one common technique for guaranteeing application flexibility is separation of the persistence layer from the remainder of the application. This approach is also very useful for measuring the performance of specific database tasks.

Performance measurements with a façade design are pretty straightforward and can also be done as regression tests. When a use case requires a sequence of operations, adding a small amount of test code can group the tasks into a test resembling normal use case operation.

Measuring performance for user interface use cases may appear difficult. The common case is that operations are driven by user interactions. An approach commonly used in such cases is *instrumenting* the application to facilitate testing. Instrumenting an application means building in probing points to the key locations of the application code. In our contacts example, use case 3.2 measures overall application performance, rather than just database iteration. Therefore, the application may be instrumented to keep the time when scrolling has started and report the result when the user has scrolled to the 100th contact. It is usually a good idea to keep the instrumentation code out of the final deliverable by using conditional compilation or other techniques.

One of the concerns when measuring performance is that the accuracy of measurements depends on the granularity of the timing device. Many mobile devices, especially older models, have very low granularity – e.g. 64 ticks per second. This means that, for such a device, we cannot measure times shorter than 15.625 ms. This is generally not a problem – most common use cases take longer than that. However, it may pose a problem if instrumentation or test code attempts to measure and sum the time for many small tasks in order to measure use case performance. Such short tasks may finish between ticks, i.e., they may take less than 15 ms to execute. If the test measures many such tasks – possibly in a loop – the measurement result may significantly deviate from the true value due to cumulative error.

It is advisable to measure all code that executes during a use case. Apart from guarding against timer resolution problems and taking the UI into account, it also includes other environmental effects, such as the effects of background processes. However, the influence of other factors can make it more difficult to discern the time spent performing database operations – it is particularly problematic if measurements are used for optimization. In such cases, the only way to assess the performance may be by using a profiler; however, profiling is a one-off operation for optimizing the application rather than a recurring cost.

Keeping all the difficulties in mind, the objectives are still extremely useful from the product usability perspective and are worth every bit of the engineering effort required to implement and achieve them.

8.1.4 Develop Performance Tests

Performance tests should implement testing of key use cases defined during design. Automated performance tests have huge benefits as they can be used for regression testing and can be run at very low cost.

Performance tests can be difficult to devise and execute because of the difficulties in replicating conditions present in real-life situations. This is, however, a central element in achieving good performance and we must make our best effort to emulate the environment.

8.1.5 Follow up Test Results

What we do about a failed performance test depends on the stage of development we are in. Early on, we may investigate to ensure the result is not caused by a design problem. Unsatisfactory performance test results ultimately calls for optimization. Incidentally, this is exactly what the next section explains in detail.

8.2 An Introduction to Optimization

When a use case does not meet an objective, there are several things to consider:

- inspecting the implementation
- inspecting the application design
- revisiting the use cases.

The remainder of this chapter contains a wealth of information on how to investigate the implementation. If implementation inspection does not yield the desired results, the application design may be revisited with this use case in view. It may not be possible to achieve the objective for an ill-conceived use case, so this could be another line of action. Changing the objective is usually not an option or, at least, not a good option at this stage.

It is clear that optimization is an iterative process. In essence, we devise a possible improvement strategy – be it an implementation change, a configuration tweak or a hardware change – and measure key use case performance.

A well-guided optimization process would follow the law of diminishing returns. Our first targets are ‘low hanging fruit’ – improvements that affect results significantly with little effort. A second round may be focused on relatively big gains with some refactoring effort. Finally, further marginal improvements could be achieved but at significant cost. Such changes may affect other operations and extend development time while offering little benefit.

8.2.1 Understand the Operation of the Application

An understanding of the complexity of available functions is useful in helping to reduce the amount of data processing that an application typically does.

When using the SQLite command line interface, the EXPLAIN keyword can be placed before an SQL command to obtain a report of the sequence of internal steps that would have been taken to execute the command. This can be very useful in helping developers to understand whether, for example, an index is used when executing the statement, which index is used and whether a temporary table is created. See the SQLite manual at ***www.sqlite.org*** for further information.

8.2.2 Identify Bottlenecks Before Optimizing

Before optimizing, we must clearly know why the application failed a performance objective in the first place. Having an instrumented application can be a great help in doing this. Even if we need to add some probing points in the application, the existence of instrumentation infrastructure could make this task simpler.

Another approach is to use a profiler. Symbian offers profiling support when working on reference boards via the Symbian Sampling Profiler which is available in the Series 60 SDK. You can find further information about the Symbian profiler in the article at ***developer.symbian.org/wiki/index.php/Symbian_C++_Performance_Tips***. Profiler support is also available from the ARM RealView Profiler (RVP), available from ***www.arm.com/products/DevTools/RVP.html***.

8.2.3 Avoid Early Optimization

Early optimizations can lead to complex and unintelligible code and, ultimately, to code that is difficult to maintain. If in doubt about the efficiency of an implementation, it is usually a good idea to provide an abstraction that enables us to easily replace a problematic component later. Early on, it is much more important to focus on good design than on the efficiency of a particular implementation.

8.2.4 Maintain Performance Test Records

Continuous integration systems play a very important role in many modern development efforts. Apart from nightly build–test cycles, many continuous integration tools support tracing of performance metrics produced by tests.

This can prove crucial when analyzing the effects of code changes on application performance. The performance record is a near-perfect tool for pinpointing a change or a range of changes causing a significant performance drop.

However, it must be noted that performance test changes can introduce some confusion as they may affect measured results without the underlying code change.

8.2.5 Carry Out One Optimization at a Time

There are a very large number of possible parameters we could tweak, so it is essential to remain systematic in our optimization process. Changing one parameter at a time allows us to build up a picture of the effects of particular optimization strategies. However, we often find that we must tweak several parameters to achieve the required performance.

8.2.6 Optimize in the Right Environment

One interesting question, particularly relevant in the mobile space, is whether to optimize on the emulator or on the target device. The Symbian emulator offers great convenience for running regression and performance tests. While the emulator may highlight major performance problems, in most cases it is not a very reliable performance indicator.

There are several reasons for this. At the time of writing, applications built to run on the emulator are built as native Windows executables and benefit from the speed of the desktop PC processor. The same applies to I/O – particularly to storage. Hard drives have completely different characteristics for read–write and seek times than the flash media normally used in Symbian devices. There are many further details that affect performance – operating system caching, disk buffers, and so on.

In summary, while the emulator is ideal for development and debugging, only performance tests run on the target device can be considered relevant.

8.3 Optimizing Applications

While the remainder of the chapter offers lots of detailed advice, here we offer some general optimization tips. The scope for optimizations is very large so this section deals separately with optimization in different contexts: in the application layer, the data model, indexing, queries, memory usage and system tuning.

8.3.1 Application Layer

Use Active Objects with Care

Most Symbian applications are single-threaded thanks to the non-preemptive multitasking afforded by active objects. While this is great in many ways, it also requires very careful programming because any blocking operation executing in a `CActive::RunL()` has the potential to block the user interface and event processing which, in turn, causes the application to appear ‘frozen’.

Where possible, use asynchronous APIs. If your application must use a synchronous (blocking) API inside `RunL()`, you must ensure that the call completes as quickly as possible. For example, it may be fine to synchronously retrieve a few integers using a simple indexed query but running a complex query or retrieving large amounts of data is not advisable.

Inspect Loops

Code executing in a loop has great potential to degrade application performance. Most commonly, loop bounds are determined at run time and are almost never sanity checked in terms of ‘how long *could* this take to complete.’ It is our responsibility to avoid using heavy, blocking I/O inside loops, at least when they execute on an application’s main thread.

Use Asynchronous APIs Carefully

Due to the non-preemptive multitasking nature of Symbian applications, most Symbian components offer asynchronous APIs. Unfortunately, not all of the asynchronous APIs actually have asynchronous implementations. Also, there is a small set of slow, blocking operations that do not have asynchronous variants.

It follows that it may be a good idea to do I/O-intensive operations in a separate thread. While this may require somewhat more development effort, it has the benefit of allowing the UI to remain responsive regardless of the I/O load. This also removes the need to account for different implementations of asynchronous functions. Finally, with the upcoming symmetric multiprocessing support and multi-core devices, this can also dramatically improve application performance by using parallelization features.

Use Incremental Retrieval

One great performance killer is loading all query results in one go. Another example is loading the whole content of a BLOB in one go. Always perform these operations incrementally as this allows other tasks executing in the active scheduler to continue running normally. Incremental retrieval also typically uses less memory.

While it is true that using this technique could result in marginally slower retrieval, we gain in application stability, scalability and – most importantly – we ensure a good user experience.

8.3.2 Data Model

Creating a well-designed data model has the potential to become a tradeoff between design 'beauty' and performance. In some cases, this appears unavoidable. Here, we discuss some techniques that can help to create a balanced tradeoff – small sacrifices in 'beauty' that can produce substantial performance gains.

Normalization

There are cases where we have to carefully consider the way we use particular tables before deciding on the *level of normalization* appropriate to our application. There are essentially two theoretical models – online transactional processing (OLTP) and online analytical processing (OLAP). OLTP databases tend to have a more balanced relation between read and write operations and are typically more normalized. In OLAP databases, read operations are much more frequent and critical than write operations and hence these databases tend to be less normalized.

While normalized data encourages good design, the key challenge with normalized schemas is the need for JOIN operations in data retrieval. Joins with large tables or on non-indexed columns may pose significant performance challenges. Further, we often need to maintain derived data in a database. It is in these cases that a level of denormalization can be tolerated in order to complete particular operations quickly (Sanders and Shin 2001).

There are some ground rules for dealing with denormalized and derived data in a database. First, it is important to start with a normalized schema, at least to 3NF level (see Section 3.4). Only if the normalized schema does not fulfill our requirements should we look at some level of denormalization. We must always maintain consistency of any duplicated and derived data and do this before any data presentation. A good approach would be to keep the logical design normalized and store additional redundant information in separate tables that can be maintained with triggers.

Denormalization is a controversial and hotly contested topic. Clearly, denormalized schemas have potential for speeding up some read operations at the expense of write operations. On the other hand, it is often argued that denormalization introduces design impurity and hidden costs.

Large Data Types

It is important to consider whether it is necessary to include large data types such as LONG text or binary large object (BLOB) data in a table, as the presence of large BLOBs can significantly impact the time taken to perform a search.

The situation is made worse if overflow pages are required to accommodate a record that contains a BLOB. Overflow pages are required when the record size exceeds the page size, which isn't unlikely when large BLOBs are included in a record. In this case, searches become even slower due to the need to retrieve and check the data on the overflow pages.

The time taken to insert or update a record is greater if it is necessary to create or retrieve an overflow page for the record. In addition, retrieval of such a record is not optimal because the linked list of pages for that record needs to be traversed to reconstruct the record and this is not an efficient operation.

Given these issues, one of the following approaches should be considered whenever BLOB data is necessary:

- store the BLOB data in a separate table linked to the original table by a common key
- place the BLOB columns as the last columns in the table schema.

The above recommendations are beneficial even when the BLOB column is not required to be selected during a record retrieval. Further optimization can be achieved by storing large BLOBs external to the database (in a separate file) and having a file location reference as the value in the table.

8.3.3 Indexing

Indexes are pivotal to vastly increasing the speed of the look-up of records from a database. This is because look-up using indexes uses binary search techniques rather than a linear full table scan.

It is recommended that no index is created until there is a clear requirement to increase the speed of data look-up. This is because an index not used correctly or sensibly may actually result in an overall degradation of the performance of a database. Whilst the addition of an index can vastly improve look-up speed, it is likely to increase the time that it takes to insert or delete a record and may also increase the time that it takes to update a record if the columns being updated are indexed.

Index Size

In SQLite, an index is stored in its own B-tree, separate from the table data. There is a minimum of four entries per B-tree leaf. The maximum size for an entry in the B-tree is approximately 230 bytes. An index on a column (or columns) that contains a large amount of data may cause the index B-tree to use overflow pages. This should be avoided as much as possible, for reasons explained earlier.

Avoiding Common Index Pitfalls

An index can be a great tool for speeding up database operations, but if not used carefully it can cause many different problems. This section documents the main things to avoid when using indexes.

Typically, an index is explicitly created using the CREATE INDEX SQL command:

```
CREATE INDEX idx1 on Table1(i3)
```

where `i3` is the third column of table `Table1`.

Specifying certain types of column (`UNIQUE` and `PRIMARY KEY`) within the table schema results in indexes being created implicitly. Note that if a column is declared as `INTEGER PRIMARY KEY` then it does not create a new index but instead treats the specified column as an alias for the in-built `ROWID` index. Care should be taken to avoid the duplication of implicit indexes by explicit indexes.

It is important to be aware that there are costs associated with each new index that is created. Each new index takes up additional space in the database file, to accommodate its B-tree. The more indexes that are created, the larger the database file becomes. Special care should be taken to avoid duplicate or redundant indexes. For example, Index `idx1` is redundant because it is a subset of `idx2`:

```
CREATE TABLE Table1(i1 INTEGER, i2 INTEGER)
CREATE INDEX idx1 on Table1(i1)
CREATE INDEX idx2 on Table1(i1,i2)
```

Index `idx3` is redundant because it duplicates the index on `ROWID`:

```
CREATE TABLE Table2(ID INTEGER UNIQUE);
CREATE INDEX idx3 on Table2(ID)
```

Indexes should not be created on columns that contain large amounts of data, e.g. BLOB columns. This is to avoid the likelihood of the index B-tree requiring to use overflow pages.

Every insert and delete operation modifies not only the B-tree for the data table but also the B-tree of each index on that table. Thus the performance of INSERT and DELETE operations decreases as the number of indexes increases. The same is true of UPDATE operations if the columns being updated are indexed

Having overlapping indexes presents the optimizer with a greater opportunity to make a poor choice of which index to use. For example, given the redundant indexes `idx1` and `idx2` listed above, the optimizer may choose to use `idx2` if `i1` is given as the column to use in a query.

Choosing the Correct Index Type

There are a number of different types of index and it is important to consider the benefits and cost of each type before a choice is made. The following list outlines the different types of index that are possible and the circumstances under which they are most appropriate:

- A *standard* index is an ordered list of values in which duplicate values are permitted. It is the default choice for an index to be used in a search field in the WHERE clause of a statement.
- A *unique* index is an ordered list of values in which duplicate values are not permitted. Unique indexes can be applied to enforce a constraint on a column or set of columns in a table to maintain data integrity. There can be only one primary key in a table, but there can be multiple unique indexes. Having the constraint of uniqueness means that operations that change a unique index are slightly slower than for standard indexes because checks have to be made to ensure that the uniqueness constraint is maintained.
- A *primary key* index is important for data normalization. A primary key value is a unique value for each record. Depending on the table, the best choice may be to have multiple columns that make up a unique primary key but in general a single integer primary key is optimal as it is aliased to the in-built `ROWID` index. This gives it the smallest index footprint and provides the best performance because the index key values are stored on the entries of the table's B+Tree and not in a separate B-tree. The cost of a primary key is that the speed of operations that change it are affected by the necessary checks to ensure that the values are unique and non-NULL.

- A *multi-key* index, also known as a compound index, is an index that is composed of two or more columns. Care should be taken when specifying the order of columns in a multi-key index. For example, unique columns should come before non-unique columns in an index.

Using `ROWID` as a Special Index

Every record of every table has a unique, signed 64-bit integer `ROWID` value. This `ROWID` is the key to the B+Tree that holds the table data and is unique within a table. Searching for data by `ROWID` is typically around twice as fast as any other search method, because the record data is stored alongside the `ROWID` index key.

The `ROWID` can be accessed using one of the special column names `ROWID`, `_ROWID_`, or `OID`. Alternatively, it is possible to declare an integer column in a table as the alias for `ROWID` by using the keyword `INTEGER PRIMARY KEY` for that column when creating the table.

Using Multiple Indexes in a Statement

The optimizer only uses one index per table during evaluation of a statement. Therefore, WHERE clauses that contain multiple columns may benefit from using multi-key indexes. For example, consider the following indexes:

```
CREATE INDEX idx1 ON Table1(a);
CREATE INDEX idx2 ON Table1(b);
```

The optimizer uses only one index and the other index is redundant when executing the following statement:

```
SELECT * FROM Table1 WHERE a = 5 and b < 10;
```

The alternative would be to use a multi-key index that contains both columns:

```
CREATE INDEX idx3 ON Table1(a,b);
```

When `idx3` is used, columns `a` and `b` may be used optimally in the evaluation. With a multi-key index, note that all index columns specified in the WHERE clause must be used with the = or IN operators except for the right-most column, which can be used with inequality operators.

Optimizing Index Selection

The optimizer chooses which index to use if more than one is available to execute a particular statement. The choice made may not always be obvious or as expected. The ANALYZE command can be used to create a table, `sqlite_stat1`, that contains statistical information about the indexes present. This information is used by the query optimizer to improve its index selection decisions.

8.3.4 SQL Queries

In the context of Symbian SQL, it is important to use the API wisely. For example, reusing a prepared statement for DML queries can help improve performance by avoiding SQL compilation. This is illustrated in the following code:

```
RSqlStatement stmt;
CleanupClosePushL(stmt);
CleanupStack::PushL(TCleanupItem(&DoRollback, &<db_conn>));
<db_conn>.Exec(_L("BEGIN"));
stmt.PrepareL(<db_conn>, _L("INSERT INTO <table> values(?,?,… )"));
for (TUint i=0; i < <no_of_rows_to_insert>; ++i)
  {
  // bind all column values
  User::LeaveIfError(stmt.Exec());
  User::LeaveIfError(stmt.Reset());
  }
<db_conn>.Exec(_L("COMMIT"));
CleanupStack::Pop(); // TCleanupItem
CleanupStack::PopAndDestroy(); //stmt
```

Further, if we perform this operation frequently, the `RSqlStatement` may also be cached between invocations – it could be kept at class scope rather than being created repeatedly.

Various APIs of the class `RSqlStatement` can be used to make the execution of operations as optimal as possible.

```
TInt RSqlStatement::ParameterIndex(const TDesC &aParameterName) const;
TInt SqlStatement::ColumnIndex(const TDesC &aColumnName) const;
```

These APIs get the indexes of the parameter and the column, respectively, with the given name.

It is more optimal to use a parameter or column index than a parameter or column name in the code that is used to execute an operation. These APIs should be called outside of the main retrieval loop and the returned index value stored and re-used within the loop, rather than using the API as a parameter to one of the bind methods. See the example code:

```
RSqlStatement stmt;
CleanupClosePushL(stmt);
TInt err = stmt.Prepare(<db_conn>,
                        _L("SELECT i1 FROM Tbl1 WHERE i1 > :Val"));
// handle error if one occurs
TInt paramIndex = stmt.ParameterIndex(_L(":Val"));
err = stmt.BindInt(paramIndex, 5);
// handle error if one occurs
TInt columnIndex = stmt.ColumnIndex(_L("i1"));
while((err = stmt.Next()) == KSqlAtRow)
  {
  TInt val = stmt.ColumnInt(columnIndex);
  RDebug::Print(_L("val=%d\n"), val);
  }
if(err == KSqlAtEnd)
  // OK - no more records
else
  // handle the error
CleanupStack::PopAndDestroy(); //stmt
```

The `TSqlScalarFullSelectQuery` interface executes SELECT SQL queries that return a single record consisting of a single column value. Using this interface is optimal as it requires fewer IPC calls. For example, the following code retrieves the ID value of a specific person in a table:

```
TSqlScalarFullSelectQuery fullSelectQuery(<db_conn>);
TInt personId = fullSelectQuery.SelectIntL(
                  _L("SELECT ID FROM PersonTbl WHERE Name = 'John'"));
```

8.3.5 Memory Usage

So far, the optimization discussion has focused on performance. In mobile devices, memory usage is equally critical. In this section, we briefly explore some practical ways to optimize memory usage and keep applications lean.

Database Connections

A symbol table is created from the database schema and stored in memory for each database with an active connection. This symbol table is approximately the size of the schema and is used by the SQL compiler. Complex schemas can result in large symbol tables; therefore, to help minimize RAM usage, it is advisable to keep schemas as simple as possible and to close database connections when they are no longer required.

Prepared Statements

You should be aware that prepared statements use approximately 1 KB of RAM each when they are cached. The memory usage increases when the statements are being used.

Deleting Multiple Records

A DELETE operation is executed using a two-pass method. The first pass determines all of the records that are to be deleted and stores their `ROWID`s in memory. The second pass then deletes the records.

Each `ROWID` uses 8 bytes of memory. Under certain circumstances, and if memory is critically low, then it may be advisable to divide the delete operation into a number of manageable parts.

Note that deleting all of the records from a table is a special case and does not require any `ROWID`s to be stored in memory.

Manipulating Large Volumes of Data

Retrieving large volumes of text or BLOB data should be carried out using streams in order to help minimize the impact on RAM usage.

Transient Tables

The page cache (see Chapter 7) holds each page that is currently in use and also recently used pages. It is maintained for the main database, attached databases and temporary and transient tables. To reduce the amount of memory that is used, it is advisable to avoid the unnecessary use of attached databases, temporary and transient tables whenever possible. It is therefore important to understand which of these must be created explicitly and which can be created implicitly, without the user's knowledge.

Attached databases and temporary tables must be created explicitly and they last for the duration of the database connection or until they are detached or dropped.

Transient tables are created implicitly to store the intermediate results of queries and only last for the duration of a query. Transient tables are used in the following circumstances:

- A statement contains sub-queries that cannot be ‘flattened’ by the optimizer. For example, the following statement causes a transient table to be created:

```
SELECT * FROM t1 WHERE (SELECT ROWID FROM t2 WHERE ROWID < 50);
```

A sub-query on the right of an IN keyword also causes a transient table to be created.

- The `DISTINCT` keyword is used.
- The `UNION`, `INTERSECT` or `EXCEPT` keywords are used.
- The `ORDER BY` or `GROUP BY` keywords are used (if no index is available to satisfy the request).

8.3.6 System Tuning

Various SQL database properties are configurable in SQL Server. These properties are discussed in this section. Some of the properties can be set at compile time and others at run time. Developers use run-time options in order to ensure compatibility across devices, while compile-time options are more interesting to device creators. Compile-time changes made to the SQL Server have a system-wide impact on all client applications and servers that use SQL Server. For this reason, such changes should only be made at compile-time if absolutely necessary and after careful consideration.

The configuration parameter rules are:

- By default, the parameters in `sqlite_macro.mmh` are used (compile-time configuration).
- If the file `SqlServer.cfg` exists then the parameters specified in it override the default parameters.
- If a configuration string is passed to an API such as `RSqlDatabase::Create()` then the parameters specified in the configuration string override the default and `SqlServer.cfg` parameters.

