SQLBase
Advanced Topics Guide

20-2119-1002

centura™

Trademarks

Centura, Centura Ranger, the Centura logo, Centura Web Developer, Gupta, the Gupta logo, Gupta Powered, the Gupta Powered logo, Fast Facts, Object Nationalizer, Quest, Quest/Web, QuickObjects, SQL/API, SQLBase, SQLConsole, SQLGateway, SQLHost, SQLNetwork, SQLRouter, SQLTalk, and Team Object Manager are trademarks of Centura Software Corporation and may be registered in the United States of America and/or other countries. SQLWindows is a registered trademark and TeamWindows, ReportWindows and EditWindows are trademarks exclusively used and licensed by Centura Software Corporation.

Microsoft, Win32, Windows, Windows NT and Visual Basic are either registered trademarks or trademarks of Microsoft Corporation in the United States of America and/or other countries.

IBM and IBM PC are registered trademarks of International Business Machines Corporation. DB2/ OS/2, and Token Ring are trademarks of International Business Machines.

NetWare and Novell are registered trademarks of Novell, Inc. Netware Requester is a trademark of Novell, Inc.

Java is a trademark of Sun Microsystems Inc.

SQL Designs and the SQL Designs logo are trademarks of SQL Designs, Inc.

All other product or service names mentioned herein are trademarks or registered trademarks of their respective owners.

Copyright

Foreword

In the early days of relational databases there was a certain purist mentality where relational databases were compared to hierarchical and network databases, and were viewed as impure because users had detailed knowledge of the data structures used to store data. Often, users would alter the design and implementation of their programs to take into account the structure of data storage. Relational database vendors took the high ground in developing the concept that it was the RDBMS's responsibility to efficiently manage the storage and retrieval of data, and it was the responsibility of the application designer to provide the logical design and schema of the data.

Coming from this purist background I have often resisted the notion of allowing our users to know about the internals of the database. However, time, experience, and the realities of relational technology have taught me that a perfect world where data storage and retrieval is optimally handled by the RDBMS does not exist for us or any other RDBMS vendor. Our customer base has been very clear in communicating their need to know more and more about the internals of SQLBase. Our close communication with customers and consultants have helped us to better understand the need and the value of clearly explaining and documenting how SQLBase does what it does.

With version 6 of SQLBase, we have taken the opposite position on exposing the internals of SQLBase. With SQLConsole 2.0, our customers can learn almost as much about the internal workings of SQLBase as the engineers that developed it. We have come to the realization that more knowledge about the internals benefits both our customers and ourselves. Our customers are able to build more robust, better performing, and more stable applications with increased knowledge of SQLBase internals. We benefit from the success of our customers in deploying these robust applications. We also benefit from receiving feedback from customers whose knowledge has been enhanced by information we have provided them about SQLBase internals. From all of this it became clear that both our customers and ourselves win by producing this book.

In 1992 I was approached by Jon Boring of SQL Designs with the idea of producing a book that would thoroughly document SQLBase internals. This notion was very exciting because of the great need for such information, and also because of the value of having this document produced by a group who is very experienced in the logical and physical design of relational databases. SQL Designs is one of our oldest and most technically respected business partners. The people at SQL Designs have large-shop, mainframe-class database know-how and have been very effective in transferring this knowledge to the PC client/server world. They have a deep understanding of the kinds of information a professional database administrator needs for database design and administration. SQL Designs have participated in some of the largest and most successful SQLBase projects and are the preferred SQLBase consultants for some of our largest SQLBase customers.

In 1991 and 1992, they were involved in several large SQLBase projects. From this project SQL Designs also became very much aware of the great need for such internal documentation. I saw the great synergism that could happen if the SQLBase development staff and SQL Designs pooled their expertise and talents to produce this book.

From October 1992 to December 1993 SQL Designs embarked upon an intense research project to learn and understand SQLBase internals. They worked closely with SQLBase development engineers to understand every aspect of the way SQLBase operates. The information and ideas were refined and reviewed over and over again by SQLBase engineering and publications staff for accuracy and clarity. For a period of time it was as though the people from SQL Designs became a part of the SQLBase 6.0 development project. In fact, the interaction with SQL Designs resulted in the decision to implement a new feature in SQLBase that allows the internal statistics SQLBase uses for its query optimization to be altered for the purpose of query modeling. This feature and several other features implemented in SQLBase 6.0 were a direct result of our interaction with SQL Designs.

Through the SQLBase 6.0 beta program we received excellent feedback about the value of this book. It was the long awaited answer to many of our customers. We believe this book will be the answer to those who have been thirsting after the detailed internal knowledge of SQLBase. We hope and believe that this book will become a vital part of the application development process involved in developing applications and systems using SQLBase.

Bruce D. Scott
Senior Vice President Research & Development,
DataBase and Connectivity Products
Gupta Corporation

About the Authors

The SQLBase Advanced Topics Guide was written by Jon Boring and Dave Gerber of SQL Designs, Inc. Jon Boring, president and cofounder of SQL Designs, is a nationally recognized expert in the area of client/server software development methods and a frequent speaker at industry conferences and trade shows.

Prior to joining SQL Designs, Dave Gerber directed the database administration and systems programming functions at several Fortune 500 corporations. Dave is a senior manager at SQL Designs and oversees the database technology group where he specializes in database administration, physical database design and query optimizers for SQL databases.

SQL Designs is a database consulting organization which specializes in the client/ server architecture and has extensive experience using the Gupta products. They offer consulting, applications development, and training in addition to a full range of support services. Interested parties may contact SQL Designs at:

SQL Designs, Inc.
206 East Pine Street
Lakeland, FL 339801
Telephone: 813.683.3886
Fax: 813.683.3997

Contents

4 External Schema Creation 4-1

5 Transaction Definition 5-1

6 Refining the Physical Design 6-1

7 Physical Design Control 7-1

8 Database Pages 8-1

9 B-Tree Indexes 9-1

Preface

This is an advanced reference manual for SQLBase. It contains detailed discussions of various technical topics, including database design and SQLBase query optimizer.

Audience

The *Advanced Topics Guide* is written for anyone using the advanced features of SQLBase. This includes:

- Application developers

  Application developers build client applications that access databases using frontend products like SQLWindows and the SQL/API.

- Database administrators (DBAs)

  Database administrators perform day-to-day operation and maintenance of the DBMS and an organization's databases. They install maintenance releases, load data, control access, perform backup and recovery, and monitor performance.

- Database designers

  Database designers, or senior database administrators, are responsible for the logical and physical design of databases. Their responsibility is to ensure the database meets the operational and performance requirements of the system.

This manual assumes you have:

- Knowledge of relational databases and SQ
- Familiarity with the material in the other SQLBase manuals.

What is in this manual

This manual is organized into the sections listed below, each of which is further subdivided into a number of chapters. There is also an index.

Section Name	Description
Database Design	Chapters 1-7 discuss logical and physical database design.
SQLBase Internals	Chapters 8-14 present several SQLBase internals topics.
SQLBase Query Optimizer	Chapters 15-18 discuss the SQLBase query optimizer.

Notation conventions

The table below show the notation conventions that this manual uses.

Notation	Explanation
You	A developer who reads this manual.
User	The end-user of applications that you write.
bold type	Menu items, push buttons, and field names. Things that you select. Keyboard keys that you press.
Courier 9	Builder or C language code example.
SQL.INI MAPDLL.EXE	Program names and file names.
Precaution	**Warning:**
Vital information	**Important:**
Supplemental information	**Note:**
Alt+1	A plus sign between key names means to press and hold down the first key while you press the second key.
TRUE FALSE	These are numeric boolean constants defined internally in Builder:

Constant	Value	Meaning
TRUE	1	Successful, on, set
FALSE	0	Unsuccessful, off, clear

Other helpful resources

Centura Books Online. The Centura document suite is available online. This document collection lets you perform full-text indexed searches across the entire document suite, navigate the table of contents using the expandable/collapsible browser, or print any chapter. Open the collection by selecting the Centura Books Online icon from the **Start** menu or by double-clicking on the launcher icon in the program group.

Online Help. This is an extensive context-sensitive online help system. The online help offers a quick way to find information on topics including menu items, functions, messages, and objects.

World Wide Web. Centura Software's World Wide Web site contains information about Centura Software Corporation's partners, products, sales, support, training, and users. The URL is http://www.centurasoft.com.

To access Centura technical services on the Web, go to http:/www.centurasoft.com/support. This section of our Web site is a valuable resource for customers with technical support issues, and addresses a variety of topics and services, including technical support case status, commonly asked questions, access to Centura's Online Newsgroups, links to Shareware tools, product bulletins, white papers, and downloadable product updates.

For information on training, including course descriptions, class schedules, and Certified Training Partners, go to http://www.centurasoft.com/training.

Other Publications. Depending on your requirements, you also use publications for these products:

- ***A Guide to the SQL Standard,*** C.J. Date

    This book describes the SQL relational database language with an emphasis on both the official standard version of SQL and on the use of SQL for programmed (versus interactive) database access.

- ***SQL & Its Applications***, Raymond A. Lorie and Jean-Jacques Daudenarde, Prentice Hall, 1991.

- ***The Art of Computer Programming***, D.E. Knuth, Addison-Wesley, 1973.

- ***The Ubiquitous B-tree,*** D. Comer, *ACM Computing Surveys*, Vol. 11, No. 2 (June 1979).

- ***Files and Databases, an introduction***, P. Smith and G. Barnes, Addison-Wesley, 1987.

- ***Algorithms + data structures = programs***, N. Wirt, Prentice-Hall, 1976.

- ***Relational Completeness of Data Base Sublanguages,*** E.F. Codd, in Data Base Systems, Courant Computer Science Symposia Series, Vol. 6. Prentice-Hall, 1972.

- ***Concurrency Control and Recovery in Database Systems***, Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman, Addison-Wesley, 1987.

- ***An Introduction to Database Systems***, C.J. Date, Addison-Wesley, 1990 (5th edition).

- ***Relational Database: Writings 1985-1989***, C.J. Date, Addison-Wesley, 1990.

- ***Relational Database: Writings 1989-1991***, C.J. Date, Addison-Wesley, 1992.

- ***The Relational Model for Database Management: Version 2***, E.F. Codd, Addison-Wesley, 1990.

- ***Computer Data-Base Organization***, James Martin, Prentice-Hall, 1977 (2nd edition).

- ***Query Evaluation Techniques for Large Databases,*** Goetz Graefe, *ACM Computing Surveys*, Vol. 25, No. 2 (June 1993).

- ***Join Processing in Relational Databases,*** Priti Mishra and Margarate Eich, *ACM Computing Surveys*, Vol. 24, No. 1 (March 1992).

- ***On Optimizing an SQL-like Nested Query,*** Won Kim, *ACM Transaction on Database Systems*, Vol. 7, No. 3 (September 1982).

- ***Join Processing in Database Systems with Large Main Memories,*** Leonard Shapiro, *ACM Transaction on Database Systems*, Vol. 11, No. 3 (September 1986).

Send comments to...

Anyone reading this manual can contribute to it. If you have any comments or suggestions, please send them to:

Technical Publications Department
Centura Software Corporation
975 Island Drive
Redwood Shores, CA 94065

or send email, with comments or suggestions to:

techpubs@centurasoft.com

Chapter 1

Introduction

This chapter introduces the following chapters on database design:

- Chapter 2 - Conceptual Schema Creation
- Chapter 3 - Internal Schema Creation
- Chapter 4 - External Schema Creation
- Chapter 5 - Transaction Definition
- Chapter 6 - Refining the Physical Design
- Chapter 7 - Physical Design Control

ANSI/SPARC three schema architecture

The Database Design Section is organized around a database architecture that was originated by the ANSI/X3/SPARC Study Group on Data Base Management Systems. This group was established in 1972 by the Standards Planning and Requirements Committee (SPARC) of ANSI/X3. ANSI is the American National Standards Institute and X3 is this organization's Committee on Computers and Information Processing. The objectives of the Study Group were to determine the areas, if any, of database technology for which standardization was appropriate, and to produce a set of recommendations. The Study Group concluded that interfaces were the only aspect of a database system that could possibly be suitable for standardization, and so defined a generalized architecture or framework for a database system and its interfaces.

This architecture includes three separate *schemas* (a schema is a model or formalism to represent an abstract concept) of an organization's data resource. Each schema provides a perspective that serves a limited purpose and is optimized towards a different set of objectives. The schemas included in the ANSI/SPARC three- schema architecture are the *conceptual*, *external*, and *internal*.

The *conceptual* schema, sometimes referred to as the *logical data model*, is based on the natural characteristics of the data without regard to how the data may be used or physically stored. The conceptual schema models the data and the relationships between data as they naturally exist. The assumption is that this perspective of an organization's data resource is the least likely to change over time and is therefore the most stable. Likewise, it is assumed that how data is used and physically stored changes frequently over the life of an organization. Consequently, the conceptual schema is the primary or base schema within the ANSI/SPARC three schema architecture to which the internal and external schemas are mapped.

The external schema is the user's views or filters of the database and specifically considers how the data is used. The views are tailored to the user's information requirements and allow the user access to the required data in the simplest possible form. Since the user's data requirements change frequently, the external schema is subject to constant revision.

The internal schema is a mapping of the conceptual model to the physical constructs of the target database management system (DBMS). In performing this translation, the database designer may introduce new physical objects or alter the conceptual objects to achieve better performance or capacity and a more usable database. These design changes can be made to exploit the hardware or software environment of an implementation platform. Since these changes should not be visible at the conceptual level, the actual meaning of the data does not change. Since the views of the data in the external schema are also independent of this physical schema, users do not have to deal with the physical objects.

Through the three schemas and the mapping between them, the ANSI/SPARC three schema architecture provides a level of indirection which insulates user needs from database performance and capacity concerns. The major advantage of using this architecture is that as the physical characteristics of the data and/or the target DBMS changes over the life of the database, database designers can adjust the physical constructs of the internal schema *without affecting existing program modules*. Similarly, the user's views of the data can be changed as their requirements change without affecting the manner in which the data is physically stored. Without the three schema (or similar) architecture, changes to the physical database can require costly effort wasted on maintenance of existing program modules.

Although no commercial DBMS now provides full support for the ANSI/SPARC three schema architecture, this architecture provides an excellent goal for the data base administrator (DBA) to work towards. It also provides a good framework for organizing database design activities.

Structure of this section

In the remaining six chapters of this section we reveal a step by step path through the database design process which specifies and optimizes each schema of the ANSI/SPARC three schema architecture. This process is shown in Figure A.
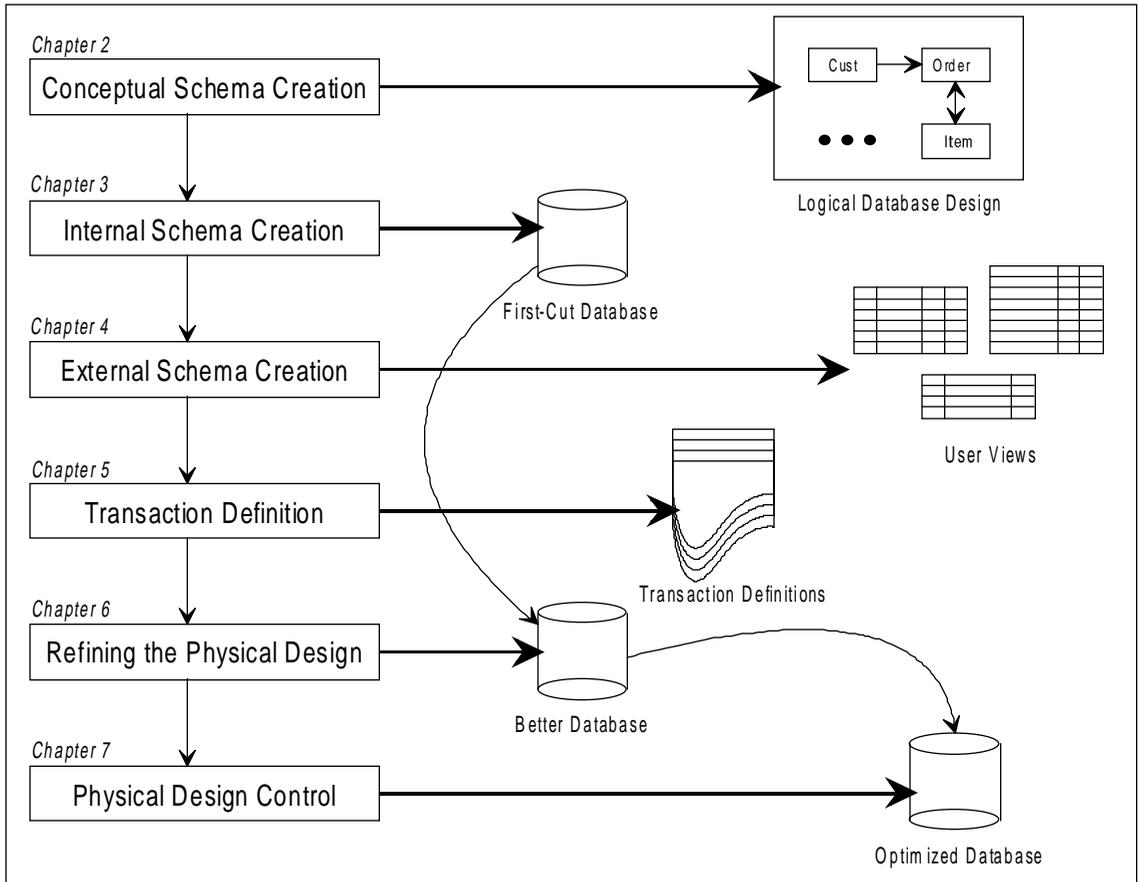
Figure 1A - Database design chapters and deliverables

Some DBAs do not like the work required to create and maintain the external schema level of the ANSI/SPARC standard. Since the benefits of using this set of user views are limited by the current state of the art in relational database theory (see chapter 4 for further details), they avoid this work until the payoff is more substantial. This is a valid viewpoint, and organizations that wish to take this approach may omit the work in Chapter 4, *External Schema Creation*. They can then code all of their transaction data manipulation language (DML), as well as user queries, directly against the base

tables in the internal schema. The risk associated with doing this is that alterations made to the physical design have a greater chance of affecting existing DML and causing some re-coding to keep the DML valid. Note that creating the external schema does not totally eliminate this risk.

Chapter 2 - Conceptual Schema Creation

Logical data modeling, the creation of the conceptual schema, is a (relatively) well-known process within the database industry. Many books and training courses are available on the subject. Consequently, this chapter does *not* attempt to present an exhaustive treatment of this material. Rather, this chapter introduces you to the logical database design process. Readers who are interested in a more detailed presentation of the material are encouraged to read one of the many excellent books available on the subject or attend a local training course.

In addition, the database industry has standardized on the "Entity-Relationship Diagram," or ERD, to represent the conceptual schema. Consequently, many Computer Aided Software Engineering (CASE) vendors market tools for entity relationship diagramming. You may find one of these tools beneficial in the creation of your conceptual schema. Centura has established partnerships with several CASE vendors to provide tighter degree of integration between the CASE tool and the Centura software. At the time this manual was written, partnerships exist with LBMS, Intersolv, Bachman, Popkin, Visual Systems and LogicWorks.

Chapter 3 - Internal Schema Creation

In this chapter we cover the tasks necessary to create the first-cut internal schema or physical database design. The database constructed in this chapter is an initial prototype only, representing the logical data requirements in their simplest form, without any processing requirements. The first-cut internal schema is constructed by applying some simple conversion rules to the entity relationship diagram. These rules are discussed in detail in this chapter.

Also, many of the CASE tools discussed above automatically generate the first-cut internal schema. You may find using a CASE tool which converts an entity relationship diagram to SQLBase compatible SQL DDL (data definition language) beneficial.

Chapter 4 - External Schema Creation

In this chapter we cover the tasks required to create the first user views needed for the external schema. These views are mainly reflections of underlying tables without any complex user requirements. We introduce you to the on-going process of view administration, since the external schema evolves as new needs arise. We discuss tasks of view administration and the underlying security requirements of views.

Chapter 5 - Transaction Definition

We create transaction definitions at this point because they represent detailed and critical processing requirements. The final database must meet these requirements in order to be completed successfully. Completion of the tasks outlined in this chapter provides the yardstick by which the rest of the database design process is judged.

Chapter 6 - Refining the Physical Design

In this chapter we re-visit the internal schema created in chapter three with the goal of introducing physical database design optimization methods which improve its performance and capacity. At this point in the database design process, general guidelines and heuristics are applied to the first-cut physical database design to produce a better physical database design, or internal schema. Over the years, database designers have discovered many rules-of-thumb or heuristics to improve overall database performance. Many of these common guidelines are detailed in this chapter.

Chapter 7 - Physical Design Control

In this chapter we conclude the database design effort by entering the next stage, the iterative process of adjusting the internal schema to meet the objectives set in the transaction definitions. Transactions that fail to meet processing requirements are examined for possible improvements that target their particular database accesses. Next we evaluate the effect on the other transactions that use the database. Our goal is to find an acceptable trade-off between conflicting requirements of all of the transactions which make up the complete system. This process is reiterative until all critical transactions are performing effectively.

Chapter 2

Conceptual Schema Creation

The conceptual schema is the primary or base schema within the ANSI/SPARC three schema architecture and is sometimes referred to as the logical data model. It models an organization's data resource and the relationships between data as they naturally exist without regard to how the data may be used or physically stored. Of the three schemas, this is the least likely to radically change over time.

This chapter introduces the process of logical data modeling and several related topics. In this chapter we:

- Introduce the data administration function and responsibilities.
- Cover logical database design.
- Explain types of entity-relationship data models.
- Explain the data normalization technique.

Data administration

The data administration (DA) function plays a consulting role in the process of logical database design. This contrasts with the traditional mission of the database administration (DBA) function, which is mainly responsible for physical database design and implementation tasks. The mission of data administration is to ensure that an organization's data resources are managed responsibly and recognized as an important asset. A data administration department is responsible for formulating the plans that guide the evolution of all corporate databases. By providing this master plan, the DA function ensures that corporate computing systems will be compatible in the future.

This compatibility guarantees that future system development will not adversely affect existing systems and users in some way, and that information in separate databases today can be combined in the future without undue cost or delay. Without this assurance, a company can be exposed to financial and operational risk. This can arise from either the disruption of mission critical computer systems or through the company's inability to meet future information requirements effectively.

Many corporations have not formally established a DA function. In the absence of this formal mandate, it is important for those personnel responsible for operating and maintaining the database environment to establish an agreement to "self-rule." This helps to avoid the pitfalls caused by a lack of standardization. Since database professionals are often the first people to realize the benefit of establishing DA practices, it behooves them to either attempt to persuade management to formally establish the DA function, or to make it part of their regular procedures as best they can.

In this chapter we intend to provide a starting point for database administrators who are working without formal DA functions and who have not yet adopted any DA practices. There are many books and articles that treat these and other DA subjects in great detail, and we encourage you to search out those subjects that interest you.

Standards creation and enforcement

One of the first tasks of data administration is the creation of numerous standards. The policing of these standards is one of the major ongoing efforts of the DA function. There are many opinions regarding the exact composition of these standards, particularly naming standards. However, the most important aspect of these standards is that they exist, are fully documented and distributed, and that they are followed.

The following list represents some of the areas in which standards are generally created:

- Abbreviations

  Standards for the abbreviations used as roots for object names are useful in maintaining naming consistency.

- Data element (or attribute) names

  These standards benefit users who access the database using tools such as Quest. These users must find the information they need in a consistent way. When several people are creating new elements, variations of the same name can be created. Good element and abbreviation standards can solve this problem. Element naming standards should also address the capitalization and word separators of multi-name elements.

- Table names

  The previous comments on data element names apply here as well. Names must be simple and meaningful to users.

- View names

  Many sites want naming conventions that allow easy differentiation between views and tables. Some sites like to highlight views that result in join operations since these may be time consuming and have restrictions on update operations.

- Program names

  Most installations try to indicate application ownership of programs by mnemonics in the name. Mnemonics are short abbreviations, often two or three letters, which contain enough of the original phrase to prompt an association to the original. An example of a mnemonic is "AR" for accounts receivable. Also, some sites like to be able to tell whether a program is read-only from the name alone.

- File names

  File names should bear some relationship to the database that owns the file.

- User names

  These standards have to do with who a person is and where that person can be found, either by department or physical location.

  The definition and utilization of standardized templates is valuable in performing security authorizations for users based on the data requirements of their job. This procedure is very useful when an audit is being performed.

- Database names

    Standardizing database names simplifies operational control of the databases for day-to-day use. The same applies to the following areas or items:

    - Server names

    - Workstation names

    - Printer names

Data dictionary

One of the key tools of data administration is a data dictionary or repository. This is a special database which contains metadata (data about data such as element names, meaning, data type, null characteristics, masks, and validations) describing the business's database environment. Within the dictionary is information about each table, its columns, indexes, views, and relationships to other tables. Through the use of the dictionary, users and programmers can find the data items, tables, or views they need. Also, data administrators can monitor naming standards, locate data, and eliminate inconsistencies in the ways various departments describe and use data.

Since the dictionary is the main tool needed for data administration, database administrators who attempt to provide basic DA functions for their organizations can design and implement a simple dictionary for their own use. A database describing tables, views, and columns can be created and then populated with data extracted from the SQLBase system catalog tables. These are included in every SQLBase database, and have names which start with "SYSADM.SYS", such as "SYSADM.SYSTABLES." By combining this information across all databases, the DBA can locate and correct non-standard names, inconsistent key definitions, and duplicate table or column definitions.

The Team Object Manager component of Centura Team Developer contains a repository which you may use to store your metadata. Similarly, many CASE tools include a data dictionary and you may find it beneficial to use the repository of the CASE tool you have selected to perform database design tasks.

Enterprise-wide data planning

Corporations which conduct strategic management as an organization-wide activity usually perform some type of strategic planning within the MIS division. Although various strategic planning methods may be used, most techniques lead to the creation of a strategic data plan. This plan serves as a guide for the development of the corporation's data resource throughout its development. Often, one of the outputs of the strategic data plan is an enterprise-wide data model that can be used as a roadmap by future database designers.

The format of an enterprise-wide data model is generally that of an entity-relationship diagram (ERD). This diagram shows the major business entities in the organization and the way they relate to each other. The entities should have descriptions attached, but most attributes can be omitted when the modeling is done at this high level. Include primary key attributes, along with their data types, in order to enforce consistency among these critical attributes.

It is up to database designers working in organizations that have an enterprise-wide data model to be familiar with the model and to keep database designs consistent with it. While the enterprise model will be missing some low-level entity types and most of the attributes, it should dictate the general structure of the major entities and their relationships in any lower level design. If this is not true, then adjust the enterprise model. Analyze and document any impact from making these changes.

Subject area databases

A database may be designed around either application systems or subject areas. Research and experience has shown that building databases around subject areas rather than application systems to be far superior.

For example, an organization could build a database around an accounts receivable application, with its own customer entity, and then design another database around an order entry application, also containing another customer entity. This results in both a proliferation of databases as well as data duplication. While the customer entities in the two databases just mentioned may differ in some ways, they probably have an overlap of many attributes such as name and address. Keeping track of these two entities' common attributes is difficult and error prone. Most application systems designed with their own database simply ignore the similar information in other systems and make no effort to maintain consistency. Consequently, most organizations with a formal data administration function design databases around business subject areas rather than application systems.

Center the design around subject areas, such as product or customer, and your result will be many different applications and user departments that can create, update, and share information in a seamless fashion. In addition, you are likely to find that the time necessary to develop application systems decreases significantly as the corporate database is completed.

This is because a significant portion of most application development projects is spent building the programs necessary to add, change, and delete persistent data. If databases are designed around application systems, then a database must be built for every application system and the programs to add, change, and delete persistent data must be written for every application. On the other hand, if the database is designed around subject areas with a high degree of data sharing, at some point in an organization's development cycle all subject areas are built, the database is complete, and all the programs necessary to add, change, and delete persistent data are written.

Once an organization reaches this stage, new applications can be developed rapidly since no new programs must be written to add, change, and delete persistent data.

Organizations that have a formal DA function and have completed a strategic data plan and an enterprise data model often move on to performing data modeling on certain key business areas before starting development work on applications. These business area data models are usually fully normalized (normalization is covered later in this chapter) and complete, with specifications of all keys and attributes in each entity included within the model. These data models serve as good sources of material for a database design and can often be used as is, but two or more may have to be merged to obtain the complete set of data required by one application.

Logical data modeling

The most rigorous technique now in use to perform the logical database design task is logical data modeling. Using diagram symbols, a model (or picture) of the data required to meet the objectives of the system is developed. The resulting model is called an entity-relationship diagram (ERD).

Techniques are applied to this model to verify its accuracy, completeness, and stability. The model can then be translated into a physical database design, possibly for multiple implementation platforms, by performing a transformation process that meets specific performance requirements as well as any special characteristics of the chosen physical architecture.