PRAGMA command settings are disabled for secure shared databases. Symbian does not recommend that the PRAGMA method of configuration is used.

Auto-Vacuum

The auto-vacuum setting is a compile-time configuration of SQL Server. In basic terms, when auto-vacuum is OFF, a database file remains the same size after data is deleted from the database. With auto-vacuum ON, the file may become smaller after a deletion if the deletion results in pages becoming empty.

Note that databases that have auto-vacuum set to ON have a slightly larger file size than those with auto-vacuum set to OFF. This is because the database has to store extra information to keep track of pages.

The default auto-vacuum setting of SQL Server is ON.

Page Size

The page size is a run-time option and can be set when the database is created using `RSqlDatabase::Create()`. The page size value cannot be changed once the database is created. The supported page sizes are 512 bytes, 1024 bytes, 2048 bytes, 4096 bytes, 8192 bytes, 16,384 bytes and 32,768 bytes. The upper limit may be modified by setting the value of macro `SQLITE_MAX_PAGE_SIZE` during compilation. The maximum upper bound is 32,768 bytes. The default page size in SQL Server is 1024 bytes.

It is important to recognize that the ratio of page size to record size can significantly impact the performance of a database. Since data is written to and read from a database in terms of pages, it is critical to have a page size that ensures that as much as possible of the space available on a page is used, without triggering the need for overflow pages. For example, having an average record size of 1.1 KB and a page size of 2 KB means that only one record can fit on any one page and so around 50% of each page is not used but is written to and read from the database, incurring a sizeable performance hit if, for example, two records, and therefore two pages, need to be retrieved.

If the page size is set so that a large number of records can fit onto a single page and the requirement is to perform mainly bulk operations on all or a high percentage of the records in a table, then the speed of operations is increased. However, if there are some operations that update only a few bytes of a single record then extra time is required to write to and read from a larger page. A balance needs to be reached based on the operational requirements of the application.

This may result in the need to perform customized tests in order to determine the optimal page size for the given application. However, it is important to have a page size that is large enough to store at least one record, thus avoiding the immediate need for overflow pages.

Cache Size

Cache size is configurable at the time of creating a database. The cache size determines the number of pages that are held in memory at the same time. The larger the cache size, the greater the likelihood that SQL Server does not have to resort to performing disk I/O in order to execute an operation. However, a large cache size can significantly impact the amount of RAM that an application needs to use.

The default cache size of SQL Server is 64 pages.

Database Encoding

SQL Server supports both UTF-8 and UTF-16 database encodings (i.e. the underlying format of the database file is either UTF-8 or UTF-16). The desired encoding can be set at database creation time.

The default database encoding of SQL Server is UTF-16.

A UTF-8 encoded database clearly has a smaller disk footprint than a UTF-16 encoded equivalent. However, the choice of database encoding should be based on knowledge of the text alphabet encoding that the applications handle. For example, when UTF-16 text is inserted into a UTF-8 encoded database then there is extra cost from converting the UTF-16 text into UTF-8 format for storage.

Query Encoding

SQL Server supports both UTF-8 and UTF-16 query encoding. This allows an application to call SQL Server APIs with SQL statements that are encoded in either UTF-8 or UTF-16. This is convenient as different applications handle different types of text. As with database encoding, an application's choice of query encoding should be based on knowledge of the text alphabet/encoding that the application handles. The choice of database encoding and query encoding should be as compatible as possible given the requirements of the application or applications that use the database.

8.4 Symbian SQL Optimization Tips

The remainder of this chapter describes some important features of Symbian SQL and SQLite in more depth and explains how a programmer can exploit these features to make SQLite run faster or use less memory. Topics are loosely grouped into six optimization domains: data organization, expressions, statements, indexing, using the optimizer and resource usage.

8.4.1 Data Organization

As has already been discussed, small changes in how we organize data can yield significant performance improvements. Some further techniques are included below.

Put Small and Frequently Accessed Columns at the Beginning of a Table

SQLite stores a row of a table by gathering the data for all columns in that row and packing the data together in column order into as few bytes as possible. This means that the data for the column that appears first in the CREATE TABLE statement is packed into the bundle first, followed by the data for the second column, and so on.

SQLite goes to some trouble to use as few bytes as possible when storing data. A text string that is 5 bytes long, for example, is stored using just 5 bytes. The '\00' terminator on the end is omitted. No padding bytes or other overhead is added even if the column specifies a larger string such as VARCHAR(1000). Small integers (between −127 and +127) are stored as a single byte. As the magnitude of the integer increases, additional bytes of storage are added as necessary up to 8 bytes. Floating point numbers are normally stored as 8-byte IEEE floats, but if the floating point number can be represented as a smaller integer without loss of information, it is converted and stored that way to save space. BLOBs are also stored using the minimum number of bytes.

These efforts to save space help to keep database files small and to run SQLite faster, since the smaller each record is the more information fits in the space and the less disk I/O needs to occur.

The downside of using compressed storage is that data in each column is an unpredictable size. So within each row, we do not know in advance where one column ends and the next column begins. To extract the data from the *n*th column of a row, SQLite has to decode the data from the *n*−1 prior columns first. Thus, for example, if you have a query that reads just the fourth column of a table, the bytecode that implements the query has to read and discard the data from the first, second, and third columns in order to find where the data for the fourth column begins in order to read it.

For this reason, it is best to put smaller and more frequently accessed columns of a table early in the CREATE TABLE statement and put large CLOBs and BLOBs and infrequently accessed information toward the end.

Store Large BLOBs Outside the Database

SQLite does not place arbitrary constraints on the size of the BLOBs and CLOBs that it stores. So SQLite can and does store really big BLOBs. There are deployed applications that use SQLite and store huge files – up to 100 MB or more – inside BLOB columns. And those applications work fine.

But just because you can do this does not mean you should. Though SQLite is able to handle really big BLOBs, it is not optimized for that case. If performance and storage efficiency is a concern, you are better to store large BLOBs and CLOBs in separate disk files and store the name of the file in the database in place of the actual content.

When SQLite encounters a large table row – large because it contains one or more large BLOBs perhaps – it tries to store as much of the row as it can on a single page of the database. The tail of the row that does not fit on a single page spills into overflow pages. If more than one overflow page is required, then the overflow pages form a linked list. So if you have, for example, a 1 MB BLOB that you want to store in a SQLite database and your database uses the default 1 KB page size, the BLOB is broken up into a linked list of over 1000 pieces, each of which is stored separately. Whenever you want to read the BLOB, SQLite has to walk this 1000-member linked list in order to reassemble the BLOB. This works and is reliable, but it is not especially efficient.

If you absolutely need to store large BLOBs and CLOBs in your database (perhaps so that changes to these data elements are atomic and durable) then at least consider storing the BLOBs and CLOBs in separate tables that contain only an `INTEGER PRIMARY KEY` and the content of the BLOB or CLOB. So, instead of writing your schema this way:

```
CREATE TABLE image(
  imageId INTEGER PRIMARY KEY,
  imageTitle TEXT,
  imageWidth INTEGER,
  imageHeight INTEGER,
  imageRefCnt INTEGER,
  imageBlob BLOB
);
```

factor out the large BLOB into a separate table so that your schema looks more like this:

```
CREATE TABLE image(
  imageId INTEGER PRIMARY KEY,
  imageTitle TEXT,
  imageWidth INTEGER,
  imageHeight INTEGER,
  imageRefCnt INTEGER
);
CREATE TABLE imagedata(
  imageId INTEGER PRIMARY KEY,
  imageBlob BLOB
);
```

Keeping all of the small columns in a separate table increases locality of reference and allows queries that do not use the large BLOB to run much faster. You can still write single queries that return both the smaller fields and the BLOB by using a simple (and very efficient) join on the `INTEGER PRIMARY KEY`.

Use CROSS JOIN to Force Ordering

The SQLite query optimizer attempts to reorder the tables in a join in order to find the most efficient way to evaluate the join. The optimizer usually does this job well, but occasionally it makes a bad choice. When that happens, it might be necessary to override the optimizer's choice by explicitly specifying the order of tables in the SELECT statement.

To illustrate the problem, consider the following schema:

```
CREATE TABLE node(
  id INTEGER PRIMARY KEY,
  name TEXT
);
```

```
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
  orig INTEGER REFERENCES node,
  dest INTEGER REFERENCES node,
  PRIMARY KEY(orig, dest)
);
CREATE INDEX edge_idx ON edge(dest,orig);
```

This schema defines a directed graph with the ability to store a name on each node of the graph. Similar designs (though usually more complicated) arise frequently in application development. Now consider a three-way join against the above schema:

```
SELECT e.*
FROM edge AS e,
     node AS n1,
     node AS n2
WHERE n1.name = "alice"
  AND n2.name = "bob"
  AND e.orig = n1.id
  AND e.dest = n2.id;
```

This query asks for information about all edges that go from nodes labeled 'alice' to nodes labeled 'bob'. There are many ways that the optimizer might choose to implement this query, but they all boil down to two basic designs. The first option looks for edges between all pairs of nodes. The following pseudocode illustrates:

```
foreach n1 where n1.name='alice' do:
  foreach n2 where n2.name='bob' do:
    foreach e where e.orig=n1.id and e.dest=n2.id do:
      return e.*
    end
  end
end
```

The second basic design is to loop over all 'alice' nodes and follow edges from those nodes looking for nodes named 'bob' (the roles of 'alice' and 'bob' might be reversed here without changing the fundamental character or the algorithm):

```
foreach n1 where n1.name='alice' do:
  foreach e where e.orig=n1.id do:
    foreach n2 where n2.id=e.dest and n2.name='bob' do:
      return e.*
    end
  end
end
```

The first algorithm above corresponds to a join order of n1−n2−e. The second algorithm corresponds to a join order of n1−e−n2. The optimizer has to decide which of these two algorithms is likely to give the fastest result. It turns out that the answer depends on the nature of the data stored in the database.

Let the number of ‘alice’ nodes be M and the number of ‘bob’ nodes be N. Consider two scenarios. In the first scenario, M and N are both two but there are thousands of edges on each node. In this case, the first algorithm is preferred. With the first algorithm, the inner loop checks for the existence of an edge between a pair of nodes and outputs the result if found. But because there are only two alice and two bob nodes, the inner loop only has to run four times and the query is very quick. The second algorithm would take much longer. The outer loop of the second algorithm only executes twice, but because there are a large number of edges leaving each ‘alice’ node, the middle loop has to iterate many thousands of times. It will be much slower. So in the first scenario, we prefer to use the first algorithm.

Now consider the case where M and N are both 3500: ‘alice’ nodes are abundant. But suppose each of these nodes is connected by only one or two edges. In this case, the second algorithm is preferred. With the second algorithm, the outer loop still has to run 3500 times, but the middle loop only runs once or twice for each outer loop and the inner loop only runs once for each middle loop, if at all. So the total number of iterations of the inner loop is around 7000. The first algorithm, on the other hand, has to run both its outer loop and its middle loop 3500 times each, resulting in 12 million iterations of the middle loop. Thus in the second scenario, the second algorithm is nearly 2000 times faster than the first.

In this particular example, if you run ANALYZE on your database to collect statistics on the tables for the optimizer to reference, the optimizer can figure out the best algorithm to use. But if you do not want to run ANALYZE or if you do not want to waste database space storing the `sqlite_stat1` statistics table that ANALYZE generates, you can manually override the decision of the optimizer by specifying a particular order for tables in a join. You do this by substituting the keyword phrase CROSS JOIN in place of commas in the FROM clause. This forces the table to the left to be used before the table to the right. For example, to force the first algorithm, write the query this way:

```
SELECT *
FROM node AS n1 CROSS JOIN
     node AS n2 CROSS JOIN
     edge AS e
WHERE n1.name = "alice"
  AND n2.name = "bob"
  AND e.orig = n1.id
  AND e.dest = n2.id;
```

And to force the second algorithm, write the query like this:

```
SELECT *
FROM node AS n1 CROSS JOIN
     edge AS e CROSS JOIN
     node AS n2
WHERE n1.name = "alice"
  AND n2.name = "bob"
  AND e.orig = n1.id
  AND e.dest = n2.id;
```

The CROSS JOIN keyword phrase is perfectly valid SQL syntax according to the SQL standard, but it is syntax that is rarely used in real-world SQL statements. Because it is so rarely used, SQLite has appropriated the phrase as a way to override the table reordering decisions of the query optimizer.

The CROSS JOIN connector is rarely needed and should probably never be used until the final performance tuning phase of application development. Even then, SQLite usually gets the order of tables in a join right without any extra help. But on those rare occasions when SQLite gets it wrong, the CROSS JOIN connector is an invaluable way of tweaking the optimizer to do what you want.

8.4.2 Expressions

SQLite has some peculiarities in its interpretation of expressions and sometimes understanding the implementation details can greatly improve query performance. The following sections describe the most important of these rules.

Rewrite Expressions for Optimization

We have described how the optimizer only makes use of an index if one of the columns being indexed occurs on one side of certain equality and comparison operators: =, IN, <, >, <=, >=, and sometimes `IS NULL`. This is technically true. But before the stage of the analysis where the optimizer is looking for these kinds of expressions, it might have modified the WHERE clause (or the ON or HAVING clauses) from what was originally entered by the programmer. The next few paragraphs describe some of these rewriting rules.

The query optimizer always rewrites the BETWEEN operator as a pair of inequalities. For example, if the input SQL is this:

```
SELECT * FROM demo324 WHERE x BETWEEN 7 AND 23;
```

the query optimizer rewrites it as:

```
SELECT * FROM demo324 WHERE x>=7 AND x<=23;
```

In this revised form, the optimizer might be able to use an index on the `x` column to speed the operation of the query.

A disjunction of two or more equality tests against the same column are changed into a single IN operator. So if you write:

```
SELECT * FROM demo324 WHERE x=7 OR x=23 OR x=47;
```

the WHERE clause is rewritten into the following form:

```
SELECT * FROM demo324 WHERE x IN (7,23,47);
```

The original format was not a candidate for the use of indexes. After the disjunction is converted into a single IN operator, the usual index processing logic applies and the query can be made much faster. In order for this rewriting rule to be applied, however, all terms of the disjunction must be equality comparisons against the same column. It does not work to have a disjunction involving two or more columns or involving expressions. So, for instance, the following statements are not optimized:

```
SELECT * FROM demo324 WHERE x=7 OR y=23;
SELECT * FROM demo324 WHERE x=7 OR +x=23;
```

Another rewriting rule is associated with the LIKE and GLOB operators. This final rule only applies if the operation of LIKE and GLOB have not been overridden using `sqlite3_create_function()`. And the rule only applies to LIKE if SQLite is configured to make LIKE case sensitive using either a compile-time option or

```
PRAGMA case_sensitive_like=off;
```

The GLOB operator is case sensitive by default and so the PRAGMA above is not necessary for this rewriting rule to apply to GLOB. If all of the previous conditions are met and the right-hand side of either the GLOB or LIKE operator is a literal string (not a bound parameter, a column value or an expression) that does not begin with a wildcard character, then some additional comparison terms are added to the WHERE clause to help restrict the search. For example, if the input SQL is:

```
SELECT albumId FROM media WHERE composer LIKE "mozart%";
```

the query optimizer actually sees:

```
SELECT albumId FROM media WHERE composer LIKE "mozart%"
          AND composer>="mozart" AND composer<"mozaru";
```

The terms `composer>="mozart"` and `composer<"mozaru"` were added to help restrict the number of entries in the table that have to be checked using the relatively expensive LIKE operator. These two added terms might also allow an index to be employed to speed the search substantially.

Notice how the last character in the second term has changed from 't' to 'u'. That is not a misprint. The string 'mozaru' is the lexically smallest string that does not match the 'mozart%' pattern and is thus used as an upper bound. The query optimizer has to have access to the original search pattern ('mozart%') in order to compute this upper-bound string, which is why the LIKE and GLOB rewriting rule only applies if the search string is a literal string and not a bound parameter or expression.

Put Constant Subexpressions Inside Subqueries

The query parser and compiler in SQLite is designed to be small, fast, and lean. A consequence of this design is that SQLite does not do much in the way of folding constants or elimination of common subexpressions. SQLite evaluates SQL expressions mostly as written.

One way to work around this is to enclose constant subexpressions within a subquery. SQLite does optimize constant subqueries – it evaluates them once, remembers the result, then reuses that result repeatedly.

An example will help to clarify how this works. Suppose you have a table that contains a timestamp recorded as the fractional Julian day number:

```
CREATE TABLE demo325(tm DATE, data BLOB);
```

A query against this table to find all entries after November 8, 2006 might look like the following:

```
SELECT data FROM demo325 WHERE tm>julianday("2006-11-08");
```

This query works fine. The problem is that the `julianday("2006-11-08")` function is called repeatedly, once for each row tested. But the function returns the same value each time. It is much more efficient to call the function only once and reuse the result over and over. You can accomplish this by moving the function call inside a subquery as follows:

```
SELECT data FROM demo325 WHERE tm>(SELECT julianday("2006-11-08"));
```

There are, of course, some cases where multiple evaluations of a function in the WHERE clause is desirable. For example, suppose you want to return roughly one out of every eight records, chosen at random. A suitable query would be:

```
SELECT data FROM demo325 WHERE (random()&7)==0;
```

In this case, moving the function evaluation into a subquery would not work as desired:

```
SELECT data FROM demo325 WHERE (SELECT random()&7)==0;
```

The above statement would result in either all of the records in the table (probability 12.5%) or none of them (probability 87.5%). The difference here, of course, is that the `random()` function is not constant whereas the `julianday()` function is. But the SQL compiler does not have any way of knowing this so it always assumes the worst: that every function works like `random()` and can potentially return a different answer even with the same inputs. Use a subquery if you truly want to make a subexpression constant.

`MIN()` and `MAX()` Optimization

SQLite includes some special case optimizations for the `MIN()` and `MAX()` aggregate functions. The normal way of evaluating a `MIN()` or `MAX()` aggregate function is to run the entire query, examine every row of the result set and pick the largest or smallest value. Hence, the following two queries take roughly same amount of work:

```
SELECT x FROM demo326 WHERE y>11.5;
SELECT min(x) FROM demo326 WHERE y>11.5;
```

The only difference in the above two SELECT statements is that the first returns every possible value for `x` and the second returns only the smallest value. Both require about the same amount of time to run.

But there are some special cases where `MIN()` and `MAX()` run very fast. If the result set of a SELECT consists of only the `MIN()` function or the `MAX()` function and the argument to that function is an indexed column or the `ROWID` and there is no WHERE, HAVING or GROUP BY clause then the query runs in logarithmic time. So these two queries are very quick:

```
SELECT min(x) FROM demo326;
SELECT max(x) FROM demo326;
```

Note that the result set must contain a single column. The following query is much slower:

```
SELECT min(x), max(x) FROM demo326;
```

If you need the results from this last query quickly, you can break the query up into two subqueries both of which can be optimized using the `MIN()` and `MAX()` optimization, as follows:

```
SELECT (SELECT min(x) FROM demo326), (SELECT max(x) FROM dem326);
```

Note also that the result set must not be an expression on `MIN()` or `MAX()` – it needs to be a plain `MIN()` or `MAX()` and nothing else. So the following query is slower:

```
SELECT max(x)+1 FROM demo326;
```

As before, you can achieve the same result quickly using a subquery:

```
SELECT (SELECT max(x) FROM demo326) + 1;
```

Use UNION ALL Rather Than UNION

SQLite, in accordance with the SQL standard, allows two or more SELECT statements to be combined using operators UNION, UNION ALL, INTERSECT, and EXCEPT.

The UNION and UNION ALL operators do very nearly the same thing, but with one important difference. Both operators return the union of all rows from their left and right queries. The difference is that the UNION operator removes duplicates whereas the UNION ALL operator does not. To look it another way, the following two queries are equivalent:

```
SELECT * FROM tableA UNION SELECT * FROM tableB;
SELECT DISTINCT * FROM
  (SELECT * FROM tableA UNION ALL SELECT * FROM tableB);
```

When you look at it this way, you should clearly see that UNION is just UNION ALL with some extra work to compute the DISTINCT operation. And, you should also see that UNION ALL is noticeably faster than UNION and uses less temporary storage space.

If you need uniqueness of output values, then by all means use UNION. It is there for you and it works. But if you know in advance that your results are unique, or if you do not care, UNION ALL almost always run faster.

Avoid Using OFFSET for Scrolling Cursors

A common design pattern is to show the results of a large query result in a scrolling window. The query result might contain hundreds or thousands of rows, but only a handful are shown to the user at one time. The user clicks Up and Down buttons or drags a scrollbar to move up and down the list.

A common example of this is in media players where a user has requested to see all albums of a particular genre. There might be 200 such albums stored on the device, but the display window is only large enough to show five at a time. So the first five albums are displayed initially. When the user clicks on the Down button, the display scrolls down to the next five albums. When the user presses the Up button, the display scrolls back up to the previous five albums.

The naïve approach to implementing this behavior is to store the index of the topmost album currently displayed. When the user presses Up or Down, this index is decremented or incremented by five and a query similar to the following one is run to refill the display:

```
SELECT title FROM album WHERE genre="classical" ORDER BY title
  LIMIT 5 OFFSET ?
```

The bound parameter on the offset field would be filled in with the index of the topmost album to be displayed and the query is run to generate five album titles. Presumably the album table is indexed in such as way that both the WHERE clause and the ORDER BY clause can be satisfied using the index so that no accumulation of results and sorting is required. Perhaps the index looks like this:

```
CREATE INDEX album_idx1 ON album(genre, title);
```

This approach works as long as the offset value is small. But the time needed to evaluate this query grows linearly with the offset. So as the user scrolls down toward the bottom of the list, the response time for each click becomes longer and longer.

A better approach is to remember the top and bottom titles currently being displayed. (The application probably has to do this already in order be able to display the titles.) To scroll down, run this query:

```
SELECT title FROM album WHERE genre="classical" AND title>:bottom
  ORDER BY title ASC LIMIT 5;
```

And to scroll back up, use this query:

```
SELECT title FROM album WHERE genre="classical" AND title<:top
  ORDER BY title DESC LIMIT 5;
```

For scrolling down, the addition of `title>:bottom` (where `:bot-tom` is a parameter which is bound to the title of the bottom element currently displayed) causes SQLite to jump immediately to the first entry past the current display. There is no longer a need for an OFFSET clause in the query, though we still include 'LIMIT 5'. The same index works to optimize both the WHERE clause and the ORDER BY clause.

The scrolling up case is very similar, though in this case we are looking for titles that are less than the current top element. We have also added the 'DESC' tag (short for 'descending') to the ORDER BY clause so that titles will come out in 'descending' order. (The sort order is descending, but the order is ascending if you are talking about the order in which the titles are displayed in the scrolling window.) As before, the same `album_idx1` index is able handle the terms of both the WHERE clause and the descending ORDER BY clause.

Both of these queries should be much faster than using OFFSET, especially when the OFFSET is large. OFFSET is convenient for use in ad hoc queries entered on a workstation, but it is rarely helpful in an embedded application. An indexing scheme such as that described here is only slightly more complex to implement but is much faster from the user's perspective.

Use Conjunctions in WHERE Clause Expressions

SQLite works best when the expression in a WHERE clause is a list of terms connected by the conjunction, AND, operator. Furthermore, each term should consist of a single comparison operator against an indexed column. The same rule of thumb also applies to ON clauses in a join and to HAVING clauses.

SQLite is able to optimize queries such as this:

```
SELECT * FROM demo313 WHERE a=5 AND b IN ("on","off") AND c>15.5;
```

The WHERE clause in the example above consists of three terms connected by the AND operator and each term contains a single comparison operator with a column as one operand. The three terms are:

- `a=5`
- `b IN ("on","off")`
- `c>15.5`

The SQLite query optimizer is able to break the WHERE clause down and analyze each of the terms separately, and possibly use one or more of those terms with indexes to generate bytecode that runs faster. But consider the following similar query:

```
SELECT * FROM demo313 WHERE (a=5 AND b IN ("on","off") AND c>15.5)
                                                      OR d=0;
```

In this case, the WHERE clause consist of two terms connected by an OR. The SQLite query optimizer is not able to break this expression up for analysis. As a result, this query is implemented as a full table scan in which the complete WHERE clause expression is evaluated for each row. In this case, refactoring the WHERE clause does not help much:

```
SELECT * FROM demo313 WHERE (a=5 OR d=0)
         AND (b IN ("on","off") OR d==0)
                AND (c>15.5 OR d=0)
```

The WHERE clause is now a conjunctive expression but the terms of the expression are not simple comparison operators against a table column. The query optimizer can break the WHERE expression into three smaller subexpressions for analysis, but each subexpression is disjunct, no index can be used and a full table scan results.

If you know in advance that all rows in the result set are unique (or if that is what you want to achieve) then the following query can be used for an efficient implementation:

```
SELECT * FROM demo313 WHERE a=5 AND b IN ("on","off") AND c>15.5
UNION
SELECT * FROM demo313 WHERE d=0
```

In this form, the two queries are evaluated separately and their results are merged to get the final result. The WHERE clause on each query is a conjunction of simple comparison operators so both queries may be optimized to use indexes.

If the result set might contain two or more identical rows, then you can run the preceding query efficiently as follows:

```
SELECT * FROM demo313 WHERE RowID IN (
  SELECT RowID FROM demo313 WHERE a=5 AND b IN("on","off") AND c>15.5
  UNION ALL
  SELECT RowID FROM demo313 WHERE d=0
)
```

The subquery computes a set containing the `ROWID` of every row that should be in the result set. Then the outer query retrieves the desired rows. When a WHERE clause contains OR terms at the top level, most enterprise-class SQL database engines such as PostgreSQL or Oracle automatically convert the query to a form similar to the above. SQLite omits such advanced optimizations to minimize its complexity and size. In the rare cases where such queries are required, they can be optimized manually by the programmer by recasting the query statements as shown above.

8.4.3 Statements

This section discusses some SQLite-specific rules on statements.

Enclose INSERT and UPDATE Statements in Transactions

When programmers who are new to SQLite first start looking at it, they often write a test program to see how many INSERT statements per second it can do. They create an empty database with a single empty table. Then they write a loop that runs a few thousand times and does a single INSERT statement in each iteration. Upon timing this program, they find, to their chagrin, that SQLite appears to only be doing a couple of dozen INSERTs per second.

'Everybody I talked to says SQLite is supposed to be really fast', the new programmer typically complains, 'But I'm only getting 20 or 30 INSERTs per second!' Comments like this appear on the public SQLite mailing list at regular intervals.

In reality, SQLite can do around 50,000 or more INSERTs per second on a modern workstation (fewer on a mobile device). Constraints on the physics of disk drives and other media and the fact that SQLite does atomic updates to the database mean that SQLite only does a few dozen COMMIT operations per second. And unless you take specific action to tell SQLite to do otherwise, SQLite automatically inserts a COMMIT operation after every INSERT. So, the programmers described above are really measuring the number of transactions per second that SQLite does, not the number of INSERTs. This is a very important distinction.

Why is COMMIT so much slower than INSERT? Recall that all changes to a SQLite database are ACID (atomic, consistent, isolated, and durable). The atomic and durable parts are what take time. In order to be atomic, SQLite has to go through an elaborate dance with the underlying file system in which every modified page of the database file must be written twice. In order to be durable, the COMMIT operation must not return until all content has been safely written to non-volatile media. If mass storage is on a disk drive, at least two rotations of the disk platter are required to do an atomic and durable commit. If the platter spins at 7200 RPM then the fastest that a COMMIT can occur is 60 times per second. Flash memory is not constrained by the rotation rate of a spinning disk, but writing to flash is still a relatively slow operation. At least two consecutive non-concurrent writes to flash memory must occur in order to COMMIT and, depending on the flash memory technology used, this can take as long as or longer than waiting for a disk drive platter to spin twice.

The fact that SQLite does an atomic and durable COMMIT is a very powerful feature that can help you to build a system that is resilient, even in the face of unplanned system crashes or power failures. But the price of this resilience is that COMMIT is a relatively slow operation. Hence, if you want to make an application that uses SQLite go quickly, you should strive to minimize the number of COMMITs.

If you need to do many INSERT, UPDATE or DELETE operations (where 'many' means 'more than one at a time'), you are advised to put them all inside a single explicit transaction by running the BEGIN statement prior to the first changes and executing COMMIT once all the changes have finished. In this way, all your changes occur within a single transaction and only a single time-consuming COMMIT operation must occur. If you omit the explicit `BEGIN...COMMIT`, then SQLite automatically inserts an implicit `BEGIN...COMMIT` around each of your INSERT, UPDATE, and DELETE statements, which means your application ends up doing many COMMITs, which is always slower than doing just one.

Batch INSERT, UPDATE, and DELETE Operations

When you have many changes to make to a SQLite database, you are advised to make all those changes within a single explicit transaction by preceding the first change with a BEGIN statement and concluding the changes with a COMMIT statement. The COMMIT operation is usually the slowest operation that SQLite has to do. If you do not put your updates all within an explicit `BEGIN...COMMIT`, then SQLite automatically adds a `BEGIN...COMMIT` around each INSERT, UPDATE, and DELETE statement and you end up doing many more time-consuming COMMIT operations than are necessary.

The problem with `BEGIN...COMMIT` is that BEGIN acquires an exclusive lock on the database file which is not released until the COMMIT completes. That means only a single connection to the database can be in the middle of a `BEGIN...COMMIT` at one time. If another thread or process tries to start a `BEGIN...COMMIT` while the first is busy, the second has to wait. To avoid holding up other threads and processes, therefore, every BEGIN should be followed as quickly as possible by COMMIT.

Sometimes you run into a situation where you have to make periodic INSERTs or UPDATEs to a database based on timed or external events. For example, you may want to do an INSERT into an event log table once every 250 milliseconds or so. You could do a separate INSERT for each event, but that would mean doing a separate COMMIT four times per second, which is perhaps more overhead than you desire. On the other hand, if you BEGIN and accumulate several seconds worth of INSERTs you can avoid doing a COMMIT except for every 10 seconds or so. The trouble there is that other threads and processes are unable to write to the database while the event log is holding its transaction open.

The usual method for avoiding this dilemma is to store all of the INSERTs in a separate temporary (TEMP) table, then periodically flush the contents of the TEMP table into the main database with a single operation.

A TEMP table works just like a regular database table except that it is only visible to the database connection that creates it and the TEMP table is automatically dropped when the database connection is closed. You create a TEMP table using the `CREATE TEMPORARY TABLE` command, like this:

```
CREATE TEMP TABLE event_accumulator(
  eventId INTEGER,
  eventArg TEXT
);
```

Because TEMP tables are ephemeral (meaning that they do not persist after the database connection closes), SQLite does not need to worry about making writes to a TEMP table atomic or durable. Hence a COMMIT to a TEMP table is very quick. So a process can do multiple INSERTs into a TEMP table without having to enclose those INSERTs within an explicit `BEGIN...COMMIT` for efficiency. Writes to a TEMP table are always efficient regardless of whether or not they are enclosed in an explicit transaction.

So as events arrive, they can be written into the TEMP table using isolated INSERT statements. But because the TEMP table is ephemeral, one must take care to periodically flush the contents of the TEMP table into the main database where they will persist. So every 10 seconds or so (depending on the application requirements) you can run code like this:

```
BEGIN;
  INSERT INTO event_log SELECT * FROM event_accumulator;
  DELETE FROM event_accumulator;
COMMIT;
```

These statements transfer the content of the ephemeral `event_accumulator` table over to the persistent `event_log` table as a single atomic operation. Since this transfer occurs relatively infrequently, minimal database overhead is incurred.

Use Bound Parameters

Suppose you have a string value in a C++ variable named `aName` and you want to insert that value into a database. One way to proceed is to construct an appropriate INSERT statement that contains the desired string value as an SQL string literal, then run that INSERT statement. Pseudo-code for this approach is as follows:

```
TBuf<256> query;
LIT(KStatement, "INSERT INTO namelist VALUES('%S')");
query.Format(KStatement, &aName);
User::LeaveIfError(database.Exec(query));
```

The INSERT statement is constructed by the call to `TDes::Format()` on the third line of the example above. The first argument is a template for the SQL statement. The value of the `aName` variable is inserted where `%S` occurs in the template. Notice that `%S` is surrounded by single quotes so that the string is properly contained in quotes.

This approach works as long as the value in `aName` does not contain any single-quote characters. If `aName` does contain one or more single-quotes, then the string literal in the INSERT statement is not well-formed and a syntax error might occur. Or worse, if a hostile user is able to control the content of `aName`, they might be able to put text in `aName` that looks something like this:

```
hi'); DELETE FROM critical_table; SELECT 'hi
```

This would result in the query variable holding

```
INSERT INTO namelist VALUES('hi'); DELETE FROM critical_table;
                                                     SELECT 'hi'
```

Your adversary has managed to convert your single INSERT statement into three separate SQL statements, one of which does things that you probably do not want to happen. This is called an ‘SQL Injection Attack’. You want to be very, very careful to avoid SQL injection attacks as they can seriously compromise the security of your application.

Symbian SQL, like most other SQL database engines, allows you to put parameters in SQL statements and then specify values for those parameters prior to running the SQL. In Symbian SQL, parameters can take several forms, including:

- `?`
- `?NNN`
- `:AAA`
- `@AAA`
- `$AAA`

In the above, `NNN` means any sequence of digits and `AAA` means any sequence of alphanumeric characters and underscores. In this example, we stick with the first and simplest form – the question mark. The operation above would be rewritten as shown below (error checking is omitted from this example for brevity):

```
Tint KParamIndex = 0 ; // first query parameter

RSqlStatement stmt;
TInt err = stmt.Prepare(database, "INSERT INTO namelist VALUES(?)");
User::LeaveIfError(stmt.BindText(0,aName));
User::LeaveIfError(stmt.Exec());
stmt.Close();
```

The `RSqlStatement::Prepare()` compiles an SQL statement that contains a single parameter. The value for this parameter is initially NULL. The `BindText()` call then converts the value of this parameter to be the content of the `aName` variable. Then we invoke the `Exec()` routine to run the SQL statement and the `Close()` interface to destroy the statement when we are done.

The value bound to a parameter need not be plain text. There are many variations on the `RSqlStatement::BindXXX()` routine to bind other kinds of value such as UTF-16 strings, integers, floating-point numbers, and binary large objects (BLOBs). See the documentation for details. The key point to observe here is that none of these values need to be quoted or escaped in any way. And there is no possibility of being vulnerable to an SQL injection attack.

A similar effect can be achieved using the SQLite C API, using the `sqlite3_prepare()`, `sqlite3_bind_xxx()` and `sqlite3_step()` functions. In this case, besides reducing your vulnerability to SQL injection attacks, the use of bound parameters is also more efficient than constructing SQL statements from scratch, especially when inserting large strings or BLOBs. If the final parameter to the `sqlite3_bind_text()` or `sqlite_bind_text16()` or `sqlite3_bind_blob()` is `SQLITE_STATIC`, the value of the variable being bound does not change for the duration of the SQLite statement. In that case, SQLite does not bother to make a copy of the value but just uses the value that is in the variable being bound. This can be a significant time saving if the value being inserted is a long string or BLOB.

Cache and Reuse Prepared Statements

Evaluating an SQL statement in SQLite is a two-step process. First the statement must be compiled using `RSqlStatement::Prepare()`. Then the resulting prepared statement is run using `RSqlStatement::Next()`. The relative amount of time spent doing each of these steps depends on the nature of the SQL statement. SELECT statements that return a large result set and UPDATE or DELETE statements that touch many rows of a table normally spend most of their time in the Virtual Machine module and relatively little time being compiled. Simple INSERT statements, on the other hand, can take twice as long to compile as they take to run in the virtual machine.

A simple way to reduce the CPU load of an application that uses SQLite is to cache the prepared statements or `RSqlStatement` objects and reuse them. Of course, one rarely needs to run exactly the same SQL statement more than once. But if a statement contains one or more bound parameters, you can bind new values to the parameters prior to each run and thus accomplish something different with each invocation.

This technique is especially effective when doing multiple INSERTs into the same table. Instead of preparing a separate INSERT for each row, create a single generic INSERT statement like this:

```
INSERT INTO important_table VALUES(?,?,?,?,?)
```

Then for each row to be inserted, use one or more of the `RSqlStatement::BindXXX()` interfaces to bind values to the parameters in the INSERT statement, call `RSqlStatement::Exec()` to do the insert, then call `sqlite3_reset()` to rewind the program counter of the internal bytecode in preparation for the next run. For INSERT statements, reusing a single prepared statement in this way typically makes your code run two or three times faster.

You can manually manage a cache of prepared statements, keeping around only those prepared statements that you know will be needed again and destroying prepared statements using `RSqlStatement::Close()` when you are done with them or when they are about to fall out of scope. Depending on the application, it is often more convenient to create a wrapper class around the SQLite interface that manages the cache automatically. The wrapper class can keep around the five or 10 most recently used prepared statements and reuse those statements if the same SQL is requested. Handling the prepared statement cache automatically in a wrapper has the advantage that it frees you to focus more mental energy on writing a great application and less effort on operating the database interface. It also makes the programming task less error prone since, with an automatic class, there is no chance of accidentally omitting a call to `RSqlStatement::Close()` and leaking prepared statements. But the downside is that a cache wrapper does not have the foresight of a human programmer and often caches prepared statements that are no longer needed, thus using excess memory, or discards prepared statements just before they are needed again. This is a classic tradeoff between ease-of-programming and performance.

For applications that are intended for a high-powered workstation, we have found that it is usually best to go with a wrapper class that handles the cache automatically. But when designing an application for a resource-limited mobile device where performance is critical and engineering design talent is plentiful, it might be better to manage the cache manually. Regardless of whether or not the prepared statement cache is managed manually or automatically, reusing prepared statements is always a good thing and can, in some cases, double or triple the performance of the application.

8.4.4 Indexing

Use Indexes to Speed Up Access

Suppose you have a table like this:

```
CREATE TABLE demo5(
  id INTEGER,
  content BLOB
)
```

Further suppose that this table contains thousands or millions of rows and you want to access a single row with a particular `id`. Like this:

```
SELECT content FROM demo5 WHERE id=?
```

The only way that SQLite can perform this query and be certain to get every row with the chosen `id` is to examine every single row, check the `id` of that row, and return the content if the `id` matches. Examining every single row this way is called a 'full table scan'. Reading and checking every row of a large table can be very slow, so you want to avoid full table scans.

The usual way to avoid a full table scan is to create an index on the column you are searching against. In the example above, an appropriate index would be created like this:

```
CREATE INDEX demo5_idx1 ON demo5(id);
```

With an index on the `id` column, SQLite is able to use a binary search to locate entries that contain a particular value of `id`. If the table contains a million rows, the query can be satisfied with about 20 accesses rather than one million accesses. This is a huge performance improvement.

One of the beautiful features of the SQL language is that you do not have to figure out what indexes you might need in advance of coding your application. It is perfectly acceptable, even preferred, to write the code for your application using a database without any indexes. Once the application is running and you make speed measurements, you can add whatever indexes are needed in order to make it run faster. When you add indexes, the query optimizer within the SQL compiler is able to find new more efficient bytecode procedures for carrying out the operations that your SQL statements specify. In other words, by adding indexes late in the development cycle, you have the power to completely reorganize your data access patterns without needing to change a single line of code.

Create Indexes Automatically Using `PRIMARY KEY` and `UNIQUE`

Any column of a table that is declared to be the primary key or that is declared unique is indexed automatically. There is no need to create a separate index on that column using the CREATE INDEX statement. So, for example, this table declaration:

```
CREATE TABLE demo39a(
  id INTEGER,
  content BLOB
);
CREATE INDEX demo39_idx1 ON demo39a(id);
```

is roughly equivalent to the following:

```
CREATE TABLE demo39b(
  id INTEGER UNIQUE,
  content BLOB
);
```

The two examples above are roughly, but not exactly, equivalent. Both tables have an index on the `id` column. In the first case, the index is created explicitly. In the second case, the index is implied by the `UNIQUE` keyword in the type declaration of the `id` column. Both table designs use exactly the same amount of disk space and both use exactly the same bytecode to run queries such as

```
SELECT content FROM demo39 WHERE id=?
```

The only difference is that table `demo39a` lets you insert multiple rows with the same `id` whereas table `demo39b` raises an exception if you try to insert a new row with the same `id` as an existing row.

If you use the `UNIQUE` keyword in the statement that creates the index of `demo39a`, like this:

```
CREATE UNIQUE INDEX demo39_idx1 ON demo39a(id);
```

Then both table designs really would be exactly the same in every way. In fact, whenever SQLite sees the UNIQUE keyword on a column type declaration, all it does is create an automatic unique index on that column.

In SQLite, the PRIMARY KEY modifier on a column type declaration works like UNIQUE; it causes a unique index to be created automatically. The main difference is that you are only allowed to have one PRIMARY KEY. This restriction of only allowing a single PRIMARY KEY is part of the official SQL language definition. The idea is that a PRIMARY KEY is used to order the rows on disk. Some other SQL database engines actually implement primary keys this way.

In SQLite, a PRIMARY KEY is just like any other UNIQUE column. INTEGER PRIMARY KEY is a special case which is handled differently, as described in the next section.

INTEGER PRIMARY KEY is a Fast Special Case

Every row of every SQLite table has a signed 64-bit integer ROWID. This ROWID is the unique key for the B-tree that holds the table content. When you find, insert, or remove a row from a table in SQLite, the row is first located by searching for its ROWID. Searching by ROWID is very fast. Everything in SQLite tables centers around ROWIDs.

In the CREATE TABLE statement, if you declare a column to be of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID. It is generally a good idea to create such a column whenever it is practical.

Looking up information by ROWID or INTEGER PRIMARY KEY is usually about twice as fast as any other search method in SQLite. Consider, for example, an indexed table like this:

```
CREATE TABLE demo1(
  id1 INTEGER PRIMARY KEY,
  id2 INTEGER UNIQUE,
  content BLOB
);
```

Queries against the INTEGER PRIMARY KEY:

```
SELECT content FROM demo1 WHERE id1=?;
```

are about twice as fast as queries like this:

```
SELECT content FROM demo1 WHERE id2=?;
```

Note that the following two queries are identical in SQLite – not just equivalent but identical. They generate exactly the same bytecode:

```
SELECT content FROM demo1 WHERE id1=?;
SELECT content FROM demo1 WHERE RowID=?;
```

For stylistic and portability reasons, the first form of the query is preferred.

In order for the `INTEGER PRIMARY KEY` to truly be an alias for the `ROWID`, the declared type must be exactly `INTEGER PRIMARY KEY`. Variations, such as `INT PRIMARY KEY`, `UNSIGNED INTEGER PRIMARY KEY`, or `SHORTINT PRIMARY KEY`, become independent columns instead of aliases for the `ROWID`. So be careful to always spell `INTEGER PRIMARY KEY` correctly in CREATE TABLE statements.

Use Small Positive Integers for `INTEGER PRIMARY KEY` and `ROWID`

The `ROWID` also occurs in index entries and is used to refer the index entry back to the original table row. When writing the `ROWID` to disk, SQLite encodes the 64-bit integer value using a Huffman code over a fixed probability distribution. The resulting encoding requires between 1 and 9 bytes of storage space, depending on the magnitude of the integer. Small, non-negative integers require less space than larger or negative integers. Table 8.1 shows the storage requirements.

Table 8.1 Storage requirements for `ROWID`

`ROWID` Magnitude		**Bytes of storage**
Minimum value	**Maximum value**	
0	127	1
128	16,383	2
16,384	2,097,151	3
2,097,152	268,435,455	4
268,435,456	34,359,738,367	5
34,359,738,368	4,398,046,511,103	6
4,398,046,511,104	562,949,953,421,311	7
562,949,953,421,312	72,057,594,037,927,935	8
$-\infty$	0	9
72,057,594,037,927,936	∞	9

The table makes it clear that the best way to reduce the number of bytes of disk space devoted to storing ROWIDs is to keep them as small non-negative integers. SQLite works fine with large or negative ROWIDs but the database files take up more space.

SQLite normally works by assigning ROWIDs automatically. An automatically generated ROWID is normally the smallest positive integer that is not already used as a ROWID in the same table. So the first ROWID assigned is 1, the second 2, and so forth. The automatic ROWID assignment algorithm begins by using one-byte ROWIDs, then moves to two-byte ROWIDs as the number of rows increases, then three-byte ROWIDs, and so forth. In other words, the automatic ROWID assignment algorithm does a good job of selecting ROWIDs that minimize storage requirements.