The three most common styles of ERDs in use today are the Bachman diagram, the Information Engineering diagram and the Chen diagram. Two are named after their authors, while the other was specified by James Martin as a component of his information engineering (IE) diagram set. Each style represents similar types of objects, and examples of each follow. The most basic objects are the entity and the relationship. Keys, attributes, and domains are not assigned diagram symbols, but are important background objects for logical data modeling. Each of these concepts are discussed in the following sections.

Entities and relationships

Entities are the basic information units of an ERD and are equivalent to tables in the finished physical database design. Entities represent a person, place, concept, or event in the real world about which information is needed. Examples of entities are customers, orders, invoices, employees, warehouses, and so on. Entities are represented by boxes in all three diagram styles.

Relationships are connections between one or more entities. These are represented graphically by lines containing various symbols that convey characteristics of the relationship. The majority of relationships involve only two entities and are called

binary relationships. These are often broadly classified according to how many occurrences of an entity type can appear on each side of the relationship, as shown in Figure A.
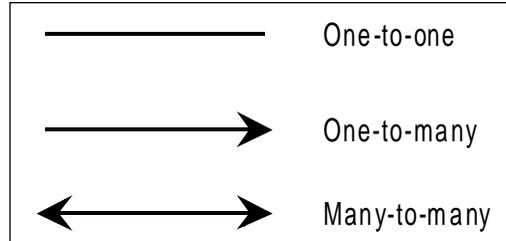
Figure 2A - Types of relationships classified by maximum cardinality or relationship volume.

An example of a simple ERD for each of the three diagramming styles follows. Each diagram depicts the same model, a common example of a business's basic order processing data requirements.

The Bachman ERD style
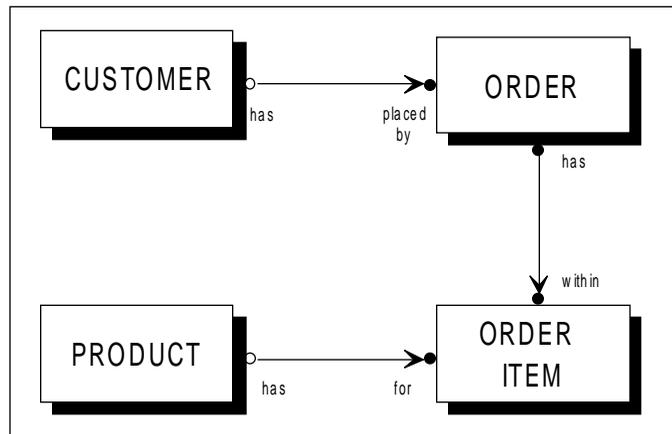
Figure 2B - Bachman ERD of order processing data requirements

In the Bachman ERD, the CUSTOMER entity has a relationship with the ORDER entity, as indicated by the line between the two entities. An important characteristic of the relationship is its cardinality, which describes the allowable number of occurrences of each entity type involved in the relationship. In the Bachman notation, minimum cardinality (which can be either zero or one) is described through the use of circles at the ends of the lines (a relationship with a minimum cardinality of zero is often called optional, while those with a minimum of one are termed mandatory), while maximum cardinality (which can be either one or many) is described through

the use of arrows. In the case of minimum cardinality, the circle can either be hollow, indicating zero, or filled in, indicating one. To describe maximum cardinality, the absence of the arrow means a maximum of one, while the presence of the arrow means a maximum of many (more than one).

In Bachman terminology, a relationship is called a *partnership set*, which is made up of two *partnerships*. Each of these two partnerships is owned by one of the entities involved in the relationship. In this example, CUSTOMER has a partnership with ORDER, and ORDER also has a partnership with CUSTOMER. These two partnerships together form the CUSTOMER-ORDER partnership set.

For CUSTOMER's partnership with the ORDER entity, the minimum cardinality of orders for a customer is specified by the hollow circle next to the CUSTOMER entity, which is labelled "has." The minimum cardinality is zero therefore this circle is hollow. This means that a customer may exist without any orders. The maximum cardinality of the ORDER entity from the perspective of the CUSTOMER is described by the arrow on the ORDER side of the relationship, which specifies that a customer may have many orders. If there were no arrow, then the maximum cardinality would be one rather than many, and a customer could have only a single order at any one time. The way to read this relationship in terms of the CUSTOMER entity is to say, "A customer *has* from zero to many orders."

From the ORDER perspective, the minimum cardinality of CUSTOMER is 1. This is noted by the circle next to the ORDER entity which is labelled "placed by" and is filled in. The maximum cardinality of the relationship is also one, since there is no arrow on the CUSTOMER side of the relationship. This type of relationship, where an entity has both minimum and maximum cardinality of one with another entity, is often called a dependency. The ORDER entity is said to be dependent on CUSTOMER, since it could not exist without a relationship to a single occurrence of CUSTOMER. The reading from the ORDER entity's perspective is, "An order is *placed by* one, and only one, customer."

In a similar manner, the other relationships read:

- An order *has* 1 or many items.
- An item is *within* 1, and only 1, order.
- A product *has* from zero to many items.
- An item is *for* 1, and only 1, product.

The Information Engineering ERD style
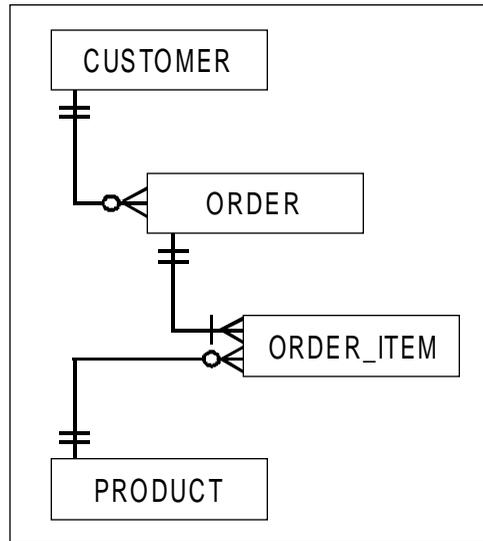
Figure 2C - Information Engineering ERD of order processing data requirements

In the Information Engineering ERD, entities are also depicted by boxes and relationships as lines. The significant difference from the Bachman ERD is the graphic description of relationship cardinality. In this diagram, each entity's minimum and maximum cardinality in a relationship is described immediately adjacent to itself. For instance, the CUSTOMER's cardinality with respect to ORDER is shown in two horizontal bars immediately below CUSTOMER. The symbol closest to an entity describes the maximum cardinality and can be either a bar (indicating one), or a crow's foot (indicating many). Following the relationship line, the next symbol describes the minimum cardinality and can be either a bar (indicating one), or a circle (denoting zero). The meaning of each relationship in Figure C is the same as described for Figure B.

The Chen ERD style
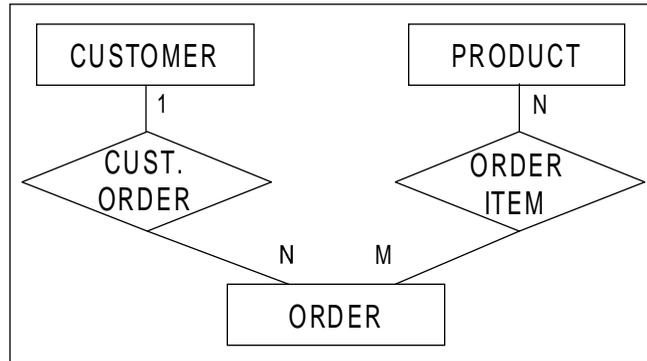
Figure 2D - Chen ERD of order processing data requirements

In the Chen ERD, there is a more significant deviation from the previous examples. Entities are still represented by boxes, and lines between entities still play a part in representing relationships, but now there are also diamonds that appear on the relationship lines which contain the name of the relationship. Also, relationships may have attributes assigned to them directly. These are called *information bearing* relationships. In the Chen diagram, attributes may appear directly on the diagram as bubbles which contain the attribute's name and are attached to the entity or relationship to which they belong. Attributes have been omitted from this diagram in order to remain uncluttered and consistent with prior diagrams. When this happens, the relationship containing the attributes would be transformed into a table in the physical database design in the same way that entities become physical tables.

The ORDER ITEM relationship is an example of this. In previous diagrams, ORDER ITEM was an entity rather than a relationship. In the Chen diagram, the relationship between ORDER and PRODUCT is indicated as having a maximum cardinality of "many" on both sides (shown by the "M" and "N"), and the ORDER ITEM relationship between them is information bearing and would appear as a table in the resulting physical database design. The table that results from two entities having a many-to-many relationship is said to *resolve the relationship*. The data contained in the table is called *junction data*, because it is the information that is unique for each occurrence of the relationship between the two entities. In this example, the junction data would be attributes such as QUANTITY-ORDERED. Also, while the Chen diagram indicates maximum cardinality by means of a number or symbol (the "1", "N", and "M" in Figure D) next to the entity involved in a relation, minimum cardinality is not depicted. In Figure D, "N" orders occur for each CUSTOMER (where "N" indicates some number greater than 1), thereby depicting the customer to order relationship as one-to-many. The "one" side of this relationship is indicated by the "1" appearing next to the CUSTOMER entity.

Attributes and domains

Attributes are simply characteristics of entities. They either describe or identify an entity. Identifying attributes are also called keys, which are further explained in the next section and are data items such as EMPLOYEE-NUMBER, CUSTOMER-NUMBER and SSN (social security number). Examples of descriptive attributes are EMPLOYEE-NAME, CUSTOMER-ADDRESS and PRODUCT-DESCRIPTION. Following the transformation of the logical design into a physical database, attributes will become columns of tables.

Domains are the logical value sets that attributes possess. There is some variation on the definition and use of domains in different diagram styles. Notably, the Bachman ERD calls this definition of domains a *dimension*, which is further classified into domains which then possess a particular display characteristic. The definition of domain given here is the usage specified by E. F. Codd when he published the relational data model. Attributes such as TOTAL-SALE-DOLLARS and PRODUCT-COST could share a domain called CURRENCY, for example. This is because the possible values of these attributes is a discrete set of those real numbers that are represented by the picture 999,999...,999.99. Likewise, the attributes of ORDER-DATE and BIRTHDATE could share the DATE domain, in which 02/29/1992 is a set member, but 02/29/1993 is not.

Domains provide a means to perform semantic validation of query statements, particularly for join operations. When tables are joined on columns that are not defined on the same domain, the results will not be meaningful. An example of this is SELECT * FROM CUSTOMER, ORDER WHERE CUSTOMER.CUSTOMER-NUMBER = ORDER.TOTAL-SALE-DOLLARS. Although domains do not have to be specified for a data model since they are not at this point transformed into any physical design object, their inclusion can provide a means of manually validating the join criteria used in SQL statements. They are also useful in the physical design process for determining physical data types of table columns.

Primary and foreign keys

The two most important types of keys for database design are primary and foreign keys. These provide the means by which SQLBase supports referential integrity. For a logical data model to be complete, every entity must have a primary key identified. This primary key can consist of either a single attribute, or may be composed of two or more attributes in which case it is called a concatenated or composite key. Some diagramming techniques allow mandatory relationships to be specified as an alternative to attributes when defining primary keys. When this is done the effect is for the primary key of the entity on the other side of the relationship to be *inherited* as a component of a concatenated key in the dependent entity.

The two critical characteristics of a primary key are uniqueness and non-redundancy. Uniqueness refers to the fact that the value of a primary key for a given occurrence of an entity must not be duplicated in any other occurrence of the same entity type. The physical design implication is that a unique index must be able to be placed on a primary key successfully. The non-redundancy characteristic of a primary key, which applies to concatenated keys only, means that each of the attributes which comprise the primary key must be essential to its uniqueness, so that if any attribute was removed, then the key would no longer be unique. This last rule prevents extraneous information from being placed into keys. These rules for forming primary keys are important in the normalization process.

CUSTOMER(<u>CUSTOMER-NO</u>,CUSTOMER-NAME,CUSTOMER-ADDRESS)

ORDER(<u>ORDER-NO</u>,TOTAL-SALES-DOLLARS,ORDER-DATE,*CUSTOMER-NO*)

ORDER_ITEM(*<u>ORDER-NO</u>*,<u>LINE-NO</u>,QUANTITY-ORDERED,*PRODUCT-NO*)

PRODUCT(<u>PRODUCT-NO</u>,PRODUCT-DESC)

Legend:
<u>Primary Key</u>
Foreign Key
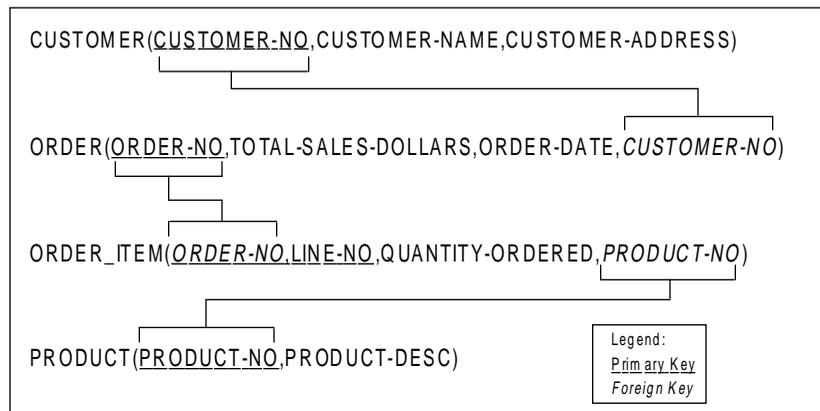
Figure 2E - Example of mapping between primary and foreign keys. Dependent entity ORDER_ITEM inherits part of its composite key from ORDER. The ORDER-NO attribute also serves as the foreign key to ORDER.

Another rule many organizations follow in formulating primary keys is to avoid building concatenated keys through stringing together many coded attributes. As an example, consider a part number where codes are present that indicate the product family, weight, color, primary vendor, and voltage of the part. Some users like this type of key because they can tell a great deal about the part merely by looking at the key. The drawback, however, is that when these attributes change (such as a new vendor), the primary key has to change. When the primary key changes, any references to it as a foreign key in related entities must change as well. All of these changes to data rows consume processing time and also contribute to the need to perform more frequent database reorganizations to prevent performance degradation. Since keys often have indexes placed on them, all these indexes will become increasingly disorganized as well.

Foreign keys must be created in an entity type to indicate the specific occurrence of another entity with which this entity has a relationship. The attributes of the foreign key must match the primary key of the related entity exactly. The foreign key appears in the entity that is on the many side of a one-to-many relationship, or may appear on either side of a one-to-one relationship. Many logical data modeling diagram techniques do not explicitly name foreign key attributes, since their existence can be assumed through the relationships that the entity participates in. When the relationship is many-to-many, an intermediate table will contain the foreign keys of both entities involved in the relation. See Figure E for the relational notation of the primary to foreign key mapping for the data model used in the previous examples.

Normalization

The normalization process is a logical design technique that is especially well suited for use with SQLBase (or for that matter, any DBMS which similarly adheres to the relational model of data) because of its capability to directly implement the results without any translation. Performing normalization results in reduced data redundancy, protects against update and deletion anomalies, and provides a more flexible, longer lasting database design. The resulting database will be easier to maintain and will not require major, high-cost structural changes as often as unnormalized designs. This is due to the inherent stability of basing the data structures on the natural organization of the basic business information flowing through the corporation rather than structuring it with only a limited user or application perspective.

While the processes that a business performs on its data may change often, the data items that are operated on by those processes can be structured to reflect the nature of the company's business. This means that the application processes, embodied in program logic, can be more quickly changed to keep pace with company needs. The underlying database remains the same, reflecting the basic information inherent in the environment. Attaining this level of stability is the goal of data normalization.

One of the fundamental requirements of a designer who wishes to perform normalization is to know the meaning of the data intimately and to have a thorough understanding of its use in the real world of business. The way that a table's attributes interact with and depend on each other is the basis for normalization. The designer performing the normalization process must be knowledgeable of the data at this detailed level in order to successfully complete the process without undue risk of error arising from misunderstandings or assumptions.

Normal forms

The normal form is a specific level of normalization achieved by a set of tables comprising a database design. Each successive normal form represents a more stable and controlled stage of design. Higher levels of normal forms always require the design to be in each lower level of normalization as a prerequisite. As an example, second normal form requires the design to be in first normal form, third normal form requires both first and second forms, and so on.

Form	Description	Transform
Unnormalized Data		Decompose all data structures into flat, two-dimensional tables.
First Normal Form	Records with no repeating groups	
Second Normal Form	All nonkey data items fully functionally dependent on primary key	For tables with composite primary keys, ensure that all other attributes are dependent on the whole key. Split tables apart as required to achieve this.
Third Normal Form	All nonkey data items independent of each each	Remove all transitive dependencies, splitting the table apart as required. No nonkey attribute should represent a fact about another nonkey attribute.

Figure 2F - Summary of normal forms, rules, and steps needed to achieve the next form

There are a number of normal forms defined and work is continuing so that more forms may appear in the future. Most current texts on database design will address normal forms one through five, although only the first three are needed for most tables. For this reason, only the first three forms are summarized in Figure F. Consult more detailed database design texts for a more in-depth discussion of additional normal forms.

First normal form

STUDENT	**STUDENT**	**CERTIFICATE**
Student_Number	Student_Number	Student_Number
Student_Name	Student_Name	Certificate_Code
Major_Dept_Code	Major_Dept_Code	Certificate_Name
Major_Dept_Name	Major_Dept_Name	Certificate_Date_Earned
Major_Dept_Location	Major_Dept_Location	Certificate_Hours_Reqd
Admission_Office		
Admitted_Date	**ADMISSION**	
Certificate_Code	Student_Number	
Certificate_Name	Admission_Office	
Certificate_Date_Earned	Admitted_Date	
Certificate_Hours_Reqd		
Unnormalized	*First Normal Form*	

Figure 2G - Conversion of an unnormalized relation, STUDENT, to three separate relations in first normal form. Primary keys are indicated by the attributes which

Conversion of an unnormalized relation called STUDENT into first normal form is shown in Figure G. This conversion involves breaking the table apart, with each table having its own primary key identified. Any repeating elements would also be removed from the relations at this point, so that each resulting table is "flat", meaning that each column has a unique name and no columns consist of multiple elements (such as an array). This simple step places these data items into first normal form.

Second normal form

First Normal Form	Second Normal Form
CERTIFICATE Student_Number Certificate_Code Certificate_Name Certificate_Date_Earned Certificate_Hours_Reqd	**CERTIFICATE** Certificate_Code Certificate_Name Certificate_Hours_Reqd **STUDENT_CERTIFICATE** Student_Number Certificate_Code Certificate_Date_Earned
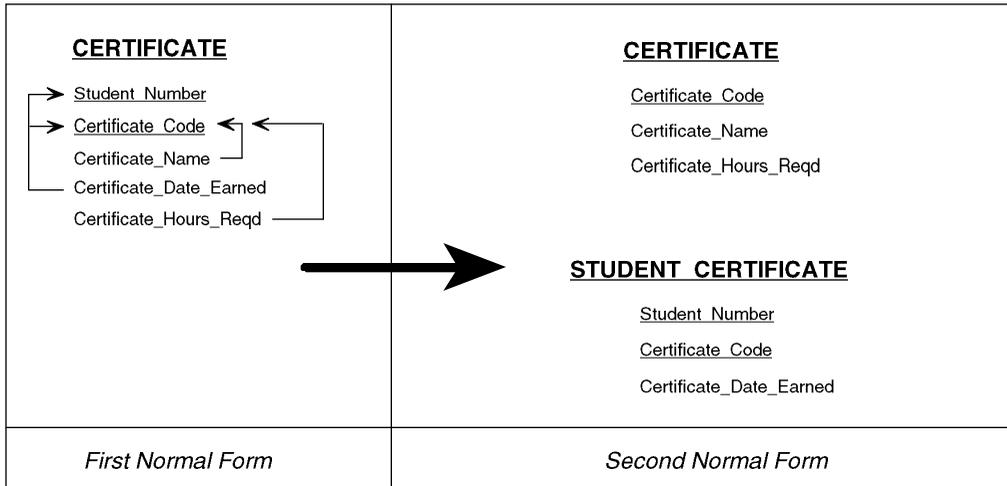
Figure 2H - Further normalization of the previous example, where the CERTIFICATE relation violates second normal form. The small arrows indicate which key attributes each non-key attribute is dependent on. The other relations in the previous example are already in second normal form.

Further transformations are required to bring the data structure from Figure G into second normal form. As Figure H shows, the CERTIFICATE relation violates second normal form, and must be broken apart. The non-key attributes of Certificate_Name and Certificate_Hours_Reqd are only dependent on part of the primary key, Certificate_Code. The table is therefore transformed so that the Certificate_Code is the entire primary key, with the two attributes which depend on it remaining in the table. The Certificate_Date_Earned attribute, which depended on the entire key of the original CERTIFICATE relation, is placed into a new relation, STUDENT_CERTIFICATE, along with the original composite key. The entire set of relations is now in second normal form.

Third normal form
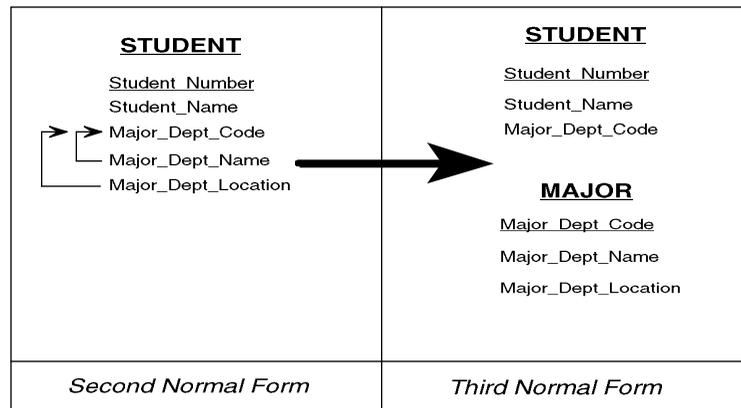
Figure 2I - Further normalizing our example to third normal form. The student relation violated third normal form due to non-key attributes which are dependent on other non-key attributes, as shown by the arrows. These attributes are broken out into new relations, the original relation as a foreign key to the new relation. This foreign key attribute is MAJOR_DEPT-CODE in the STUDENT relation.

One remaining transformation must be done on our data from Figure H in order to get all the relations into third normal form. As shown in Figure I, the STUDENT relation violates third normal form, and must be transformed into two tables. The two non-key attributes Major_Dept_Name and Major_Dept_Location depend on another non-key attribute, Major_Dept_Code. Even though the are currently in the STUDENT table, they do not depend on the Student_Number primary key directly, but are dependent through the Major_Dept_Code, which is dependent on the primary key. This is called a transitive dependency, and is removed by creating a new table with a primary key of the transitive attribute in the original relation. This transitive attribute then becomes a foreign key to the new table. The attributes in the original relation that were transitively dependent are now placed in the new table, where they are fully dependent on the primary key only. The MAJOR relation in Figure I is formed through this process. The data originally described in Figure G is now entirely in third normal form.

Denormalization

Now that we have looked at the benefits associated with data normalization, a few words about denormalization are in order. Denormalization is the process of taking a database design out of third normal form (or some higher form) for performance reasons. One may denormalize by either redundantly storing prime attributes, or by adding aggregate attributes to the database.

Some database designers routinely denormalize during logical database design. However, the only reason to denormalize is to achieve greater performance. In addition, because of the added complexities of a denormalized database (refer to chapter 7 for further details), denormalization should only occur as a last resort after all other means of increasing performance have failed. Consequently, denormalization should only take place during physical design control. The logical design should still be maintained in (at least) third normal form in order to facilitate clear thinking about its structure and meaning.

Chapter 3

Internal Schema Creation

In this chapter we cover the tasks necessary to create the first-cut physical database design (internal schema). The database constructed in this chapter is an initial prototype only. The entities, relationships and the other logical constructs of the conceptual schema are mapped to SQLBase physical constructs without regard to performance or capacity issues. The objective of this step is to produce SQLBase compliant SQL DDL (data definition language) which represents the initial physical database design. The first-cut internal schema is constructed by applying several simple conversion rules to the entity relationship diagram. These rules are discussed in detail in this chapter.

Many of the CASE tools discussed in chapter 1 automatically generate the first-cut internal schema. You may find using a CASE tool which converts an entity relationship diagram to SQLBase compatible SQL DDL beneficial.

Building the initial internal schema

Following completion of the logical database design, the designer must convert the entity relationship diagram (ERD) to SQL data definition language (DDL) which specifies the desired physical database to SQLBase. This transformation may be done either manually by the designer, or in an automated fashion through the facilities of a CASE tool.

The following steps are required to perform the conversion:

- Create the base tables. These are the basic building blocks of the database, and are derived from the entities contained within the ERD. Each possesses a number of characteristics which must be considered by the designer:
    - The table-wide options specified on the CREATE TABLE statement, such as owner, name, and PCTFREE.
    - The list of columns contained within the table, which is derived from the attributes of the original ERD entity.
    - The specific data type associated with each column. These may have to be determined now, or may have been specified as part of the ERD domain specification.
- Create any required junction tables needed to resolve many-to-many relationships in the ERD.
- Add referential integrity (RI) specification to the DDL. This implements the relationships specified in the ERD, and requires the following steps:
    - Identify the primary key of each table within the CREATE TABLE statement.
    - Build unique indexes for each primary key of each table.
    - Build ALTER TABLE statements which identify the foreign keys contained within child tables.

Creating tables

The first step in building the ANSI/SPARC internal schema is to identify the relational tables which will be managed by SQLBase. The primary input to this task is a completed ERD containing entities and relationships. On completion, a set of CREATE TABLE skeleton statements may be coded that will serve as the starting point for the addition of columns and other details.

A base table will be created for each entity on the ERD. Base tables are the major information-bearing objects within a relational database, and contrast with views in

that no projection or selection operations may restrict the visibility of any columns or rows.

For each base table identified, define a long identifier which uniquely names the table within this database. This name should follow site standards for base table names while also being as descriptive as possible about the business information contained within the table. Also, use an explicit authorization-id instead of allowing the authorization-id to default to the user name of the current user. Select an authorization-id that meets your site standards, and can serve as a consistent owner for all application objects within the database.

When you have finished, verify that the number of tables matches the count of entity boxes in the E-R diagram.

Adding columns to tables

The columns needed by the tables you have created are present in the ERD as attributes. These attributes need to be transformed into column specifications in the middle of the CREATE TABLE statements you will code. The attribute names themselves must be transformed into column names, and must adhere to the SQLBase long identifier restrictions. This is a good time to check that the root portions of the column names (that part between underscore delimiters) conform to the standardized list of abbreviations. New abbreviations should be evaluated for acceptability and added to the list. Column names which are not clear and consistent may cause confusion for programmers and users throughout the life of the application.

Data typing for columns

After identifying the columns, data types need to be added. This task will be greatly simplified if the ERD contains domains and the attributes have all been given domain assignments. First convert each domain into an appropriate data type, if necessary. Some ERDs may have domains which are already specified in terms of RDBMS data types as implemented by SQLBase. These require no further transformation. When performing this conversion, consider the following factors:

- Internally, SQLBase stores all columns as variable length fields. Therefore, choosing between some data types becomes a matter of documentation, which may be useful for range validation purposes. As an example, defining a classification code column such as LEDGER_SUBCODE in a general ledger application as CHAR(3) documents the fact that values must always consist of three characters (perhaps falling into the range "10A" to "99Z") more clearly than defining the data type as VARCHAR(3). Each of these two data type definitions results in an identical internal implementation in SQLBase, however.

- If the usage of a numeric data item requires it to be of a fixed width, use the DECIMAL data type with the required precision and an appropriate scale. If an integer is desired, use DECIMAL with a scale of zero. This helps to communicate external formatting requirements more clearly.

- Avoid the use of INTEGER and SMALLINT data types in general. Use these only for internal control fields such as cycle counters. Formatting these fields for external display use can be problematic due to the indeterminate display width.

- Avoid using LONG VARCHAR unless absolutely required. Each LONG VARCHAR column must be stored on a separate page, apart from the rest of the row to which it belongs. Also, each of these columns will occupy at least an entire page by itself, regardless of how much free space on the page may be wasted. Therefore, using LONG VARCHAR increases both the number of I/O operations to read a row, as well as the amount of space required to store the data. In addition, some @ functions cannot be used with LONG VARCHAR columns.

- Avoid using CHAR or VARCHAR for data that will always be numeric. Using a numeric data type will ensure that the field will always be populated with valid numeric data.

- Avoid the use of the REAL (or FLOAT or DOUBLE PRECISION) data type except for situations calling for scientific measurement information containing a wide range of values. The exponent-mantissa format of these data types lacks the exact precision associated with other numeric data types, and also causes floating point operations to be performed for their manipulation. On most CPUs, floating point arithmetic incurs greater cycle times than integer operations.

- Use the DATE and TIME formats when expressing chronological data.

- Use DATETIME (or TIMESTAMP) for control purposes. Many designers include a TIMESTAMP field for tracking row creation and updates.