If a table contains an INTEGER PRIMARY KEY column, that column becomes an alias for the ROWID. If you then specify a particular ROWID when doing an INSERT, the ROWID you specify overrides the default choice made by SQLite itself. For example, if your table is like this:

```
CREATE TABLE demo311(
  id INTEGER PRIMARY KEY,
  content VARCHAR(100)
);
```

Then if you write the following statement:

```
INSERT INTO demo311(id,content) VALUES(-17,"Hello");
```

The value −17 is used as the ROWID. Because it is negative, −17 requires nine bytes of space in the disk file. If you leave id blank:

```
INSERT INTO demo311(content) VALUES("Hello");
```

or specify NULL:

```
INSERT INTO demo311(id,content) VALUES(NULL,"Hello");
```

SQLite automatically selects a ROWID value that is the smallest positive integer not already in use, thus minimizing the amount of disk space required.

So, unless you have specific requirements to the contrary, it generally works best to let SQLite pick its own ROWID and INTEGER PRIMARY KEY values.

Use Indexed Column Names in WHERE Clauses

Suppose you have a table with an indexed column you want to search against, like this:

```
CREATE TABLE demo312(
  id INTEGER PRIMARY KEY,
  name TEXT
);
```

When you use `demo312.id` in a query, it is important that the column name appear by itself on one side of the comparison operation and that it not be part of an expression. The following query works very efficiently:

```
SELECT name FROM demo312 WHERE id=?;
```

But the following variations, though logically equivalent, end up doing a full table scan:

```
SELECT name FROM demo312 WHERE id-?=0;
SELECT name FROM demo312 WHERE id*1=?;
SELECT name FROM demo312 WHERE +id=?;
```

To improve performance, you should make sure that the indexed column name appears by itself on one side or the other of the comparison operator, and not inside an expression of some kind. Even a degenerate expression, such as a single unary operator (+), disables the optimizer and causes a full table scan.

Some variation in terms of the WHERE clause is permitted. The column name can be enclosed in parentheses, it can be qualified with the name of its table, and it can occur on either side of the comparison operator. All of the following forms are efficient and, in fact, generate identical bytecode:

```
SELECT name FROM demo312 WHERE id=?;
SELECT name FROM demo312 WHERE demo312.id=?;
SELECT name FROM demo312 WHERE ?=id;
SELECT name FROM demo312 WHERE (id)=?;
SELECT name FROM demo312 WHERE (((demo321.id))=?);
```

The previous examples have all shown SELECT statements. But the same rules apply to the WHERE clause in DELETE and UPDATE statements:

```
UPDATE demo312 SET name=? WHERE id=?;
DELETE FROM demo312 WHERE id=?;
```

Use Multi-Column Indexes

SQLite is able to make use of multi-column indexes. The rule is that if an index is over columns X0, X1, X2, . . . , X*n* of some table, then the index can be used if the WHERE clause contains equality constraints for some prefix of those columns X0, X1, X2, . . . , X*i* where *i* is less than *n*. As an example, suppose you have a table and index declared as follows:

```
CREATE TABLE demo314(a,b,c,d,e,f,g);
CREATE INDEX demo314_idx ON demo314(a,b,c,d,e,f);
```

The index might be used to help with a query that contains a WHERE clause like this:

```
... WHERE a=1 AND b="Smith" AND c=1
```

All three terms of the WHERE clause would be used with the index in order to narrow the search. But the index could not be used if the WHERE clause said:

```
... WHERE b="Smith" AND c=1
```

This WHERE clause does not contain equality terms for a prefix of the columns in the index because it omits a term for `a`. Similarly, only the `a=1` term in the following WHERE clause can be used with the index:

```
... WHERE a=1 AND c=1
```

The `c=1` term is not part of the prefix of terms in the index which have equality constraints because there is no equality constraint on the `b` column.

SQLite can only use a single index for each table within a simple SQL statement. For UPDATE and DELETE statements, this means that only one index can be used, since those statements can only operate on one table at a time. In a simple SELECT statement, multiple indexes can be used if the SELECT statement is a join – one index for each table in the join. In a compound SELECT statement (two or more SELECT statements connected by UNION or INTERSECT or EXCEPT), each SELECT statement is treated separately and can have its own indexes. Likewise, SELECT statements that appear in subexpressions are treated separately.

Some other SQL database engines (for example, PostgreSQL) allow multiple indexes to be used for each table in a SELECT. For example, if you have the following table and indexes in PostgreSQL:

```
CREATE TABLE pg1(a INT, b INT, c INT, d INT);
CREATE INDEX pg1_ix1 ON pg1(a);
CREATE INDEX pg1_ix2 ON pg1(b);
CREATE INDEX pg1_ix3 ON pg1(c);
```

and you run a query such as the following:

```
SELECT d FROM pg1 WHERE a=5 AND b=11 AND c=99;
```

then PostgreSQL may attempt to optimize the query by using all three indexes, one for each term of the WHERE clause.

SQLite does not work this way. SQLite is compelled to select a single index to use in the query. It might select any of the three indexes shown, depending on which one the optimizer thinks will give the best speed. But it can only select a single index and only a single term of the WHERE clause can be used.

SQLite prefers to use a multi-column index such as this:

```
CREATE INDEX pg1_ix_all ON pg1(a,b,c);
```

If the `pg1_ix_all` index is available for use when the SELECT statement above is prepared, SQLite is likely to choose it over any of the single-column indexes because the multi-column index is able to make use of all three terms of the WHERE clause.

You can trick SQLite into using multiple indexes on the same table by rewriting the query. If you rewrite the SELECT statement shown above as:

```
SELECT d FROM pg1 WHERE RowID IN (
  SELECT RowID FROM pg1 WHERE a=5
  INTERSECT
  SELECT RowID FROM pg1 WHERE b=11
  INTERSECT
  SELECT RowID FROM pg1 WHERE c=99
)
```

then each of the individual SELECT statements can use a different single-column index and their results are combined by the outer SELECT statement to give the correct result.

Other SQL database engines, such as PostgreSQL, that can use multiple indexes per table do so by treating simpler SELECT statements as if they were the more complicated SELECT statement shown here.

Use Inequality Constraints on the Last Index Term

Terms in the WHERE clause of a query or UPDATE or DELETE statement are mostly likely to trigger the use of an index if they are an equality constraint – in other words, if the term consists of the name of an indexed column, an equals sign (=), and an expression. So, for example, if you have the following table, index, and query:

```
CREATE TABLE demo315(a,b,c,d);
CREATE INDEX demo315_idx1 ON demo315(a,b,c);
SELECT d FROM demo315 WHERE a=512;
```

The `a=512` term of the WHERE clause qualifies as an equality constraint and is likely to provoke the use of the `demo315_idx1` index.

SQLite supports two other kinds of equality constraint, the IN operator and the `IS NULL` constraint:

```
SELECT d FROM demo315 WHERE a IN (512,1024);
SELECT d FROM demo315 WHERE a IN (SELECT x FROM someothertable);
SELECT d FROM demo315 WHERE a IS NULL;
```

SQLite allows at most one term of an index to be constrained by an inequality, such as less than (<), greater than (>), less than or equal to (<=), or greater than or equal to (>=). The column that the inequality constrains is the right-most term used with the index. For example, in the following query:

```
SELECT d FROM demo315 WHERE a=5 AND b>11 AND c=1;
```

only the first two terms of the WHERE clause are used with the `demo315_idx1` index. The third term, the `c=1` constraint, cannot be used because `c` occurs to the right of `b` in the index and `b` is constrained by an inequality.

SQLite allows up to two inequalities on the same column as long as the two inequalities provide upper and lower bounds on the column:

```
SELECT d FROM demo315 WHERE a=5 AND b>11 AND b<23;
```

All three terms of the above WHERE clause can be used because the two inequalities on `b` provide upper and lower bounds on the value of `b`.

SQLite only uses the four inequalities mentioned above (<, >, <=, and >=) to help constrain a search. Other inequality operators, such as not equal to (!= or <>) and `NOT NULL` are not helpful to the query optimizer; they are never used to control an index and help make the query run faster.

Use Indexes to Help ORDER BY Clauses Evaluate Faster

The default method for evaluating an ORDER BY clause in a SELECT statement is to first evaluate the SELECT statement and store the results in a temporary table, sort the temporary table according to the ORDER BY clause, and scan the sorted temporary table to generate the final output. This method always works, but it requires three passes over the data (one pass to generate the result set, a second pass to sort the result set, and a third pass to output the results) and it requires a temporary storage space sufficiently large to contain the entire result set.

Where possible, SQLite avoids storing and sorting the result set by using an index that causes the results to emerge from the query in sorted order in the first place.

The way to get SQLite to use an index for sorting is to provide an index that covers the columns specified in the ORDER BY clause. For example, if you have the following table, index, and query:

```
CREATE TABLE demo316(a,b,c,data);
CREATE INDEX idx316 ON demo316(a,b,c);
SELECT data FROM demo316 ORDER BY a,b,c;
```

then SQLite uses the `idx316` index to implement the ORDER BY clause, obviating the need for temporary storage space and a separate sorting pass.

An index can be used to satisfy the search constraints of a WHERE clause and to impose the ordering of outputs at the same time. The trick is for the ORDER BY clause terms to occur immediately after the WHERE clause terms in the index. For example, one can write:

```
SELECT data FROM demo316 WHERE a=5 ORDER BY b,c;
```

The `a` column is used in the WHERE clause and the immediately following terms of the index, `b` and `c`, are used in the ORDER BY clause. So, in this case, the `idx316` index can be used both to speed up the search and to satisfy the ORDER BY clause.

The following query also uses the `idx316` index because, once again, the ORDER BY clause term, `c`, immediately follows the WHERE clause terms, `a` and `b`, in the index.

```
SELECT data FROM demo316 WHERE a=5 AND b=17 ORDER BY c;
```

Now consider the following query:

```
SELECT data FROM demo316 WHERE a=5 ORDER BY c;
```

Here there is a gap between the ORDER BY term and the WHERE clause term so the `idx316` index cannot be used to satisfy both the WHERE clause and the ORDER BY clause. The index can be used on the WHERE clause and a separate sorting pass occurs to put the results in the correct order.

Add Result Columns to the End of an Index

SQLite database queries sometimes run faster if their result columns appear in the right-most entries of an index. Consider the following example:

```
CREATE TABLE demo317(a,b,c,data);
CREATE INDEX idx317 ON demo316(a,b,c);
```

A query where all result columns appears in the index:

```
SELECT c FROM demo317 WHERE a=5 ORDER BY b;
```

typically runs about twice as fast as a query that uses columns that are not in the index:

```
SELECT data FROM demo317 WHERE a=5 ORDER BY b;
```

The reason for this is that when all information is contained within the index entry only a single search has to be made for each row of output. But when some information is in the index and other information is in the table, first there must be a search for the appropriate index entry then a separate search for the appropriate table row, based on the `ROWID` found in the index. Twice as much searching has to be done for each row of output generated.

The extra query speed does not come for free, however. Adding additional columns to an index makes the database file larger. So when developing an application, the programmer must consider the trade-off between space and time to determine whether the extra columns should be added to the index or not.

To summarize, the best sets of columns to put in an index can be described, in order, as follows:

1. columns that have equality constraints in the WHERE clause;
2. columns that are specified in the ORDER BY clause;
3. columns that are used in the result set.

If any column of the result set must be obtained from the original table, then the table row must be located. There is no speed advantage in selecting some columns from the index, so you might as well omit the extra columns from the end of the index and save on storage space. The performance improvement described in this section can only be realized when every column in a table is obtainable from the index.

Resolve Indexing Ambiguities Using the Unary + Operator

The SQLite query optimizer usually does a good job of choosing the best index to use for a particular query, especially if ANALYZE has been run to provide it with index performance statistics. But occasions do arise where it is useful to give the optimizer hints.

One of the easiest ways to control the operation of the optimizer is to disqualify terms in the WHERE clause or ORDER BY clause as candidates for optimization by using the unary + operator. In SQLite, a unary + operator makes no change to its operand, even if the operand is something other than a number. So you can prepend + to any expression in SQLite without changing the meaning of the expression. On the other hand, the optimizer only uses an index for terms in WHERE, HAVING, or ON clauses that have an index column alone on one side of a comparison operator. If you want to prevent such a term from being used by the optimizer, all you have to do is prepend '+' to the column name.

For example, suppose the database has the following schema:

```
CREATE TABLE demo321(a,b,c,data);
CREATE INDEX idx321a ON demo321(a);
CREATE INDEX idx321b ON demo321(b);
```

If you issue a query such as this:

```
SELECT data FROM demo321 WHERE a=5 AND b=11;
```

The query optimizer might use either `idx321a` or `idx321b`. The choice is arbitrary. If you want to force the use of `idx321a`, you can accomplish that by disabling the second term of the WHERE clause as an optimization candidate:

```
SELECT data FROM demo321 WHERE a=5 AND +b=11;
```

The term `+b=11` is an expression instead of an indexed column name and the optimizer does not recognize that it can be used with an index. The optimizer is compelled to use an index on the first term.

The unary '+' operator can also be used to disable ORDER BY clause optimizations. Consider this query:

```
SELECT data FROM demo321 WHERE a=5 ORDER BY b;
```

The optimizer has the choice of using `idx321a` with the WHERE clause to restrict the search or doing a full table scan and using `idx321b` to satisfy the ORDER BY clause, thus avoiding a separate sorting pass. To force the use of `idx321a` on the WHERE clause:

```
SELECT data FROM demo321 WHERE a=5 ORDER BY +b;
```

To go the other way and force the `idx321b` index to be used to satisfy the ORDER BY clause, disqualify the WHERE term:

```
SELECT data FROM demo321 WHERE +a=5 ORDER BY b;
```

You should not overuse the unary + operator. The SQLite query optimizer usually picks the best index without any help and premature use of unary + can confuse the optimizer and cause less optimal performance. However, in some cases, it is useful to be able to override the decisions of the optimizer and the unary + operator is an excellent way to do this.

Avoid Indexing Large BLOBs and CLOBs

Recall that SQLite stores indexes as B-trees. Each B-tree node uses one page of the database file and the default page size is 1024 bytes. In order not to fan out too much, the B-tree module within SQLite requires that at least four entries must fit on each page of a B-tree. There is also some overhead associated with each B-tree page. At most, about 250 bytes of space are available on the main B-tree page for each index entry.

If an index entry exceeds the space allocation, excess bytes spill onto overflow pages. There is no arbitrary limit on the number of overflow pages or on the length of a B-tree entry but for maximum efficiency, it is best to avoid overflow pages, especially in indexes, which means that you should strive to keep the number of bytes in each index entry below 250.

If you keep the size of index entries significantly smaller than 250 bytes, then the B-tree fans out less, the binary search algorithm used to search for entries in an index has fewer pages to examine and the algorithm, therefore, runs faster. So the fewer bytes used in each index entry the better, at least from a performance perspective.

For these reasons, it is recommended that you avoid indexing large BLOBs and CLOBs. SQLite works when large BLOBs and CLOBs are indexed, but there is a performance impact.

If you need to look up entries using a large BLOB or CLOB as the key, then by all means use an index. An index on a large BLOB or CLOB is not as fast as an index using more compact data types such as integers, but it is still many orders of magnitude faster than doing a full table scan. So to be more precise, the advice of this section is that you should design your applications so that you do not need to look up entries using a large BLOB or CLOB as the key. Try to arrange to have compact keys consisting of short strings or integers.

Note that many other SQL database engines disallow the indexing of BLOBs and CLOBs. SQLite is more flexible than most, in that it does allow BLOBs and CLOBs to be indexed and it uses those indexes when it is appropriate. But, for maximum performance, it is best to use smaller search keys.

Avoid Creating Too Many Indexes

Some developers approach SQL-based application development with the attitude that indexes never hurt and that the more indexes you have the more likely your application will run fast. This is definitely not the case. There are costs associated with each new index you create:

- Each new index takes up additional space in the database file. The more indexes you have, the larger your database files become for the same amount of data.
- Every INSERT and UPDATE statement modifies both the original table and all indexes on that table. So the performance of INSERT and UPDATE decreases linearly with the number of indexes.
- Compiling new SQL statements using `sqlite3_prepare()` takes longer the more indexes that the optimizer has to choose between.
- Surplus indexes give the optimizer more opportunities to make a bad choice.

Your policy on indexes should be to avoid them wherever you can. Indexes are powerful and can work wonders to improve the performance of a program. But, just as too many drugs can be worse than none at all, so too many indexes can cause more harm than good.

When building a new application, a good approach is not to explicitly declare any indexes at the beginning and only add indexes as needed to address specific performance problems.

Take care to avoid redundant indexes. For example, consider this schema:

```
CREATE TABLE demo323a(a,b,c);
CREATE INDEX idx323a1 ON demo323(a);
CREATE INDEX idx323a2 ON demo323(a,b);
```

The `idx323a1` index is redundant and can be eliminated. Anything that the `idx323a1` index can do, the `idx323a2` index can do better. Other redundancies are not quite as apparent as the above. Recall that any column or columns that are declared `UNIQUE` or `PRIMARY KEY` (except for the special case of `INTEGER PRIMARY KEY`) are automatically indexed. In the following schema, both indexes are redundant and can be eliminated with no loss in query performance:

```
CREATE TABLE demo323b(x TEXT PRIMARY KEY, y INTEGER UNIQUE);
CREATE INDEX idx323b1 ON demo323b(x);
CREATE INDEX idx323b2 ON demo323b(y);
```

Occasionally one sees a novice SQL programmer use both `UNIQUE` and `PRIMARY KEY` on the same column:

```
CREATE TABLE demo323c(p TEXT UNIQUE PRIMARY KEY, q);
```

This has the effect of creating two indexes on the `p` column – one for the `UNIQUE` keyword and another for the `PRIMARY KEY` keyword. Both indexes are identical so clearly one can be omitted. A `PRIMARY KEY` is guaranteed to be unique so the `UNIQUE` keyword can be removed from the `demo323c` table definition with no ambiguity or loss of functionality.

It is not a fatal error to create too many indexes or redundant indexes in SQLite. SQLite continues to generate the correct answers. But it might take longer to produce those answers and the resulting database files might be a little larger. So for best results, keep the number of indexes to a minimum.

8.4.5 Using the Optimizer

The ANALYZE Command

A big part of the job of the SQL compiler within SQLite is trying to figure out which of multiple implementation options should be used for a particular SQL statement. A table often has two or more indexes and the optimizer needs to choose one of those indexes to use. The optimizer may have to decide between using an index to implement the WHERE clause or the ORDER BY clause. For a join, the optimizer has to pick an appropriate order for the tables in the join, and so forth.

By default, the only information the optimizer has to go on when trying to choose between two or more indexes is the database schema. In many cases, the schema alone is not sufficient to make a wise choice. Suppose, for example, you have a table with two indexes:

```
CREATE TABLE demo318(a,b,c,data);
CREATE INDEX idx318a ON demo381(a);
CREATE INDEX idx318b ON demo381(b);
```

And further suppose SQLite is trying to compile a statement such as the following:

```
SELECT data FROM demo318 WHERE a=0 AND b=11;
```

The optimizer has to decide whether to use `idx318a` or `idx318b`. Without additional information, the two approaches are equivalent and so the choice is arbitrary. In reality, the choice of index might have a big impact on performance. Suppose that `demo318` contains one million rows of which 55% have a=0, 40% have a=1, and the rest have a=2 or a=3. If the `idx381a` index is chosen, it only narrows down the set of candidate rows by half. You are unlikely to notice any performance gain at all. In fact, in this extreme case, using the index is likely to make the query slower!

On the other hand, let us assume that there are never more than two or three rows with the same value for column `b`. If the `idx381b` index is used, the set of candidate rows is reduced from one million to two or three, which is a huge speed improvement. Clearly, in this example, we would want the optimizer to choose index `idx318b`. But without knowledge of the table contents and the distribution of values in the columns, the optimizer has no way of knowing which of the `idx318a` and `idx318b` indexes will work better.

The ANALYZE command is used to provide the SQLite query optimizer with statistical information about the distribution of values in a database so that it can make better decisions about which indexes to use. When you run the ANALYZE command, SQLite creates a special table named `sqlite_stat1`. It then reads the entire database, gathers statistics and puts those statistics in the `sqlite_stat1` table so that the optimizer can find them.

The ANALYZE command is relatively slow, since it has to read every single row of every single table in your database. The statistical information generated by ANALYZE is not kept up to date with subsequent changes to the database. However, most databases have a relatively constant statistical profile and so you can normally get by with running ANALYZE once and using that statistical profile thereafter.

For embedded applications, you can often get by without ever having to run ANALYZE on a real database on the embedded device. Instead, you can generate a template database that contains typical data for your application and run ANALYZE on that database on a developer workstation. You can then copy the content of the resulting `sqlite_stat1` table over to the embedded device. You cannot CREATE or DROP the `sqlite_stat1` table but you can SELECT, DELETE, and UPDATE it. To create the `sqlite_stat1` table on the embedded device, run ANALYZE when the database is empty. That takes virtually no time and leaves you with an empty `sqlite_stat1` table. You can then copy the contents of the `sqlite_stat1` table from the developer workstation into the database on the embedded device and the optimizer uses it just as if it had been created locally.

There is one entry in the `sqlite_stat1` table for each index in your schema. Each entry contains the name of the table being indexed, the name of the index, and a string composed of two or more integers separated by spaces. These integers are the statistics that the ANALYZE command collects for each index. The first integer is the total number of entries in the table. The second integer is the average number of rows in a result from a query that had an equality constraint on the first term of the index. The third integer (if there is one) is the average number of rows that would result if the index were used with equality constraints on the first two terms of the index, and so on for as many terms as there are in the index.

The smaller the second and subsequent integers are, the more selective an index is. And the more selective the index is, the fewer rows have to be processed when using that index. So the optimizer tries to use indexes with small second and subsequent numbers. The details are complex and do not really add any understanding to what is going on. The key points are that the second and subsequent integers in the `sqlite_stat1` table are a measure of the selectivity of an index and that smaller numbers are better.

The statistics gathered by the ANALYZE command reflect the state of the database at a single point in time. While making the final performance tweaks on an application, you may want to adjust these statistics manually to better reflect the kinds of data they expect to encounter in the field. This is easy to do. You can read and write the `sqlite_stat1` table using ordinary SQL statements. You are free to adjust or modify the statistics in any way that seems to give a performance benefit.

EXPLAIN QUERY PLAN

When tweaking an SQL statement to try to get it to run faster, it is helpful to know what indexes the optimizer is choosing and, in the case of a join, in what order the tables are being processed. You can discover this information on a workstation using the SQLite command-line shell by prefixing the SQL statement with the keywords EXPLAIN QUERY PLAN. When you do this, the result of the SQL statement is a table with three columns. The first two columns are integers that show the order in which the tables are evaluated in the join and the order in which the tables appear in the original SQL of the join, respectively. The third column contains text which shows the tables being accessed and the indexes used to access them. The third column is usually the most instructive.

The following is the output of EXPLAIN QUERY PLAN for an eight-way join used in an embedded application:

```
0|3|TABLE pidenums WITH INDEX pidenumvalueidx ORDER BY
1|2|TABLE pids WITH INDEX pids_pidenumid
2|4|TABLE units USING PRIMARY KEY
3|7|TABLE pidformulaconversions USING PRIMARY KEY
4|1|TABLE pidgroupstopids WITH INDEX pidgrppid_pidid
5|5|TABLE controllers USING PRIMARY KEY
6|6|TABLE controllerprotocols USING PRIMARY KEY
7|0|TABLE userpiddetails WITH INDEX pidgroup2pidididx
```

The first column of integers shows the original order of the tables as the query optimizer has chosen to evaluate them. The second column shows the order of the tables as they were originally presented in the FROM clause of the SELECT statement. The third column shows the name of the table and the index used to access it. The first line shows that the pidenums table is accessed using the pidenumvalueidx index. The ORDER BY keywords at the end of the first line shows us that the pidenumvalueidx index is also used to guarantee that the results appear in the correct order and that no separate sorting pass is required. The USING PRIMARY KEY phrase on lines 3, 4, 6, and 7 show that those tables are accessed by ROWID instead of by an index. Sometimes a table appears with neither WITH INDEX nor USING PRIMARY KEY clauses. Consider the following two-way join:

```
0|0|TABLE pidenums
1|1|TABLE pids WITH INDEX pids_pidenumid
```

No index is used for accessing the pidenums table. That means that a full table scan occurs. If pidenums contains many rows, the query could be slow.

To save space on embedded devices, the EXPLAIN QUERY PLAN capability is typically omitted on the version of SQLite that goes onto the device. This is rational since EXPLAIN QUERY PLAN is not particularly useful on the device itself. It is intended for use on a developer workstation when doing tuning and analysis.

8.4.6 Resource Usage

Shrink a Database File

One way to shrink a SQLite database is to run the vacuum command, which essentially rebuilds the entire database from scratch. It creates a temporary database file and copies the content of the current database to the temporary database, taking care to pack the data into the temporary database as tightly as possible. When the temporary database has been fully populated, the original database is truncated and the temporary database is copied back in place of the original. All this occurs under the protection of a rollback journal so that the vacuum operation is guaranteed to be atomic even if the application crashes or suffers a power failure in the middle of it.

The vacuum command works well for what it does but you need to be aware that it can use a lot of temporary disk space. The amount of temporary space used can be as much as twice the size of the original database file. If you are running short of persistent storage and want to vacuum in order to reclaim some space, then this need for a large amount of temporary storage can be a serious problem and can effectively prevent vacuum from running.

Another problem with vacuum is that it has to read the entire database file twice and write it three times. It can be slow for large databases.

An alternative to issuing the vacuum command is to create the database using ‘auto-vacuum’ mode. The auto-vacuum mode is selected by default when creating a Symbian SQL database. Using the SQLite C API, auto-vacuum mode must be selected using a PRAGMA when a database is initially created – before any CREATE TABLE statements have been issued and when the size of the database file is still 0 bytes. The PRAGMA looks like this:

```
PRAGMA auto_vacuum=ON;
```

The auto-vacuum PRAGMA only works on a new empty database file and only persists if one or more tables are created in the database before the connection that issued the PRAGMA closes. After creating a database, it is prudent to make sure that auto-vacuum was set correctly by invoking the following command:

```
PRAGMA auto_vacuum;
```

This statement returns either 0 or 1 to indicate whether or not auto-vacuum has been set for the database.

A database in auto-vacuum mode is slightly larger than one that is not, since an auto-vacuum database is required to store some additional information in the database file to facilitate the auto-vacuuming function. Updates to an auto-vacuumed database are slightly slower since this additional information must be kept up to date. But the advantage of an auto-vacuumed database is that when B-tree pages are no longer needed, they are moved to the end of the database file and then the database file is truncated, thus returning the unused pages back to the filesystem. Running a large DELETE on an auto-vacuumed database causes the database file to shrink.

The auto-vacuum functionality was implemented specifically to support applications on embedded devices. For workstation-based applications, the vacuum command seems to work better. But on embedded devices with limited mass storage, auto-vacuum is the preferred approach.

Avoid Running out of Filesystem Space

SQLite's efforts to make sure that all database changes are atomic and durable results in the following irony: You cannot delete information from a SQLite database if the filesystem is full. SQLite requires some temporary disk space in order to delete data. This is true regardless of whether or not auto-vacuum is set.

The temporary disk space is needed to hold the rollback journal that SQLite creates as part of its COMMIT processing. Before making any changes to the original database file, SQLite copies the original content of the pages to be changed into the rollback journal file. Once the original content is preserved in the rollback journal, modifications can be made to the database file. Once the file has been updated, the rollback journal is deleted and the transaction commits.

The purpose of the rollback journal is to provide a means of recovering the database if a power failure or other unexpected failure occurs during the delete (or other updating operation). Whenever a SQLite database is opened, a check is made to see if rollback journal exists for that database. The presence of a rollback journal indicates that previous changes to data failed to complete. The original database content stored in the rollback journal is copied back into the database file thus reverting the database back to its state at the start of the transaction.

Since the rollback journal stores the original content of pages that are changing, the size of the rollback journal depends on how many pages are changing in the transaction. The more records in the database that are modified, the larger the rollback journal becomes.

A consequence of this is that any change to a SQLite database requires temporary disk space that is roughly proportional to the size of change. Any delete requires some temporary space and a large delete requires correspondingly more temporary disk space than a small delete. Designers of applications for embedded systems should therefore takes steps to make sure that filesystem space is never completely exhausted. And when filesystem space is low, large updates and deletes should be broken up into smaller transactions to avoid creating excessively large rollback journals.

Keep Schemas Small and Unchanging

SQLite stores the original text of all CREATE statements in the `sqlite_master` table of the database. When a new database connection is created using `sqlite3_open()` or when the database schema is changed, SQLite has to read and parse all of the CREATE statements in order to reconstruct its internal symbol tables. The parser and symbol table builder inside SQLite are very fast but they still take time, which is proportional to the size of the schema. To minimize the computational overhead associated with parsing the schema:

- Keep the number and size of CREATE statements in the schema to a minimum.
- Avoid unnecessary calls to `sqlite3_open()`. Reuse the same database connection where possible.
- Avoid changing the database schema while other threads or processes might be using the database.

Whenever the schema is reparsed, all statements that were prepared using the old schema are invalidated. If you try to pass a statement to `sqlite3_step()` that was prepared prior to the schema reparsing, `sqlite3_step()` immediately returns `SQLITE_ERROR` and the subsequent `sqlite3_reset()` or `sqlite3_finalize()` return `SQLITE_SCHEMA` to let you know that the cause of the error was a schema change. Any prepared statements that were cached have to be discarded and prepared again. Refilling the prepared statement cache can be as computationally expensive as parsing the schema in the first place.

Prepared statements can be invalidated for reasons other than schema changes. Some of the other causes include:

- Executing ATTACH or DETACH
- Calling `sqlite3_create_function()`
- Calling `sqlite3_create_collation()`
- Calling `sqlite3_set_authorizer()`

Avoid all of the above events if you have a large cache of prepared statements. The best strategy is to make all necessary calls to `sqlite3_create_function()`,`sqlite3_create_collation()`, and `sqlite3_set_authorizer()` and to ATTACH all associated databases as soon as a new database connection is created and before any statements are prepared.

Avoid Tables and Indexes with an Excessive Number of Columns

SQLite places no arbitrary limits on the number of columns in a table or index. We have seen commercial applications using SQLite that construct tables with tens of thousands of columns. And these applications actually work.

However SQLite is optimized for the common case of tables with no more than a few dozen columns. For best performance, you should try to stay in the optimized region. Furthermore, we note that relational databases with a large number of columns are usually not well normalized. So, even apart from the performance considerations, if you find your design has tables with more than a dozen or so columns, you really need to rethink how you are building your application.

There are a number of places in `sqlite3_prepare()` that run in time $O(N^2)$ where N is the number of columns in the table. The constant of proportionality is small in these cases so you should not have any problems when N is less than 100 but for N on the order of 1000, the time to run `sqlite3_prepare()` can start to become noticeable.

When the bytecode is running and it needs to access the *i*th column of a table, the values of the previous $i-1$ columns must be accessed first. So if you have a large number of columns, accessing the last column can be an expensive operation. This fact also argues for putting smaller and more frequently accessed columns early in the table.

There are certain optimizations that only work if the table has 30 or fewer columns. The optimization that extracts all necessary information from an index and never refers to the underlying table works this way. So in some cases, keeping the number of columns in a table at or below 30 can result in a twofold speed improvement. Indexes are only used if they contain 30 or fewer columns. You can put as many columns in an index as you want, but if there are more than 30, the index will not improve performance and will not do anything but take up space in your database file.

Avoid Corrupting Database Files

One of the key benefits of using an atomic and durable database engine, such as SQLite, is that you can be reasonably confident that the database will not be corrupted by application crashes or power failures. SQLite is very resistant to database corruption but it is possible to corrupt a SQLite database. This section describes all of the known ways to corrupt a SQLite database so that you can make sure you avoid them.

A SQLite database is just an ordinary file in the filesystem. Any process with write permission can open that file and write nonsense into the middle of it, corrupting the database. Similarly, an operating system malfunction or a hardware fault can cause invalid data to be written into the database file. Both of these issues are beyond the control of SQLite or of application developers. We only mention them here for completeness and because these are the most common causes of database corruption we see in actual deployments.

At critical times during the commit process, SQLite pauses and waits for prior writes to reach the disk (or be burned into flash) before continuing with subsequent writes. These pauses are what makes COMMIT run so slowly. Sometimes programmers, in their quest for better performance, set this PRAGMA:

```
PRAGMA synchronous=OFF;
```

When synchronous is turned off, SQLite does not pause and wait for disk I/O to complete before continuing with its commit process. This dramatically increases write performance, but it also means that an unexpected power failure can corrupt the database file.

After a power loss or system crash and subsequent system reboot, a rollback journal file can be found in the same directory as the original database. The presence of this rollback journal file is the signal to subsequent users of the database that the database is in an inconsistent state and needs to be restored (by playing back the journal) before it is used. A rollback journal file that is left over after a crash is called a 'hot journal.'

If after a post-crash reboot some kind of filesystem recover operation occurs which deletes, renames, or moves a hot journal, then applications that try to access the database file will have no way of knowing that the hot journal should exist. They will not know that the database is in an inconsistent state and will have no way to restore it to a consistent state. Deleting or renaming a hot journal results in a corrupted database nearly every time.

The name of the hot journal is related to the name of the original database file. If the database file is renamed, a process that tries to open the database by the new name will not see the hot journal which is based on the original name, no database recovery will be undertaken and the database will become corrupt. On systems that support multiple names for the same file (hard or soft links or aliases) if the database is open under one name when the crash occurs but is opened using a different name after the crash, the hot journal will not be found on the post-crash open, since the name of the hot journal was based on a different database file name, and the database will become corrupt.

A database can be corrupted if the locking mechanism used to serialize access to the database file is faulty. The OS Layer implements different locking mechanisms for different operating systems. If any of the locking mechanisms do not work correctly, database corruption can occur. This happens most commonly on network filesystems where locking protocols are difficult to design and implement correctly. It is rarely a problem on locally mounted filesystems.

Finally, a database might be corrupted by bugs in SQLite. This has happened in the past, but only rarely, and not with any recent version of SQLite.

These are all of the known ways to corrupt a SQLite database file. As you can see, none of these ways are easy to achieve. SQLite databases have proven to be remarkably reliable and trouble-free. By following a few simple precautions – leave the synchronous PRAGMA turned on, avoid deleting or renaming hot journals, avoid database filename aliases, and use well-tested and reliable system software and hardware – you can ensure that your databases is safe and intact even after system crashes and untimely power failures.

Avoid Queries that Require Transient Tables

Complex queries sometimes require SQLite to compute intermediate results that are stored in 'transient' tables. Transient tables are nameless entities that exist for the duration of a single SQL statement and are automatically deleted at the conclusion of the statement. We use the term 'transient' to describe these tables rather than 'temporary' to avoid confusion with TEMP tables (named tables that are private to a particular database connection and which persist for the duration of that database connection). Transient tables are invisible, internal holding areas for intermediate results that exist only until the end of the current SQL statement. The SQLite source code sometimes refers to transient tables as 'ephemeral' tables.

Transient tables are stored either in RAM or on disk, depending on compile-time and run-time settings. The `TEMP_STORE` preprocessor macro can be set to control the storage of transient tables at compile time. If `TEMP_STORE`=0, then transient tables are always stored on disk. If `TEMP_STORE`=1 then transient tables are stored on disk by default. If `TEMP_STORE`=2 then transient tables are stored in RAM by default. Finally, if `TEMP_STORE`=3 then transient tables are always stored in RAM.

At run time, you can issue one of the following PRAGMAs to influence where transient tables are stored if the `TEMP_STORE` compile-time setting is 1 or 2:

```
PRAGMA temp_store=memory;
PRAGMA temp_store=file;
PRAGMA temp_store=default;
```

The `temp_store=memory` PRAGMA causes transient tables to be stored in RAM. The `temp_store=file` PRAGMA causes transient tables to be kept on disk. The `temp_store=default` PRAGMA causes transient tables to be saved according to the default compile-time setting. If `TEMP_STORE` is set to 0 or 3 at compile time, it always overrides the run-time PRAGMA.

Note that when transient tables are stored on disk, in reality a combination of memory and file space is used to hold the table. Transient tables go through the same page cache mechanism that regular database files go through. As long as the transient tables do not grow too large, they are always held in memory in the page cache rather than being written to disk. Information is only written to disk when the cache overflows. The real difference between storing transient files in memory and in a file is that with memory storage, the cache is considered infinitely large, never overflows and thus nothing is ever written out to the filesystem. Storing transient tables in memory rather than in files avoids the overhead of file I/O but requires potentially unbounded amounts of memory. Using file-based transient tables puts an upper bound on the amount of memory required but adds file I/O overhead. This is a tough trade-off. It is hard to know which approach will work best. SQLite provides both compile-time and run-time options for the transient table storage strategy in order to give you maximum flexibility in choosing an approach that works well in your particular application.

One strategy for dealing with transient tables is to avoid them all together. If you never use transient tables then it does not matter if they would be stored in memory or on disk. The remainder of this section enumerates the things that might provoke SQLite to create a transient table. If you avoid all of these things in your code, then you never need to worry about where your transient tables are stored.

A transient table is created whenever you use the DISTINCT keyword in a query:

```
SELECT DISTINCT name FROM artist;
```

The DISTINCT keyword guarantees that each row of the result set is different from all other rows. In order to enforce this, SQLite creates a transient table that stores all prior output rows from the query. If a row is already found in the transient table, then it is skipped. The same situation occurs when DISTINCT is used within an aggregate function:

```
SELECT avg(DISTINCT cnt) FROM playlist;
```

In this context, the DISTINCT keyword means that the aggregate function is only applied to distinct elements of the result. As before, a transient table is used to record prior values of the result so that SQLite can tell if new results have been seen before.

The UNION, INTERSECT, and EXCEPT operators always generate a distinct set of rows. Even though the DISTINCT keyword does not appear, it is implied and a transient table is used to enforce the distinctness. In contrast, the UNION ALL operator does not require a transient table.

A transient table might be used to implement an ORDER BY or GROUP BY clause. SQLite always tries to use an index to satisfy an ORDER BY or GROUP BY clause if it can. But if no indexes are available which can satisfy the ORDER BY or GROUP BY, then the entire results set is loaded into a transient table and sorted there.

Subqueries on the right of the IN operator use a transient table. Consider an example:

```
SELECT * FROM ex334a WHERE id IN (SELECT id FROM ex334b WHERE amt>7);
```

The results of the subquery are stored in a transient table. Then the value of the `id` column is checked against this table for each row of the outer query to determine if that row should be included in the result set.

Sometimes subqueries in the FROM clause of a query result in a transient table. This is not always the case because SQLite tries very hard to convert subqueries in the FROM clause into a join that does not use subqueries. SQLite uses the term ‘flattening’ to describe the conversion of FROM clause subqueries into joins. Flattening is an optimization that makes queries run faster. In some cases, flattening cannot occur. When the flattening optimization is inhibited, the results of the subqueries are stored in transient tables and then a separate query is run against those transient tables to generate the final results.

Consider the following schema and query:

```
CREATE TABLE t1(a,b,c);
CREATE TABLE t2(x,y,z);
SELECT * FROM t1 JOIN (SELECT x,y FROM t2);
```

The subquery in the FROM clause is plainly visible in the SELECT statement above. But if the subquery were disguised as a view, it might be less noticeable. A view is really just a macro that serves as a place-holder for a subquery. So the SELECT statement above is equivalent to the following:

```
CREATE VIEW v2 AS SELECT x, y FROM t2;
SELECT * FROM t1 JOIN v2;
```

In either case above, whether the subquery is stated explicitly or is implied by the use of a view, flattening occurs and the query is converted into this:

```
SELECT a,b,c,x,y FROM t1 JOIN t2;
```

Had flattening not occurred, it would have been necessary to evaluate the `v2` view or the subquery into a transient table and then execute the outer query using the transient table as one of the two tables in the join. SQLite prefers to flatten the query because a flattened query generally uses fewer resources and is better able to take advantage of indexes. The rules for determining when flattening occurs and when it does not are complex. Flattening occurs if all of the following conditions in the outer query and in the subquery are satisfied:

- The subquery and the outer query do not both use aggregates.
- The subquery is not an aggregate or the outer query is not a join.
- The subquery is not the right operand of a left outer join, or the subquery is not itself a join.
- The subquery is not DISTINCT or the outer query is not a join.
- The subquery is not DISTINCT or the outer query does not use aggregates.
- The subquery does not use aggregates or the outer query is not DISTINCT.
- The subquery has a FROM clause.
- The subquery does not use LIMIT or the outer query is not a join.
- The subquery does not use LIMIT or the outer query does not use aggregates.
- The subquery does not use aggregates or the outer query does not use LIMIT.
- The subquery and the outer query do not both have ORDER BY clauses.
- The subquery is not the right term of a LEFT OUTER JOIN or the subquery has no WHERE clause.
- The subquery and outer query do not both use LIMIT.
- The subquery does not use OFFSET.

Nobody really expects a programmer to memorize or even understand the above set of flattening rules. As a shortcut, perhaps it is best to remember that a complicated subquery or view in the FROM clause of a complicated query might defeat the flattening optimization and thus require the use of transient tables.

An obscure use of transient tables is when there is an `INSTEAD OF DELETE` or `INSTEAD OF UPDATE` trigger on a view. When such triggers exist and a DELETE or an UPDATE is executed against that view, then a transient table is created to store copies of the rows to be deleted or updated. Since it is unusual to have `INSTEAD OF` triggers in the first place, this case rarely arises.

In summary, transient tables are used to implement the following features:

- the DISTINCT keyword and other situations where distinct results are required, such as compound queries using UNION, INTERSECT, or EXCEPT
- ORDER BY or GROUP BY clauses that cannot be satisfied by indexes
- subqueries on the right-hand side of the IN operator
- subqueries or views in the FROM clause of a query that cannot be flattened
- DELETE or UPDATE against a view with `INSTEAD OF` triggers.

If you are running SQLite in a resource-limited environment where file I/O is expensive and memory is scarce, you are well advised to avoid these constructs and thus avoid the need for transient tables.

Avoid Using Excess Memory

SQLite is judicious in its use of memory. On workstations where memory is abundant, it is practically impossible to get SQLite to use more memory than is easily available. However, on embedded devices with limited resources, memory always seems to be in short supply and in that kind of environment it is helpful to know which features within SQLite use more memory than others.

The largest user of memory in SQLite is the page cache. The page cache sits between the B-tree layer and the file system. The page cache keeps a copy of every page that is currently in use as well as copies of recently used pages that might be reused again in the near future. It does this for the main database file, for each ATTACHed database, and for the various temporary files used to implement TEMP and transient tables.

Generally speaking, a larger page cache gives better performance since a larger cache generally reduces the need for file I/O. The default build for SQLite (designed for use on workstations) sets the maximum size of the page cache to 2000 pages for regular database files and 500 pages for TEMP and transient databases. Depending on the number of bytes per page, the page cache can easily use a dozen or more megabytes of memory in a workstation configuration. The Pager module does not normally use this much memory even on a workstation, however. The maximum page cache size is just that: a maximum. The Pager uses only as much memory as it really needs and so if you are working with a small database or with simple queries, the amount of memory used by the Pager is typically much smaller than the maximum. Nevertheless, the maximum page cache sizes for embedded devices are usually much smaller than for workstations because embedded devices usually have much less memory. SQLite works well with a maximum page cache size as small as 10.

Note that a separate page cache is used for the main database file, the database file used to hold TEMP tables, each ATTACHed database, and each transient table. So a good way to keep down the amount of memory used by the pager is to avoid using TEMP tables, ATTACHed databases, and transient tables. TEMP tables and ATTACHed database are easy to control since they are specified explicitly. Transient tables, on the other hand, are implicit and so controlling their use is more subtle (as discussed in the previous section).

The programmer has some control of the size of the page cache in the main database file, the TEMP database, and all ATTACHed databases. The maximum page cache size can be set using a PRAGMA:

```
PRAGMA main.cache_size=200;
PRAGMA temp.cache_size=50;
PRAGMA aux1.cache_size=10;
```

In the example above, the `main.cache_size=200` PRAGMA sets the maximum size of the page cache used by the main database file to 200 pages. The second PRAGMA sets the maximum size of the page cache for TEMP tables and the third PRAGMA sets the maximum cache size for an ATTACHed database named `aux1`. There is currently no way to set the maximum cache size for transient tables, except at compile-time using the `SQLITE_DEFAULT_TEMP_CACHE_SIZE` preprocessor macro.

Another big user of memory in SQLite is the symbol table. When a SQLite database connection is created, the database schema is read out of the `sqlite_master` table and parsed into internal data structures that are accessible to the SQL compiler. This symbol table exists for the life of the database connection. The size of the symbol table is roughly proportional to the size of the database schema. For a typical schema, the symbol table size is less than 20 KB – sometimes much less – but for a really complex schema, the size of the symbol table can be much larger. So if memory is tight, it is best to keep the complexity of the database schema to a minimum. Keeping the schema simple has the added benefit of making it easier to parse and thus speeding database connection setup.