Null usage

When defining data types for columns, you will also need to specify the column's ability to assume the null value. Note that null is a special value that indicates either "unknown" or "not applicable". Users and programmers not accustomed to the use of nulls may have problems remembering to include host variable declarations and tests for nulls in columns that allow them. Nulls can also cause problems when they are in columns that are used for joining tables together. When the join is performed, any rows that contain nulls in the join column of either table will not be returned to the result set. This "missing data" may be disconcerting to a user of the query, who may have expected to see all of the rows of at least one of the tables. Of course, an outer join specification (using the "+" operator) may be used to force all rows of one of the

tables to be returned, but this requirement could be easily overlooked. Consider the following factors when deciding whether to allow a null value in a column:

- Columns participating in primary keys should always be defined with NOT NULL, since they are required to be populated with unique values for each row.

- Foreign keys should generally be defined with NOT NULL. Whenever a child table is dependent on a parent, the foreign key to the parent table cannot be null, since the existence of a child occurrence without a parent would violate the dependency rule of the relationship. Only foreign keys to tables with optional relationships can be considered as candidates for the use of nulls, to signify that no relationship exists. Also, foreign keys defined with a delete rule of SET NULL must be defined as nullable.

- Use NOT NULL WITH DEFAULT for columns with DATE, TIME, or DATETIME data types to allow current date and time data to be stored in the field automatically. This is particularly useful for row creation timestamps.

- Allow nulls for those columns that will actually have an anticipated need for the unknown or inapplicable meaning of the null value. The null value works especially well in numeric fields that may be subjected to aggregation uses such as SUM or AVERAGE.

- Use NOT NULL WITH DEFAULT for all columns not meeting one of the above criteria.

Example 1

We will carry forward the example ERD used in the previous chapter to demonstrate diagramming styles to show the transformation from external to internal schema. The following SQL statements represent the current state of the design at this point. Note that a site standard requires base table names to be suffixed with a "_T" to facilitate easy identification.

```
CREATE TABLE OESYS.CUSTOMER_T (
CUSTOMER_NO DECIMAL(6,0) NOT NULL,
CUST_NAME VARCHAR(45) NOT NULL,
CUST_ADDR1 VARCHAR(35) NOT NULL,
CUST_ADDR2 VARCHAR(35) NOT NULL WITH DEFAULT,
CUST_ADDR3 VARCHAR(35) NOT NULL WITH DEFAULT,
CUST_CITY VARCHAR(45) NOT NULL,
CUST_STATE_CD CHAR(2) NOT NULL,
CUST_ZIP DECIMAL(5,0) NOT NULL WITH DEFAULT)
PCTFREE 10;
CREATE TABLE OESYS.ORDER_T
(ORD_SALES_CENTER_NO DECIMAL(2,0) NOT NULL,
ORD_SEQ_NO DECIMAL(6,0) NOT NULL,
```

```
ORD_CUSTOMER_NO DECIMAL(6,0) NOT NULL,
ORD_PLACED_DATE DATE NOT NULL WITH DEFAULT,
ORD_CUST_PO VARCHAR(35) NOT NULL WITH DEFAULT,
ORD_CUST_PO_DATE DATE NOT NULL WITH DEFAULT,
ORD_STATUS CHAR(1) NOT NULL,
ORD_CHANGED_DATE DATE,
ORD_INVOICED_DATE DATE)
PCTFREE 10;
CREATE TABLE OESYS.ORDER_ITEM_T
(OITM_SALES_CENTER_NO DECIMAL(2,0) NOT NULL,
OITM_SEQ_NO DECIMAL(6,0) NOT NULL,
OITM_LINE_NO DECIMAL(3,0) NOT NULL,
OITM_PRODUCT_NO DECIMAL(9,0) NOT NULL,
OITM_ORDER_QTY DECIMAL(3,0) NOT NULL,
OITM_SHIPPED_QTY DECIMAL(3,0) NOT NULL WITH DEFAULT,
OITM_MEASURE_UNIT CHAR(2) NOT NULL,
OITM_COST_AMT DECIMAL(5,2) NOT NULL,
OITM_SELL_PRICE_AMT DECIMAL(6,2) NOT NULL)
PCTFREE 10;
CREATE TABLE OESYS.PRODUCT_T
(PRODUCT_NO DECIMAL(9,0) NOT NULL,
PROD_DESC VARCHAR(50) NOT NULL,
PROD_DESC_SHORT VARCHAR(15) NOT NULL WITH DEFAULT,
PROD_MEASURE_UNIT_SMALL CHAR(2),
PROD_COST_SMALL_AMT DECIMAL(5,2),
PROD_MEASURE_UNIT_MED CHAR(2),
PROD_COST_MED_AMT DECIMAL(5,2),
PROD_MEASURE_UNIT_LARGE CHAR(2),
PROD_COST_LARGE_AMT DECIMAL(5,2),
PROD_STATUS CHAR(1) NOT NULL,
PROD_ON_HAND_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT,
PROD_AVAIL_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT,
PROD_ON_ORDER_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT)
PCTFREE 10;
```

Resolving many-to-many relationships

If the ERD contains any many-to-many relationships, these cannot be directly implemented in the relational model without creating an intermediate table. This process is called *resolving the relationship*. The relationships affected are those that exist between two tables that have a maximum cardinality of many on both sides of the relationship. These relationships are transformed into two one-to-many relationships with a new table which serves as a placeholder for the purpose of representing occurrences of related parent tables. Figure A, below, contains an

example of resolving a many-to-many relationship between CUSTOMER and SALESMAN.
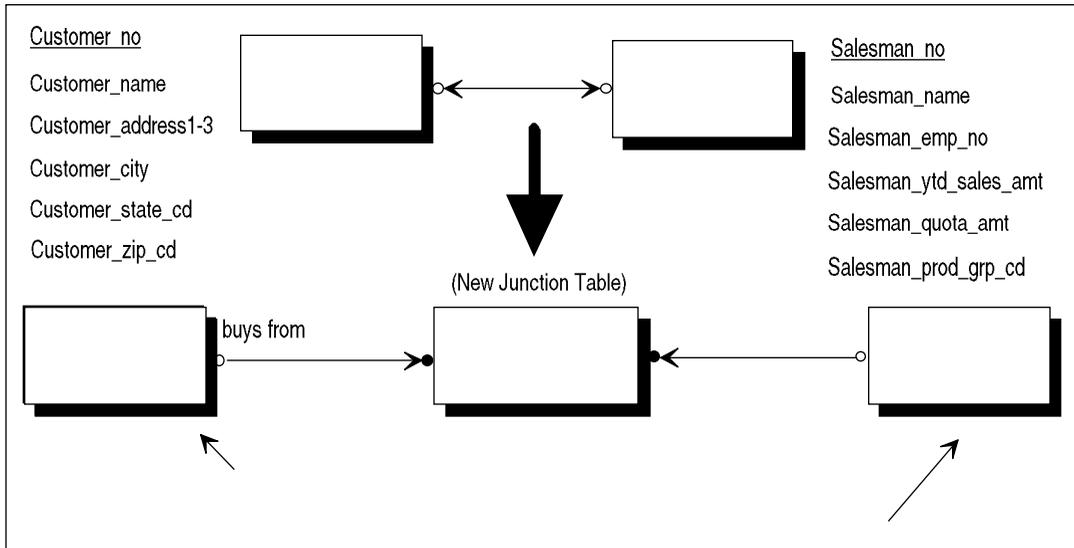
Figure 3A - An example of resolving a many-to-many relationship through the creation of a junction table. The junction table CUST_SLSMAN contains the primary key of each parent and is also dependent on both. Examples of other columns that could be contained within the junction table would be SALESMAN_DATE_ASSIGNED or SALES_YTD_AMT. Columns within the CUSTOMER and SALESMAN tables remain unchanged. Columns which are primary keys are underlined, while foreign keys columns are in italics.

Resolving the relationship

1. Create a new table, called a *junction table*, between the two existing tables. Often, designers will name this table as a conjunction of the names of the two tables involved in the relationship. Figure A shows an example of this by creating a CUST_SLSMAN junction table between the CUSTOMER and SALESMAN tables.

2. Add columns to the new table. The required columns are the primary keys of *both* tables in the relationship. Be sure to include all columns of any composite primary keys.

3. Define primary and foreign keys for the junction table. The primary key will be a composite key made up of the concatenation of the primary keys of both tables in the relationship. The foreign keys will be each parent table's half of the primary key, which references back to that parent table. This completes the transformation

of the single many-to-many relationship into two one-to-many relationships with an intermediate junction table.

Now that the many-to-many relationship has been resolved, take a moment to think about other attributes that may exist within the relationship. This information, called *junction data*, can be added as columns to the new table that resolved the relationship. There are often data items that will appear in these tables as the project progresses and user requirements become better known. Two examples of such columns are given in Figure A.

Adding referential integrity

Referential integrity is the verification that links between table rows are valid at all times. These links are the physical implementation of the relationships modeled in the Entity-Relationship diagram during logical design. The links are implemented through common values contained in primary and foreign keys in the related rows. These rows may exist in the same table, or in separate tables. The table containing the primary key is said to be the *parent* table in the relationship, and the table containing the foreign key is termed the *child* table. (Although these terms "parent" and "child" are widely accepted in relational terminology, they connote a hierarchical, dependent relationship between the tables. Of course, a relationship implemented through primary and foreign keys in SQLBase, or any other RDBMS, could be of several types other than a hierarchy.) It is important to remember that these terms are only meaningful in the context of a single relationship, since a child table in one relationship could be a parent table in another relationship, or vice-versa.

By defining a table's primary and foreign keys when you create the table, SQLBase is able to enforce the validity of relationships within the DBMS itself without requiring any special coding conventions by the user. For instance, without system-enforced referential integrity, an order entry program would have to retrieve a row from the customer table (corresponding to the customer number entered by the operator) in order to verify that the customer actually existed. With referential integrity, the child order table contains the customer number as a foreign key to the parent customer table and the program no longer needs to check for the existence of the customer. When the order row is inserted, the DBMS will realize that the referential integrity definition requires it to perform a read of the customer table to verify that a row with a primary key that matches the contents of the foreign key in the order record exists. If this row is found in the customer table, then the order row can be inserted successfully. Otherwise the insert will fail and an appropriate SQL error code will be returned to the program to indicate that a referential integrity error occurred.

Sometimes concerns are raised about the possible performance penalties that can be incurred through the use of system defined and enforced referential integrity. While there may be some small minority of cases where user enforced referential integrity

could be performed more efficiently, in the vast majority of cases, the DBMS itself is able to maintain a higher level of consistency with a lower cost. The two most important factors contributing to this are its ability to perform integrity checks for *all* database accesses, and its ability to *always* use the most efficient access technique available to perform the referential checks.

Selection of appropriate referential integrity constraints requires an understanding of the business rules governing the way applications can delete and update data. Use of this knowledge allows the database designer to select the appropriate delete rules from among the three options available in SQLBase. These options are:

- *Restrict*, which disallows deletion of parent rows if any child rows exist. This is the default, and is generally the safest option.
- *Set Null*, which sets the foreign key values of child rows to null when parent rows are deleted.
- *Cascade*, which deletes child rows when parent rows are deleted.

There are also implications of delete rules resulting from the transitive relationships between three or more tables that can cause restrictions to be placed on the designer's selection. Furthermore, some DML coding options can be constrained through the use of referential integrity. The designer must consider all these factors when selecting appropriate referential integrity constraints for the database.

Defining keys

Name the columns comprising the primary key of each table in the PRIMARY KEY clause of the CREATE TABLE statement. Since the primary key is also required to be unique, the next step is required before the primary key definition is complete.

Following each table's CREATE TABLE statement, code a CREATE UNIQUE INDEX statement that builds a unique index on the primary key columns of the table. Include only those columns that are in the primary key, and index them in the same order as they are specified in the PRIMARY KEY definition clause of the CREATE TABLE statement. Consider using some indicator (such as a preceding, or following, "XPK") in the index name to signify that this is a primary key index, since inadvertently dropping this index would have detrimental effects on the desired referential integrity for the database. It is important to remember that *both* the PRIMARY KEY clause of the CREATE/ALTER TABLE statement *and* the CREATE UNIQUE INDEX are required before SQLBase will consider the primary key definition for the table to be complete.

At the end of the DDL file (following all CREATE TABLE/INDEX statements), code an ALTER TABLE statement that adds the foreign key definitions needed for each child table in the database. Adding foreign keys through this ALTER technique guarantees that each table's primary key definition is completed (with its associated index) prior to being referenced in a foreign key clause by some other table.

Example 2

Continuing from example 1, we now incorporate the required referential integrity definitions to define the relationships in the ERD. These modifications include the PRIMARY KEY clause of the CREATE TABLE statement, the CREATE UNIQUE index statement for each primary key, and finally, the ALTER TABLE statements needed to define the foreign keys.

```
CREATE TABLE OESYS.CUSTOMER_T (
CUSTOMER_NO DECIMAL(6,0) NOT NULL,
CUST_NAME VARCHAR(45) NOT NULL,
CUST_ADDR1 VARCHAR(35) NOT NULL,
CUST_ADDR2 VARCHAR(35) NOT NULL WITH DEFAULT,
CUST_ADDR3 VARCHAR(35) NOT NULL WITH DEFAULT,
CUST_CITY VARCHAR(45) NOT NULL,
CUST_STATE_CD CHAR(2) NOT NULL,
CUST_ZIP DECIMAL(5,0) NOT NULL WITH DEFAULT,
PRIMARY KEY(CUSTOMER_NO))
PCTFREE 10;
CREATE UNIQUE INDEX OESYS.XPKCUSTOMER
ON OESYS.CUSTOMER_T
(CUSTOMER_NO)
PCTFREE 10;
CREATE TABLE OESYS.ORDER_T (
ORD_SALES_CENTER_NO DECIMAL(2,0) NOT NULL,
ORD_SEQ_NO DECIMAL(6,0) NOT NULL,
ORD_CUSTOMER_NO DECIMAL(6,0) NOT NULL,
ORD_PLACED_DATE DATE NOT NULL WITH DEFAULT,
ORD_CUST_PO VARCHAR(35) NOT NULL WITH DEFAULT,
ORD_CUST_PO_DATE DATE NOT NULL WITH DEFAULT,
ORD_STATUS CHAR(1) NOT NULL,
ORD_CHANGED_DATE DATE,
ORD_INVOICED_DATE DATE,
PRIMARY KEY(ORD_SALES_CENTER_NO,ORD_SEQ_NO))
PCTFREE 10;

CREATE UNIQUE INDEX OESYS.XPKORDER
ON OESYS.ORDER_T
(ORD_SALES_CENTER_NO,ORD_SEQ_NO)
PCTFREE 10;
CREATE TABLE OESYS.ORDER_ITEM_T (
OITM_SALES_CENTER_NO DECIMAL(2,0) NOT NULL,
OITM_SEQ_NO DECIMAL(6,0) NOT NULL,
OITM_LINE_NO DECIMAL(3,0) NOT NULL,
OITM_PRODUCT_NO DECIMAL(9,0) NOT NULL,
OITM_ORDER_QTY DECIMAL(3,0) NOT NULL,
OITM_SHIPPED_QTY DECIMAL(3,0) NOT NULL WITH DEFAULT,
```

```
                OITM_MEASURE_UNIT CHAR(2) NOT NULL,
                OITM_COST_AMT DECIMAL(5,2) NOT NULL,
                OITM_SELL_PRICE_AMT DECIMAL(6,2) NOT NULL,
                PRIMARY KEY (ITM_SALES_CENTER_NO,OITM_SEQ_NO,
                OITM_LINE_NO))
                PCTFREE 10;

                CREATE UNIQUE INDEX OESYS.XPKORDERITEM
                ON OESYS.ORDER_ITEM_T
                (OITM_SALES_CENTER_NO,OITM_SEQ_NO,OITM_LINE_NO)
                PCTFREE 10;
                CREATE TABLE OESYS.PRODUCT_T (
                PRODUCT_NO DECIMAL(9,0) NOT NULL,
                PROD_DESC VARCHAR(50) NOT NULL,
                PROD_DESC_SHORT VARCHAR(15) NOT NULL WITH DEFAULT,
                PROD_MEASURE_UNIT_SMALL CHAR(2),
                PROD_COST_SMALL_AMT DECIMAL(5,2),
                PROD_MEASURE_UNIT_MED CHAR(2),
                PROD_COST_MED_AMT DECIMAL(5,2),
                PROD_MEASURE_UNIT_LARGE CHAR(2),
                PROD_COST_LARGE_AMT DECIMAL(5,2),
                PROD_STATUS CHAR(1) NOT NULL,
                PROD_ON_HAND_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT,
                PROD_AVAIL_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT,
                PROD_ON_ORDER_QTY DECIMAL(4,0) NOT NULL WITH DEFAULT,
                PRIMARY KEY(PRODUCT_NO))PCTFREE 10;
                CREATE UNIQUE INDEX OESYS.XPKPRODUCT
                ON OESYS.PRODUCT_T (PRODUCT_NO)
                PCTFREE 10;
                ALTER TABLE OESYS.ORDER_T
                FOREIGN KEY PLACEDBY (ORD_CUSTOMER_NO)
                REFERENCES OESYS.CUSTOMER_T
                ON DELETE RESTRICT;
                ALTER TABLE OESYS.ORDER_ITEM_T
                FOREIGN KEY FKWITHIN (OITM_SALES_CENTER_NO,
                OITM_SEQ_NO)
                REFERENCES OESYS.ORDER_T
                ON DELETE CASCADE,
                FOREIGN KEY FKFOR (OITM_PRODUCT_NO)
                REFERENCES OESYS.PRODUCT_T
                ON DELETE RESTRICT;
```

Chapter 4

External Schema Creation

In this chapter we cover the creation of the initial external schema. This requires coding the data definition language necessary to build views on the underlying tables of the internal schema, called base tables. It also means establishing the procedures needed to allow the external schema to evolve as requirements for new user views of the data become known. Security mechanisms may also be put into place to insure that all data access is accomplished through the external schema, rather than by direct access to base tables. In this way, the base tables are allowed to change while isolating the user views from some of the impact of these modifications.

Using views as the external schema

In the ANSI/SPARC three schema architecture, the external schema is used to isolate the user's (or program's) data requirements from the physical implementation. This isolation is intended to benefit users by allowing them to be unaware of the physical aspects of the database structure, which were possibly decided for reasons of physical DBMS and hardware features and limitations. This external schema is also isolated from the conceptual schema so that the user's data requirements are not impacted by any esoteric modeling processes (such as normalization) or organization-wide data policies that may seem to have unnatural effects on their own data. In other words, the goal of the external schema is to allow data to be presented to the user in as natural and simple a way as possible, and to allow for their data requirements to change dynamically as the business needs evolve. The facility that can be used to accomplish the goal of the external schema in RDBMSs, such as SQLBase, is the SQL *view*.

However, with the current SQL standard used by most RDBMSs, including SQLBase, the support for this external schema is limited. The current SQL standard fails to differentiate between conceptual and physical objects, and currently fails to provide an external view mechanism that transcends the ability of the physical design to impact user views. However, just because these limitations currently exist does not mean that they will not be solved in the future as the SQL standard evolves. The relational view mechanism can currently be used to isolate users' data requirements from many types of physical changes. Theorists and researchers are currently working to extend the ability of views to be isolated from physical database dependencies. Much of the current limitations manifest themselves as certain view DML limitations, the most notable being the inability to process any update commands when a view accesses multiple tables, such as in a join.

Examples of physical schema level database changes that can be 'hidden' by views include the following:

- Object owners changes. Changing the owner of views can have an impact, however.

- Column names changes. The view definitions can map the old column names to the new names, thereby making column name changes invisible to users' views.

- Table names changes. These can be handled the same as column names mentioned above.

- Table column additions, since these new columns need not be added to existing views.

- Table column re-ordering. Views can explicitly order their columns independent of the underlying table's column ordering.

- Index creation, modification, or removal. Of course, these are also 'invisible' to base table users. Note that this does not apply to primary key indexes, where changes or deletions can impact referential integrity, which is not an external schema impact but rather has an impact on the conceptual schema.

- The combination of two or more tables into one. This assumes that the primary keys of the two tables are the same.

- The splitting of a table into two or more tables. This can be hidden if the table is split so that no view has to access both new tables to access its required columns.

This list could expand significantly as new releases of SQLBase become available. By enforcing the use of the external schema through the view mechanism now, an organization will be positioned to take advantage of present and future benefits accruing to the three schema architecture without incurring costly source code changes.

The primary costs incurred by the extensive use of views fall into two administrative areas. The impact on the database administration area is the overhead of creating and managing the many views that will exist. The second impact is on the task of security administration, since more authorizations will need to be managed to ensure that all database access is through views as opposed to base tables.

The view administration impact can be minimized by adopting good conventions for managing the views. Views should be named in a meaningful manner and cataloged according to the business data they supply. This will allow DBAs to maximize the reuse of views by facilitating the identification of appropriate existing views as an alternative to user requests to create new views. In many cases, programmers and users may be able to locate a view on their own that provides the data they need without involvement from the DBA.

The major pitfalls to avoid in view administration are the temptation to delegate the responsibility of creating views too widely and the use of a very few broad views to fulfill all access requests. The first pitfall comes about when the DBA grants the ability to create views to nearly everyone in an attempt to eliminate the work of creating views to DBA. While a few trusted individuals can be given this ability (if they are carefully monitored), when many users have this ability, they may succumb to the practice of simply creating a new view for every new access requirement they have, without first searching for an existing view to satisfy their requirement.

The second pitfall occurs when the DBA avoids the workload of view creation by satisfying every view request with the name of a view that mirrors the base table definition of each table needed by the user. While this action is sometimes appropriate, the DBA must be willing to create new views when no existing view definition comes close to satisfying the new request. The DBA needs to find an appropriate balance between these extremes by comparing the number of columns

and rows that may be returned from a view with the result set that would exactly satisfy the user's request. For example, a rule of thumb could be set that specifies a desired result set being within 75% of the columns and rows returned from a view as acceptable.

The security administration work involved in enforcing the use of views can be minimized as well. Techniques such as adopting automated procedures for the creation and execution of the data control language (DCL, the SQL GRANT and REVOKE statements) can greatly improve the consistency of security authorizations as well as ease the workload. Making routine use of these procedures at key points in the application development life cycle lowers the requirement for unscheduled, on-demand security administration.

Creating the initial external schema

To create the initial external schema for a new database, start by creating a view that mirrors each base table in the internal schema. These views can be used by programmers when they are coding DML statements to initially populate the database, since they will have no update restrictions placed on them. These views are also likely to be widely used by application programs that perform insert and update operations. The name of each view should reflect the fact that it is a view of the underlying base table, such as PART for the mirror view of the base table PART_T (where the _T signifies a base table).

This mirror view should *explicitly* name each column in the base table, as opposed to using the 'SELECT *' form to implicitly reference table columns, and contain no WHERE clause that restricts row retrieval. If 'SELECT *' is used to implicitly reference table columns, then application programs may require modifications when columns are added or dropped from the base table. This is because application programs must *bind* attributes returned from SQL statements to application program variables and these bind clauses depend on a strict number and ordering of attributes returned from SQL statements.

Additional views can be created for programs which perform database queries when the data requirements of the program are fully documented. Also, views for ad-hoc database users can also be created when their data requirements have been adequately described.

While these CREATE VIEW statements build the initial external schema view of the database, this task is best viewed as a continual evolutionary process. As additional information requirements become known, additional views will be added to the external schema to satisfy these requirements in an efficient and flexible manner. In order to facilitate the administration of the external schema's DDL, these views should be kept together in a subdirectory where they can be conveniently accessed and re-executed, should the need arise.

No programs or queries should be written which access the base tables, either by the DBAs or anyone else. In order to enforce the use of the external schema as the only means of accessing the database, do not grant authorization on the base tables to anyone other than DBAs. Even the DBAs should exercise restraint in their use of the base tables, using them only for utilities which exclude views from being named. The views themselves can either be granted to PUBLIC or to individual users, depending on how restrictive the organization's security policies are.

While these security restrictions may seem excessive to some, it is only through their usage that the goal of using the external schema to isolate user data views from internal implementation can be approached and eventually achieved.

Example

Carrying forward the example of the previous two chapters, we will now create the mirror views of the base tables:

```
CREATE VIEW OESYS.CUSTOMER
AS SELECT
CUSTOMER_NO,
CUST_NAME,
CUST_ADDR1,
CUST_ADDR2,
CUST_ADDR3,
CUST_CITY,
CUST_STATE_CD,
CUST_ZIP
FROM OESYS.CUSTOMER_T;
CREATE VIEW OESYS.ORDERS
AS SELECT
ORD_SALES_CENTER_NO,
ORD_SEQ_NO,
ORD_CUSTOMER_NO,
ORD_PLACED_DATE,
ORD_CUST_PO VARCHAR(35),
ORD_CUST_PO_DATE DATE,
ORD_STATUS CHAR(1),
ORD_CHANGED_DATE,
ORD_INVOICED_DATE
FROM OESYS.ORDER_T;
CREATE VIEW OESYS.ORDER_ITEM
AS SELECT
OITM_SALES_CENTER_NO,
OITM_SEQ_NO,
OITM_LINE_NO,
OITM_PRODUCT_NO,
```

```
                    OITM_ORDER_QTY,
                    OITM_SHIPPED_QTY,
                    OITM_MEASURE_UNIT,
                    OITM_COST_AMT,
                    OITM_SELL_PRICE_AMT
                    FROM OESYS.ORDER_ITEM_T;
                    CREATE VIEW OESYS.PRODUCT
                    AS SELECT
                    PRODUCT_NO,
                    PROD_DESC,
                    PROD_DESC_SHORT,
                    PROD_MEASURE_UNIT_SMALL,
                    PROD_COST_SMALL_AMT,
                    PROD_MEASURE_UNIT_MED,
                    PROD_COST_MED_AMT,
                    PROD_MEASURE_UNIT_LARGE,
                    PROD_COST_LARGE_AMT,
                    PROD_STATUS,
                    PROD_ON_HAND_QTY,
                    PROD_AVAIL_QTY,
                    PROD_ON_ORDER_QTY
                    FROM OESYS.PRODUCT_T;
```

Now we will create two views for known user requirements. The first view is for the aged open order line inquiry and the second view is for the daily cancelled order report:

```
CREATE VIEW OESYS.OPENLINES
AS SELECT
ORD_SALES_CENTER_NO,
ORD_SEQ_NO,
OITM_LINE_NO,
OITM_PRODUCT_NO,
OITM_ORDER_QTY - OITM_SHIPPED_QTY AS ORDER_QTY,
ORD_PLACED_DATE
FROM OESYS.ORDERS,OESYS.ORDER_ITEM
WHERE ORD_SALES_CENTER_NO = OITM_SALES_CENTER_NO
AND ORD_SEQ_NO = OITM_SEQ_NO
AND ORD_STATUS = 'O'
AND OITM_ORDER_QTY > OITM_SHIPPED_QTY;
CREATE VIEW OESYS.CANCELED_ORDERS
AS SELECT
CUSTOMER_NO,
CUST_NAME,
ORD_SALES_CENTER_NO,
ORD_SEQ_NO,
ORD_CUST_PO,
ORD_CUST_PO_DATE,
```

```
ORD_PLACED_DATE,
ORD_CHANGED_DATE,
OITM_SELL_PRICE_AMT,
OITM_ORDER_QTY - OITM_SHIPPED_QTY AS OPEN_ORDER_QTY,
OITM_PRODUCT_NO,
PROD_DESC
FROM OESYS.CUSTOMER,OESYS.ORDER,OESYS.ORDER_ITEM, OESYS.PRODUCT
WHERE ORD_CHANGED_DATE = CURRENT DATE
AND CUSTOMER_NO = ORD_CUSTOMER_NO
AND ORD_SALES_CENTER_NO = OITM_SALES_CENTER_NO
AND ORD_SEQ_NO = OITM_SEQ_NO
AND ORD_STATUS = 'C'
AND OITM_ORDER_QTY > OITM_SHIPPED_QTY
AND OITM_PRODUCT_NO = PRODUCT_NO;
```

Chapter 5

Transaction Definition

This chapter:

- Explains the purpose of well-defined transactions:
    - Database design benefits
    - Data integrity considerations
- Covers the detailed information required for each transaction:
    - Identifying information
    - Categorical information
    - Volume information
    - Performance requirements
    - Data requirements

Benefits of defining transactions

Transaction definitions are required for proper physical database design and to ensure data integrity.

Physical database design

The primary goal of *physical* database design is to ensure that the database provides the required level of performance. Typically, database performance is measured in terms of transaction performance. So, the objective of physical database design is to ensure every database *transaction* meets or exceeds its performance requirements.