Each prepared statement (each `sqlite3_stmt` object created by `sqlite3_prepare()`) uses memory that is roughly proportional to the complexity of the SQL statement it implements. Most prepared statements use less than 1 KB of memory when they are not running. All the memory associated with a prepared statement is freed when the prepared statement is finalized. Prepared statements use some additional memory when they are running. The additional memory is freed when the prepared statement is reset using `sqlite3_reset()`.

Most internal values used during SQL statement execution are stored in preallocated space inside the prepared statement. However, additional memory is allocated to store large strings and BLOBs. This additional memory is freed when the use of the string or BLOB ends, and efforts are made to make sure that no unnecessary copies of the string or BLOB are made during processing. While they are held in memory, strings and BLOBs use memory space proportional to their size.

SQLite implements the DELETE command using a two-pass algorithm. The first pass determines the `ROWID`s for all rows in the table that are to be deleted and stores them in memory. The second pass does the actual deleting. Because the first pass stores the `ROWID`s, the DELETE operation can use memory that is proportional to the number of rows to be deleted. This is not normally a problem. Each `ROWID` fits in only 8 bytes. But if tens of thousands of rows are being deleted from a table, even 8 bytes per row can be more memory than an embedded device has available. A reasonable workaround is to do the DELETE in two or more smaller chunks.

If you are deleting all the rows in a table and the DELETE statement has no WHERE clause, then SQLite uses a special optimization that deletes the entire table all at once without having to loop through and delete each row separately. When this optimization is employed, no memory is used to hold `ROWID`s. Thus the following statement

```
DELETE FROM ex335;
```

is much faster and uses less memory than this statement:

```
DELETE FROM ex335 WHERE 1;
```

Another memory usage in SQLite that is worth mentioning is the page bitmask used by the transaction logic in the Pager. Whenever a new transaction is started, the Pager has to allocate a chunk of memory that contains two bits for every page in the entire database file. On embedded devices, this bitmask is not normally a big deal. A 10 MB database file with 4 KB pages only requires 640 bytes for the bitmask. The transaction bitmask usually only becomes a factor on workstations with databases that are tens or hundreds of gigabytes in size.

8.5 Summary

Having read this chapter, you have a sound basis for making good design decisions and optimizing your application. We have covered a lot of ground (performance and optimization principles, application and system tuning) and delved into intricate details of SQLite. Don't forget to check back when you are writing code or performing a code review – it is then that you'll find this chapter most useful.

9
Using Symbian SQL: Three Case Studies

This chapter presents three case studies that describe the creation of applications that use Symbian SQL. The first case study describes changes to the phonebook application engine within the Symbian platform, known as the contacts model, to migrate from DBMS to Symbian SQL. The contacts model was the first application to use the new database and it enables the most widely used and basic functionality on the phone, managing users' contacts and phone numbers.

The second case study describes the Columbo search service, which provides generic searching facilities across the many types of user generated and managed data on a mobile device. Columbo uses an innovative approach to indexing with a simple but efficient database to store metadata, enabling 'live' incremental searching of user data similar to that found on desktop operating systems.

Finally, there is a discussion of the Symbian Wikipedia application, which was written as a demonstration of the capabilities of Symbian SQL. It uses an SQL database that caches the title and abstract of more than two million Wikipedia entries. Its good performance, through database optimization, provides the phone user with the capability to search the whole of Wikipedia on their Symbian device.

9.1 Contacts Model

The Symbian contacts model is an application engine upon which phonebook and contacts applications can be built. The contacts model provides APIs to enable application developers to carry out various phonebook use cases such as creating and storing contact items (with names, phone numbers, addresses, etc.), retrieving contact items, creating and filtering lists of contacts for display, matching phone numbers and so on. The contacts database class (`CContactDatabase`) is an abstraction that simplifies storing and retrieving contact items to a persistent database store. The contacts model follows the classic Symbian client–server design pattern. One or more clients may be served by the contacts server which, in turn, is a client of a database server.

The contacts model is a long-established component that has its roots in the contacts application on the Psion Series 5. In its original incarnation, it provided fairly basic phonebook operations. Over time, functionality has been added to make it more feature-rich in order to support the requirements and expectations of a mobile device aimed at the enterprise market. This includes phone number lookup, support for groups (such as home, work, friends, etc.), the ability to filter and sort contacts on different criteria, speed-dial functionality, multiple contact database file support, and the importing and exporting of contact data in vCard format for synchronization with other devices.

Up to and including Symbian^1, the contacts model used the Symbian database engine, DBMS. In Symbian^2, the contacts model moved to using Symbian SQL. This case study briefly describes the experiences of the development team that performed the migration.

9.1.1 Contacts Model on DBMS

The contacts model originally used DBMS to persist contact information. As new functionality and features were added, the database schema was modified and augmented. There was a strong imperative to maintain data and file format compatibility as far as possible in order that a contacts database file could be moved from one device to another and still work, even if the new device supported more features. As a result, there was a tendency to add to the database schema, leaving the existing structure in place, rather than modify the schema and break compatibility.

It is for this reason, and others relating to specific characteristics of the DBMS database engine, that the schema diagram looks as it does (Figure 9.1). The main table is the Contacts table which contains the bulk of the contacts information. Contact items are stored here in a binary format, supported by some columns containing metadata to aid searching. Similarly, the Identity table is there for improved lookup, containing data that is useful for searching, sorting and displaying such as first, last and company name. Support for contacts groups (e.g. 'work' or 'personal') is provided in two tables as an optimization for fast retrieval in DBMS, owing to issues with indexing and file size. The Phone and Email tables, through a certain amount of redundancy, provide fast phone number matching of incoming calls (through storing a pre-processed hash of the number) and email lookup for messaging applications. Finally, the Preferences table stores database metadata.

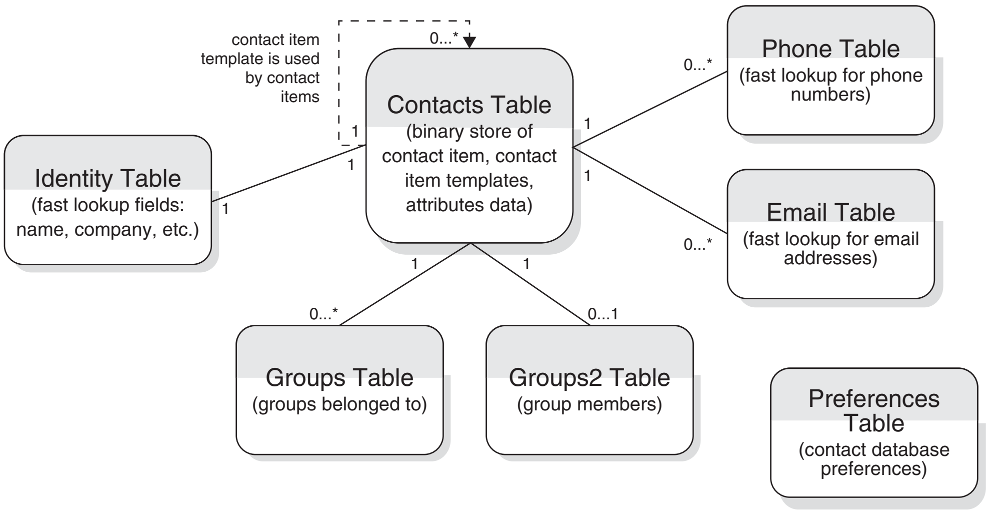

contact item template is used by contact items
0...*
1
1
Contacts Table
(binary store of contact item, contact item templates, attributes data)
Identity Table
(fast lookup fields: name, company, etc.)
1
Phone Table
(fast lookup for phone numbers)
0...*
1
1
Email Table
(fast lookup for email addresses)
0...*
1
1
0...*
0...1
Groups Table
(groups belonged to)
Groups2 Table
(group members)
Preferences Table
(contact database preferences)

Figure 9.1 Contacts model database schema in DBMS

9.1.2 Aims of Migration

The primary goals of migrating the contacts model to Symbian SQL were scalability and memory usage. There is increasing demand for larger and richer contacts data sets that need to be stored and accessed on a user's device. For example, many companies now want the ability to provision devices with the entire corporate directory before distributing them to employees. This requires a database solution that has predictable memory usage and efficient file management to support such scaling.

The constraints placed on the migration project related to maintaining source and binary compatibility, performance, and the existing functionality. Some of these issues were eased by the layered architecture in the contacts model. A separate persistence layer provides the server with an abstraction of the persistent data store. Most of the change to the code, therefore, was isolated to this layer and took the form of swapping out the use of one database engine for another.

There were also areas in which the DBMS schema could be improved that arose from historical issues and strategies to maintain data compatibility. The ways of accessing and modifying data in DBMS are quite different from most SQL relational database engines. All interaction with DBMS has to be programmatic, through the emulator or on a hardware reference device. With other database systems, one is usually able to analyze database files on a PC using some form of database administration tool. However, testing even simple things with DBMS generally means writing some code. Also, the management of the database file itself is more difficult. Conventional database engines take care of a lot of the file management. In DBMS, the contacts model had to take care of file corruption problems, had more work to do for transactions, locking, rollback and recovery, and had to open and close tables for all clients.

9.1.3 Contacts Model on Symbian SQL

There were four main benefits that came from the migration of the contacts model to Symbian SQL:

- scalability through a more flexible and extendable schema
- improved performance in some key use cases
- lower RAM usage in certain use cases
- improved ease of development and maintenance through a standard SQL database API.

Scalability

While certain aspects of the old schema were retained, the schema was refactored to make it more flexible and maintainable (see Figure 9.2). In moving from DBMS to Symbian SQL, data compatibility was broken and this gave an opportunity to improve the schema, removing some of the earlier peculiarities (such as separate Contacts and Identity tables).

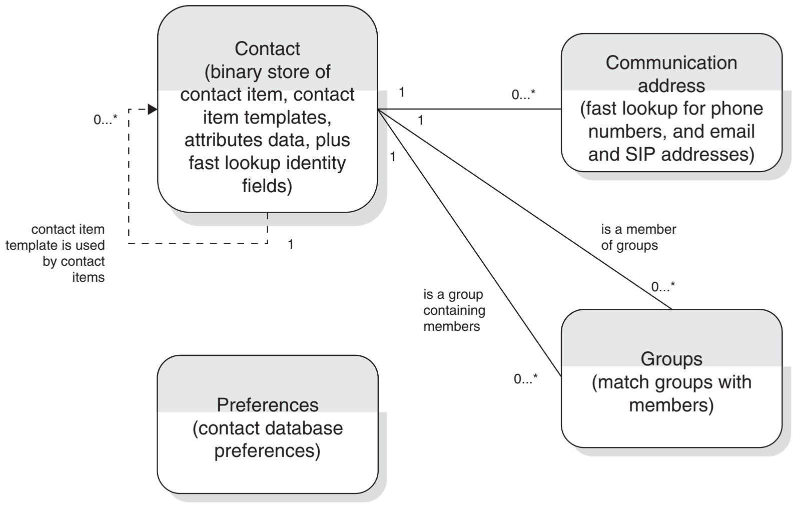

Contact (binary store of contact item, contact item templates, attributes data, plus fast lookup identity fields)
Communication address (fast lookup for phone numbers, and email and SIP addresses)
Groups (match groups with members)
Preferences (contact database preferences)
contact item template is used by contact items
is a member of groups
is a group containing members
1
0...*
1
1
1
0...*
0...*
0...*

Figure 9.2 Contacts model database schema in Symbian SQL

The underlying SQLite database's better support for indexing and faster read times enabled improvements such as merging the two group tables into a single one, with a `<group, group-member>` mapping, capable of doing the lookup in both directions. The Phone and Email tables were also combined into a single Communication Address table which also has support for SIP addresses and can hold any other type of communication address. As a result of the move to Symbian SQL, it will be easier in the future to modify and extend the schema to support new requirements.

Performance

The improvements in performance were seen particularly in reading from the database. This improved use cases such as selecting individual contact items (which is important, for example, when displaying the name of the caller for an incoming phone call).

Performance in deleting records saw an improvement. The team also implemented optimization suggestions from the documentation, such as appropriate use of indexes and improving the ordering of database columns (for example, putting BLOBs at the end of the table).

RAM Usage

Symbian SQL uses disk space efficiently and its journaling approach to file management provides robustness and reliability, helping to underpin scaling of data size. The RAM usage improvements were seen with contact views, sorted or filtered lists of one or more fields from a set of contact items. When a user goes to the phonebook, the sorted list of `<first name, last name>` labels are usually generated using a contact view.

The contacts model allows phone manufacturers to supply a plug-in to carry out customized sorting since it may be desirable to sort the list of contacts in a non-standard, non-alphabetical order, perhaps handling empty fields in a special way. As a result, this sorting has to be done outside the usual ORDER BY clause. The faster read times in SQLite made it possible to change the way in which views are created such that the operation now has significantly reduced RAM usage. In turn, this has improved scalability as it is possible to create larger views than before. During development, the contacts model was tested with views of 10,000 contacts.

Ease of Development and Maintenance

An additional benefit is the improved development experience. The Symbian SQL APIs are well documented with a lot of code examples to demonstrate best practice and optimal usage. It is also easier to work with the Symbian SQL database as it is possible to use existing knowledge of standard SQL and to use tools for the desktop environment to aid development. The team were able to design and test schemata with freely available tools on a PC, running queries and analyzing data in a much easier fashion than on a phone. This separation of the database design from the development of the business logic on the phone platform simplified things greatly. Then, when integrating the database schema with the persistence layer code, it was simply a matter of implementing the data definition statements and the relevant create, read, update, and delete (CRUD) queries using the familiar SQL API.

As well as an improved development experience, perhaps more importantly, we expect to see similar benefits in maintenance of the code and enhancement of its features in the future.

9.1.4 Possible Future Improvements

Probably the most obvious improvement to the contacts model's use of Symbian SQL would be to redesign the schema. Elements of the schema on DBMS were brought through to the Symbian SQL version, such as the use of large BLOB and CLOB fields to store much of the contact item data, owing to constraints on the migration project.

More of relational database functionality could be used by changing to a redesigned, more normalized schema. Redesigning the schema would push some of the work of searching, retrieving and manipulating the data stored in the contact items into the database engine, something for which it is optimized. This would simplify the contacts model code by removing some of the internal business logic code.

In addition, improvements could be made in the contacts model to make better use of new features and functionality and improved performance as they are added to Symbian SQL. This could perhaps be said of any software but, given the wide industry interest in SQLite and that it is open source, there is likely to be a great deal of development in the future.

9.1.5 Case Study Summary

The migration from DBMS to Symbian SQL was a positive experience. The overall performance improvement, although not dramatic, improved the functionality of the contacts model. For engineers working on the project, it was more enjoyable and productive to be able to leverage existing database knowledge when writing the new code and it was satisfying to be able to simplify the contacts model code on the database end. It was also a great benefit being able to use freely-available development tools from the Web and the online documentation from the SQLite community.

9.2 Columbo Search Service

In the world of device convergence, it is no longer unreasonable to anticipate comparable amounts of personal and professional data on a mobile device to those on a desktop or laptop. The convenience of synchronizing calendar events, contacts, tasks and other such daily changing information provides a substantial catalog of data. A user may then use this same device to take photos, to browse and bookmark the Web, to receive email and to store a year's worth of SMS messages. A considerable amount of information sits in a user's pocket, most of the time away from its desktop archive.

It is simple to locate an individual item on a desktop PC. The large screen allows a highly detailed month view in a calendar application or tens of contact items in an address book window. However, is it practical to try to emulate this desktop-style approach on a device with a fraction of the screen size?

Columbo presents an alternative to application developers. Content can be searched incrementally, efficiently and quickly in the form of a generic document representation. Its central search index enables not only the system-wide search facility seen in Apple Spotlight and Microsoft Windows Vista but also local in-application information retrieval, with the aim of quickly retrieving a user's data without complicated window furnishing arrangements.

The service is based on Symbian SQL; it makes extensive use of the technology, most notably for persisting and indexing the abstracted document content representation. Here we look at the application of the technology to a problem full of complications, highlighting how the SQL technology takes a good number of them away.

9.2.1 Why Symbian SQL?

There are a good number of persistence options available to application developers on the Symbian platform. So, what made Symbian SQL the best choice for this particular project? Alternatives, such as Symbian DBMS or any of the file store APIs from the STORE library, are mature and well-proven solutions; what makes Symbian SQL attractive to new projects such as this?

Reuse of Developer Skills

SQL language compliance significantly reduces the learning curve for developers. For this project, it meant the difference between being able to start work immediately and having to research a good deal of API documentation first. A basic understanding of SQL should stimulate immediate interest in this technology for those seeking a persistence solution for a Symbian application.

Agile-Friendly API

Throughout the Columbo project, iterative development strategies were employed and Symbian SQL's ability to provide base functionality with little initial effort makes for an excellent incremental development and testing approach. Subsequently, addition of more sophisticated persistence features (such as streaming) can be carried when a degree of confidence and maturity is established in the already tested application code.

Highly Competitive Performance

Aside from the widely appreciated SQLite performance characteristics underlying the Symbian API, there is the ability to trivially and intuitively attach indexes to your database schema. Support for the familiar `CREATE INDEX` commands gives the engineer the required toolset for optimizing the schema design without code changes.

In Columbo, performance demands for user queries are met by use of the SQLite indexing facilities and required no real developer effort, aside from thinking through the schema design.

Portability

APIs for SQLite databases exist for many platforms, scripting languages and database management tools. This makes prototyping and testing as flexible and intuitive as it can be in many respects. Much of the prototyping and automated generation of test data for this project was done using Python. The creation of a simple script to populate a database file, which could then be transferred to a handset for testing, greatly simplified the performance testing aspect of this project.

Reliability and Fault-Tolerance

SQLite supports transaction management with standard `SQL BEGIN`, `COMMIT` and `ROLLBACK` commands. When working alongside the platform's idiomatic error-handling mechanism of leaves and traps, this provides the developer with the tools needed to create a truly robust product. Data integrity can be assured with a good degree of simplicity with this database solution. For an always-on service such as Columbo, data integrity is a priority and Symbian SQL takes a lot of effort away from the developer in providing it.

9.2.2 Developing with SQLite

To provide context, let us start by summarizing the requirements of this search service:

- persist and maintain a document content reference for any data type
- provide keyword matching and incremental metadata searching
- present results in a useful manner, so that the original document can easily be retrieved
- protect user data (the search index must never leak confidential information).

To meet such demands, Columbo provides an API to applications that enables the registering of new generic documents in the service index and searching through an interface. Figure 9.3 shows the overall system architecture, demonstrating that the service is provided as a Symbian server with a client-side service API in the form of a DLL.

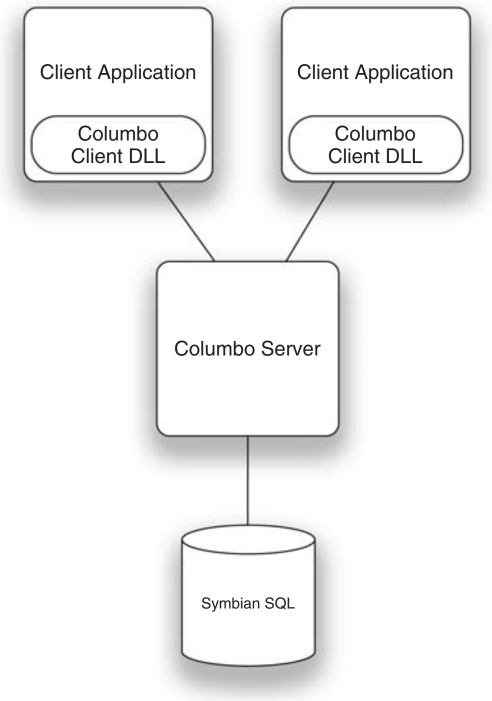


Figure 9.3 Columbo system architecture

The Columbo Server makes use of the Symbian client–server framework. It acts as a business logic tier between the client application and the SQL database server, queuing and processing client requests, scheduling them appropriately and presenting results data to the client applications. A useful approach taken to separate the server implementation details from the persistence mechanism is shown in Figure 9.4. Componentizing database interaction and defining the requirements of this component with a mixin class is an effective way of isolating external dependencies. In this case, the dependency on the SQL server has been broken out into a particular implementation of the `MColumboDatabase` interface.

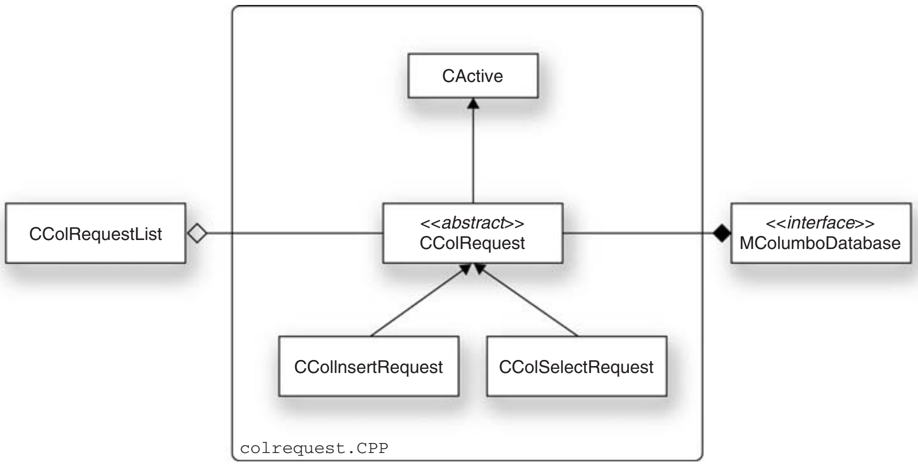


Figure 9.4 Separating the server implementation details from the persistence mechanism

Should it be necessary to distribute the service to phones without the SQLite component, it would be trivial to provide an alternative implementation of this interface; perhaps `CColDbmsDatabase`. To maximize handset compatibility, it may even prove sensible to package this implementation selection in an ECOM plug-in, thus allowing dynamic loading of the best available persistence service for the particular variant installed (see Figure 9.5).

A simplified version of the database schema is illustrated by Figure 9.6. It provides a very simple data model to represent something quite obvious: 'a document contains many words.' The actual implementation is essentially a number of complications built around this trivial concept, but this information is enough to show a clear example without being overwhelmed by dull caveats from the field of metadata searching.