The process of physical database design involves mapping the logical data structures (entities and relationships in an ER diagram) to the physical constructs of SQLBase. For example, entities in the logical database model become one or more clustered or non-clustered SQLBase tables. The design criteria which guides this mapping are transaction performance requirements and volume statistics. The deciding factor for the designer choosing between the various physical constructs that may be used is their effect on transaction performance. For any particular construct, the effect will probably enhance the performance of some transactions while degrading the performance of others. So, physical database design can be viewed as an optimization problem. The goal is to select the SQLBase physical constructs so that every transaction meets or exceeds its performance requirements.

Hence, well-defined transactions are a prerequisite for physical database design. Without this information, the database designer has no rule to guide in the selection of physical database constructs. For example, the database designer will be unable to determine whether a clustered or non-clustered table will result in better performance. A similar argument can be made for index selection.

Data integrity

Database transactions usually involve *multiple* database accesses. For example, consider the banking transaction of withdrawing $1000 from a savings account and depositing the funds into a checking account. This transaction entails two separate database operations: subtracting $1000 from the savings table and adding $1000 to the checking table. If either one of these database operations successfully completes but the other does not, then the account balances would be wrong. Both of these operations must complete successfully or neither of them should complete successfully.

Thus, a database transaction is a logical unit of work which causes the database to advance from one consistent state to another. In other words, a transaction moves the database from one state which does not violate the data integrity requirements of the database to another state which does not violate the data integrity requirements of the

database. In addition, a database transaction is a grouping of logically related database operations so that all of the database accesses should complete successfully or none of them should be applied to the database. Otherwise, if only some of the database accesses complete successfully, the database is left in an inconsistent state.

So, in order to ensure the integrity of the database, it is necessary to rigorously define the database transactions. This chapter describes how to define transactions, and identifies the information which is typically included in transaction definitions.

Defining transactions

Transaction definitions can take many forms. Some organizations may possess an upper CASE tool with a transaction definition facility as part of a data repository. Other organizations may choose to define transactions using paper-based or automated forms. Regardless of the media, all good transaction definitions include several key ingredients. Each of these is discussed below.

Transaction name, number, and description

The first step in defining transactions is to uniquely identify each database transaction. This is accomplished by assigning a name and unique identifier to each transaction.

In addition, a short narrative should be composed, referred to as the transaction description, which describes the transaction in business terms:

- The description should specify what the transaction accomplishes for the user rather than how it is accomplished.
- The description should be understandable by users (it should not include technical jargon).

The purpose of transaction names and descriptions is to provide users and MIS staff with a means for distinguishing one transaction from another.

Example: Transaction Number: 001

Example: Transaction Name: Transfer Funds

Example: Transaction Description: The Transfer Funds transaction verifies that sufficient funds exist in the account to be debited. If sufficient funds exist, the transaction debits the account by the specified amount and credits the other account for the same amount.

In OnLine Transaction Processing (OLTP) systems, the database transactions are known ahead of time. For these types of systems, a one-to-one relationship will exist between the transaction definitions and the actual transactions submitted to SQLBase.

In Decision Support Systems (DSS), transactions are not known ahead of time. These systems are characterized by ad hoc and exception reporting. Therefore, it is not possible to describe each and every transaction. With these systems, the transaction definitions are illustrative of the actual transactions which will be submitted to SQLBase once the system is implemented. DSS systems require the database designer to predict the type of transactions which the users will likely run.

Transaction type and complexity

Frequently, transaction definitions also categorize transactions in terms of type and complexity. For example, transactions may be defined as either "Batch" or "Online" and given a complexity rating of either "High," "Medium," or "Low." This information is useful for producing listings or totals of like transactions. For example, management may require an estimate of the time required to complete physical database design. To accomplish this, one might need to know how many low, medium, and high complex transactions a system has. The database for a system with a large percentage of high complexity transactions would take longer to design than a system with mostly low complexity transactions.

The complexity rating is a matter of judgement, but should be based on the degree of difficulty the database designer is likely to have meeting the performance requirements for the transaction (see below). A transaction of high complexity has at least two of the following characteristics:

- Contains many SQL statements (more than 10).
- Contains WHERE clauses with many predicates (for example, three or more table joins or subqueries).
- The SQL statements affect many rows (> 100).

On the other hand, a low complexity transaction has the following characteristics:

- Contains few SQL statements (three or fewer).
- Contains WHERE clauses with only one or two predicates.
- The SQL statements affect few rows (< 25).

Transaction volumes

Transaction definitions must also include volume information. This typically includes the average and peak frequency of each transaction. For example, a branch bank might estimate that the funds transfer transaction described in the introduction of this chapter will occur 50 times an hour with peak loads reaching 65 transactions per hour.

The transaction volume statistics are extremely important for physical database design; consequently, care should be taken to be as accurate as possible. For example,

the database designer will handle a transaction which occurs 1000 times per hour quite differently from one that will occur only once per hour.

If the new computer system is replacing an existing system, it may be possible to derive transaction volumes from the existing system. On the other hand, new systems rarely duplicate the transaction logic of their predecessors so these differences should be considered.

Typically, the hardest situation in which to deduce accurate transaction volumes is one in which no automated or manual predecessors exist. These situations occur when organizations enter new lines of business. In this case, volume statistics may be no more than educated guesses.

Transaction performance requirements

The transaction definitions should also document the performance requirements of each transaction. As mentioned in the introduction, the transaction performance requirements are the basis of physical database design. During physical database design, the database designer chooses the physical constructs from among the various options provided by SQLBase, such that all transactions meet or exceed their transaction performance requirements. Without specifying performance requirements, the database designer has no means by which to evaluate whether or not the database performance is acceptable.

Some organizations prefer to specify exact performance requirements for each transaction. The performance requirements typically take the form of number of execution seconds. For example, one requirement might state that the transfer funds transaction must complete in three seconds or less.

Other organizations find it easier to define performance requirements in terms of transaction types and complexity. For example, an organization might require:

- high complexity, online transactions to execute in ten seconds or less.
- medium complexity, online transactions to execute in six seconds or less.
- low complexity, online transactions to execute in three seconds or less.
- high complexity, batch transactions to execute in 60 minutes or less.
- medium complexity, batch transactions to execute in 30 minutes or less.
- low complexity, batch transactions to execute in 15 minutes or less.

Relative priority

Transaction definitions also should rank each transaction in terms of relative priority, which describes how important to the business one transaction is compared to all other transactions. Physical database design ensures that each transaction meets or exceeds its performance requirements. Consequently, the transactions must be

evaluated one at a time. The relative priority of each transaction provides the sequence in which transactions are evaluated. This ensures that the most important transactions receive the most attention.

In addition, during the process of physical database design, the database designer will tailor the physical database design so that one given transaction executes faster. Making one transaction go faster frequently negatively affects the performance of other transactions. Relative priority ranking also provides the basis for any transaction arbitration scheme.

The ranking can take many forms. For example, some organizations assign a number from one (high) to ten (low) to each transaction.

Transaction SQL statements

Each transaction definition must also specify the SQL statements which perform the necessary database operations. These SQL statements are either UPDATE, INSERT, DELETE or SELECT commands, which are known collectively as SQL DML (data manipulation language). The SQL statements are required during physical database design to forecast the execution times of the transactions. In addition, a detailed understanding of the database design and significant experience with SQL is required to accurately code many SQL statements. Therefore, it is sometimes preferable for the database designer to provide the SQL statements for the application programmers rather than allowing them to code the SQL themselves. However, the database designer and the application programmers must coordinate their efforts closely to ensure that the SQL statements support the program's data requirements.

The transaction definitions should also include, for each SQL statement, a brief (non-technical) narrative explaining:

- What the command accomplishes
- Why it is required (if it is not obvious)
- Number of rows in the database affected by the command

The number of rows affected means either the number of rows returned (in the case of a SELECT command), or the number of rows inserted, updated, or deleted. The performance of any SQL command is dependent on the number of database rows affected. For example, it would take longer to read 10,000 customer rows than one customer row. Hence, the number of records affected by each SQL command is an important element of physical database design.

Once coded and described, each SQL command should be validated for accuracy. To accomplish this, it will be necessary to construct a small test database. Many users find it useful to build a SQLTalk script containing all the transactions' SQL commands for this purpose.

Also keep in mind that the DML should be coded against the external, rather than the internal, schema. The initial external schema created in chapter 4 will probably contain many of the views needed to support the transactions. Other views may need to be created in order to meet the tailored needs of some transactions. If the programs are coded directly against the base tables contained in the internal schema, there will be a greater risk of impacting them during the physical design control phase of the database design.

Example: The table below shows how the SQL for the funds transfer transaction might appear in a transaction definition.

# Rows	SQL Statement	Notes
1	select amt from account where acct_nbr = :DebitAcct	This SQL command retrieves the balance of the debit account. If it is less than the transfer amount, the program should report an "insufficient funds" error to the teller. Also, the SQL statement should use either the RR isolation level or the SELECT FOR UPDATE command to ensure that no other transactions withdraw funds from the debit account while this transaction is in progress.
1	update account set amt = amt - :TransferAmt where acct_nbr = :DebitAcct	This SQL command decreases the funds in the debit account by the transfer amount.
1	update account set amt = amt +:TransferAmt where acct_nbr = :CreditAcct	This SQL command increases the funds in the credit account by the transfer amount.

Chapter 6

Refining the Physical Design

In this chapter we re-visit the internal schema created in chapter 3 with the goal of introducing physical database design optimization methods which improve its performance and capacity. At this point in the database design process, general guidelines and heuristics are applied to the first-cut physical database design to produce a better physical database design, or internal schema. Over the years, database designers have discovered many rules-of-thumb or heuristics to improve overall database performance. These guidelines are detailed in this chapter.

Introduction

In this chapter we present rules which may be used to refine the first cut physical database design. These rules are typically applied in the following manner:

- The database designer recognizes a scenario in the database which is often the subject of a particular design technique.
- The designer then applies that technique to the database on the assumption that the benefits will be both substantial and required.

As such, these rules do *not* offer specifically proven benefits for *every case*, but represent common database design rules that are usually incorporated into physical designs by experienced DBAs. You should carefully evaluate every database as a unique application, where some unusual characteristics may cause the assumptions on which these rules are based to be violated. For these cases, as well as for further refinement of any design, the more rigorous predictive techniques presented in Chapter 7 may be applied.

These guidelines detailed in this chapter are discussed under the following general categories:

- Splitting tables into two or more tables
- Clustering of table rows
- Indexing
- Database partitioning
- Freespace calculation

Splitting tables

One of the most common denormalization techniques that may be applied to a physical database design is the vertical or horizontal splitting apart of tables. Vertical splitting is the process of moving some of a table's *columns* to a new table that has a primary key identical to that of the original table. Horizontal splitting is the moving of some of the *rows* of one table to another table that has a structure identical to the first. The resulting tables can be managed in ways that tailor each to its particular performance requirements. The basic motivation for performing a split is that either the original table has some performance problem that must be solved, or one or more subsets of the original table has significantly different performance requirements that cannot be met in combination.

Vertically splitting large rows

The most common reason for vertically splitting apart a table due to performance problems is that the row length of the original table is too large. There are two common situations when the row length is too large: when the row length is larger than the database page size and when a clustered hashed index is used.

When the row length is larger than the database page size

Some entities in the ERD may contain a large number of attributes or a few unusually long attributes. When these entities are converted to a SQLBase table, the size of a row can be actually larger than the usable database page size. In this case, when a row is inserted into the table, SQLBase will allocate one or more extent pages on which to store the row (see chapter 8, Database Pages). Consequently, multiple physical I/O will be required to retrieve the row, one for the base page and one for each extent page (worst case scenario). These multiple I/O operations will degrade performance. To resolve this situation, the table can be vertically split in order to bring the row size down to a more manageable size.

SQLBase's usable page size is approximately 938 bytes. If the average row length for a table is greater than the usable page size and optimum performance is required when accessing the table, the database designer should split the table into two or more tables. The method used to perform the split is simple, since the split is simply the physical manifestation of performing a relational projection on the table. This means that some of the columns in the original table will be moved to a new table, while the remaining columns will reduce the row length of the table. The new tables that are created using this technique have not changed any inter-column relationships in any way (which means each column should still be fully dependent on the entire primary key), therefore each new table should have an exact copy of the original table's primary key as its own primary key.

Determining which columns to place in each table is the most critical decision affecting the successful application of this technique. This is because the split can only result in a benefit to transactions that can satisfy their data requirements by accessing only one of the resulting tables. Any transactions that must perform a join to reassemble the original row will fail to benefit from the higher blocking factor due to the additional I/O operations required to read the new table. To determine which columns to place in each table, the database designer should analyze the access patterns of all high priority transactions and place together in the same table those columns which are accessed together the most frequently.

When using a clustered hashed index

When a table's data is clustered through the use of a clustered hashed index and the row is too large, it fails to achieve the desired page clustering objective. Consequently, the number of rows that will be retrieved for each physical I/O operation may no longer be sufficient to provide adequate transaction performance. This long row length is of concern when transactions commonly access a number of rows with each execution, and the rows are physically clustered through a clustered hashed index on the key that defines this group of rows.

The concept of page clustering or *rows per page* is the same as that of the blocking factor for a file in general, where the number of rows that will fit on a page is described by:

$$\text{NumberOfRows} = \frac{\text{pagesize}}{\text{rowsize}}$$

The purpose of splitting a table apart is to attempt to achieve a number of rows per page that is at least as great as the number of rows that will be retrieved together by common transactions. Since the size of the page is fixed by the implementation of SQLBase on each hardware platform, the only way to influence the number of rows per page is by adjusting the size of the row. Since there is *usually* no such thing as too many rows per page, the direction this adjustment takes is to reduce the row size in an attempt to increase the number of rows on a page. After eliminating any spurious column definitions and reducing the maximum width of VARCHAR and CHAR columns to a minimum, the only remaining technique is to split the table apart.

Horizontally splitting tables

As defined above, horizontal splitting of a table is the moving of some of the *rows* of one table to another table that has a structure identical to the first. Horizontal splitting occurs most often to isolate current data from historical data so that the performance of accessing current data is increased.

For example, frequently organizations must store several years of order information to produce comparison and historical reports. If an organization creates several hundred thousand orders per year, an order table which contains five years of order information might exceed one million rows. Such a large table would slow the access to current year's orders, which are probably the most frequently accessed. In this situation, the database designer might choose to horizontally split the order table(s) into two tables, one for the current year orders and another for all other orders.

Tailoring performance requirements

Whenever some rows or columns of a table have significantly different performance requirements than the remainder, splitting the table apart is a viable option. This is worthwhile if there are tangible benefits to be accrued from allowing the portion of the table with the lowest requirements to accede to that level.

A good example of vertically splitting a table for this purpose is a LONG VARCHAR column that actually contains a binary CAD/CAM drawing which is historical in nature and rarely accessed by the users. Since this is a lot of data that has a much lower usage rate than the remainder of the table, it could possibly be placed in some lower cost storage medium, such as optical storage, in the future when appropriate hardware and software become available. (Future releases of SQLBase *may* allow the database designer to map specific tables to specific physical files and devices.) You can accomplish this by splitting the table apart, with the LONG VARCHAR in a new table that has the same primary key. This technique allows the LONG VARCHAR to be stored economically (in the future) without affecting the other columns in the table. If the column remains with the original table and future releases of SQLBase do add this functionality, all existing programs which access the table will have to be modified before the column may be mapped to these types of "near-line" storage.

Referential integrity considerations

When splitting a single table into two or more tables, the primary keys of each table should be identical if the table was correctly normalized prior to the split. The new tables' relationships to other tables in the database, therefore, *could* also be the same as the original table. However, the DBA should consider the effect of the delete rules in the original table's relationships when deciding how to define the referential integrity constraints for the new tables. In some cases, it may be simpler not to define referential constraints at all, since the actual row deletions from the new tables will have to be maintained in parallel with the original table through programmatic means anyway.

An alternative technique is to make the new tables dependents of the original table, with one-to-one relationships back to the original table. When this technique is used, the new tables will have a foreign key back to the original table that is an exact match to the primary key of the table. If the delete rule for this foreign key is cascade, then SQLBase will automatically delete rows of the new tables whenever a corresponding row of the original table is deleted. This eliminates the programmatic work required to keep the tables in parallel for deletions, although insertions still have to be made explicitly.

Splitting apart a table

1. Determine which new tables will contain which columns of the original table.

2. Create the new tables with primary keys identical to the original table. This means that each primary key column must be duplicated in each new table. Remember to build the primary key index on each new table as well.

3. If you would like SQLBase to handle referential integrity constraints for the new tables in the same manner as it does the original table as regarding its participation as a *child* in relationships, then the foreign key columns for each parent must also be duplicated in each new table. Furthermore, the new tables must contain the FOREIGN KEY clauses that identify the parent table of each relationship.

    OR

    If you would like to set up the new tables as dependents of the original table, specify the primary key as a foreign key back to the original table. If you wish SQLBase to handle row deletions from the new tables in parallel with the original table, specify CASCADE as the delete rule for this foreign key. No other foreign keys need to be defined in this alternative.

4. If you would like SQLBase to handle referential integrity constraints for the new tables in the same manner as it does the original table regarding its participation as a *parent* in relationships, then you must add a new FOREIGN KEY clause to each child table of the original table that identifies this new table as an additional parent.

    OR

    If you have selected the alternative of keeping the new tables as dependents of the original table in step 3, then the new tables should not participate as parents in any other relationships. The original table should keep all of the relationships in which it is the parent.

5. The new tables may be populated by means of an INSERT statement with a subselect referencing the columns in the original table that need to be moved.

6. Drop the unneeded columns from the original table through an ALTER TABLE command.

7. Change any views that access the original table to access whichever tables contain the columns needed by the view. If a view has to be changed from a single table access to a join, determine whether any programs using the view are performing updates. These programs will need source code changes to eliminate the attempt to update through a view.

Clustering of table rows

Controlling the physical storage location of rows within a table using SQLBase is accomplished through the use of a *clustered hashed* index. Using this indexing technique, which is actually not a physical index at all but a method of storing the underlying subject table, can enhance some aspects of database performance in particular cases. There are two major reasons why this technique should be applied to a table: one allows for faster access to single table rows while the other accommodates the collocation of table rows that are often accessed together.

Direct random access of rows

Whenever fast direct access to a single row of a SQLBase table is critical, and a unique key exists to identify that row, you should consider a clustered hashed index. A unique clustered hashed index on the table column that is specified with an equal predicate in the WHERE clause of a SELECT statement will cause the SQLBase optimizer to access that row with a hash technique that will, in most cases, allow retrieval of the page containing the row with a single read operation. This provides the best possible access time for the row within the SQLBase system, exceeding the capability of B+ index trees.

This technique would be appropriate, for example, when a high volume transaction has to access a large operational master, such as an account table in a bank's teller system. Since the teller knows the customer account number, the program can access the account information directly with a single I/O, eliminating not only the need to traverse a large index tree, but also the I/O and locking overhead associated with that task. Since many tellers are performing similar transactions simultaneously, the overall load on the DBMS and disk subsystem is decreased through the use of a clustered hashed index, while the response time for the transactions is enhanced.

In order for a table to be a good candidate for this technique, two conditions must exist:

- A good primary key must exist that can be fully specified for each access request. Since a clustered hashed index does not provide any partial key search capability (such as NAME LIKE 'JO%'), the key must be one that can be reliably specified in full. Furthermore, this key must also provide satisfactory mapping to the physical database pages through the SQLBase hash function. Since the hash function provided with SQLBase has been designed to give good results for the majority of keys, this will probably be no problem. Chapter 10 provides the details on how to verify the mapping of the hash function for a given key.

- The number of rows in the table must be fairly constant and predictable. This is because the clustered hashed index CREATE statement must include an estimate of the total number of rows that the table will contain. If the table

size varies significantly over time, the benefits of creating the clustered hashed index may be lost. A possible technique to get around this problem, though, would be to drop and recreate the table and index often, with the row occurrence estimate updated each time the index is recreated. This method also requires an unload and load of the table as well, and for large tables may be too time consuming.

Clustering rows together

When a fairly small number of rows of a single table are often accessed as a group, the clustered hashed indexing technique can allow for these rows to be located physically adjacent to each other. If the rows will all fit on one page, then a query requesting all of the rows will be satisfied with a single I/O operation by SQLBase. This allows for the maximum database performance to be realized for a commonly accessed group of rows.

This technique is sometimes called *foreign key clustering* because it is often a foreign key within the table that defines the group of data that should be collocated. That is, when the value of a foreign key is the same for a group of records, SQLBase attempts to store this group of records in proximity to each other. This foreign key becomes the subject key of the *non-unique* clustered hashed index that is placed on the table.

A good example of this situation is in the familiar order entry system. The ORDER and ORDER_ITEM tables in this system are often accessed together, and the database designer wishes to make this access as quick as possible, by defining a clustered hashed index on the ORDER_ITEM table, with the index column being the ORDER_NO column. This column is the ORDER_ITEM table's foreign key back to the ORDER table. Since transactions first access the ORDER table to get the header row for the order, they can then access the ORDER_ITEM table with a WHERE ORDER_ITEM.ORDER_NO = ORDER.ORDER_NO clause. This will cause SQLBase to use the clustered hashed index to locate this group of rows within the ORDER_ITEM table. These rows will (hopefully) all be on one page and can be retrieved with a single read operation.

An important qualifying requirement for the use of this method is the physical size of the cluster that will be formed by grouping with the selected foreign key. If this cluster size is too large, then the grouping will be spread out over many pages anyway and the potential benefits of this technique will not be realized. Also, if the number of rows within the grouping is large, then the chances of encountering collisions within the hashing algorithm increase, causing an increase in overflow page allocations, resulting in degraded performance.

In order to determine the average size of the foreign key grouping in terms of rows, the following formula may be used:

$$grpsize \; = \; \frac{rows}{fkeys}$$

The numerator, which is the total number of rows in the table, may be determined through estimation or through the use of the following query if the table already exists:

```
SELECT COUNT(*) FROM tablename;
```
The denominator, which is the total number of unique foreign key values, may be estimated based on the number of rows in the parent table where this foreign key is the primary key. Alternatively, the following query may be run if the child table already exists:

```
SELECT COUNT(DISTINCT ForeignKeyColumnName) FROM tablename;
```
This calculation yields the average number of rows for each foreign key group occurrence.

If the resulting group size is very large (a good rule of thumb is greater than 20), then the table is probably not a good candidate for clustering. The group size may be offset somewhat by the row length, however. The smaller the row length, the greater the group size that can be tolerated. This is because the true measure of how quickly SQLBase will be able to retrieve all the rows in a foreign key grouping is determined by how many pages are required to store these rows. The following formula can be applied to determine the number of pages which will typically be required (remember to calculate the correct page size for the implementation platform on which you will run SQLBase):

$$pages \; = \; \frac{(grpsize \times rowlength)}{pagesize}$$

If this formula yields a page spread of more than one, overflow will become a problem and this table is probably not a very good candidate for a hashed clustered index based on this foreign key.

Creating a clustered hashed table

1. If the table already exists, unload the data and drop the table.

2. Execute the CREATE TABLE command to create the table, along with the CREATE UNIQUE INDEX statement that completes the primary key definition. You may omit the CREATE UNIQUE INDEX only if the clustered hashed index you will be creating is both unique and based solely on the primary key.

3. Code and execute the CREATE INDEX commands which builds the clustered hashed index. Make sure that the SIZE clause is included, and that your estimate for the number of rows includes enough space for growth that will occur in

between database reorganizations. Also, consider allowing for free space of between 20 and 30 percent which will achieve the minimization of hash collisions (multiply your final estimate by 1.2 or 1.3 to do this).

4. Recreate any foreign keys in other tables which reference the primary key in this table.

5. Reload the data unloaded in step 1, if any.

Following this procedure, the entire space needed for the table will have been pre-allocated within the database.

Indexing

An index in a relational database is a data structure that allows rapid access of table rows based on the value of data in one or more columns of those rows. (For a detailed explanation of SQLBase's implementation of BTree indexes, see chapter 9). There are only two reasons to build an index on a relational table. The first is to ensure the uniqueness of the values being indexed, which is why all primary key columns must have a unique index defined on them. The other reason for creating indexes (and the only reason to create a nonunique index) is to enhance the performance of the database.

The drawback of indexes is the amount of resources that they consume. Every index (other than a clustered hashed index) requires disk space for the storage of the index tree structure. Each of these indexes also requires CPU cycles in order to be created and maintained. In particular, indexes consume processing time whenever a transaction performs an INSERT or DELETE operation against an indexed table, or performs an UPDATE of a table row column that participates in one or more indexes. Exactly how much overhead is created depends on the characteristics of the index and subject table, but the overhead is not trivial in any case.

The challenge for the database designer is to create indexes only for those cases where the benefits realized from the access time improvements outweigh the penalties suffered from the overhead of building and maintaining the index. While some scenarios can be identified as potentially good (or bad) candidates for an index, no irrevocable 'rules of thumb' exist. For each index that a database designer creates, the costs and benefits should be measured in order to justify the index's continued existence. If the designer discovers that an index is no longer worth its cost, then it should be dropped.

As complex as this cost/benefit analysis is, it is further complicated by the fact that the benefits are not actually within the control of the DBA. In SQLBase (as with any DBMS implementation using the SQL standard), the decisions controlling how the database is accessed are not made by the programmer, the user, or the DBA. Rather, they are made by a software module within the DBMS called the optimizer.

Therefore, the indexes which are finally selected by the DBA are subsequently reviewed through a pre-determined statistical analysis by the optimizer, where they may be found devoid of value and not used at all. Actually, the optimizer will select some indexes for some DML statements, other indexes for other statements, and no indexes for still other statements. For this reason, an accurate evaluation of the benefits side of the index problem requires a good understanding of the optimizer and how to monitor the decisions made by it.

The purpose of this section is to help the designer select possible indexes that are worth considering within a physical database design. These must then be evaluated for their effectiveness with particular DML statements through an analysis of the decisions made by the optimizer.

Column cardinality and selectivity factor

The cardinality of a table column is the number of discrete values that the column assumes across all rows of the table. As an example, the cardinality of the EMPLOYEE-SEX column in the EMPLOYEE table is two, assuming that 'M' and 'F' are the only acceptable values. Another example is the EMPLOYEE-NUMBER column in the same table, which is the primary key of the table and has a unique index placed on it as the system requires. The cardinality of this column is equal to the number of rows in the table, since for each row the column must assume a unique value. These two examples illustrate the extreme cases of cardinality that may be found.

The reason cardinality is important to index design is that the cardinality of an index's subject columns determines the number of unique entries that must be contained within the index. In the above example concerning the EMPLOYEE-SEX column, an index placed on this single column would only have two unique entries in it, one with a symbolic key of 'M' and one with a symbolic key of 'F'. Each of these entries would be repeated many times, since between the two entries all the table's rows would be accounted for. If there were 100,000 rows in the EMPLOYEE table, then each index entry would be repeated 50,000 times if the distribution of employees were equal between the sexes. This reiteration of the same symbolic key causes the index to be unattractive to the SQLBase optimizer and can also have a significant impact on the efficiency of index operations.

Determining the cardinality of a potential index key is simple if the data is already loaded into the subject table. Use the following query to find the cardinality:

```
SELECT COUNT (DISTINCT key1||key2||...||keyn) FROM tablename;
```
In this query, *key1* through *keyn* represent the potential index columns. The count returned will be the exact cardinality that the index would have if created on *tablename* at the current point in time.

The way that the optimizer considers the effect of cardinality is through the use of a statistic called the s*electivity factor.* The selectivity factor of an index is simply the inverse of the combined cardinality of all index columns:

$$\text{SelectivityFactor} = \frac{1}{\text{Cardinality}}$$

The selectivity factor is critical because this is the criteria used by the SQLBase optimizer to decide whether an index is acceptable for use in the access path that is built to satisfy the data requirements of a given DML statement. The optimizer uses a formula that combines the selectivity factor of the chosen index, along with information about the index's width and depth, in order to compute an estimate of the amount of I/O needed for a particular access path. The lower the selectivity factor, the lower the resulting I/O estimate will be, which increases the possibility of the index being selected for use.

These characteristics of columns and indexes will be used during the descriptions of good and poor index candidates which follows.

Composite indexes

When an index is created on two or more columns, it is called a *composite index.* When creating composite indexes, consider the affect that the order of columns within the index has on performance. The best performance is achieved by placing the column that usually has the most restrictive selection criteria used against it in the high order index position. If you do not know which column meets this criteria, then place the column with the highest cardinality first. Following these guidelines will result in the lowest possible selectivity factor to be achieved by the index. Of course, if the intended use of the index is to provide a generic search capability, then the columns which will be always fully specified must be placed first in the index or the searching capability will be lost.

One good reason to use composite indexes is to achieve *index only access.* This is when all of the data which is required by a query can be found within the index itself. A good example is a lookup table that finds tax rates for various county codes. If a composite index is placed on the county code followed by the tax rate, then a query that specifies an equality predicate on the county code within the WHERE clause, and requests retrieval of the associated tax rate may be completed with an access to the index entry without any access to the page containing the actual row.

Example:

A table is used by transactions to perform a lookup of county tax rates. Since this is done often, the DBA wishes to allow the operation to be performed with a minimum number of I/O operations. This is the SQL that is the most efficient in this case.

- This is the table:

  ```
  CREATE TABLE COUNTY_TAX
  (COUNTY_CODE INTEGER NOT NULL,
  TAX_RATE DECIMAL(5,2) NOT NULL)
  PRIMARY KEY(COUNTY_CODE);
  ```

- This select statement is used by transactions which perform the lookup:

  ```
  SELECT TAX_RATE
  FROM COUNTY_TAX
  WHERE COUNTY_CODE = :county_code;
  ```

- This index was required to complete the primary key definition of the table:

  ```
  CREATE UNIQUE INDEX XPKCNTYTAX
  ON COUNTY_TAX (COUNTY_CODE);
  ```

- The DBA realized that adding a new index which includes the TAX_RATE column, in the low order of the key, would allow the lookup to perform *index only access*, resulting in an I/O savings:

  ```
  CREATE UNIQUE INDEX XCNTYTAX
  ON COUNTY_TAX (COUNTY_CODE, TAX_RATE);
  ```

An alternative to composite indexes is creating separate indexes for each column. One advantage is that these separate indexes may be used when processing the indexed columns in a WHERE clause that specifies predicates that are OR'd together, since a composite index cannot be used in this case. Another advantage is that if all the columns are indexed separately, then you do not have to concern yourself with having the leading columns of a composite index always being specified in WHERE predicates.

Example:

Suppose that the following set of queries access the STUDENT table, and each query is contained within a number of transactions whose response time is critical. These are the SQL query statements:

```
SELECT NAME FROM STUDENT
WHERE ID = :id;
SELECT ID FROM STUDENT
WHERE NAME = :name
OR HOME_CITY = :home_city;
SELECT NAME FROM STUDENT
WHERE HOME_CITY = :home_city
AND MAJOR = :major;
SELECT AVG(GPA) FROM STUDENT
WHERE MAJOR = :major;
```

- Since ID is the primary key of the STUDENT table, query number one will never be a problem. A unique index must exist on ID in order for the primary key definition to be complete.

- A composite index on NAME and HOME_CITY can never be used by query number 2, regardless of the column order, since the logical connector is OR. This statement can only be satisfied by a table scan, or an index merge between two indexes where the specified predicates occupy the high order position.

- A composite index built on HOME_CITY followed by MAJOR could be used by query number 3, but would not be acceptable for query number 4, since MAJOR is not in the most significant position.

- The following set of indexes could be considered for use by every one of the above queries, and represents the most efficient way to satisfy the above queries:

```
CREATE UNIQUE INDEX XPKSTUDENT
ON STUDENT (ID);
CREATE INDEX XNKSTUNAME
ON STUDENT (NAME);
CREATE INDEX XNKSTUHOME
ON STUDENT (HOME_CITY);
CREATE INDEX XFKSTUMAJOR
ON STUDENT (MAJOR);
```

Good index candidates

The following list identifies some columns that may make good index candidates:

- Primary key columns. By definition, primary key columns must have unique indexes placed on them in order for the definition to be complete and allow for the use of referential integrity.

- Foreign key columns. These are usually very good index subjects for two reasons. First, they are often used to perform joins with their parent tables. Second, they can be used by the system for referential integrity enforcement during delete operations against the parent table.

- Columns that are referenced in WHERE predicates or frequently the subjects of ORDER BY or GROUP BY clauses. Indexes on such columns can speed up these operations significantly by preventing the creation of an intermediate result table.

- Any column that contains unique values.

- Columns that are either searched by, or joined with, less than 5 to 10 percent of all table row occurrences.

- Columns that are often the subject of aggregate functions (involving SUM, AVG, MIN, and MAX).
- Columns often used for validity editing in programs, where a value entered by the operator is checked for its existence in some table.

Poor index candidates

Small tables

One common rule is to avoid creating indexes on tables which span less than five pages. The costs associated with creating and maintaining these indexes are typically greater than the cost of performing a scan of the entire table. Of course, a unique index required to define a primary key in order to provide needed referential integrity is an exception to this rule.

Large batch updates

Tables which experience updates in large batches usually have problems with index overhead, particularly if the batches update more than 10% of the table. One possible way to avoid these problems while still experiencing the benefits of indexes is to drop all of the indexes on the table prior to the update, then recreate them after the update.

Skewness of keys

Another index pitfall is when the index's key value distribution is significantly skewed. In this situation, the cardinality of the index will appear to be quite high, allowing the optimizer to compute a sufficiently low selectivity factor to make use of the index often. In spite of this, the performance results of queries that utilize the index are often unacceptable. This is because when there is a large percentage of rows that have the same value for the index key, the index performance when accessing these rows is much worse than when retrieving rows whose index key value occurs less often. The following query will show the distribution of index values in a subject table:

```
SELECT key1, key2...keyn, COUNT(*) FROM tablename GROUP BY
key1,key2..keyn ORDER BY LastColumnNo
```
This query creates a report of index keys ordered from those occurring least often to those occurring most frequently. If the difference between the occurrences at the top of the list and those at the bottom is several orders of magnitude or greater, then this index is probably going to cause performance problems when the rows being retrieved fall into the more often occurring values. The fact that the total cardinality of this index appears to be acceptable, and therefore its use is considered desirable by the optimizer, makes this an insidious situation. It is only when the skewness of the keys themselves is examined that the database designer discovers the performance penalties that will accrue from the creation of this index.

Other poor index candidates

The following column characteristics generally lead to indexes which are either too expensive to maintain or of little benefit to system performance:

- Low cardinality. These columns make poor index candidates since their high selectivity factor influences the optimizer to avoid their use. Use of a low cardinality index would often cause the DBMS to incur more I/Os than a table scan, since many pages may be accessed multiple times.

- Many unknown values (nulls). Null values represent a skew in the distribution of what may otherwise be a column with acceptable cardinality.

- Frequently changing values. Columns which are frequently updated cause excessive index maintenance overhead. Each time the column is updated, the DBMS must delete the old value from the index tree and insert the new value.

  Also, columns which are updated by transactions that occur very frequently may cause lock contention problems within the index structure. This is particularly true if the column is a combination of date and/or time, or if the column is a serially created key value. These situations cause transaction activity to be concentrated in a narrow section of index values, which increases the probability of lock contention occurring.

- Excessive length. Single or composite index keys that are longer than 50 bytes will cause the index to grow to a large number of levels unless the number of rows is low. Since at least one I/O is incurred for each level when searching the index, this will degrade performance.

Database partitioning

The ability to partition a database across a number of database areas allows for the throughput of the database to be greatly increased. Much of the benefit depends on the way the operating system environment allows SQLBase to manage these database areas. This varies depending on the platform on which the server is implemented (such as NetWare, or OS/2). In general SQLBase will create a unique I/O thread for each database area, which allows for a greater level of parallelism to be achieved in the disk subsystem. Whether or not this will create bottlenecks at the disk controllers or the drives themselves depends on the physical configuration of the I/O subsystem.

The other benefit realized from database partitioning is the ability to have databases which are greater than the physical size of any individual disk drive. For very large databases, there is no alternative to database partitioning since there may be no disk drive available which is sufficient to store the entire database on a single drive.

The drawback to partitioned databases is that the physical disk location of a database may be harder to identify. This becomes even more true if the DBA decides to allow

database areas to be shared by different storage groups, which are in turn used by different databases. Monitoring free space in database areas is more difficult that monitoring free space on a physical device, and determining the total free space available to a storage group may involve the examination of a number of database areas. Also, the DBA must be careful to maintain synchronization between the MAIN database, which stores *all* of SQLBase's database area extent information, and *all* of the databases which are partitioned.

The bottom line to partitioning is that it is the best way to enhance throughput and capacity in an appropriate operating environment, but it should only be undertaken when the benefits are truly needed. Those DBAs who decide to implement database partitioning must be prepared to spend time ensuring that their monitoring, backup, and recovery procedures are all up to the task of handling the more complex details associated with this environment.

When deciding how to partition a database consider the following factors:

- Do not share database areas between databases and logs. Always specify different storage groups for the data and for the log files, and ensure that no database areas are shared between these two storage groups.

- Place database areas intended for logs on a different physical device than any of the database areas which are used for the data storage of tables. If a database's data files share a disk drive with their own log files, then the log files may be damaged along with the database in a disk drive failure, and will not exist to perform a full recovery of the database.

- Consider *not* sharing storage groups between different databases. This prevents databases from being interspersed in the same database area files, so that if a failure occurs, the affected database will be more easily isolated.

- If you are implementing database partitions in UNIX raw partitions, make sure you have a technique developed to monitor space in the partition.

- Do not set a default storage group unless you are absolutely sure that you are prepared to manage *all* databases as partitioned.

- The following query may be executed against the MAIN database using the guest/guest account in order to ascertain available free extents in a database's database areas:

```
select a.name,b.stogroup,b.areaname,
@trim(c.pathname), c.areasize,
@nullvalue(100*(d.extsize/c.areasize),0)
from syssql.databases a, syssql.stoareas b,
syssql.areas c, syssql.freeexts d
where b.areaname = c.name
and c.name = d.name(+)
and ((a.stogroup = b.stogroup)
```

```
or (a.logstogroup = b.stogroup))
order by 1,2
```

Free space calculations

SQLBase provides a keyword, PCTFREE, to allow database designers the ability to tune the amount of available space which is reserved within the data pages of tables and indexes. Through the judicious use of this parameter, the designer may allow for the growth of the database over a period of time without any degradation in performance. Following this time period, a reorganization of the database should be performed, marking the beginning of a new time interval for database growth. This time interval, which is chosen by the designer, is termed the *reorganization frequency.*

The key considerations for choosing a reorganization frequency are the rate at which the database grows and the availability of time in which to perform the reorganization. If the database grows very slowly, then a great deal of time may be allowed to pass between reorganizations without affecting its performance. On the other hand, if the database is constantly growing, frequent reorganization may be required to maintain consistent performance even with a generous amount of freespace specified. Also, very large databases may take a great deal of time to reorganize and this time may not be easy to find in the schedule of demand usage of the computer system by the users. Finding the right trade-off between frequent reorganizations and system availability can exercise the political ability of even the most organizationally astute DBA.

The purpose of the PCTFREE parameter in the CREATE TABLE statement is to reserve space in the database page for future growth of rows already allocated to that page. No new row addition will be made to the page if it would cause the free space on the page to drop below the PCTFREE value. You can ensure that some space will remain on a page that will be available for the expansion of its rows.

This expansion can come about in three ways. First, a column of any data type may be updated with data which is longer than what was previously stored. Second, a column whose value was originally NULL is updated to possess a non-null value. Finally, the DBA may decide to alter a table definition to add new columns to the table which are then updated with non-null values.

Whenever any of these events occur and cause an existing row to expand on the page, there must be sufficient free space on the page in order to avoid the creation of an extent page. Whenever an extent page is created, additional I/O becomes required to obtain rows which have spilled over onto that page. This additional I/O causes the performance of the database to degrade. When enough of this activity has taken place, with each data page having a string of extent pages connected to it, execution times for transactions may be extended by several times their original objective.

When determining the appropriate values for the PCTFREE specification, consider the following factors:

- Tables which are read-only require no free space, and neither do indexes on these tables. Specify PCTFREE 0.

- If many table columns are often increased in length, specify a high (30-50) PCTFREE value to accommodate this growth.

- If a table is expected to have columns added to it after its initial creation, specify a high PCTFREE to avoid the creation of many extent pages when the new columns are added. However, if the new column will be populated with non-null values only gradually, then specify a medium PCTFREE value (10-30).

- If transactions are experiencing problems with waiting for locks or deadlocks due to a large number of active rows being located on the same page, the concurrency for this table may be increased through the PCTFREE value. Alter the table definition with a high PCTFREE value (80-90), then perform a reorganization. The table will now have very few rows on each page (possibly as few as one), which will decrease the likelihood of page locks contending with the access of other rows. The downside is that the table will occupy more room on the disk, and any table scan operations will take longer.

Following the implementation of the database, the DBA needs to observe its performance over time to determine if any degradation is being realized prior to reorganizations. If there is a significant performance drop due to excessive extent pages being created, then appropriate action should be taken. The reorganizations could be done on a more frequent basis. Alternatively, the PCTFREE specifications could be increased prior to the next reorganization to stretch the effective performance life of the database.

Chapter 7

Physical Design Control

This chapter describes a technique for performing physical database tuning on a reiterative basis. The activities involved are:

- Measuring or predicting transaction response times for a given database design state.

- Making a change to the design of the database in order to improve the response time of one or more transactions.

- Measuring or predicting the response time for all critical transactions with the new design.

- Deciding whether to accept or reject the database change based on the results indicated.

Introduction

As explained in chapter 1 of this section, database design is a six step process. These six steps, which correspond to chapters two through seven of this section, are:

1. Build the logical data model, or the conceptual schema, of the ANSI/SPARC three schema architecture.

2. Convert the logical model to a "first cut" physical database design which is the internal schema in the ANSI/SPARC three schema architecture. This conversion is accomplished by applying a few simple rules to the standard Entity Relationship diagram.

3. Define the external schema, or DBMS views.

4. Define the database transactions.

5. Refine the "first cut" physical database design by applying some heuristics to common situations.

6. Perform physical design control.

Physical design control is the iterative process whereby the physical database design is adjusted, or tuned, so that the execution time of a database transaction, which is not meeting its performance objective, is decreased. The goal of physical database design is to ensure that every database transaction meets or exceeds its performance objectives. (In reality, it is sometimes prohibitively costly for every transaction to meet or exceed its performance objectives. In these cases, the goal of physical design is to ensure that every high priority transaction meets its performance requirements, and all other transactions do not deviate unacceptably from their performance requirements.) Consequently, physical design control is complete when every transaction meets or exceeds its performance objectives.

This chapter presents the procedure of physical design control. In addition, common tuning scenarios are discussed along with tips and techniques which may be used to speed up specific types of transactions.

The process

The process of physical design control is accomplished in three steps:

1. Determine the execution time of each transaction and locate the highest priority transaction which is not meeting its performance objectives.

2. Adjust the physical constructs of the database so that the transaction takes less time to execute.

3. Go back to step one, unless all transactions are now performing within their required response times.

Each of these steps are discussed in the following sections.

Step one

The execution time of individual transactions may be determined by actually executing the transactions or by estimating the execution time using some forecasting model. The former method is referred to as the "brute force method" and the latter as the "forecast method."

Brute force method

To apply the brute force method, the database designer must compile the physical schema and load the database tables with data. Once this is complete, SQLTalk scripts or application programs may be developed which submit each transaction to SQLBase and record the execution time.

However, in order for the results to be meaningful, the database size (in terms of the number of rows in each table) must be approximately equivalent to the size of the production database. This is because the query optimizer chooses the access path based on statistics kept in the catalog. If the database is relatively small, as compared to the production database, it is likely the access path chosen by the optimizer will differ from the one which would have been chosen had the transaction been run against the production database.

Therefore, for the results of the brute force method to be meaningful, data volumes which approach the size of the production database must be loaded in the database. This fact limits the usefulness of the brute force method to relatively small databases. (Of course, "small" is a relative term. On a 386/33MHz computer running DOS, small probably means less than 50MB. On a 486/66MHZ computer running a 32 bit operating system like NetWare, a small database is probably less than 150MB.) This reasoning centers on the time required to process large volumes of data.

For example, consider a 500 megabyte SQLBase NLM database running on a 486/66MHz Compaq SystemPro. It takes several *days* of processing time just to load a 500 megabyte database on this hardware platform. Furthermore, a database of this size typically contains at least one table with several hundred thousand rows. On a 486/66MHz computer, it requires several hours of processing time to build *one index* on a table containing several hundred thousand rows. It is not uncommon for a 500 megabyte database to have dozens of tables, each containing several indexes.

Perhaps if the database tables and indexes only have to be loaded or built one time, the logistics and computing time necessary to handle a large database would not be insurmountable, but this is not the case. As we shall see in step two, physical design control involves altering the physical constructs of the database to decrease execution

time. For example, several different indexes may have to be tried before a specific transaction meets its performance objectives. Similarly, the database designer may choose to cluster a table on a different key to see if performance increases. To accomplish this, the table must be dropped, the data reloaded and all the indexes rebuilt. Sometimes a design iteration may require the data load programs to be altered. This can occur, for example, if the database was denormalized in an attempt to increase the performance of a transaction. Of course, re-coding application programs may involve an enormous amount of effort.

Consequently, each design iteration in the physical design control process potentially requires the data load procedures to be repeated. On a complex database, ten or more design iterations may be required before all the transactions are meeting or exceeding their performance objectives. Hence, the database tables and indexes may have to be built ten or more times. Therefore, as stated above, the usefulness of the brute force method is limited to relatively small databases.

Forecast method

The forecast method is ideally suited for medium to large databases since many different design strategies may be tried without having to actually load data into the database. This technique involves estimating the execution time of a particular transaction through the use of some forecast model.

Forecast models typically try to predict how many physical block reads or writes and CPU cycles a specific transaction will consume. These numbers are then multiplied by the average execution time of a physical I/O or CPU cycle. For example, most disk drives commonly used today in Intel-based database servers have a total data access time of less than 20 milliseconds. This is the average time required to position the read/write head on the desired track plus the average rotational delay time (the time required for the desired sector to spin around to the read/write head). According to this model, if a given transaction required 1000 physical I/O, on average the transaction will execute in twenty seconds (1000 * .02). We are ignoring CPU time here for the sake of simplicity. In fact, CPU time is often omitted from forecast models because CPU time is usually a small percentage of overall transaction execution time.

Of course, these forecast models are an over-simplification of true performance and overlook many factors such as disk controller caches, operation system file buffers, and database page buffers. However, the measurements are usually sufficiently accurate for the desired purpose. It is not necessary to know *exactly* the execution time of a given transaction. If the forecast is only 50% to 75% accurate, we still have an adequate measuring rod to perform physical design control. The estimate of execution time provides a *relative measurement*, as long as the estimating algorithm is applied evenly. This relative measure can then be used to identify those transactions which are expected to grossly exceed their performance objectives.

Before the relational model and SQL became the de facto standard, forecast models were used almost universally by database designers. This is because most DBMS of that time employed a navigational database manipulation language (DML). With these navigational DMLs, the programmer determined the access path for each database access. Once the access path is known, it is relatively straightforward to estimate physical I/O. Hence, forecast models were relatively easy to develop.

However, with the rise of the relational DBMS and SQL, forecast models have become much more difficult to build. This is because SQL is a declarative DML. With SQL, the programmer does not determine the access path. This task is delegated to the query optimizer. So, in order to accurately forecast the number of physical I/O, the forecaster must first "guess" which access path the query optimizer will employ for a given database operation. The cost model of query optimizers are extremely complex and beyond the reach of most database designers.

The Centura advantage

Fortunately, SQLBase provides an elegant solution to the problem of forecasting execution time. SQLBase contains a number of features which, when used together, allow the SQLBase query optimizer to serve as the forecast model.

First, the Explain Plan facility has been enhanced to provide:

- More detailed information describing the access path
- The "cost" of the access path

Consequently, when in "explain mode," SQL commands are not actually executed or result rows returned. Instead, the query optimizer determines the lowest cost path and returns a description of that path plus the cost. The cost returned by the query optimizer is the optimizer's own projection of the execution time of the SQL command. The cost is calculated by estimating the number of physical I/O and CPU cycles required to actually execute the SQL command based on various database statistics. Please refer to section three of this manual for a detailed discussion of the cost.

Since the query optimizer uses current database statistics, care must be taken by the database designer to ensure that these statistics reflect the production environment. Otherwise, the query optimizer might choose a different path or return an inappropriate cost.

One way to ensure that the query optimizer is using the right statistics is to load the database with production data volumes. However, this approach suffers from the same limitations as the brute force method. Fortunately, SQLBase also provides several new commands which allows the database designer to override the statistics with user calculated values.

In SQLBase, all statistics used by the query optimizer are now stored in the catalog as columns of the various system tables automatically created and maintained by SQLBase. In addition, the UPDATE STATISTICS command may be used to store user-desired statistics in these columns in order to influence the choices made by the optimizer.

Consequently, the procedure which the database designer should follow when attempting to use the query optimizer as a forecasting tool is:

- Compile the physical schema DDL.

- For each table and index, estimate the values for the optimizer statistics which will be stored in the catalog once production data volumes are loaded.

- Store these user-determined statistics in the catalog using the UPDATE STATISTICS command.

- Put the database in explain mode using either the *sqlexp* function or the SQLTalk command, SET PLANONLY ON.

- Submit SQL commands to SQLBase in the usual way and record the access path and cost returned by the query optimizer.

- Calculate the total forecasted cost of the transaction by summing the cost of each SQL command in the transaction.

Step two

During step one of physical design control, the execution time of each transaction is determined and the highest priority (during transaction definition, each transaction is ranked in terms of its relative priority and its performance requirements specified) transaction which does not meet its performance objectives is located. Thus, the input to step two is *one transaction*. The objective of step two is to adjust the physical constructs of the database so that this transaction's execution time is decreased. This is often referred to as "tuning the schema" for a transaction.

Most database transactions consist of many SQL commands. Consequently, the first task in step two is to determine which of the SQL commands that make up the transaction are contributing the most to the overall cost of the transaction. The designer should then focus his attention on these commands first.

Once a specific SQL command is targeted, the database designer may then begin to consider alterations to the physical schema or the SQL syntax which may increase performance. The general design goal is to reduce the number of physical I/Os required to satisfy the SQL command. Three fundamentally different alterations may be made to the physical schema:

- Add or change row clustering, primarily via hashing
- Add or change an index

- Denormalize a table

Similarly, the SQL syntax may be changed in two fundamental ways:

- Restrict the selection criteria
- Replace the SQL command with an equivalent one

Add or change row clustering

The mechanics of the clustered hashed index are covered in chapter 10 of this manual. You are encouraged to review this section. In addition, general principles for determining whether a hashed table will result in improved performance in a given situation was discussed in the previous chapter.

Also, note that in some circumstances, transaction performance may be increased by unhashing the table and clustering it through a different technique. For example, one could improve performance during table scans for tables which contain static data (data rows which are never updated or deleted) by unloading the data, physically sorting the rows in some specific order, and inserting the rows in this specific physical order.

Add or change an index

Determining the optimal index set for a given database and transaction is perhaps the most difficult task the database designer faces and it requires a thorough understanding of the optimizer. In addition, metrics are presented in the previous chapter to evaluate the quality of a proposed index as well as guidelines to follow when creating an index.

Denormalize a table

Denormalizing a database is a common technique to improve the performance of a SQL command. Denormalization refers to the process of converting a database which is in third normal form to some lower level of normalization for performance reasons. The concepts of first, second, and third normal form are presented in chapter two and you are encouraged to review this material.

While in the majority of situations, the benefits of direct implementation of normalized data structures outweigh any disadvantages, in a few cases there may be sufficient justification to perform denormalization in a controlled, deliberate manner. For example, whenever many joins will be required across large tables on a frequent basis, the performance penalty of these join operations may be lessened by combining some of the tables or introducing selective data redundancy among the tables. This denormalization must be done with careful consideration to the costs involved in working around update and deletion anomalies, performing processing to update redundant data, and increasing the recurring maintenance costs for the database. Of course, the benefits realized from the denormalization include fewer join operations

and fewer foreign keys along with their usual indexes. When these benefits are more critical to the success of the system than the drawbacks of a non-normal database, then denormalization is done as a part of the physical database design process.

Denormalizing with prime attributes is one common form of denormalization. This involves redundantly storing a "prime" (meaning not derived or calculated) data element in both a primary table and a secondary table. The primary table is the table where the data element "belongs" in terms of the normalization rules. By storing the data element on the secondary table, the normalization rules are violated.

For example, consider the query, "display a list of all customer zip codes for which there currently exist open leads, and rank the zip codes in descending order by the number of open leads." The SQL command for this query might be:

```
SELECT C.ZIP_CODE, COUNT(C.ZIP_CODE)
FROM CUSTOMER C, LEAD L
WHERE L.CUST_ID = C.CUST_ID
AND L.LEAD_STATUS = 'OPEN'
GROUP BY C.ZIP_CODE
ORDER BY 2 DESC;
```

Note that the ZIP_CODE of the customer is stored in the CUSTOMER table because that is where the attribute belongs. For any one CUST_ID, there exists only one ZIP_CODE. (We are ignoring here the fact that a company may have many locations and each location may have a postal and delivery zip code.) However, to execute this query, the LEAD table must be joined with the CUSTOMER table (see the WHERE clause above). If the CUSTOMER table contains several hundred thousand rows and at any one point in time there exists only a relatively small number of open leads, then the performance of this query could be significantly increased by denormalizing the LEAD table.

To denormalize the LEAD table to speed up this query, the customer ZIP_CODE field is redundantly stored in the LEAD table. The new SQL command to produce the same result is:

```
SELECT ZIP_CODE, COUNT(ZIP_CODE)
FROM LEAD L
WHERE LEAD_STATUS = 'OPEN'
GROUP BY ZIP_CODE
ORDER BY 2 DESC;
```

Thus, the join of the CUSTOMER table with the LEAD table is avoided resulting in increased performance of the target query. However, the LEAD table is now *not* in third normal form because the ZIP_CODE attribute is not dependent on the entire primary key of the LEAD table. Presumably, the primary key of the LEAD table is (CUST_ID, LEAD_NBR). Given this, the ZIP_CODE attribute depends only on the CUST_ID; regardless of the LEAD_NBR, the customer's ZIP_CODE remains the same. Thus, the LEAD table now violates the second normal form rule.

Violating the second normal form rule, in this situation, does increase performance of the target query. However, this increased performance comes at the price of an update anomaly. We cannot now change the ZIP_CODE of a customer in the CUSTOMER table without changing every lead for that customer. Hence, the update customer transaction has to be written so that any change of a customer's ZIP_CODE is reflected in all leads (otherwise the data in the LEAD and CUSTOMER tables would be inconsistent). Denormalizing the LEAD table to speed up the target transaction also resulted in:

- Decreased performance of the update customer transaction.
- Placing the data integrity of the CUSTOMER and LEAD tables in the hands of the application.

This particular example of denormalization results in an update anomaly. Insertion and deletion anomalies are also possible. (See C.J. Date, *An Introduction to Database Systems*, Addison-Wesley for a detailed explanation of update, insertion and deletion anomalies.) Every time a table is denormalized, the database designer must be aware of, and devise a strategy to deal with, update, insertion, and deletion anomalies. In addition, one must rely just a little bit more on the application programs to enforce the data integrity of the database. If the programs contain bugs relating to their handling of these anomalies, data integrity will be compromised. *For these reasons, denormalization should only be considered as a last resort.* The database designer should try all other alternatives to speed up the query before resorting to denormalization.

Further, one should resort to denormalization only in the case when a specific transaction cannot otherwise meet its performance objectives. Some database design methodologies suggest that the designer denormalize the database as a general rule. In other words, certain tables are denormalized based on broad guidelines *before* any transactions are timed. The table is thus denormalized whether *it is needed or not*. The only valid reason to denormalize a table is to achieve increased performance of a given transaction that cannot otherwise be obtained. To denormalize a table before it has been demonstrated that the transaction performance objectives cannot otherwise be met is reckless given the cost of dealing with update, insertion, and deletion anomalies or a loss in data integrity.

Denormalizing with aggregate data is another common form of denormalization. This deals with adding aggregate or derived data attributes to database tables. Aggregate data are data items derived from other lower level data and are frequently required in decision support and executive information systems in order to satisfy the large degree of summary and exception reporting these type systems provide.

For example, consider the query "produce a list of salesmen and their total year-to-date sales in descending order from the top selling salesman to the salesman with the

least sales." To produce this report, all orders for the current year must be read. Most order entry systems store order data in two tables: the ORDER_HEADER and ORDER_DETAIL. The ORDER_HEADER table contains data germane to the entire order, such as SALESMAN_ID, SHIPTO_ADDRESS and CUST_ID. The ORDER_DETAIL table stores information pertaining to one order line item. Consequently, to produce this report, the ORDER_HEADER table must be joined with the ORDER_DETAIL table. The SQL statement to produce this report is:

```
SELECT H.SALESMAN_ID, SUM(D.AMOUNT)
FROM ORDER_HEADER H, ORDER_DETAIL D
WHERE H.ORDER_ID = D.ORDER_ID
GROUP BY H.SALESMAN_ID
ORDER BY 2 DESC;
```

If there are 50,000 rows in the ORDER_HEADER table and 300,000 rows in the ORDER_DETAIL table (on average each order has six lines items), the performance of this query can be significantly improved by adding an aggregate data element, TOTAL_ORDER_AMOUNT, to the ORDER_HEADER table. Whenever a row in the ORDER_DETAIL table is inserted, updated or deleted, the TOTAL_ORDER_AMOUNT is decremented or incremented appropriately so that this summary field contains the total sales amount of the entire order (which is the sum of all the line items). The SQL command can then be written to avoid the join:

```
SELECT SALESMAN_ID, SUM(TOTAL_ORDER_AMOUNT)
FROM ORDER_HEADER
GROUP BY SALESMAN_ID
ORDER BY 2 DESC;
```

Technically, the ORDER_HEADER table is now in violation of the normalization rules and update, insert, and delete anomalies exist. The create/modify order transaction must be written so that this total field in the ORDER_HEADER table is maintained as order lines are added, changed, or deleted.