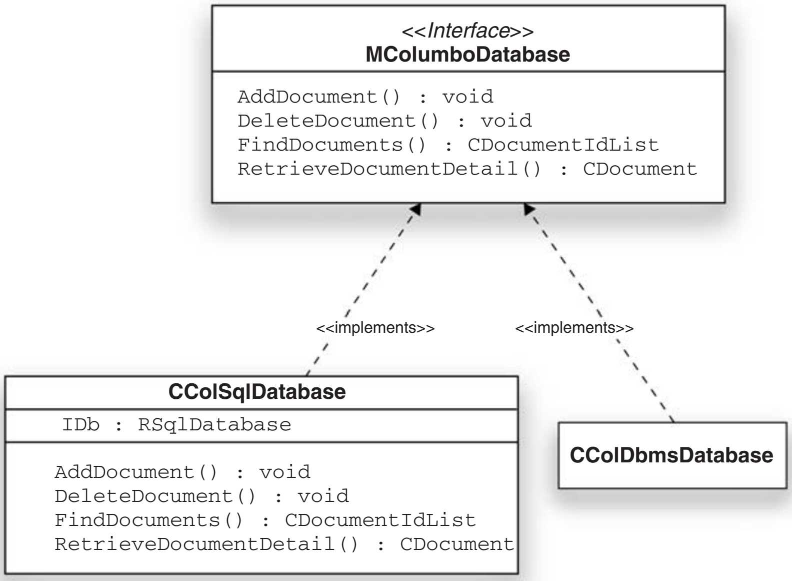


Figure 9.5 Using the best persistence service for the installed variant

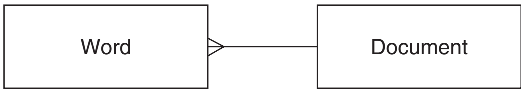


Figure 9.6 Simplified version of the database schema

Translating a schema like this to a SQLite implementation is simple:

```
void CColSqlDatabase::SetupDatabaseL()
  {
  // Word entity represents tokens in documents
  _LIT(KCreateWordTblSql, "\
    CREATE TABLE word\
      (\
      token INTEGER NOT NULL,\
      docId INTEGER NOT NULL,\
      tfQuot REAL NOT NULL\
      );\
    ");

  // Document entity represents an abstract document
  _LIT(KCreateDocTblSql, "\
    CREATE TABLE doc\
      (\
      docId INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,\
      uri BLOB NOT NULL,\
      mime BLOB NOT NULL,\
      userKey BLOB,\
      description BLOB\
      );\
    ");

  // Creating an index on the tokens
  _LIT(KCreateWordIdxSql, "\
    CREATE INDEX word_token_idx\
      ON word (token);\
    ");
  // Execute statements
  User::LeaveIfError(iDb.Exec(KCreateWordTblSql));
  User::LeaveIfError(iDb.Exec(KCreateDocTblSql));
  User::LeaveIfError(iDb.Exec(KCreateWordIdxSql));
  }
```

The code fragment above also includes CREATE INDEX statements. The `word` table is a list of numeric hashes for the tokens describing the document content, but the service is entirely dependent on the indexing mechanisms of the underlying database to guarantee performance. Most performance issues identified at later stages of development are likely to be solvable by altering this constant alone.

Retrieving information from the database is also intuitive. The iterator pattern usage in the `RSqlStatement` class makes for simple and readable code in this area. The following code snippet shows this API being used to retrieve potential candidate matches for a query in the Columbo search service:

```
void CColSqlDatabase::DoGetDocumentsL(CQueryDetails& aQuery,
                                      CResults& aResults)
  {
  // SQL Query constants
  _LIT(KMatchedDocSql, "\
      SELECT docId, tfQuot \
      FROM word \
      WHERE token = :token\
    ");
  _LIT(KToken, ":token");
  _LIT(KDocIdColumn, "docId");
  _LIT(KTfQuotColumn, "tfQuot");

  // Open an SQL statement handle
  RSqlStatement selectStmnt;
  CleanupClosePushL(selectStmnt);

  // Preparing the statement to use the above SQL,
  // Also the offsets for each column parameter must be retrieved.
  // (iDb is already open and owned by the object instance)
  selectStmnt.PrepareL(iDb, KMatchedDocSql);
  TInt tokenIdx = selectStmnt.ParameterIndex(KToken);
  TInt docIdColumn = selectStmnt.ColumnIndex(KDocIdColumn);
  TInt tfQuotColumn = selectStmnt.ColumnIndex(KTfQuotColumn);

  // Iterating through each token in the phrase
  // Issuing an SQL query for each.
  CTermCollection& soughtTerms = aQuery.QueryPhrase();
  const TInt termCount = soughtTerms.Length();
  for (TInt currTermOffset = 0; currTermOffset < termCount;
                                          ++currTermOffset)
    {
    aResults.NextTermL();
    selectStmnt.BindInt64(tokenIdx, soughtTerms[currTermOffset].iTerm);

    while (selectStmnt.Next() == KSqlAtRow)
    {
    // We're at a row in the result set,
    // read the data into variables
    TInt documentId = selectStmnt.ColumnInt(docIdColumn);
    TReal tfQuotient = selectStmnt.ColumnReal(tfQuotColumn);

    // Calculate the score and record the result
    TReal score = ScoreL(tfQuotient);
    aResults.AppendMatchL(currMatch.iDocument,
                          soughtTerms[currTermOffset].iTerm, score);
    }

    // Reset the statement before moving on to the next token.
    selectStmnt.Reset();
    }

  CleanupStack::PopAndDestroy(&selectStmnt);
  }
```

The call to `Next()` is synchronous. When designing an application, it is worth considering whether a significant number of results are to be returned in a query. If so, one should assess what kind of processing is to be done with these results. Columbo simply formats the results to be returned to the client process for further evaluation, meaning the expense per row is relatively small in comparison to the overall request-handling operation.

If, however, extensive calculation is to be performed on each returned row, some planning needs to be done about how to break this operation up with an active object state machine. Yielding control after *x* results have been processed, hoping to resume later, involves holding this `RSqlStatment` handle open. This keeps a read lock on the database file, meaning no write operations may obtain a lock until the handle is closed. Similarly, long write operations block all other operations until their completion. If designing a request queue for database operations based on active objects, the following should be taken into account:

- After opening a statement handle, a write operation locks the database file until it is closed.
- A read operation prevents only write operations until the handle is closed.

This simply means that more care has to be taken when breaking up lengthy operations. For standard database operations, code is straightforward and this complication can largely be ignored.

9.2.3 Case Study Summary

The development of this search service has been simplified greatly by the use of Symbian SQL.

To demonstrate a reasonable approximation of the relative project-wide effort spent in persistence operations for this project, Figure 9.7 represents the number of lines of code interacting with the database with respect to that of other service components.

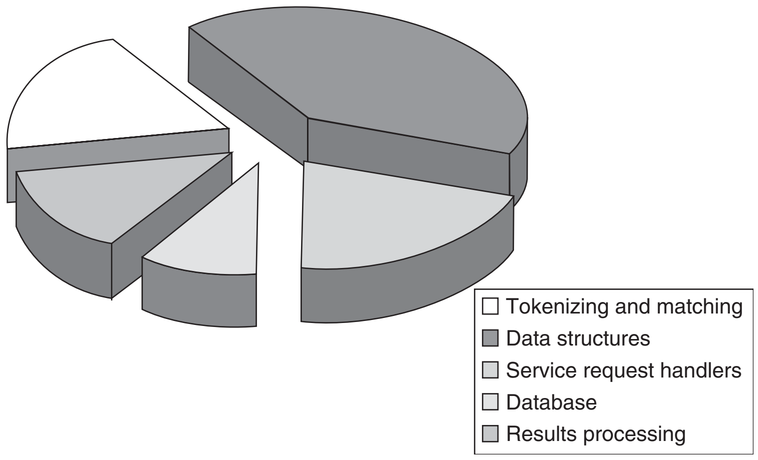


Figure 9.7 Lines of code in the service components

For what is essentially the heart of the service, 400 lines of code in a project with a total of around 7000 lines of code seems somewhat surprising. This demonstrates clearly that the database substantially reduces programming effort, making it a highly attractive persistence solution for a wide variety of data services problems.

9.3 Wikipedia Demo Application

This case study demonstrates the scalability of Symbian SQL by discussing how an application can be designed and implemented to efficiently search a database containing a very large number of records. The aim of the case study was to develop an application on the Symbian platform to allow the user to search and browse 2.2 million Wikipedia article abstracts.

Wikipedia (***www.wikipedia.org***) is a free, online encyclopedia project supported by the non-profit Wikimedia Foundation. Wikipedia works on a collaborative basis; it has over 3 million articles written in English,[1] created by volunteers all over the world. Wikipedia was formally launched on January 15, 2001; with over 253 active language editions, it has grown to be a global phenomenon.

9.3.1 Downloading the Symbian Wikipedia Demo Application

The example application and database can be downloaded from the wiki page for this book at ***developer.symbian.org/wiki/index.php/Inside_Symbian_SQL***. The source code is also available on that page, along with a SIS file for immediate installation and use.[2]

9.3.2 Storing Wikipedia on a Phone

Since Wikipedia is an online resource, it is possible to use it from a Symbian device over an Internet connection. But what if the Internet connection isn't available or the data rates are prohibitively expensive? In these cases, it would be useful to have a local copy of Wikipedia on the phone.

It is possible to download the entire Wikipedia database but how would you interface to it in a performant way? And, since it is so big, how could it fit on a phone where the usual size of cards or internal memory is typically (at the time the application was developed) only up to 8 GB?

Due to the current restraints on flash memory, we decided to condense the Wikipedia content. Instead of the full version, the application uses only the article abstracts, along with a small portion of the beginning of some articles and the links within them. The content was downloaded from the Wikipedia website in the form of two XML files, one for the abstracts and one for the article contents. Once the schema of the

[1] Correct in October 2009 and taken from ***en.wikipedia.org/wiki/Special:Statistics***.
[2] Available for S60 3rd Edition FP1 (Symbian OS v9.2) at time of writing.

database was decided upon, a Java application used the contents of these XML files to populate the database.

A simple application interface was all that was needed for this use case. As the user begins to type their criteria into the search box, the list of matches is constantly updated (known as a 'prefix-search' approach). Once the user can see the title of the article they are searching for, they can then simply select that article to view it.

9.3.3 Designing the Schema (the Wrong Way)

The main thing to consider when designing the application was to make sure that even though the database contains such a large number of records, it must still be responsive and give instantly updated results every time the user presses a key, otherwise it would be unusable and the interface would have to change to a less desirable one.

At the beginning of the database design phase, we assumed that a simple approach would not be sufficient to meet the demands of the application, due to the large number of records in the database.

The initial approach was to split the content of the database into separate tables depending on the prefix of the article title. The database schema would then depend on how big a prefix was used. For instance, if using a two-letter prefix, then all articles beginning 'AA' would be stored in one table, all articles beginning with 'AB' would be stored in another table, and so on. The idea was that storing the data in different tables would make the lookup quicker, as the tables would be smaller. However, looking at the content of the database, a two-letter prefix would give the following characteristics:

- 3212 distinct prefixes (i.e. tables)
- an average of 671 entries per prefix (i.e. records per table)
- prefix 'MA' has 58,734 entries (i.e. records in the largest table).

So although the table size has been reduced, there are still tables with tens of thousands of records. At this point, we decided to investigate increasing the size of the prefix to three, four or five characters. The five-character prefix would have the following characteristics:

- 295,675 distinct prefixes (i.e. tables)
- an average of seven entries per prefix (i.e. records per table)
- prefix 'JOHN' has 17,880 entries (i.e. records in the largest table).

So there are a huge number of individual tables and still tens of thousands of records in some of them. The approach was becoming unreasonable, especially due to the fact that to actually split the data into this format would have required a complex script to deal with the large overheads of these different prefixes, tables and indexes. Also the application design to generate the SQL would be more complex.

9.3.4 Designing the Schema (the Right Way)

It was quickly becoming clear to us that the initial approach had become far too complex, with no benefit for it. We then decided to try a simpler approach:

```
CREATE TABLE pages (
  id INTEGER PRIMARY KEY,
  title NVARCHAR COLLATE nocase,
  content BLOB,
  links BLOB
);
CREATE INDEX table_index ON pages (title);
```

This schema is very simple, and means all of the data can be stored in a single table. This means that generating the database is significantly more straightforward than the previous approaches. The final database that was created had the following characteristics:

- 2.15 million articles
- 843 MB content (an average of 391 bytes per article)
- 40 MB of titles (an average of 18 bytes per title).

The articles ranged from ‘A: First letter of the Latin Alphabet’ to ‘ZZZZZ: An episode of *The Outer Limits* aired 1964’, which demonstrates the diversity of the content!

Another advantage of this approach is the ease with which the application can perform searches on the database. For instance, if the user types in ‘john’, then the following query is all that is needed to return, in order, all of the records that start with ‘john’:

```
SELECT title FROM pages WHERE title LIKE 'john%' ORDER BY title
```

Even though the data is arranged in a single table, a simple schema and a simple query performed on it, the performance was still extremely fast, and more than sufficient for the demands of the application.

9.3.5 Displaying the List of Articles

The query above returns many thousands of records. Since the screen on the phone is small and can only typically fit around six article titles on it at a time, how can this be displayed?

The approach we took was to split the article titles into pages, each containing six items. When the user types the search criteria, the display shows the first page, containing the first six items. The user can move to the next page to display the next six items and so on. From there, the user can move to the previous page, as shown in Figure 9.8.

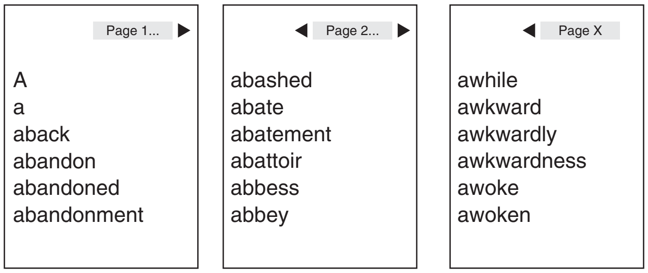


Figure 9.8 Wikipedia application interface

To move down the list of records is straightforward. One can simply call `RSqlStatement::Next()` six times to populate the list with the next six items, without performing another query on the database. But what if the user wants to go back to a previous page?

For instance, if the user is looking at page 3, then `Next()` has been called three times and the statement points to the 18th row (the last item of the third page of six records). To move to the previous page, the statement needs to point to the 7th item (the first item of the second page of six records). There is no `RSqlStatement::Previous()` function, as statements only work one way, so how can this be done?

This is where SQL's `LIMIT` and `OFFSET` features can be used:

```
SELECT title FROM pages WHERE title LIKE 'john%' ORDER BY title
                                            LIMIT -1 OFFSET 6
```

`OFFSET 6` means that records are returned beginning from the 7th row rather than from the 1st row. `LIMIT -1` means that all records are returned, with no limit. Now it is simply a case of calling `Next()` six times to re-populate the list with the items on page 2.

9.3.6 Retrieving the Content of Articles

Due to the simple nature of the schema, retrieving the content when the user selects an article is trivial:

```
SELECT content, links FROM pages WHERE title = 'john'
```

It is now simply a case of the application formatting and displaying the content and links fields on the device.

9.3.7 Case Study Summary

Due to the sheer volume of the dataset to be ported to the application, it was assumed at the start of the project that a complex approach to the database organization would be necessary in order to gain the very high performance that was essential for the intended application interface. Originally, it was not certain that this interface would even be possible. In actual fact, a very simple design was all that was required. This meant that the code, database creation and interrogation were kept simple and maintainable while achieving the desired interface and high performance.

9.4 Summary

Symbian SQL makes a big difference to the Symbian Platform by providing a beautiful, standards-compliant and easy-to-use persistence mechanism. We discussed the initial motivation for Symbian SQL in Chapter 1. Some benefits were clear immediately, such as improvements to overall system performance and efficiency. However, the full consequences were impossible to predict.

A good database management system is an enabler. Things that were inconceivable have become possible and real. All it takes is some creativity and we get a content search engine running on a phone and a usable, off-line, Wikipedia knowledge base with millions of articles. The future is, clearly, very exciting indeed.

Appendix A

Troubleshooting

In this appendix, we discuss troubleshooting SQL statements and language, memory and disk usage, transactions, and error handling.

A.1 Executing SQL Statements

`RSqlStatement::Next()` returns `KSqlErrMisuse` following the preparation and execution of a SELECT statement:

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.PrepareL(iDatabase, KSelectStmt);
User::LeaveIfError(stmt.Exec());
TInt err = stmt.Next();
if (err == KSqlAtRow)
  {
  // Do something
  // ...
  }
CleanupStack::PopAndDestroy(&stmt);
```

The call to `Exec()` is unnecessary; it would only be necessary when binding values. In this case, `Next()` can be called directly after `PrepareL()` as it executes the statement itself and moves to the next row.

```
stmt.PrepareL(iDatabase, KSelectStmt);
TInt err = stmt.Next();
if (err == KSqlAtRow)
```

```
  {
  // Do something
  }
CleanupStack::PopAndDestroy(&stmt);
```

RSqlStatement::Next() should only be called after SELECT statements. It panics[1] after preparing INSERT, UPDATE or DELETE statements:[2]

```
_LIT(KInsertStmt, "INSERT INTO MyTable (id, data)
                        VALUES (1, 'Record 1')");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.PrepareL(iDatabase, KInsertStmt);
User::LeaveIfError(stmt.Next());
CleanupStack::PopAndDestroy(&stmt);
```

It is incorrect to call Next() for INSERT, UPDATE or DELETE statements as they do not return any records. Exec() should be called instead.

Panics can occur when calling certain functions, such as, RSqlStatement::Next(), RSqlStatement::AtRow(), RSqlStatement::ColumnXXX():

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.Prepare(iDatabase, KSelectStmt);

// Iterate the rows
while (stmt.Next() == KSqlAtRow)
  {
  // Do something
  }

CleanupStack::PopAndDestroy(&stmt);
```

Care must be taken when executing functions that return error codes:

```
TInt Prepare(RSqlDatabase& aDatabase, const TDesC& aSqlStmt);
```

In this case, the call to Prepare() could have returned an error. If it did, then it was not checked and handled, therefore the RSqlStatement is

[1] The panic only occurs in Symbian OS v9.4 and earlier versions.

[2] The panic does not occur in Symbian OS v9.5 and later; instead, Next() completes and executes the statement correctly. This is bad practice though and should not be used. In particular, the return value is not applicable to the intended action.

invalid. This would cause the call to `Next()` to panic. Always take care to check for errors when functions return them:

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
TInt err = stmt.Prepare(iDatabase, KSelectStmt);
if (err == KErrNone)
  {
  // Iterate the rows
  TInt rowErr = KErrNone;
  while ((rowErr = stmt.Next()) == KSqlAtRow)
    {
    // Do something
    }
  if (rowErr != KSqlAtEnd)
    {
    // There has been an error so handle it
    }
  }
else
  {
  // Handle the error
  }
CleanupStack::PopAndDestroy(&stmt);
```

Note that, upon completion of iterating through the records, the return value is checked to be `KSqlAtEnd`. It is good practice to check this; an error may have occurred and should be handled.

Where applicable, one can also use the leaving version, i.e. `PrepareL()`:

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.PrepareL(iDatabase, KSelectStmt);
// Iterate the rows
CleanupStack::PopAndDestroy(&stmt);
```

An infinite loop can occur when trying to return records with `RSqlStatement::Next()`:

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.PrepareL(iDatabase, KSelectStmt);
```

```
while (stmt.Next())
  {
  // Do something
  }
CleanupStack::PopAndDestroy(&stmt);
```

The code assumes that `Next()` returns an error (i.e. a negative return value), which it does not necessarily do. Unless there are any errors, `Next()` returns `KSqlAtRow` (which has a value of 1), until it can read no more rows, in which case it returns `KSqlAtEnd` (which has a value of 2). In the code above, that would result in an infinite loop, as the return value is always positive. Always make sure to check that the statement is pointing to a valid row:

```
_LIT(KSelectStmt, "SELECT * FROM MyTable");
RSqlStatement stmt;
CleanupClosePushL(stmt);
// RSqlDatabase iDatabase already connected
stmt.PrepareL(iDatabase, KSelectStmt);
while (stmt.Next() == KSqlAtRow)
  {
  // Do something
  }
CleanupStack::PopAndDestroy(&stmt);
```

What could cause `KErrPermissionDenied` to be received when executing a SQL statement on a secure shared database? One reason could be a lack of sufficient capabilities. You can check that the capabilities of the application match the security policy of the secure database using the `RSqlSecurityPolicy::Policy()` function. The object returned defines what capabilities the calling application must have in order to perform particular database operations. There are three policy types and each one refers to a set of capabilities encapsulated in the `TSecurityPolicy` object:

- `ESchemaPolicy`: The application can modify the database schema, write to the database and read from the database.
- `EReadPolicy`: The application can read from the database.
- `EWritePolicy`: The application can write to the database.

Another reason could be if you're using the following SQL statements; they are not supported on secure shared databases:

- `ATTACH`

- DETACH
- PRAGMA

Statement parameter bindings can appear to retain their values, even after the RSqlStatement is reset. Is this a defect? For example:

```
_LIT(KCreate, "CREATE TABLE MyTable (T1 TEXT, T2 TEXT)");
// RSqlDatabase iDatabase already connected
User::LeaveIfError(iDatabase.Exec(KCreate));

_LIT(KInsert, "INSERT INTO MyTable VALUES(:Prm1, :Prm2)");

RSqlStatement stmt;
CleanupClosePushL(stmt);
stmt.PrepareL(iDatabase, KInsert);

_LIT(KValue1, "Ringo");
_LIT(KValue2, "John");
User::LeaveIfError(stmt.BindText(0, KValue1));
User::LeaveIfError(stmt.BindText(1, KValue2));
User::LeaveIfError(stmt.Exec());
// Record inserted = Ringo | John
User::LeaveIfError(stmt.Reset());

_LIT(KValue3, "Paul");
User::LeaveIfError(stmt.BindText(0, KValue3));
// T2 not set
User::LeaveIfError(stmt.Exec());
// Record inserted = Paul | John

CleanupStack::PopAndDestroy(&stmt);
```

No, this is expected behavior. Any SQL statement parameters that had values bound to them retain their values, even after a call to RSqlStatement::Reset(). This function resets the prepared SQL statement to its initial state and makes it ready to be executed again.

A.2 SQL Language

A.2.1 Datatypes

SQLite is loosely typed and does not perform type checking. Care must be taken not to introduce defects into your SQL code. What happens when no error is received when inserting a value of type X into a column of type Y? For example:

```
CREATE TABLE MyTable (MyInt INTEGER);
INSERT INTO MyTable VALUES (1);
INSERT INTO MyTable VALUES ("Hello");
```

The `CREATE TABLE` SQL statement is incorrect, but it works. Why? Look at these examples:

```
CREATE TABLE MyTable (INTEGER MyInt);  // Incorrect
CREATE TABLE MyTable (MyInt INTEGER);  // Correct
```

SQLite makes assumptions when presented with unknown datatypes. When specifying a data type that contains the string ‘INT’, SQLite assumes that it is an INTEGER. So in this case, the table is created with a column called INTEGER, with data type INTEGER. Technically, it works in exactly the same way as the correct SQL, but the column name is not as expected. More details on the assumptions that SQLite makes can be found at ***www.sqlite.org.***

A.2.2 SQLite Command Line

SQLite command-line program commands (e.g., `.schema`, `.tables`, etc.) do not appear to work in Symbian SQL. The reason is that these commands are issued to the SQLite Shell Interface, and are not supported in Symbian SQL.

A.2.3 Dates

Symbian DBMS has data types that are specific to dates (e.g., `DATE`, `TIMESTAMP`) but there is no `DATE` type in Symbian SQL. How are dates handled?

SQLite has a limited number of data types:

- `NULL`
- `INTEGER`
- `REAL`
- `TEXT`
- `BLOB`

All data is stored in this way (see ***www.sqlite.org/datatype3.html***). This mean that there are a few ways to handle dates; some ideas are presented here.

Storing Dates as Strings

You can use dates in DBMS like this:

```
CREATE TABLE MyTable (RowID INTEGER, Date TIMESTAMP)
```

```
SELECT RowID FROM MyTable
WHERE Date >= #2007/01/01 00:00# AND Date <= #2007/12/31 23:59#
```

This works in SQLite if the string format is maintained manually. For example, you must always format manually to have ‘`YYYY/MM/DD HH:MM`’. If using this method, the SQL changes to:

```
CREATE TABLE MyTable (RowID INTEGER, Date TEXT)

SELECT RowID FROM MyTable
WHERE Date >= "2007/01/01 00:00" AND Date <= "2007/12/31 23:59"
```

The `TTime` class provides `FormatL()` functions that can be used to convert dates to strings:

```
void FormatL(TDes& aDes, const TDesC& aFormat) const;
void FormatL(TDes& aDes, const TDesC& aFormat,
                         const TLocale& aLocale) const;
```

To convert back again, `TTime` provides a constructor overload that takes a string as parameter:

```
TTime(const TDesC& aString);
```

Storing Dates as Long Integers

Before creating the SQL statement, convert the timestamp into a long integer (for instance, micro-seconds since date X). When reading back out with SQL, convert the long integer back into a date. So the generated SQL would look something like this:

```
CREATE TABLE MyTable (RowID INTEGER, Date INTEGER)

SELECT RowID FROM MyTable
WHERE Date >= 1221991075 AND Date <= 4210476723
```

These values 1221991075 and 4210476723 are the long integer versions of the dates specified.

The `TTime` class provides a `MicroSecondsFrom()` function that can be used to convert dates to long integers:

```
TTimeIntervalMicroSeconds MicroSecondsFrom(TTime aTime) const;
```

To convert back again, `TTime` provides a constructor overload that takes a long integer as a parameter:

```
inline TTime(const TInt64& aTime);
```

Storing Dates as Fixed-format Integers

This option lies somewhere between the other two options. It works if the format is fixed, for example to `YYYYMMDDHHMM`, and the date is stored as a long integer. For instance:

```
CREATE TABLE MyTable (RowID INTEGER, Date INTEGER)

SELECT RowID FROM MyTable
WHERE Date >= 200701010000 AND Date <= 200712312359
```

Advantages and Disadvantages of the Options

In all three options, sorting and searching works because lexical and numerical order are the same as natural date order, but they rely on the same date format being used in all cases. Each option has advantages and disadvantages, and these are detailed here.

Since there are no Symbian APIs to facilitate this conversion, one would need to create one. Such an API would be trivial to create. A simple example follows:

```
// Setup the masks for conversion
const TInt64 yearMask = 100000000;
const TInt monthMask = 1000000;
const TInt dayMask = 10000;
const TInt hoursMask = 100;
const TInt minutesMask = 1;

// Example individual components for the date
// 2007-12-31 23:59
TInt year = 2007;
TInt month = 12;
TInt day = 31;
TInt hours = 23;
TInt minutes = 59;

// Create the integer version of the date
TInt64 dateInt = yearMask * year; // 200700000000
dateInt += monthMask * month; // 200712000000
dateInt += dayMask * day; // 200712310000
dateInt += hoursMask * hours; // 200712312300
dateInt += minutesMask * minutes; // 200712312359
```

```
// Convert this back into individual date components
year = dateInt / yearMask; // 2007
dateInt -= (year * yearMask);
month = dateInt / monthMask; // 12
dateInt -= (month * monthMask);
day = dateInt / dayMask; // 31
dateInt -= (day * dayMask);
hours = dateInt / hoursMask; // 23
dateInt -= (hours * hoursMask);
minutes = dateInt / minutesMask; // 59
```

To summarize:

- Storing dates as strings is rated high for readability and usability; it is rated low for disk usage, index usage and speed.
- Storing dates as long integers is rated high for disk usage, index usage and speed; it is rated low for readability and usability.
- Storing dates as fixed-format integers is rated medium in all categories.

A.3 Memory and Disk Usage

`KSqlErrFull` may be received even though there is plenty of free disk space available. This error does not occur due to a lack of disk space; rather, it indicates that an insertion operation has failed because an `Autoincrement` column has used up all available values of `ROWID`. For disk full errors, look for `KErrDiskFull`.

Symbian SQL's server may appear to leak memory. Its heap usage appears to be too big and it grows. By default, Symbian SQL uses up to 1 MB of page cache. In some cases (such as using large BLOBS), the heap usage can reach (or exceed) the 1 MB point. This could be the reason for the apparent memory leaks.

The disk is full and when you try to delete records from the database to free up space, a disk full error is received. It is not possible to delete data from a SQLite database that is located on a full filesystem. This is because SQLite requires some temporary disk space for its rollback journal. Before making any changes to the original database file, SQLite copies the original content of the pages to be changed into the rollback journal file. Modifications are then made to the original database. Once the original file is updated, the rollback journal is deleted and the transaction commits. The rollback journal provides a means of recovering the database if a power failure or other unexpected failure occurs.

Designers of applications for embedded systems should therefore take steps to make sure that filesystem space is never completely exhausted.

When filesystem space is low, large updates and deletes should be broken up into smaller transactions to avoid creating excessively large rollback journals.

Symbian SQL provides a number of APIs to manage this problem:

```
TInt ReserveDriveSpace(TInt aSize);
void FreeReservedSpace();
TInt GetReservedAccess();
void ReleaseReserveAccess();
```

`ReserveDriveSpace()` is used to reserve some space. If the client becomes aware that space is low and needs to use the reserved space, `GetReservedAccess()` is invoked. This places the reserved space in a state where it can be used (it cannot be used until this has been called).

Once the deletions have completed, the reserved access can be relinquished by calling `ReleaseReserveAccess()` and processing can continue as usual. When the application ends, it can release the reserved space to the file system using `FreeReservedSpace()`.

Note that the `aSize` parameter of `ReserveDriveSpace()` is not currently used; this value is fixed in the current implementation (at 64 KB). In future, this parameter may be used to allow other reservation sizes to be specified.

If no reserved disk has been allocated when there is insufficient space for deletion, the database should be closed because any transaction is invalid at that point. The database could possibly be re-opened after more disk space has been obtained.

A.4 Transactions

There are two rollback scenarios in SQLite: implicit single transactions and explicit bulk transactions.

A.4.1 Implicit Single Transactions

Look at this example:

```
UPDATE table1 SET i1 = 2 WHERE i1 == 1
```

SQLite automatically rolls back the transaction if it fails to complete successfully because the presence of a journal file is detected. The rollback may be handled at different times, dependent on the nature of the failure. For instance, in the case of a power failure occurring during a database write operation, then the presence of the journal file would be detected at start up. This initiates an automatic rollback of the database to its previous state, using the data stored in the journal file. If the execution of an update fails, the rollback is handled on the next operation (e.g. a call to `Exec()` or `Close()`). Transactions are explained in more detail in Chapter 4.

A.4.2 Explicit Bulk Transactions

A sequence of single operations may be executed within a transaction within a `BEGIN...COMMIT` block. In this case, it is the responsibility of the developer to provide a rollback mechanism. For example:

```
static void DoRollback(void* aDbHandle)
  {
  // check that aDbHandle is non-NULL and cast it back to RSqlDatabase
  // call RSqlDatabase::Exec(_L("ROLLBACK") on the database
  }

RSqlDatabase db;
CleanupClosePushL(db);
User::LeaveIfError(db.Open());

// Start of bulk transaction
_LIT(KBegin, "BEGIN");
User::LeaveIfError(db.Exec(KBegin));

// Put the rollback function onto the CleanupStack,
// so it is called if a function leaves
CleanupStack::PushL(TCleanupItem(&DoRollback, &db));

_LIT(KInsert1, "INSERT INTO table1 (i1, i2) VALUES (1, 10)");
User::LeaveIfError(db.Exec(KInsert1));

_LIT(KInsert2, "INSERT INTO table1 (i1, i2) VALUES (2, 20)");
User::LeaveIfError(db.Exec(KInsert2));

_LIT(KInsert3, "INSERT INTO table1 (i1, i2) VALUES (3, 30)");
User::LeaveIfError(db.Exec(KInsert3));

// End of bulk transaction
_LIT(KCommit, "COMMIT");
User::LeaveIfError(db.Exec(KCommit));

CleanupStack::Pop();
CleanupStack::PopAndDestroy(&db);
```

In the above code, a cleanup item is pushed onto the cleanup stack. The cleanup item encapsulates the cleanup function `DoRollback()` and the object on which the function is to be performed – the database connection. The function is invoked if a leave occurs between `BEGIN` and `COMMIT`; within the function, a `ROLLBACK` operation is executed on the database connection. It rolls back each INSERT operation that was successfully executed before the leave occurred and thus the database is left in the same state that it was in before the `BEGIN...COMMIT` block was executed.

A.5 Error Handling

Is there a way to find out if the errors being returned are specific SQL errors or Symbian errors? The `SqlRetCodeClass()` function provides this information. It returns a `TSqlRetCodeClass` value as follows:

- `ESqlInformation` indicates that a return code is just for information. This category corresponds to the SQL API return codes `KSqlAtRow` and `KSqlAtEnd`.
- `ESqlDbError` indicates that a return code represents a database-specific error. This category corresponds to SQL API return codes in the range `KSqlErrGeneral` to `KSqlErrStmtExpired`.
- `ESqlOsError` indicates that a return code represents a Symbian error. This category corresponds to SQL API return codes in the range `KErrPermissionDenied` to `KErrNone`.

When executing a statement and an error of type `ESqlDbError` is received, how is it possible to get more details of what the error was? The `RSqlDatabase::LastErrorMessage()` function returns a description of the last SQL error encountered.

Appendix B

SDB Database Creation Tool

Symbian SQL uses SQLite as a database engine and therefore database files created on the emulator or device can be read and manipulated in a desktop environment using the usual SQLite tools. However, Symbian SQL server offers some additional functionality, particularly related to Symbian platform security and the `symbian_settings` table, which is not directly supported by SQLite tools.

In order to simplify the task of creating databases, Symbian offers the Symbian database (SDB) tool. This appendix provides an overview, configuration reference and some examples on how to use SDB.

Note that, at the time of writing, only SDB version 1 is publicly released. SDB version 2 will add support for creating Symbian DBMS and contacts databases as well as central repository `CommDB` files. Functionality supported by SDBv1 is carefully preserved in SDBv2 so examples in this guide can be used with both versions.

B.1 Obtaining and Installing SDB

SDB is part of the Symbian Product Development Toolkit (PDT). The toolkit contains a set of tools for Symbian device creators and developers, and can be downloaded from ***developer.symbian.org***.

B.2 Using SDB

SDB has a simple command-line interface:

```
sdb <command> [<command-specific options>]
```

SDBv1 requires a command, primarily for future proofing, which was clearly beneficial for SDBv2. SDBv1 supports only two commands: `createdb` and `help`. The `createdb` command is used as follows:

```
sdb createdb -x <XML file> -s <SQL file> -d <Existing DB file>
                                              -n <DB file name>
```

- `-x <XML file>` gives the path to the XML file.
- `-s <SQL file>` gives the path to the SQL files containing the schema and data.
- `-d <Existing DB file>` gives the full path to an existing SQLite database file.
- `-n <DB file name>` gives the name of the SQLite database file.

The XML configuration file specified via `–x <filename>` contains Symbian security and other settings to be used to create the database. This is where most of the complexity can be and removing these settings to an XML file greatly simplifies interfacing with SDB. The details of the XML configuration file are discussed in more detail in Section B.3.

We can specify one or more SQL files via `–s <filename>`. Each filename is preceded with a `-s` switch. The files are executed in the same order as specified on the command line.

SDB can either create a new database or update an existing one. This is controlled by using either `–d <filename>` to update a database or `–n <filename>` to create a database.

The following command creates a new database file, `dbname.db`, based on an XML configuration file, `security_cfg.xml`:

```
sdb createdb –x security_cfg.xml –s schema_data.sql –n dbname.db
```

Once the database is created, SDB executes the SQL statements from `schema_data.sql` sequentially on the given database file. This file would typically contain the database schema creation (DDL) statements (e.g., creating tables, indexes, and triggers) followed by DML statements inserting data (see Figure B.1).

SDB can take multiple SQL files on the command line. One example where this could be useful is for separation of the schema from the data:

```
sdb createdb –x security_cfg.xml –s schema.sql –s data.sql
                                                  –n dbname.db
```

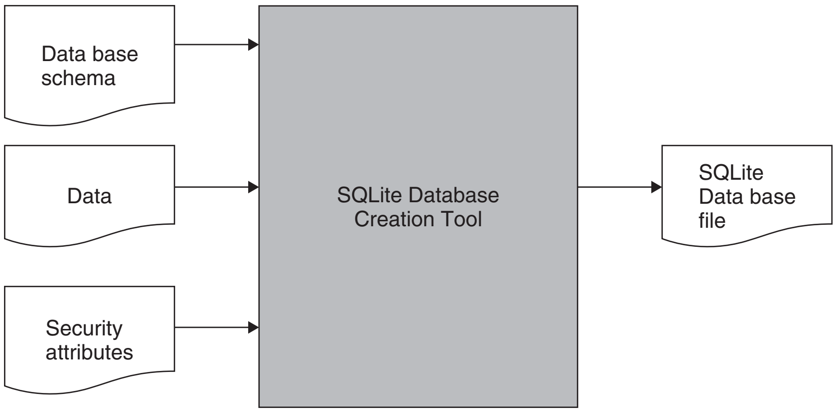


Figure B.1 SDB input and output files

This is a typical use case for Symbian; some functionality is customized by Symbian, some by device manufacturers, and some by mobile network operators.

In some cases, the schema may be split into multiple files. For example, if a device manufacturer wishes to add functionality to their application they might add a table which would be specified in its own SQL file.

B.3 SDBv1 Configuration Files

B.3.1 The Properties File

The `sqlite.properties` file is created at installation time in the `config` folder. This file is internal but it does support some settings that may come in useful when integrating SDB into other toolsets. For example, an invoking tool may customize logging settings, the default database file name, and so on. Since the settings are fairly straightforward, we show an example file and only briefly discuss interesting settings:

```
# Log File Settings
sqlitedb.log.file.enabled = true
sqlitedb.log.file.format  = %-5p [%-20.20C{1}] %-8r %4L - %m%n
sqlitedb.log.file.path    = logs\\sqlitedb.log
sqlitedb.log.file.level   = DEBUG

# Console Log Settings
sqlitedb.log.console.format = %-5p- %m%n
sqlitedb.log.console.level  = INFO
```

```
#Default Database Settings
sqlitedb.dbname = sqlitedb.db

#JDBC dll location
org.sqlite.lib.path = d:\\Symbian\\SITK\\sqlite-db-creator\\lib\\

sqlitedb.schema.location =
             d:\\Symbian\\SITK\\sqlite-db-creator\\config\\sqlitedb.xsd
```

The first two sections are essentially log4j settings. More information about log4j can be found on its website, ***logging.apache.org***.

The next section sets the default output database file name which is used if neither `-d` nor `-n` options are specified.

When called from a location other than the SDB installation directory, SDB must be able to find the SQLite driver native library. This is specified in the `org.sqlite.lib.path` property.

The final line specifies the location of the XSD schema for validation of the XML configuration file.

B.3.2 Security Settings

Security settings are specified in the XML file. Security settings are essentially a list of policies with associated capabilities. Policies are used to define the applications that can access, read, and alter particular tables or the database as a whole. Table B.1 lists the four policies that use the `type` attribute.

Table B.1 Security policies

Value	Description	Usage
Default	This policy defines the default database access policy. If an operation is attempted for which no policy has been specified, the default policy is used.	`<policy type="default">`
Schema	This policy defines applications that can update the database schema.	`<policy type="schema">`
Read	This policy defines applications that can read from the database.	`<policy type="read">`
Write	This policy defines applications that can write to the database.	`<policy type="write">`

The type of security policy is defined by options, as described in Table B.2 (see `TSecurityPolicy` in the Symbian OS documentation for more details).

Table B.2 Security policy options

Option	Description
`always="fail"`	Operations with this policy always fail the security check.
`always="pass"`	Operations with this policy always pass the security check.
`capability`	Up to seven capabilities can be specified as a `type`.
`SID="<secureid>"`	Only operations from an application with the specified SID are accepted; up to three capabilities can also be specified.
`VID="<vendorid>"`	Only operations from an application with the specified VID are accepted; up to three capabilities can also be specified.

An application that wants to carry out a specific operation must have the required capability, VID or SID.

The following XML file shows how to set the policy type and capability:

```
<!-- Default policy that always fails  -->
<policy type="default" always="fail"/>

<!-- Only the application with SID 101010fe
     and the TrustedUI capability can modify the schema -->
<policy SID="10101078" type="schema">
  <capability type="TrustedUI"/>
</policy>

<!-- Any application from vendor 101010ff can write,
     if it has WriteUserData capability -->
<policy VID="101010ff" type="write">
  <capability type="WriteUserData"/>
</policy>

<!-- Any application can read if it has ReadUserData -->
<policy type="read">
  <capability type="ReadUserData"/>
</policy>
```

More information on the Symbian SQL security model can be found in Chapter 6. The Symbian capabilities that can be specified in the XML file are listed at ***developer.symbian.org/wiki/index.php/Capabilities_(Symbian_Signed)***.

B.3.3 Database Page Size

The database page size is specified in bytes. The syntax for this configuration setting is:

```
<configuration name="page_size" value="4096"/>
```

The `page_size` is only relevant when the database file is created. The page size must be a power of two greater than or equal to 512 and less than or equal to 4096. The default value is 1024. The importance of page size has been discussed in various chapters in this book. This value is typically supplied by the device manufacturer but it may be tweaked for particular applications.

B.3.4 Text Encoding

Text encodings supported by Symbian SQL are `UTF_8` and `UTF_16`. The default value is `UTF_16`. The syntax for setting this configuration is:

```
<configuration name="encoding" value="UTF_16"/>
```

or

```
<configuration name="encoding" value="UTF_8"/>
```

Appendix C

Symbian SQL Error Codes

Identifier	Value	Description
`KSqlErrGeneral`	−311	SQL error or missing database
`KSqlErrInternal`	−312	Internal logic error
`KSqlErrPermission`	−313	Access permission denied
`KSqlErrAbort`	−314	Callback routine requested an abort
`KSqlErrBusy`	−315	Database file is locked
`KSqlErrLocked`	−316	Table in a database is locked
`KSqlErrNoMem`	−317	Out of memory (not used in the current version of Symbian SQL)
`KSqlErrReadOnly`	−318	Attempt to write to a read-only database
`KSqlErrInterrupt`	−319	Operation terminated by `sqlite3_interrupt`
`KSqlErrIO`	−320	I/O error
`KSqlErrCorrupt`	−321	Database file image invalid
`KSqlErrNotFound`	−322	Table or record not found
`KSqlErrFull`	−323	Database is full
`KSqlErrCantOpen`	−324	Unable to open database file
`KSqlErrProtocol`	−325	Database lock protocol error
`KSqlErrEmpty`	−326	Database is empty
`KSqlErrSchema`	−327	Bad schema
`KSqlErrTooBig`	−328	Too much data for one row
`KSqlErrConstraint`	−329	Constraint violation
`KSqlErrMismatch`	−330	Data type mismatch

(*continued overleaf*)

Identifier	Value	Description
`KSqlErrMisuse`	−331	Library used incorrectly
`KSqlErrNoLFS`	−332	Uses operating system features not supported on host (not used in the current version of Symbian SQL)
`KSqlErrAuthorization`	−333	Authorization denied (not used in the current version of Symbian SQL)
`KSqlErrFormat`	−334	Auxiliary database format error (not used in the current version of Symbian SQL)
`KSqlErrRange`	−335	Bind parameter is out of range
`KSqlErrNotDb`	−336	Not a database file
`KSqlErrStmtExpired`	−360	SQL statement has expired and needs to be prepared again

References

Codd, E.F. (1970) A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387.

Codd, E.F. (1972) Relational Completeness of Data Base Sublanguages. In R. Rustin (ed.), *Database Systems*, 65–98. Prentice-Hall.

Codd, E.F. (1980) Data models in database management. *Proceedings of the 1980 Workshop on Data Abstraction, Databases and Conceptual Modeling*, 112–114. New York, NY: ACM.

Codd, E.F. (1985a) Is Your DBMS Really Relational?, *ComputerWorld*, 14 October.

Codd, E.F. (1985b) Does Your DBMS Run By the Rules?, *ComputerWorld*, 21 October.

Connolly, T.M. and Begg, C.E. (2001) *Database Systems: A Practical Approach to Design Implementation and Management*. Addison-Wesley.

Date, C.J. (1999) Thirty Years of Relational: Relational Forever! *Information Week*, 2(8).

Date, C.J. (2003) *An Introduction to Database Systems*, 8th Edition. Addison-Wesley.

Lavenberg, S. (1983) *Computer Performance Modeling Handbook*. New York: Academic Press.

Sanders, G.L. and Shin, S. (2001) Denormalisation effects on performance in RDBMS. *Proceedings of the HICSS Conference*. IEEE.

Smith, C.U. (1990) *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley.

Index