When denormalizing by adding aggregate data to the database, the decision the designer must make is whether to allow SQLBase or programs to calculate these values each time they are needed, or to actually store them in the database so that they can be queried without the overhead incurred from their derivation. The key factors involved in this decision are the frequency of access for these data items, the response time required, and the frequency and types of update operations that are performed on the lower level data from which the aggregate data is derived.

Consider the case of the student's GPA. This information is probably needed for each academic inquiry transaction that the system performs, and these transactions may be executed quite frequently and require a reasonably low response time (probably two seconds or so). The lower level data, which are the grades a student has received in each course he has taken, could be quite large for a student with a long academic career. Not only would the on-line inquiry program have to retrieve all this data to perform the calculation, but the logic needed is also quite complex.

Furthermore, this grade data is updated infrequently, probably only at the end of the semester. This update could be performed with a mass insert program that can be designed to perform GPA calculations for students in an efficient manner. On-line transactions which perform grade changes are not highly used and can suffer from some slower response time while the program is calculating and updating the GPA. This is an excellent scenario for allowing aggregate data to be placed in the design, since the benefits realized will outweigh the costs.

The example of a customer's total sales for a year probably falls on the other side of the trade-off. This data is usually not used for on-line inquiry transactions, but more often appears on reports for sales or marketing departments. As such, there is little benefit from speeding up the derivation of this data. Also, it is easily computable by a SQL query which performs a GROUP BY operation. The values from which the information is derived are the sum of all customer order total sales, which could be frequently updated for active customers. Every order transaction that is processed through the system would have to perform the updating of the customer's total sales, which would cause overhead in a transaction which probably has stringent response time and throughput requirements. When everything is considered, this data item should not be placed in the customer entity and those programs needing it will have to pay the performance penalty of performing the aggregation query needed to derive its value.

Each case of aggregate data should be subject to the same level of analysis as demonstrated previously. Only in those cases where there are clearly significant benefits from actually storing the derived data should it be maintained in the database. The general rule of thumb should be to eliminate derived data and, as with the denormalization trade-off, whenever the rule is broken, the results of the analysis that lead to that decision should be thoroughly documented within the database design.

Thus, this form of normalization has the same pitfalls as redundantly storing prime attributes. However, unlike prime attributes, derived attributes add information to the database. Consequently, denormalizing with aggregate data is perhaps preferable to redundantly storing prime attributes.

Restrict the selection criteria

In addition to changing some physical construct of the physical schema, the database designer may also change the SQL command in order to increase the performance of a transaction. The first type of change the database designer can make to the SQL command is to further restrict the selection criteria. In order to do this, the database designer *takes advantage of some knowledge of the application* in order to add an additional restriction to the selection criteria.

For example, consider the query, "produce a report of all open orders." Now also suppose that:

- Orders are either "open" or "filled."

- On average, 300,000 orders exist (four years history plus the current year).

The SQL command to produce this report is (we ignore the ORDER_HEADER, ORDER_DETAIL data structure as it is not required to illustrate the point):

```
SELECT *
FROM ORDER
WHERE ORDER_STATUS = 'OPEN'
```

The optimizer is forced to execute a table scan on a table containing 300,000 rows to satisfy this query. The only index which could be used for this query would be an index on the ORDER_STATUS attribute. However, this attribute has a cardinality of two (meaning it has two distinct values); hence, an index on this attribute would have a selectivity factor so high the optimizer would never choose it.

Now suppose that the database designer also knows that:

- On average 240 new orders are added every day.

- 99.9% of all orders are usually filled by 12:00pm the same business day on which they are received or the following day.

- 99.9% accuracy is sufficient for purposes of the report.

The problem is that the optimizer must read 300,000 rows to find the 240 whose status is "open" (the ones not filled). Since 99.9% of all orders are received and filled on the same business day and the user's requirements can be met with this approximation, an index could be placed on the ORDER_DATE attribute and the query could be rewritten as:

```
SELECT *
FROM ORDER
WHERE ORDER_STATUS = 'OPEN'
AND ORDER_DATE IN (SYSDATE, SYSDATE + 1)
```

Since four years of order history is maintained in the ORDER table, the cardinality of the ORDER_DATE would be high and the selectivity factor of the index extremely low. Hence, the optimizer would probably use this new index to read the 240 open orders directly.

The point of this example is that the database designer *took advantage of some knowledge of the application* to apply a further restriction to the selection criteria which, because of the nature of the system, produced the identical results. The implication of this observation is that the database designer must understand the application or these opportunities will be missed.

Replace the SQL command with an equivalent one

SQL is a powerful and flexible database manipulation language; the same result set may be produced with several different SQL commands. A subquery may be expressed as a join. Several OR predicates may be replaced with one IN list. For example, the following SQL statement:

```
SELECT * FROM TABLE
WHERE COL1 = 'A'
OR COL1 = 'B'
OR COL1 = 'C'
```
may be rewritten as:

```
SELECT * FROM TABLE
WHERE COL1 IN ('A','B','C')
```
Equivalent SQL commands sometimes produce significantly different performance results. A SQL command may run much faster than one of its equivalent forms. The reason for the difference in performance is that the SQLBase query optimizer is able to detect certain conditions for which it is able to apply some fast access path techniques. With other forms of equivalent SQL commands, the optimizer is not able to detect these conditions.

The details of these circumstances require a detailed understanding of the query optimizer. You are encouraged to review section two of this manual which discusses the SQLBase query optimizer at length. However, in practice, when a given SQL command is executing too slowly, the database designer should re-code the command in as many different equivalent forms as possible to see if one form produces a significantly different performance result. The implication of this is that the better and more experienced the database designer is with SQL, the more equivalent forms of a given SQL command the database designer will be able to try.

Chapter 8

Database Pages

In this chapter we:

- Introduce the concept of the *database page*, which is the basic unit of disk storage used by SQLBase.

- Cover the following database page types in detail:

    - Data row pages

    - Data extent pages

    - Long varchar pages

    - Overflow pages

    - Index pages

    - Control pages

- Discuss SQLBase's page allocation and garbage collection algorithms.

Pages and page types

The fundamental data structure of a SQLBase database is the database *page*. The SQLBase database file (the .dbs file) is composed of many pages (see Figure A). The SQLBase page size is fixed; all pages have the same length. The Windows, OS/2, Windows NT, and NetWare versions of SQLBase use a page size of 1024 bytes (or 1K). In most DBMS, database pages are a multiple of 1K to optimize reading and writing to secondary storage devices. Disk controllers read and write data in *blocks* that are usually multiples of 1K. In addition, most operating systems' *file buffers* are multiples of 1K. For example, the default block and cache buffer size for NetWare is 4K. If the database page size is equal to, or a multiplier of, this common 1K block size, I/O is optimized since one physical I/O can read or write one or more database pages.
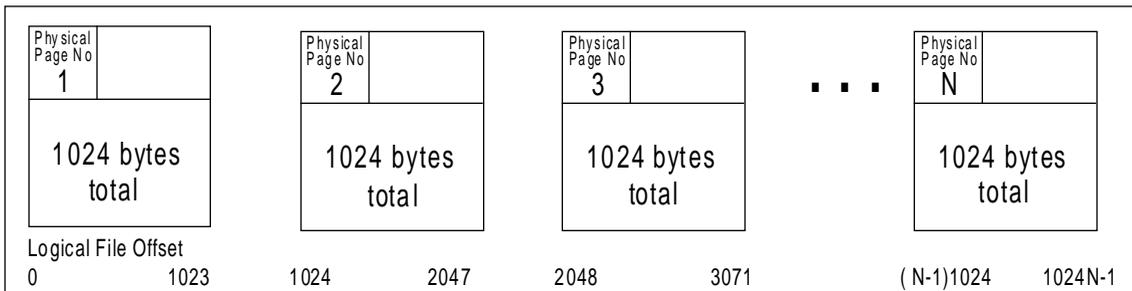
Figure 8A - The SQLBase database is a set of 1024-byte pages.

The relationship between the size of the database page and block and file buffer is an important consideration in access path selection (which is discussed at length in chapters 15 through 18). For example, assuming the default block and file buffer size in NetWare, one physical I/O actually retrieves four database pages. Thus, SQLBase can retrieve four database pages with one physical I/O for sequential reads (such as during table scans). This assumes the next physical page is also the next logical page which for sequential reads is reasonable. On the other hand, random reads require one physical I/O for each logical read request (to retrieve a row). Since one database page can store more than one row, a table scan is a much more efficient access method if more than just a small percentage of rows is needed.

Each database page is assigned a *physical page number* which is four bytes long. This number is assigned sequentially: the first page in the database is assigned page number one, the second page number two and so on. SQLBase uses the physical page number, along with the page size, to calculate the logical file address of each page. For example, the page with the physical page number of 10 would begin at byte number 9,216 (the formula is (n-1)*1024) within the database file and continue for a length of 1024 bytes. This is referred to as the logical file offset. SQLBase requests

pages from the operating system by specifying logical file offsets. With the exception of Unix raw devices, it is the job of the file system and disk controller to convert this number into a physical storage device and the actual track and sector (block) where the database page is stored, using the specific device geometry information coded into the driver module supplied by the manufacturer of the disk subsystem.

Database pages are also linked to form lists of pages. These lists are referred to as *linked lists*. Each page contains a pointer to the next, and sometimes prior, page in the list (see Figure B). The pointer consists of the physical page number and facilitates the sequential access of all the pages within a specific list.
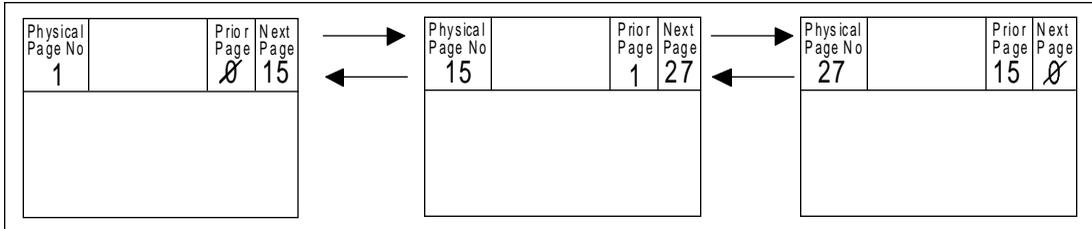
Figure 8B - Database pages are linked together in lists known as linked lists.

Each database page is a member of one *page type*. The basic page types are *data pages, index pages* and *control pages*. Data pages contain actual table row data. SQLBase has four types of data pages: row pages, extent page, long varchar pages and overflow pages. Index pages are used to store B+Tree indexes. Control pages maintain information for internal purposes such as which pages are empty or full. Each of these page types is discussed in detail in the following sections.

A database page contains data only for one page type. For example, BTree pages only contain data (consisting of symbolic keys) for a particular BTree index. Likewise, row pages only contain data.

A database page also belongs to one and only one *group* and contains data for only one group. A group is a database object such as a table or index, and a collection of pages of different page types. For example, a table is made up of *n* number of row pages, extent pages, LONG VARCHAR pages, and control pages.

Each database page is also assigned a logical page number. The logical page number is assigned sequentially within the group: the first page within the group has the page number of 1, the second page in the group 2, and so on.

	Page Type	Group No	Log. Ptr	Prior Page	Next Page
1	Data	28	1	∅	15

1st page in the customer table

	Page Type	Group No	Log. Ptr	Prior Page	Next Page
15	Data	28	2	1	27

2nd page in the customer table

	Page Type	Group No	Log. Ptr	Prior Page	Next Page
27	Data	28	3	15	∅

3rd page in the customer table
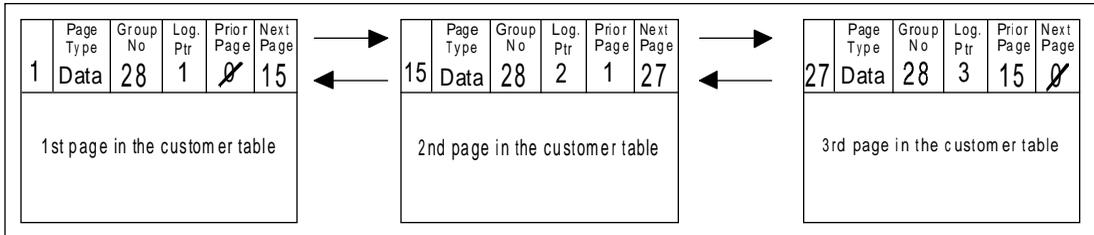
Figure 8C - All the pages of a specific group form a separate linked list. Each group has one page type and stores data only for that page type.

Data pages

As mentioned previously, SQLBase contains four basic types of data pages: row pages, extent pages, LONGVARCHAR pages, and overflow pages. Each of these pages types is discussed in the following sections.

Row pages

When a table row is originally stored in the database, it is stored on a row page. The specific location occupied by a database row on the page is called a *slot*. The number of slots that may reside on a page is a function of the size of the rows and the page size. SQLBase keeps track of the location and size of each slot within a database page through the use of a *slot table*. The slot table serves as an index to all the slots contained within a database page.

Each row page contains a generic page header, row page header, slot table, variable number of slots and a page footer. The slot table grows down the page from the row page header and the slots grow up from the page footer. The freespace within the page is between the slot table and the last slot added (see Figure D).

Page header

The page header is 53 bytes long and contains the physical page number, page type, group number, logical page number, next and previous page pointers, and a log record pointer (see Figure E). The log record pointer points to the record in the log that contains the before and after image of the page for the last SQL operation that modified the page.
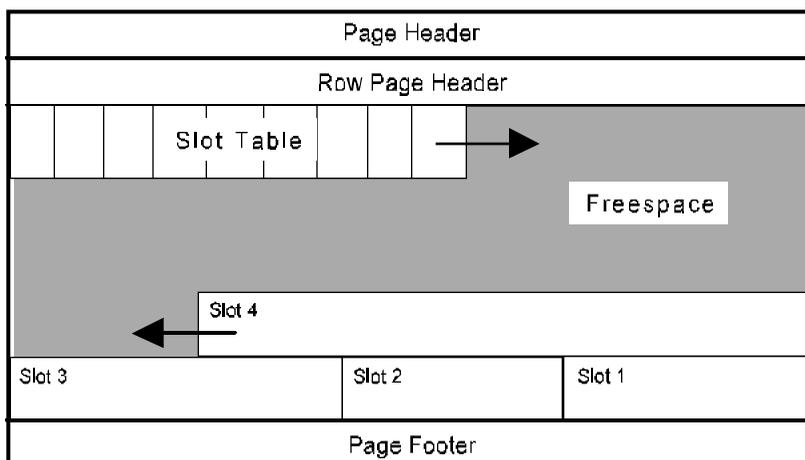
Figure 8D - Row pages contain a page header, row page header, slot table, variable number of slots and a page footer. The free space within a row page is between the last entry in the slot table and the last slot.

Row page header

The row page header is 16 bytes and contains the slot size (the number of entries in the slot table), available freespace (in bytes) and a pointer to the next (or first) extent page.

Slot table and slots

Every table row is stored in a separate slot. Each slot contains a row header followed by the set {column identifier, column length, value} for each non-null (and non-LONG VARCHAR) column in the row. The row header contains the row serial number, update serial number, and the number of columns in the row. Each slot has a corresponding entry in the slot table. A slot table entry is a two byte field. The first bit is used as an empty or full flag. The bit is on if the slot contains data; the bit is off is

the slot is empty. The remaining bits contain the byte offset of the slot within the page (see Figure E).
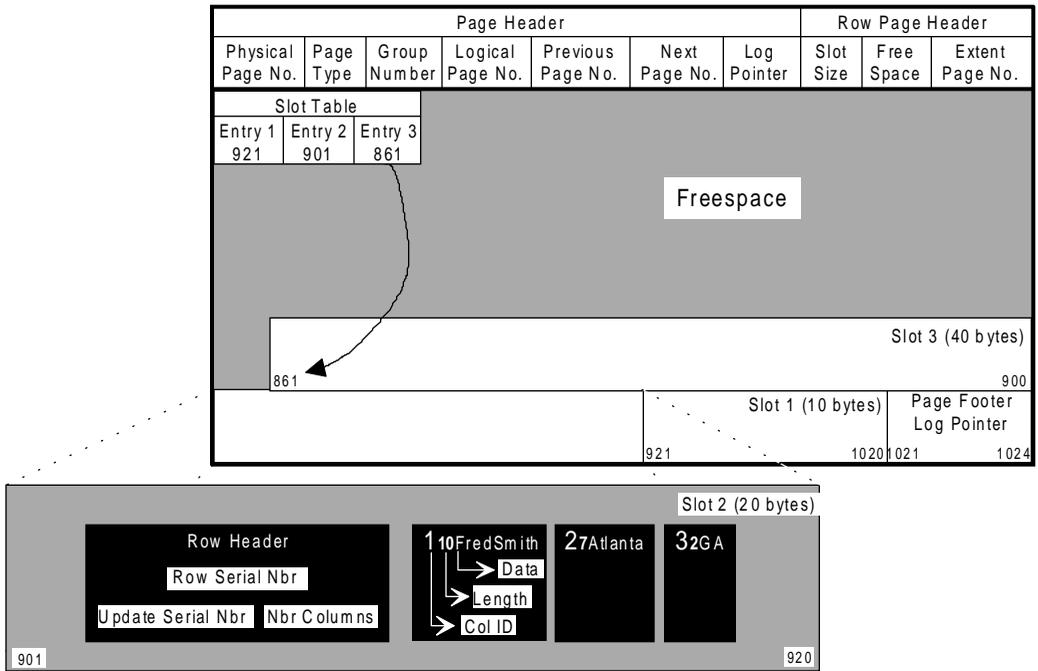
Figure 8E - Each table row is stored in a slot. The slot contains the slot header and the set {column id, length, data value} for each non-null (and non-long varchar) column.

The row serial number is a unique identifier assigned to each row in a table. The row serial number is assigned sequentially: the first row inserted into the table is assigned the row serial number of one, the second row is assigned two, and so on. The update serial number is a unique identifier assigned to every row in a common UPDATE or INSERT command. The update serial number is also assigned sequentially: all the rows updated/inserted with the first UPDATE/INSERT command are assigned an update serial number of one, all the rows updated/inserted with the second UPDATE/INSERT command two, and so on. The update serial number is used to prevent lost updates with the Cursor Stability and Release Lock isolation levels (see chapter 13) and infinite looping with UPDATE and INSERT commands.

Consider the SQL UPDATE command:

```
UPDATE TABLE Y SET X = X + 1 WHERE X > 10;
```

With this command, if the DBMS did not flag the rows in table y that met the where clause condition before the update command began, then the update command will

result in an infinite loop. The update serial number is used for this purpose in SQLBase, which contrasts with some other RDBMSs which disallow the placement of columns within the WHERE clause of an UPDATE statement which affects their contents.

Page footer

The page footer redundantly stores the log page pointer. This field serves as a consistency check and is used during transaction recovery to protect against partial page writes. Many Intel-based computers traditionally block hard disks with 512 byte sectors. This means the hard disk controller reads *and writes* in units of 512 bytes. However, the SQLBase page size is 1024 bytes. Thus, it is theoretically possible on such a computer for only a portion of a database page to be written to disk (when an undetected disk error occurs after the first sector is written but before the second 512 bytes sector is written). Thus, by storing the log page pointer in both the page header and footer, SQLBase can detect when this situation occurs and respond accordingly.

Rowid

SQLBase uses a *rowid* for all external references to table rows. A rowid is composed of the physical page number, slot number, row serial number and the update serial number. The physical page number is used to locate the row page containing the row. The slot number is used to locate the specific entry in the slot table that contains the offset of the slot (see Figure E).

The slot table/slot data structure is an indirection technique that allows a specific slot to move within a page without affecting external references (namely rowids). It is necessary to relocate slots within a page after the deletion of rows to reclaim freespace and avoid fragmentation of the page (see page garbage collection later in this chapter).

The row serial number is required to verify the row is actually stored in the specified slot. Since slots are reused, the row could have been deleted and another row stored in the same slot. The row serial number allows SQLBase to determine when a row pointed to by a rowid has been deleted. Likewise, the update serial number allows SQLBase to determine when a row referenced by a rowid has been updated by another user.

Extent pages

Extent pages are *extensions* to row pages and are used to store some of a table row's columns when all of the columns will not fit in one row page. Columns can become displaced in two circumstances, the most common of which occurs when column values are updated such that the length of the columns increase significantly in size.

As mentioned previously, when a table row is originally inserted into a table, all columns (except LONG VARCHAR columns) are stored in a row page (assuming the

row will fit into one page). However, later the row may be updated resulting in an increased row length. This can occur, for example, when a column that was null at the time of insertion is updated with a non-null value. When the row is updated such that its length is increased, sufficient freespace may not exist on the row page (the current base page) to store the new column values. When this situation occurs, SQLBase stores the new column values on an *extent* page.

The second situation which can cause columns to be displaced from their base page occurs when you attempt to insert a row with a row size greater than the usable row page size. Usable page size is the page size minus the sum of the page size reserved for later row expansions (through the use of the PCTFREE specification on the CREATE TABLE statement) and the page overhead. For example, for the NLM version of SQLBase, the page size is 1024 bytes and the page overhead is 86 bytes. Consequently, some columns will be displaced from the base page and stored on extent pages when you attempt to store a row with a row width of greater than 938 bytes, assuming the PCTFREE parameter is zero.

Extent pages, like row pages, contain a page header, row page header, slot table, variable number of slots and a page footer. However, the next and previous page pointers in the page header are not used; extent pages are linked together and to the base page in a separate linked list using the extent page pointer in the row page header (see Figure F).
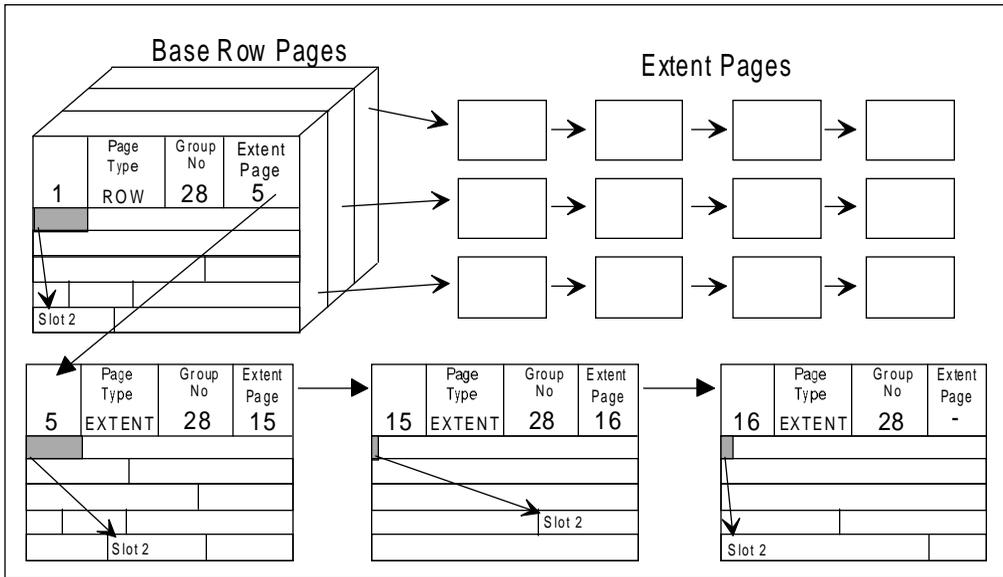

Figure 8F - Every row page may have up to ten extent pages which are linked to the base row page. Extent pages share the same group number with the base page. Displaced columns are stored in the same slot as the columns on the base page.

When an update or insert occurs such that the entire row cannot be stored on a base row page, SQLBase searches for the first extent page with sufficient freespace to contain the entire displaced columns. If no extent page is found which meets this criteria, SQLBase allocates one new extent page (up to a maximum of ten).

A given column, consisting of the set {column identifier, length, value}, is never split across pages. The slot in the base page contains no reference to the extent page or the displaced columns. The displaced columns are stored in the same position in the slot table in the extent page as the row to which they belong is stored in the base page. This means that a row's slot number remains unchanged from the base page value regardless of how many extent pages it may span. When retrieving a row, SQLBase first retrieves the base page and accesses all columns stored in the slot. If some columns are displaced, SQLBase searches sequentially through the chain of extent pages until all displaced columns are found.

Out of extent pages error

The 00802 ROW IS A "Insufficient (data page) Space Available" error occurs when a given row cannot fit within a row page and ten extent pages have already been allocated to the row page but none contains sufficient freespace to store the displaced columns. The 00802 error occurs most frequently during update commands where many columns in the row were originally inserted with null values. If this situation occurs infrequently and in conjunction with specific tables, the appropriate remedy is simply to reorganize the placement of the table rows on the database pages by exporting and importing the table. However, if the situation occurs frequently, this indicates that the nature of the data is such that rows are growing *wider* at a significant rate. In this case, the appropriate remedy is to adjust the PCTFREE parameter on the CREATE TABLE statement. The percent free parameter will cause SQLBase to reserve more freespace for expanding rows.

The 00802 error can also occur during the insertion of rows. If this situation occurs frequently, the average row width is simply too large and the database should be redesigned. One could define a table with 255 columns and attempt to store 254 characters in each column for a total row width of 65,025 bytes. This insert command would exceed the 10,318 total bytes available in the base pages plus ten extent pages (computed by multiplying the usable page space, 938, by 11) and result in the 00802 error. Because in most environments SQLBase uses 1K pages (the exception is in the UNIX implementations of SQLBase, which use a 2K page size), table width should be kept to less than 938 bytes (usable page space after overhead) to optimize row retrieval time.

Effect on performance

Extent pages negatively affect performance by requiring additional page I/O to access row data. If a row is stored without the benefit of extent pages (fitting completely on one row page), then the row can be accessed with one physical I/O. However, if the

row is spread over one row page and one or more extent pages then multiple I/Os may be required. Since extent pages are allocated on an as needed basis, it is unlikely the extent pages for a given row page would be stored in the same physical block (except following a reorganization). However, it is possible several extent pages all belonging to the same row page are stored in the same physical block.

For example, assume a "narrow" row (meaning one with a row width less than the usable row size minus reserved space) is inserted into the database and stored either on an existing page with sufficient freespace or a new page which has no extent pages allocated at the time of the insert. Now assume that at some later point in time the row is updated with increased column widths which must be stored on an extent page and that SQLBase allocates a new extent page for this purpose. At this time, it is very likely that the pages physically adjacent to the base page have already been allocated for other purposes (such as additional row pages or extent pages for other base pages). Consequently, the page which SQLBase does allocate for the extent page will not be physically adjacent to its base page.

On the other hand, consider the following. Assume a "wide" row (meaning one with a row width greater than the usable row size minus reserved space) is inserted into the database. Also assume the row can be stored on one base page and one extent page. Since the row width is greater than the usable row size minus the reserved space, SQLBase will be forced to store the row on a new page (this is because the existing page will not contain sufficient freespace to store the row). Consequently, at the time of the insert, both the base page and extent page will be allocated. If the first two pages listed in the group's free extent list are physically adjacent, then the base page and extent page will be collocated. Otherwise, the base page and extent page will be physically stored in separate blocks on the I/O device and two physical I/Os will be required to retrieve the entire row.

One way that this latter situation can occur is if the group's free extent list does not contain two free pages and SQLBase must allocate additional pages to the group, or if some pages have been de-allocated and placed back in the group's free extent list. Conversely, this latter situation cannot occur during a database reorganization since no other users are connected to the database and thus are not competing for page allocations. During a reorganization, SQLBase collocates all row pages with their respective extent pages. However, it is possible that the row page and its extent page are collocated but not stored in the same physical block. This situation would occur when the base page was physically adjacent to the extent page but also crossed a block boundary.

In any event, a database with too many allocated extent pages usually results in decreased performance. A general design goal should be to minimize the number of extent pages. To accomplish this, the DBA is encouraged to reorganize the database frequently. In addition, if it is known that most of the rows in a given table will significantly grow in width over time, then the CREATE TABLE's PCTFREE

parameter should be adjusted accordingly. Further, if the average row width is close to the usable page size, the database designer should consider splitting the table into two tables.

Long varchar pages

SQLBase stores all long varchar column data in long varchar pages. Long varchar pages contain only a page header, page footer and long varchar data. Long varchar pages are chained to the base page by pointers in the base page slots. If a row contains one or more non-null long varchar columns then its slot contains the set {column identifier, column length required in the base page, first long varchar page physical page number, last long varchar page physical page number, column size} for each non-null long varchar column. Note that the column length required in the base page is the size of the fields stored in the base page slot which for LONG VARCHAR columns is always 12 bytes (consisting of the first long varchar page physical page number, last long varchar page physical page number, column size). This page range forms a linked list in the usual way, long varchar pages contain next and previous page pointers in the page header (see Figure G).
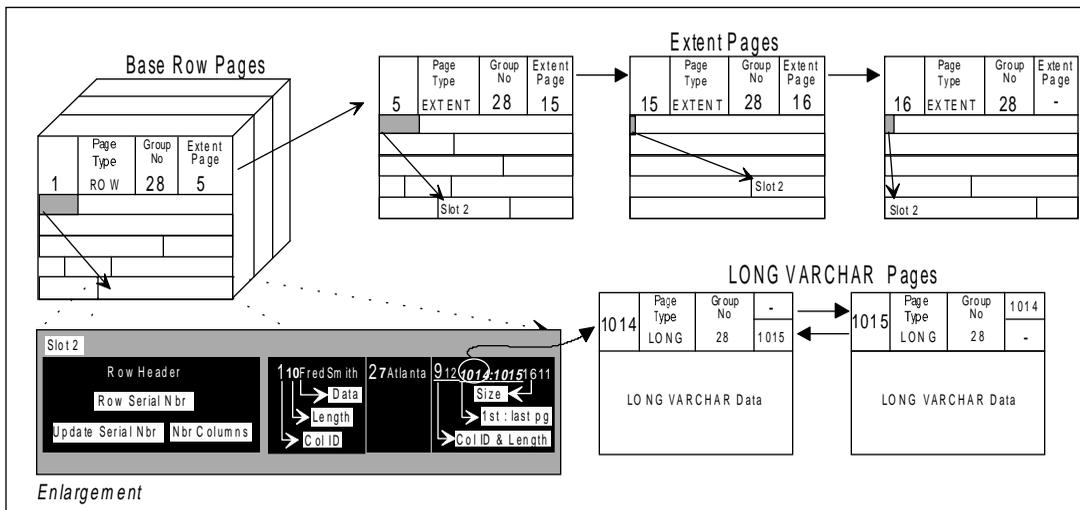
Figure 8G - Unlike EXTENT pages, LONG VARCHAR pages are linked to their base page by pointers in the slots. LONG VARCHAR pages are also linked together with next and prior pointers. Please note in the enlargement of the base slot above that displaced columns, those stored on EXTENT pages, are not referenced in the base slot but LONG columns are referenced.

The purpose of long varchar pages is to separate and isolate long varchar data from columns of other data types so that the storage of this type of data does not negatively affect the access of the other columns. Long varchar columns are used to store large quantities of unstructured data such textual comments, bitmaps, video clips, and so on. Consequently, one long varchar column, because of its size, will likely saturate many pages. If these columns were stored in row pages, the result would be heavy use of extent pages that would decrease performance to columns of all data types. By storing long varchar columns on separate pages, access to non-long varchar columns is unaffected.

On the other hand, a database access to a long varchar column as well as other columns requires at least two physical I/Os (one for the row page and one for the long varchar page). In addition, SQLBase stores only one column on long varchar pages. If a column is defined as LONG or LONGVAR but its value is only one byte, an entire long varchar page is used to store this one byte of data. Consequently, the LONG data type should only be used when the values of the attribute are known to exceed 254 characters; otherwise, the VARCHAR or CHAR data type should be used.

Overflow pages

Overflow pages store rows when an overflow condition occurs in conjunction with a clustered hashed table (see chapter 10). When a row hashes to a page which is already full, SQLBase stores this row on an overflow page. There are two types of overflow pages, designated overflow pages and dynamically created overflow pages. All overflow pages, regardless of type, participate in the same linked list as base row pages (see Figure C).

Designated overflow pages

When a CREATE CLUSTERED HASHED INDEX statement is executed, SQLBase pre-allocates and creates row pages. The number of pages allocated depends on the number of rows declared in the CREATE CLUSTERED HASHED INDEX statement and the width of the table (see chapter 10). For every N number of row pages pre-allocated, SQLBase also pre-allocates a designated overflow page (see Figure H). If

the clustered hashed index is created on a single number field, N is 100; otherwise, N is 10.
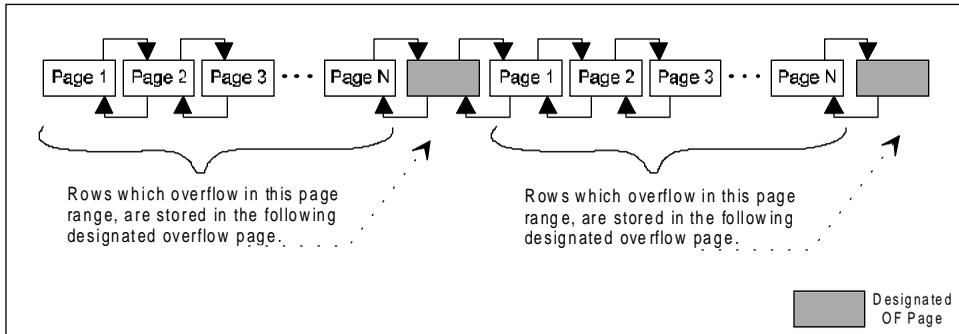
Figure 8H - Designated overflow pages are created for every N number of row pages and participate in the same linked list as the base row pages.

Dynamically created overflow pages

Dynamically created overflow pages are created whenever both of the following conditions are met:

- A designated overflow page becomes full
- SQLBase attempts to store another row in the designated overflow page

When this situation occurs, SQLBase will dynamically create another overflow page. This new overflow page is then inserted into the linked list between the designated overflow page and the next base row page (see Figure I). In this manner, SQLBase will dynamically create as many overflow pages as is necessary to store the rows which overflow in a given page range.
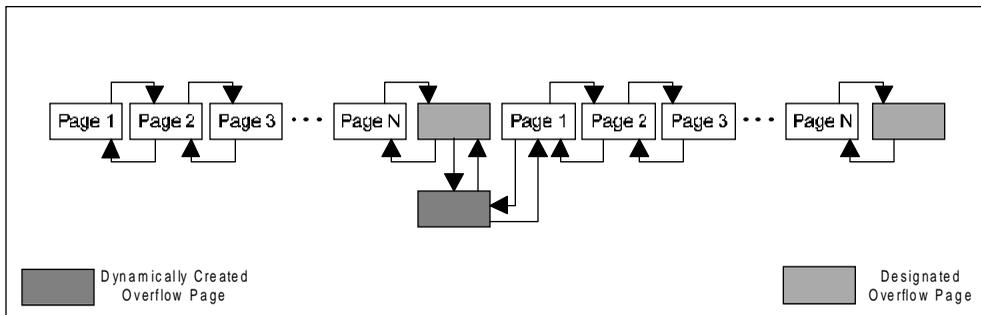
Figure 8I - SQLBase dynamically creates as many additional overflow pages as is necessary to store the rows which overflow in a given page range. These additional

overflow pages are inserted in the linked list between the designated overflow page and the next base row page.

SQLBase follows these steps when attempting to retrieve a row from a hashed table:

1. Hashes the key to obtain the target row page number;

2. Retrieves the target row page and searches the slots for the desired row;

3. If the desired row is not found on the target row page, SQLBase then retrieves the designated overflow page (for that page range) and again searches the slots for the desired row;

4. If the desired row is not found in the designated overflow page, SQLBase searches in turn the dynamically created overflow pages.

Hence, too many overflow pages degrades performance. Care must be taken when choosing a clustered hashed index key to ensure too many overflow pages will not be required for any specific page range (see chapter 10).

Index pages

Index pages contain B^+–tree indexes (refer to chapter 9, *BTree Indexes*, for a detailed description of B^+-trees). Index pages have the same basic organization of all other page types. Each index page contains a page header, index page header, slot table, slots and page footer. The page header, slot table and page footer is identical to data pages (see Figure J).
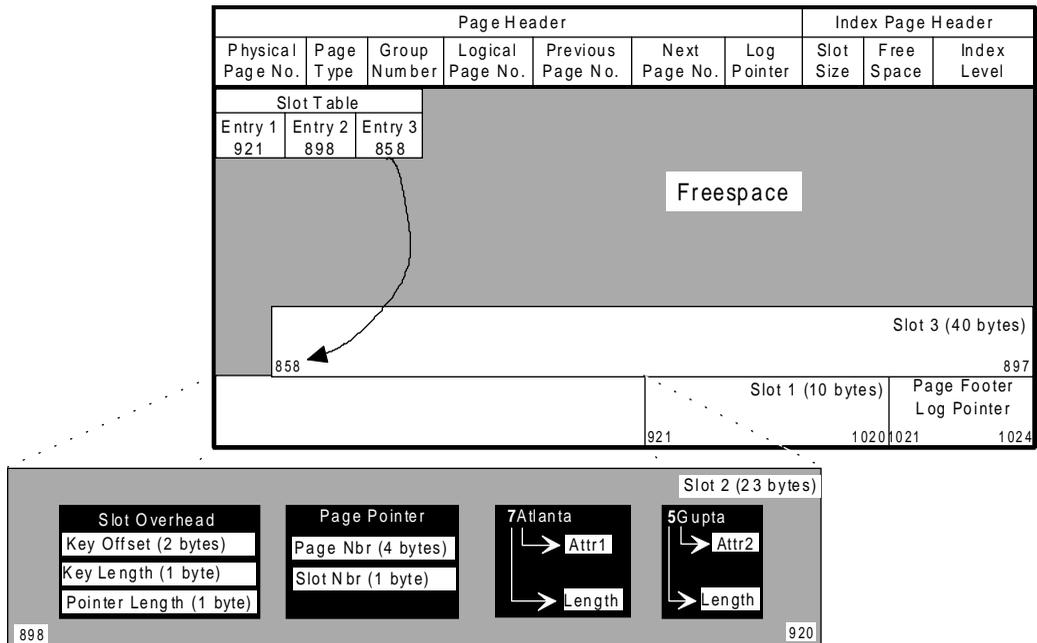
Figure 8J - Index pages have the same basic layout as row pages: each page contains a page header, index page header, slot table, and multiple slots.

The index page header and slots within an index page differ from data pages. The index page header contains the slot size, free space and the *level* in the index (see Figure J). The slot contains the slot overhead, one page pointer and one or more key attributes depending on whether the index is created on one or more table columns. In the leaf pages, the page pointer points to a data page where the key values may be found. Otherwise, the page pointer points to the next level in the index.

All index pages belong to the same group; however, non-leaf pages are not linked together. Therefore, the next and prior page pointers in the page header are not used. Leaf pages are linked together in the usual way and form what is known as the *sequence set.*

Control pages

Each SQLBase database contains a number of *control pages* for a variety of
housekeeping chores. Some of these control pages store various information about
the database as a whole. For example, the database control block is used to store
database parameters. Each SQLBase database also has one free extent list which is
used to keep track of un-allocated pages. Other control pages are used for groups and
tables. The group extent map is used to track pages allocated to a specific group.
SQLBase also has several types of control pages for each table defined in the
database. Each of these types of control pages is discussed in the following sections.

Database control block

The first page of every SQLBase database is referred to as the database control block,
or DCB. This page contains various database attributes such as:

- SQLBase version number.
- Recovery flag which indicates whether SQLBase shuts down gracefully or
  not.
- When checkpoints were done.
- Current log file.
- Next log to backup.
- Read-only flag which indicates whether the read-only isolation level is
  available.
- Logging flag which indicates whether logging is enabled.
- Log file pre-allocation flag which indicates whether log files are pre-
  allocated or not.
- Log file size field, which is maintained through the SET LOGFILESIZE
  command and indicates what size log file should be pre-allocated when using
  pre-allocation.
- A flag indicating whether the database is read-only, maintained through the
  SET READONLYDATABASE command.
- The extension size field, maintained through the SET EXTENSION
  command, which specifies the disk space increment at which the database file
  will grow.
- The log file span limit, maintained through the SET SPANLIMIT command,
  which specifies the maximum number of log files that one transaction may
  be spread across.

- The checkpoint time interval, maintained through the SET CHECKPOINT command, which specifies the frequency in minutes between recovery checkpoint operations.

When a user connects to a SQLBase database, SQLBase checks the version number in the DCB to verify that a compatible version of SQLBase created the database. Likewise, the recovery flag and current log file are used to determine whether SQLBase should initiate crash recovery operations to rollback uncommitted transactions (see Chapter 14, Logging and Recovery). The remaining fields stored in the DCB are similarly used by SQLBase for various housekeeping functions.

The DCB is also used to store user parameters for the database. Any database specific parameter set by the user and not stored in the *sql.ini* file is stored in the DCB.

Free extent list

As discussed previously, a SQLBase database file is a collection of pages. Database pages are either allocated to a group or un-allocated. The free extent list is a set of database control pages which maintain a record of which pages are un-allocated ("free" for use). This list is physically stored as a B+Tree index (see Chapter 9, BTree Indexes) where each node in the index includes the set {page number, number of contiguous un-allocated pages}.

Group free page list

If a page is not in the free extent list, then it has been allocated to a group. For each group, a list is maintained of its allocated pages. Like the free extent list, the group extent map is physically stored as a BTree index where the node of the index contains the set {page number, number of contiguous allocated pages}.

Group extent map

The Group Extent Map maps logical pages which comprise a group to physical database pages.

Bitmap pages

SQLBase uses bitmap pages to track which row pages have freespace (remember the row page header contains the amount of freespace. See Figure E). Each table within the database has a set of bitmap pages for this purpose. Each of the 8192 bits (1024 bytes * 8 bits per byte) of the bitmap page (less overhead) maps to one row page. When the bit is off, this signals the page has room for additional rows. When the bit is on, the page does not have room for additional rows. There is one bitmapped page for approximately every 8192 row pages allocated to the table's group (a bitmap page actually cannot track the freespace of 8192 pages since a portion of the bitmapped page is used for overhead purposes, these details are ignored in this example). The

first logical page within the group is a bitmap page and every 8192 pages thereafter is a bitmap page. As SQLBase inserts, updates and deletes rows from a specific row page, the freespace field in the row page header is adjusted accordingly. In addition, when the page is 80% full, the corresponding bit in the corresponding bitmap page is turned on. When allocated space drops to 50% or below, the bit is turned off.

Row count page

In addition to bitmap pages, each table defined in the database contains one row count page. This page contains only one field, the number of rows in the table. This page is updated whenever rows are inserted or deleted in the table. Consequently, this field is frequently accessed and could become a bottleneck if it were not stored on a separate page.

Table data page

Each table in the database also has one table data page. This page is used to store:

- Last row serial number assigned
- Page number of last row page
- Table size in the number of pages (hashed table only)
- Number of overflow pages (hashed table only)

The first two fields are accessed by SQLBase whenever new rows are inserted into the database. The last assigned serial number is used to determine the serial number for the new row. When a new row is inserted into the table, SQLBase uses the last row page number to quickly locate the last page. The table data page contains the latter two fields only if the table is hashed (is the subject of a clustered hashed index).

Page allocation and garbage collection

The process of allocating new pages and reclaiming space in existing pages when rows are deleted is know as page allocation and garbage collection. Each of these processes is discussed in the following sections.

Page allocation

When SQLBase receives a command to insert a row into a table, SQLBase accesses the corresponding table data page in order to retrieve the page number of the last row page in the linked list. SQLBase then retrieves this page and checks the freespace field in the row page header to determine if room exists to store this specific row. If so, the new row is inserted into the last page in the linked list and the freespace field in the row page header and corresponding bitmap page are updated accordingly.

If the last page in the linked list does not contain sufficient freespace to accommodate the new row, SQLBase begins searching each bitmap page in turn, starting with the last bitmap page accessed and wrapping around to the first bitmap page if necessary, for an existing row page with its bit turned off. If one is found, SQLBase retrieves this page and checks the freespace remaining in the row. If sufficient space exists, then the row is stored on the page. If sufficient space does not exist, SQLBase continues its search of the bitmap pages. It is also possible for SQLBase to encounter a fragmented page; that is, a page with enough total free space to store the incoming data, but not enough contiguous free space. In this case, garbage collection is performed to compact the page and coalesce all of the free space into one chunk (see *"Garbage collection"* below for details).

If all bitmap pages are searched and no page is found with sufficient freespace, SQLBase then checks the free extent list. If un-allocated pages exist in the free extent list, SQLBase allocates a set of pages, known as an extent, to the table's group by removing them from the free extent list and adding them to the group extent map. At this time, SQLBase also updates the appropriate bitmap page and/or creates new bitmap pages if necessary. Groups grow in intervals of 10% or five pages, whichever is larger. For example, if the group currently consist of 100 pages, 10 new pages are allocated to the group. On the other hand, if the group only contained 40 pages, then the extent would consist of five pages.

If no unallocated pages exist in the free extent list, SQLBase performs the following steps:

1. Obtains additional space from the file system. The SQLBase database file is not fixed to a certain size; it grows as additional space is required. When a SQLBase database requires additional storage space, additional pages are added to the database file. This process of adding additional pages to the database file is resource intensive, increasing response time for currently active transactions. Consequently, pages are not added one at a time but in groups of pages called extents. The extent size may be set by the user with the SET EXTENSION command. For non-partitioned databases, the default extent size is 100 pages.

2. Adds extent pages to the free extent list.

3. Removes pages from the free extent list and added to the group extent map.

4. Updates the row and freespace field stored in the new page and the corresponding bit in the bitmap page.

Garbage collection

When a row is deleted from a page:

- The utilization bit in the slot table is turned off, indicating that the slot is available for re-use.

- The freespace value in the row page header is increased by the row size.

- If freespace is now greater than 50% of the usable page size minus reserved space, the corresponding bit in the bitmap control page is turned off.

At the time of the deletion, the slot data is not removed and the page is not compacted. Slots are reshuffled to free up space when an insert or update (which increases the size of the subject row) occurs. In this way, garbage collection is only performed when needed.

Chapter 9

B-Tree Indexes

In this chapter we:

- Introduce the concept of tree structured indexes, particularly the B-Tree index, a variation of which is used by SQLBase.

- Discuss the general algorithms used to maintain B-Tree indexes.

- Introduce the SQLBase variation of B-Tree indexes, the B^+-Tree index, by contrasting it to the B-Tree index.

- Explain the variations between the algorithms used by SQLBase to maintain B^+-Trees and the basic algorithms already covered.

- Discuss the unique considerations that apply to SQLBase indexes.

Introduction

B-Tree indexes have become the de facto standard for random access to records (or rows) within a database file. All major RDBMS products use one variety of B-Trees or another for general purpose indexing. SQLBase uses the B$^+$-Tree variation first proposed by Knuth in 1973.

Concerning the history of B-Tree indexes, Douglas Comer documents:

> The B-tree has a short but important history. In the late 1960s computer manufacturers and independent research groups competitively developed general purpose file systems and so-called "access methods" for their machines. At Sperry Univac Corporation H. Chiat, M. Schwartz, and others developed and implemented a system which carried out insert and find operations in a manner related to the Bl-tree method. . . . Independently, B. Cole, M. Kaufman, S. Radcliffe, and others developed a similar system at Control Data Corporation . . . R. Bayer and E. McCreight, then at Boeing Scientific Research Labs, proposed an external index mechanism with relatively low cost for most [file] operations; they called it a B-tree. The origins of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others have suggested that the "B" stands for Boeing. Because of his contribution, however, it seems appropriate to think of B-trees as "Bayer"-trees.

B$^+$-Tree Before presenting the details of the B$^+$-Tree variation used by SQLBase, it is necessary for the reader to understand simple binary search trees, multiway trees and B-Trees in general. This chapter presents a general introduction to binary search trees, multiway trees, and B-Trees. Following this general introduction, B-Trees are presented along with the SQLBase implementation.

Binary search trees

When traversing a binary search tree, the path chosen is determined by comparing the target key to the key value stored at each node. If the target key is less than the node key, the left branch is followed. Otherwise, the right branch is chosen. Figure A shows part of such a search tree and the path taken for the target key "15."
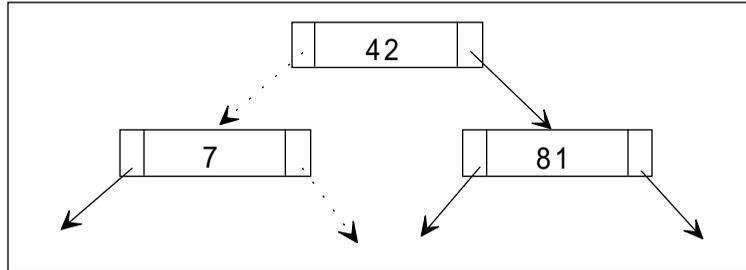
Figure 9A - This figure shows a portion of a binary search operation. The path followed for the target key '15' is illustrated with dotted lines.

Each node in a binary search tree contains exactly one key and has a branching factor of two: every node contains a maximum of two children. Thus, binary search trees are "tall" trees. The minimum height of a tree containing N records is [log2N]+1. For example, a tree with only 100 records will have at least seven levels.

When trees are used for index structures, "short" trees are preferable since they typically exhibit better performance characteristics. This is because typically one physical I/O is required for every level traversed in the index, and in order to speed database access times, physical I/O must be kept to a minimum.

Note also that in index structures, a record in a node contains the set {Key Value, Pointer}. The pointer is the physical address in the main file where the key value can be found. Figure A above contains only the key values in the nodes of the tree in order to simplify the illustration. In the remainder of this chapter we continue to follow this same practice.

Multiway trees

Multiway trees are a generalization of the binary search tree data structure. With multiway trees, each node contains N number of keys. For example, consider Figure B which illustrates a multiway tree with a node size of 2.
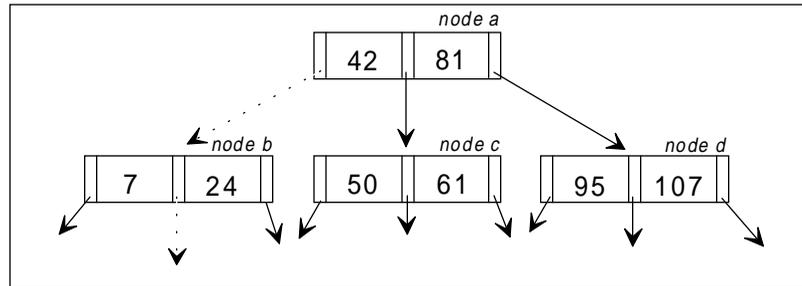

Figure 9B - This figure shows a search tree with two keys and three branches per node. The path followed for the target key "15" is illustrated with dotted lines.

Consider the path followed to locate the target key "15." First, node *a*, or the root of the tree, is examined. The target key of "15" is less than "42", the first key in node *a*, so the left most branch is followed. Next, node *b* is examined. The target key is greater than "7", the first key in node *b*, so the target key is compared to the second key in node *b*. Since "15" is less than "24" the middle branch is followed. This algorithm is repeated until either an exact match is found or a leaf node is encountered (a leaf node is any node which is the furthest from the root, in other words, a leaf node has no child). If a leaf node is examined with no exact match found, then the target key is not contained within the index.

Multiway trees are generally "shorter" than binary trees since their branching factor is greater. However, complex algorithms are required to keep multiway trees "balanced." In a balanced tree, all leaf nodes appear at the same level, and the tree has the same height regardless of which leaf node is measured. A balanced tree is desirable since it results in uniform performance.

B-Trees

B-Trees are balanced multiway trees with the following additional key property: each node need not contain exactly N number of keys. Nodes are allowed to "grow" from half full to full. This additional property greatly simplifies the balancing algorithms and significantly reduces the cost. B-Trees are a classic example of the disk space versus performance trade off. Often, as in the case of B-Trees, additional disk space can significantly increase performance. The rationale used in these cases is that,

presumably, additional disk space is relatively inexpensive and justified for the desired increase in performance.

A B-Tree *of order d* can be defined formally as a tree with the following properties:

- Intermediate nodes (meaning nodes which are not leaves or the root) have:

    - At least d records (keys) and no more than 2d records (keys);

    - At least d+1 children (pointers) and no more than 2d+1.

- The root, unless the tree has only one node, has at least two children.

- All leaf nodes appear at the same level, that is, are the same distance from the root (meaning the tree is balanced).

The order of a B-Tree

The order of a B-Tree describes the "bushiness" of the tree: the number of records stored in each node and consequently the number of child nodes. The root and leaf nodes are exceptions but all other nodes in a B-Tree of order d have between d and 2d records and between d+1 and 2d+1 children. For example, a *regular* node (meaning one that is not a leaf or root node) in B-Tree of order 10 has at least 11 and not more than 21 children. The lower bound ensures that each node is at least half full; therefore, disk space is not unnecessarily wasted. The upper bound allows the B-Tree to be mapped to fixed length records in an index file. The order also effects the performance of the B-Tree index. If the B-Tree gets to "tall" (has too many levels), performance will suffer. Likewise, if the order gets too "wide" (has too many records per node), performance will also decline.

Sequencing within a node

In addition to the data structure properties defined in the formal definition above, each node of a B-Tree index also exhibits some sequencing properties. Specifically:

- Within a node of K records, records may be numbered R_1, R_2, R_3, ..., R_K and pointers to child nodes may be numbered as P_0, P_1, P_2, ..., P_K. Figure C illustrates a typical node using these number semantics.

| P_0	R_1	P_1	...	R_K	P_K
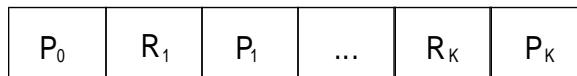
Figure 9C- This figure illustrates a typical node in a BTree index. Note that each node has K records but K+1 pointers to child nodes.

- Records in the child node pointed to by P_0 have key values less than the key of R_1. Records in the child node pointed to by P_K have key values greater than

the key of R_K. Records in the child node pointed to by P_i (where $0 < i < K$) have key values greater than the key of R_i and less than the key of R_{i+1}.

Balancing B-Trees

The popularity of B-Tree indexes results from their performance characteristics which is derived from their balancing algorithms. Records are usually inserted into and deleted from a database (and therefore an index) in an arbitrary fashion. Random insertions and deletions can cause trees to become unbalanced (see Figure D).
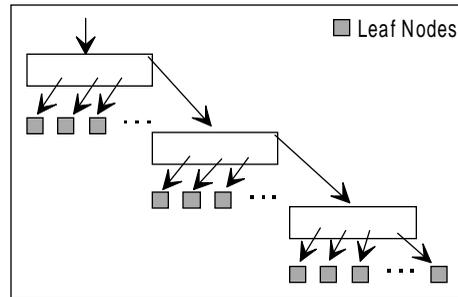
Figure 9D - Unbalanced trees have some short and some long paths from the root to the leaf nodes.

With an unbalanced tree, some leaf nodes are further from the root than others; consequently, path lengths and ultimately retrieval times vary. Some key values may be located by only visiting a few nodes; other key values may require accessing a great many nodes. Balanced trees, as mentioned previously, are trees whose leaves are all the same distance from the root. This characteristic results in uniform search paths and thus consistent retrieval times. However, in order to maintain a balanced tree through random insertions and deletions special balancing algorithms must be employed.

B-Tree balancing algorithms are specially derived to minimize balancing costs. First, B-Tree balancing algorithms confine node changes to a single path from the root to a single leaf. In addition, by utilizing extra storage (where some nodes are not 100% full), balancing costs are also lowered. Since nodes are sometimes not full at the time of an insert, the new record can simply be inserted into the node (therefore no balancing action must take place). Similarly, when deletes occur, sometimes the target node contains more than the minimum number of records so, again, the record is simply deleted. Only when the node contains 2d records (inserts) or d records (deletes) are balancing steps required.

Insertions

Figures E, F and G graphically illustrate the classic B-Tree insertion algorithm. Figure E is the base tree before any insertions. To insert the key "Q" into the B-Tree of Figure E, the find algorithm is followed to locate the leaf node where the new key belongs. If we follow this algorithm we discover that node 7 is the node where the new key belongs. Since this node has only two keys, the new key value of "Q" can simply be inserted in the third record.
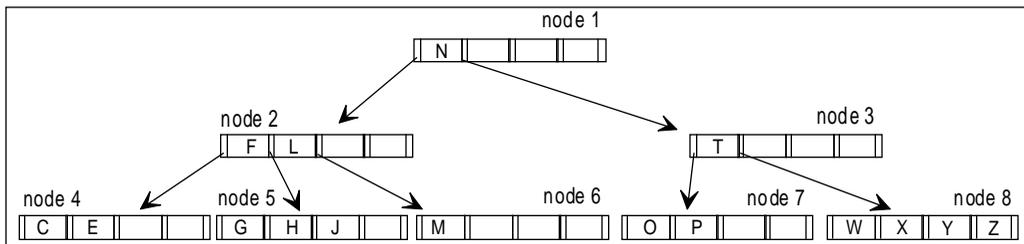
Figure 9E - This is a B-Tree of order 2, meaning each node has between two and four children.

Figure F is the same tree as Figure E after the key of "Q" has been inserted. This insertion represents the usual case: no balancing steps are required because the node is not full. B-Trees trade extra storage space (when some nodes are not full) for added performance.
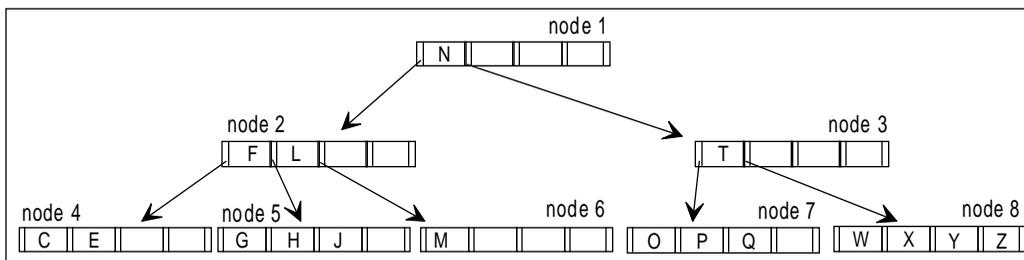
Figure 9F - This is the same B-Tree as Figure E after the key "Q" is inserted.

Let's examine a more complex example. Suppose we insert the key value of "U" into the same tree. In this case, "U" belongs in node 8; however, this node already has four keys. In this situation, a "split" must occur. To split a node, the keys with the lowest values are placed in one node and the keys with the highest values are stored in a new node. In addition, the "middle" key is propagated to the parent node and serves as a separator. Figure G contains the same tree as Figure F after "U" is inserted. In this example, the parent node contains free records for additional keys. If the "middle" key cannot be placed in the parent because the parent node is also full, then the parent

node is also split. In the worst case, this splitting occurs all the way to the root and the B-Tree grows by one level.
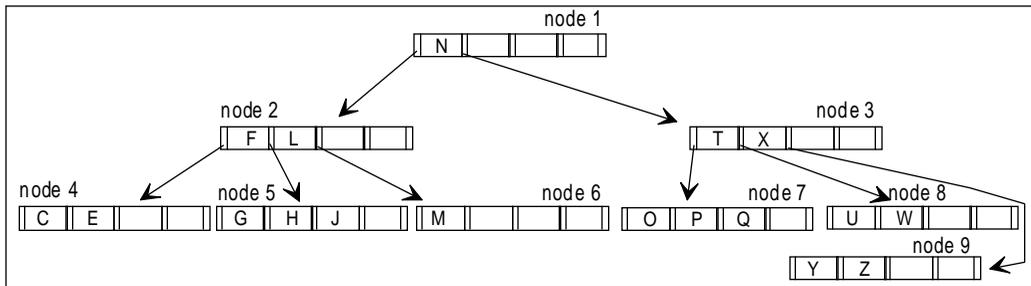
Figure 9G - A representation of the same B-Tree as Figure E after the key "U" is inserted.

Deletions

The delete algorithm, like the insert, begins with the find or retrieval operation in order to locate the node which contains the target key. The target node may be found either at the leaf level or at any other level within the tree.

If the target node is a leaf node, the key is simply removed. Since the majority of the keys are stored at the leaf level (because there are many more leaf nodes than non-leaf nodes), most of the time the target node will be found at the leaf level.

If the target node is not a leaf node then the key cannot simply be removed. The key must be replaced with a new "partitioning" key so that future find operations are guided properly to the child nodes. The general procedure is to replace the target key with the next highest key value, or what is known as the "successor" key. The successor key will always be found at the leaf level by traversing the right-most branch of the tree. Once the successor key is located, it is simply moved from the leaf node to the target node. In other words, the successor key replaces the target key (the one to be deleted).

In either case, whether the target node is a leaf or not, the above step causes one leaf node to shrink in size by 1 record. When this occurs, the affected leaf node may now have too few keys, meaning it is less than half full. This situation is termed an *underflow.* Two techniques are then used to deal with the underflow condition: redistribution and concatenation.

If either one of the adjacent siblings of the affected leaf node contain excess records (where the nodes are more than half full) then the redistribution technique may be utilized. Redistribution simply moves keys among one sibling node, the affected node and the parent such that the underflow condition is remedied.

On the other hand, if neither of the affected leaf node's adjacent nodes have excess records, then the concatenation technique must be applied. Concatenation involves merging nodes and is, in effect, the opposite of the splitting operation used in the insertion algorithm. Like splits, concatenations may propagate all the way to the root. However, in this case, if the propagation travels to the root, the tree shrinks by one level (rather than growing by one level as with splits).

Detailed examples of the deletion algorithm may be found in Comer, Smith and Barnes, Knuth, as well as any introduction text to index organizations. Wirt provides sample routines programmed in Pascal.

B-Tree operation costs

Knuth presents a detailed analysis of B-Tree retrieval, insertion and deletion costs. In the following sections we present a simplified model of B-Tree operational costs.

Retrieval costs

If we assume each node in a B-Tree is mapped to a block on a physical I/O device, then each node visited in a find operation requires one physical I/O. Consequently, retrieval cost is proportionate to the number of nodes visited to locate the target node.

The number of nodes visited during the find operation is directly related to the height of the B-Tree since most nodes will exist at the leaf level. In other words, most of the time, the find operation will traverse the tree all the way to the leaves before the target key is located. Hence, the number of physical I/Os required to locate the target key is approximately equal to the height of the tree. Knuth and others have shown that for a B-Tree of order d with n total keys:

$$\text{height} \leq \log_d \left(\frac{n+1}{2} \right)$$

Thus, the cost of the find operation grows as the logarithm of the total number of keys. The following table shows how efficient B-Trees are in terms of I/O due to their logarithmic cost function, since the height of the index is generally equivalent to the I/Os required to locate a leaf entry:

Nbr Keys => Node Size	1K	10K	100K	1M	10M
10	3	4	5	6	7
50	2	3	3	4	4
100	2	2	3	3	4

Nbr Keys => Node Size	1K	10K	100K	1M	10M
150	2	2	3	3	4

Even with 1 million keys, one can locate a target key within a B-Tree of order 50 in only four physical I/Os. In fact, Knuth and others show that this cost can be improved upon in practice by some simple implementation adjustments. The average cost is somewhat less than three physical I/Os.

Insertion and deletion costs

Both the insertion and deletion algorithms begin with the find operation. Consequently, insertion and deletion cost is a superset of retrieval costs. In addition, when a key is inserted or deleted into the tree, in the worst case splitting or concatenation continues up the tree to the root. Therefore, maximum physical I/O for an insertion or deletion is double that of the find operation.

B$^+$-Trees

As we have shown, B-Trees offer near optimal performance for random access to one record. However, sequential processing using a standard B-Tree is both inefficient and difficult. After locating the first record using the find operation, to locate the next sequential record may require traversing from the leaf to the root and back down to another leaf. Thus, in the worst case the cost of the next operation is:

$$MaxCostOfNext = \log_d n$$

For example, consider the following SQL SELECT statement:

```
SELECT * FROM CUSTOMER WHERE ID=1234;
```
This SQL statement retrieves only one row; consequently, a standard B-Tree index provides efficient random access to this one row. On the other hand, consider another SQL SELECT statement:

```
SELECT * FROM CUSTOMER ORDER BY ID;
```
Unlike the first SQL statement, this SQL statement retrieves all customers in sequence by the ID attribute. To satisfy this query, one find operation would be issued to locate the first customer followed by a find next operation for every customer stored in the customer table. Since the customer table could contain millions of rows and each find might require *MaxCostOfNext* (as defined above) physical I/Os, the cost of processing this query using the standard B-Tree would be quite large.

B$^+$-Trees are derived from the standard B-Trees. They add efficient sequential processing capabilities to the standard B-Tree yet retain the excellent random access performance.

Before defining B$^+$-Trees, it is necessary to define two terms:

- "Index nodes" refers to all nodes not at the leaf level, meaning the root and all subsequent levels excluding the leaf level;

- "leaf Nodes" or "leaves" are all nodes at the leaf level.

B$^+$-Trees differ from B-Trees in that (see Figure H):

- All pointers to the main file are removed from the index nodes;

- All key values are stored in leaves regardless of whether they exist in index nodes;

- Leaf nodes are chained together into a linked list known as the sequence set.
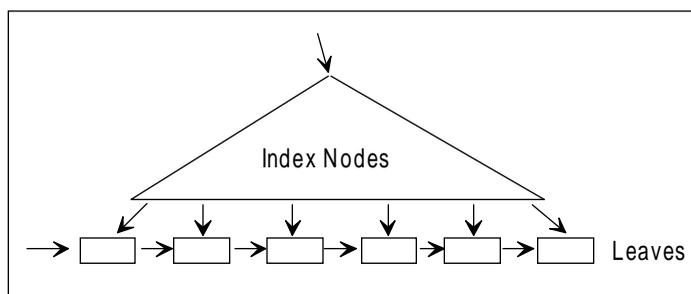
Figure 9H - A B+-Tree is composed of index nodes and leaf nodes that are linked together in a linked list known as a sequence set.

These structural differences result in several key implications:

- All find operations must travel to the leaf level to obtain the pointer to the main file; thus, the index nodes serve as a "road map" to the appropriate entry point to the sequence set.

- Since pointers to the main file are removed from the records in the index nodes, more records can be stored in the index nodes; thus, the performance is on a par with that of B-Trees.

- Sequence set provides an efficient mechanism for sequential processing: at most the cost of the next operation is one I/O.

Prefix B⁺-Trees

Prefix B$^+$-Trees are a slight variation of B$^+$-Trees. Since the index nodes only serve as a road map to the leaves and all operations must travel to the leaf nodes, the index nodes do not have to store the full key values. The index nodes must store only enough of the key value to distinguish one branch from another and direct the find operation to the appropriate leaf (see Figure I).
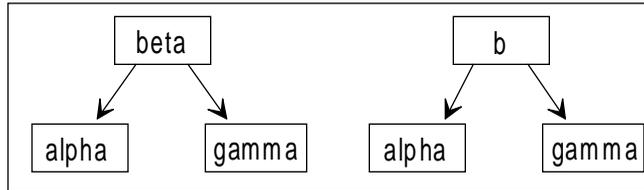
Figure 9I - The tree fragment on the left contains the full key in the index node. The tree fragment on the right contains only the "minimum distinguishing key."

Prefix B$^+$-Trees store the "minimum distinguishing key" in the index nodes. As a result, many more index records may be packed into the index nodes thus increasing the performance of the B$^+$-Tree by decreasing the height of the tree.

SQLBase implementation

SQLBase B-Tree indexes are based on the B-Tree variant described above as the prefix B$^+$-Trees. This B-Tree variant provides excellent random and sequential performance. Various aspects concerning SQLBase's implementation of prefix B$^+$-Trees are discussed below.

Index node size

The performance of a B-Tree is, as we discussed above, largely dependent on the height of the tree. As we also discussed, tree height is a function of node size (number of index records per node) and the number of rows in the table.

However, since SQLBase only stores the minimum distinguishing key in non-leaf nodes, node size is difficult to predict for non-leaf nodes. This is because the savings in secondary storage achieved through only storing the minimum distinguishing key is hard to predict because it is dependent upon the actual values. As a general rule, however, a 2:1 (or better) compression ratio is generally achieved. In other words, the node size of non-leaf nodes may be estimated as two times the size of leaf nodes.

SQLBase index nodes are based on a database page (see Chapter 8, Database Pages). For DOS, OS/2, Windows NT, and NetWare, the page size is 1024 (for UNIX

implementations of SQLBase the pages size is 2048 bytes). Index page overhead is 74 bytes. The node size of leaf nodes is calculated as:

$$NodeSize = \frac{UsableIndexPageSize}{IndexEntryLength}$$

where

$$UsableIndexPageSize = (1024 - 74) \times \left(\frac{100 - PercentFree}{100}\right)$$

and:

$$IndexEntryLength = 9 + NbrColumns + KeyLength$$

For example, if an index contains three attributes, each of which contains, on average, 20 characters, then the B-Tree node size is:

$$NodeSize = \left\lfloor \frac{950\frac{(100-5)}{100}}{9+3+60} \right\rfloor = \left\lfloor \frac{950(0.95)}{72} \right\rfloor = 12$$

Hence, in this example, the node size of leaf nodes is twelve and one would generally expect the node size for non-leaf nodes to exceed 24, in this case.

Performance considerations

In an ideal situation, a database designer could set up an index and then rarely have to perform any maintenance on it. However, index performance degrades over time due to a number of factors, foremost of which is the pattern in which insertions and deletions occur. When this happens, index reorganization becomes necessary. This reorganization can take one of two possible forms:

- Reorganization of the entire database will rebuild all indexes within the database as a subset of the process. This is overkill unless the entire database has reached the point where total reorganization is required.

- Dropping and then re-creating an index will cause the index to be built fresh. This will result in an optimal, balanced tree structure for the index, and is the simplest and quickest way to return a degraded index to peak performance.

Deletions can cause balancing problems in the tree structure and fragmentation of the index blocks. SQLBase does not perform redistributions or concatenations when underflows occur in delete processing, based on the philosophy that these operations would take too much time during regular transaction execution. Rather, it is left to the DBA to rebuild the index when the accumulated underflows are sufficient to impact performance.

Insertions are not always spread uniformly across the index tree structure. When many insertions occur within a small range of symbolic key values within the tree, the freespace within this area is rapidly used up. Once this occurs, there can be frequent splits within this portion of the tree, sometimes reaching all the way back to the root. While these operations are taking place, many exclusive locks will be held throughout

the index structure for the duration of the split operation. The locks may cause contention problems for other transactions wishing to use the index structure during this time, see chapter 13, Concurrency Control, for more details on the locking implications of B-Tree indexes in SQLBase.

This imbalance is the primary reason why indexes placed on columns whose values are incremental in nature, such as a timestamp, can cause performance problems within an application. These indexes grow only along one edge. As rows are inserted along this edge splits occur frequently, so that transactions updating this index find themselves competing for locks within a small portion of the overall index structure. One possible solution to this problem is to add columns to the index in order to make the insertion pattern more evenly distributed throughout the tree.

Setting freespace for an index

Since the effect of repetitive insertion activity into an index can cause splits to occur, which in turn can cause processing overhead and locking contention, DBAs may wish to minimize the possibility of excessive splits occurring within indexes that are the subject of high transaction activity. The PCTFREE specification of the CREATE INDEX statement allows DBAs to realize this goal. Through correct calculation of the PCTFREE figure, split activity in an index may be minimized for some period of time following the creation or rebuilding of an index. How long this time period is depends on both the rate of insertion activity against the index, and the pattern of the symbolic keys being introduced into the existing index structure.

When creating an index, the database designer can choose a PCTFREE value to specify in the CREATE INDEX statement. This value determines the amount of free space that will be left in the index nodes during the index's initial creation. This value is fairly easy to calculate for an index where the insertion activity can be expected to be evenly distributed throughout the index structure, by utilizing the following procedure:

Calculating PCTFREE for an index

1. Either estimate or obtain the current number of rows in the table. This can be obtained directly through the *sqlgnr* function.

2. Estimate the number of rows that will be in the table at the point in time in the future when you plan on rebuilding the index.

3. Calculate the PCTFREE parameter according to the following formula:

$$\text{PCTFREE} = \left(\frac{(\text{CurrentRows} - \text{FutureRows})}{\text{CurrentRows}} \right) \times 100$$

4. If the value calculated above is greater than 50%, consider rebuilding the index more frequently to lower the PCTFREE value.

This procedure can also serve as a starting point for an index that is known to have a skewed insertion pattern. After obtaining a PCTFREE value from this procedure, either increase the value by a factor of between 1.5 to 3, or simply plan on rebuilding the index more frequently than the basis of your future row estimate would indicate. The more skewed the insertion pattern, the more dramatic the adjustment should be to ward off potential performance problems.

B-Tree locking

This topic is covered extensively in *Chapter 13, Concurrency Control.* To summarize, indexes can often become points of lock contention when they are placed on an active table, and should therefore be used sparingly in these cases. When many transactions are being executed simultaneously, and rapid response time is critical, any insertion or deletion activity in index structures will have a significant adverse impact.

Non-unique indexes

Indexes which lack the "UNIQUE" clause on the CREATE INDEX statement can contain more than one row with identical symbolic key values. SQLBase handles these entries as if they were unique anyway, in that each entry exists independently of any other entry. This is in contrast to some DBMSs which build synonym chains for duplicate entries, and then incur significant overhead while maintaining these long synonym chains. For this reason, SQLBase can handle any number of duplicates within an index without incurring any significant performance degradation, and in fact some space savings will be realized due to the minimum distinguishing key technique.

One potential drawback to non-unique indexes, though, is their lower selectivity factor. This is a statistic that is kept for each index by SQLBase, and is used by the query optimizer (covered in detail in Chapters 15 through 18) when selecting indexes to use when building an access path to satisfy a query's data requirements. If an index has many duplicates as a percentage of its total rows, then there is a good chance that the index may rarely, if ever, be selected for use by the optimizer. Such indexes are of no benefit whatsoever to the system, and should be deleted.

An alternative action that will increase the selectivity factor of the index is to add one or more additional columns to the low order end of the index which will enhance its uniqueness. In fact, one technique that is sometimes used is to add the primary key of the table to the low order of an index, which can then be created as a unique index. Note, though, that this technique will not effect the optimizer's selection of the index unless these additional columns can actually be specified in the WHERE clause of the SQL referencing the table. This is because the optimizer considers the selectivity factor only for those index columns that are actually specified in the statement being prepared for execution, as opposed to the entire composite index key.

Chapter 10

Hashed Indexes

In this chapter we:

- Introduce the concept of hashing, and contrast its effect on the location of rows within a table's storage space with that of non-hashed tables.

- Analyze the theory behind the performance benefits that can be expected when hashing a table.

- Explain the details of SQLBase's implementation of hashing including the steps you need to take to ensure that your hashed tables will meet your performance expectations.

Hash table overview

SQLBase offers an alternative to the B+Tree index structure covered in the previous chapter. This alternative is the clustered hashed index, which for the remainder of this chapter will simply be called *hashing*. A database table that has a clustered hashed index placed on it is called a *hashed table*.

Row location in hashed tables

Hashing is an entirely different approach from the conventional B+Tree index in that the basic allocation algorithm for table rows is fundamentally changed. In other words, the main effect of a clustered hashed index is on the subject table itself, whereas the main effect of a B+Tree index is to allocate disk storage and build a B+Tree representation of the subject column's symbolic key values, which contains pointers into the subject table. B+Tree indexes do not alter the subject table, whereas hash indexes do. This is the reason that a hash index must be created prior to any rows being inserted into the table, if the subject table of the CREATE CLUSTERED HASHED INDEX statement is not empty an error will be returned.

It is important to remember that a clustered hashed index has no physical index object as the B+Tree index does. The effect is entirely on the allocation scheme used for the subject table. Also, the term "clustered" is something of a misnomer, since there is no sequential order to the storage of the rows that corresponds to the collating sequence of the symbolic key. The "clustered" aspect is that rows which have equal symbolic keys will be stored on the same pages, as contrasting with non-hashed table row allocation which is governed strictly by the timing of the insert commands (roughly equivalent to being clustered chronologically by insertion time).

Performance benefits of hashed tables

Hashing is a file organization technique which provides very fast access to rows in the file. A properly implemented hashed table will allow access to almost all rows with a single read operation. In order for the hash access technique to be utilized, the SELECT ... WHERE clause must specify an equality predicate on the hash field of the table. This hash field is the column, or columns, named as the subject of the CREATE CLUSTERED HASHED INDEX statement. The hash field (or the concatenation of columns which comprise the hash field) is also commonly called the hash key. Example 1 illustrates the importance of the hashing method to a database system in saving disk accesses (I/Os) when searching for a record.

Example 1:   Suppose there is a file with 50,000 records stored on a disk with block size 1,024 bytes. We assume that records are of fixed size with a record length of 100 bytes. Hence there are $\lfloor 1024/100 \rfloor = 10$ records in each block, where $\lfloor x \rfloor$ is a floor of x, which truncates the fraction part of x (for example, $\lfloor 3.1 \rfloor = 3$). The number of blocks

needed for the file is $\lceil 50{,}000/10 \rceil = 5{,}000$ blocks, where $\lceil x \rceil$ is a ceiling of x, which rounds up the fraction part of x (for example, $\lceil 3.1 \rceil = 4$). Access of the file through a BTree index would require one I/O for the leaf page, one for the actual data block, plus another I/O for each level in the BTree structure (probably at least another two I/Os, although would probably be improved through caching). This would make a total of approximately four I/Os for the BTree access. However, hashing makes it possible to find a record with very few disk accesses, typically one disk I/O. This means that the hash access would require only one-fourth of the I/O time that the BTree access would.

Potential pitfalls of hashing

This improvement in performance does not come free, of course. The following drawbacks to hashing should be considered before selecting this access technique:

- The hash function must work well with the selected hash key, eliminating bias to minimize overflow. Fortunately, SQLBase has an excellent hash algorithm built-in which works well for most keys.

- Rows of hashed tables cannot be accessed sequentially in the sequence of the primary key without undergoing an intermediate sort. If sequential access is often required, then a B+-Tree index on the primary key would have to be created to provide an efficient access technique that would bypass the need to perform a sort operation.

- Hash keys have no generic capabilities, therefore the LIKE, less than (<), greater than (>), BETWEEN, or similar predicates cannot be used with hashed access. The only predicates that can be used are the equal (=), for single value retrieval, or the IN list, for multiple value equality conditions. Also, for multi-column hash keys, all columns must be specified, each with one of the two allowable predicates. Again, these capabilities can be provided on a hashed table by creating a B+-Tree index on the table for the desired columns.

- Space for the hashed table will be pre-allocated at table creation time, specifically when the CREATE CLUSTERED HASHED INDEX statement is executed. This space allocation is one of the parameters used by the hash function when computing row locations. For this reason, changing the space allocation requires the table to be unloaded, dropped and redefined, and then reloaded. Because this may be a very time consuming process, tables which experience wide unpredictable swings in row volume are generally not well suited for hash treatment. Tables which maintain a consistent size or a slow predictable growth are the best hash candidates.

Hashing theory

We will now examine the characteristics of the hashing storage technique in general. We will cover the hash function, which performs the transformation of symbolic hash key values into storage addresses, the concept of bucket hashing, where more than one record is held at the same hash address, and provide a detailed analysis of the key performance criteria used to evaluate the success of a hashed file implementation. These criteria all relate to a hashed file's disk space allocation, and the probabilities of the hashing algorithm being able to accomplish its objective of storing and retrieving records with no more than one I/O.

The hash function

The idea of hashing is to utilize a hash function, H, which is applied to the hash key and yields the address of the row:

$$H(HashKeyValue) \Rightarrow RecordAddress$$

The purpose of a hash function is to transform the hash key (which may be a concatenation of multiple columns) to a page address that lies within the range of the pre-allocated file. The algorithm that the function uses is fairly arbitrary, so long as the final result is always within the range of the file. It is also desirable that the transformation be uniform, that is, there be no bias toward any particular address. If there is bias, rows hashing to the "favored" addresses are likely to overflow their target pages and take longer to store and retrieve (from overflow pages). The details of a simple hypothetical hash function are shown in the next example.

Example 2:   Consider a file where the hash field is a person's name. Suppose we want to reserve space for 1,000 records, which implies that the result of the function must be in the range of 1 to 1000. The hypothetical hash function H can be defined as taking two numbers from the ASCII representations of the first two characters of the name (which is the hash key), multiplying them together (in an attempt to eliminate bias), and dividing the result by 1,000 and taking the remainder for the hashing address (which insures the result will be in the proper range). Table 1 shows how three names would produce three hashing addresses. Note that although the names are listed in alphabetical order, the addresses are not in this order, but appear to be random.

Name	ASCII Codes	Product	Address
BOB	66 , 79	5214	214
STACY	83 , 84	6972	972

Name	ASCII Codes	Product	Address
TED	84 , 69	5796	796

Bucket hashing

In practice, the I/O system retrieves a group of records from a disk, instead of a single record, because records are grouped together in a block. We can extend the idea of a record address in a file to the address of a block containing a set of records, and call each block a hash bucket. In a bucket scheme, the hashing function generates a bucket number instead of a record address as the hashing address for a given hash field. The bucket number can usually be directly transformed into a relative block number in a hashed file. When this relative block number is added to the starting disk address for the file, any block in the file may be accessed directly via its disk address. Once the bucket's disk address is resolved, we need only a single block access to retrieve the required record. The cost to search for the record within the block is minimal since it can be carried out in a main memory buffer.

Collisions and overflows

A collision occurs during an insertion when two records are hashed to the same address. Referring to our hypothetical hash function in example 1, suppose there is a record in the file with the name ETHAN. Since the name ETHAN starts with the same two characters as the name TED, they produce the same hashing address 796. There is a collision between the record for TED and the one for ETHAN. One way to resolve the problem of collisions is to find a hashing function that prevents collisions from happening at all. Such a hashing function is called a *perfect hashing*. However, perfect hashing functions are rare. Knuth showed that in the set of all possible mappings of a 31-record set to a table with 41 locations, only about one out of 10^7 is the perfect function. Therefore it is not practical to find a perfect hashing function. Instead, a better strategy is to focus on determining an efficient technique to handle collisions when they occur.

In bucket hashing, a collision occurs when two records hash to the same bucket. Collisions are not serious if the bucket is not full, since records can be stored in available spaces in the bucket. However, overflow arises when a new record is hashed to a bucket which is already full. How SQLBase handles collisions in its bucket hashing algorithm will be examined in a later section of this chapter.

Collision avoidance techniques

Two techniques are available to the database designer in order to reduce the possibility of collisions in a hashed table. The first technique involves selecting a hash key which will allow for the even distribution of rows across the allocated space using the selected hash function. The other technique is to allocate a greater amount of space than is needed in order to allow the hash function a greater page range of allowable results to operate across. Application of these techniques is imperative if the desired performance benefits associated with hashing are to be achieved.

Hash key selection

The first technique entails choosing the hash field carefully. A badly chosen key leads to poor hashing results. An inappropriately selected hash key often has one of the following characteristics:

- Values of the key are not evenly distributed. In general, uniqueness is a desirable characteristic for a hash key (one significant exception to this rule is described in Chapter 6, *Refining the Physical Design*). Suppose there is an employee table with each record consisting of attributes such as employee social security number (SSN), name, and department code, among others. The primary key for the employee is the 9-digit long SSN. The department code is 2-digits long. The employee's name is not a good hash key candidate, since there is an uneven distribution of names in most organizations (many "Smith" and "Jones" occurrences). Also, the department code may not be a good candidate either, since many employee records could be clustered together in departments which have very large staffs. The best key for this case is the employee SSN, which is unique. Serially created numeric fields that contain arbitrary values are often found to be good hash key candidates.

- The key does not match well to the hash function. Suppose that the employee ID column, rather than being SSN, is now 5-digits long, and has the form of YYNNN, where YY is the year of hire, and NNN is an integer consecutively allocated from 000 up within a year. Now also consider the hypothetical hash function used back in Example 1, where the hashing address is generated based on the first two characters of the selected hash field. In this case, the employee ID would not be a good candidate since all the employees hired by the company in the same year will have the same hashing address.

- The number of all possible hashing addresses is too small. For example, it is not feasible to map millions of U.S. taxpayers' records to the 50 hashing addresses representing the 50 states.

Over-allocating disk space

The second collision avoidance technique, which is generally only effective when the hash key is fairly unique, involves allocating extra space. It is easier to find a hash function that avoids collisions if we have only a few records to distribute among many slots than if we have more records than slots. For example, if there are 1,000 slots reserved and only 100 locations will be used to store records, the result might be very good using the hypothetical hash function in Example 1. However, the disadvantage is that the storage space is wasted (90% of the space is wasted in this case). The question is how much extra space can be tolerated while still obtaining acceptable hashing performance. In the next section a technique for measuring the performance in terms of expected overflow will be shown.

Hashing disk space utilization and performance

When creating a hashed table, the database designer must consider the inter-related design issues of bucket size and packing density, both of which affect database performance.

Bucket size

Bucket size is the number of records that can be held in a bucket. For instance, since a bucket of 1,024 bytes can store 10 records each of which is 100 bytes long, the bucket size is 10. Alternatively, if the record length is doubled from 100 bytes to 200 bytes, the bucket size is decreased from 10 to 5. As bucket size increases, the probability of overflow decreases but the time taken to search for a record in the bucket may increase. However, the cost of accessing a record once the bucket is in main memory is very small compared with the time required to read a bucket in secondary storage.

Packing density

One characteristic of hashing is that the number of I/Os required to perform an insertion or retrieval tends to increase as the file fills. This seems intuitive since when most of the buckets are full, it is more likely that the hashing function will generate the address of a bucket which is already full. Overallocation of the file is an obvious way to lessen this problem, however it is not desirable to allocate space for a lot of empty buckets. A measurement called *packing density* is the ratio of the number of records (R) in a file to the number of available slots (N): We know that $N=B*S$ where

$$\mathrm{PackingDensity} \; = \; \frac{\mathrm{Records}}{\mathrm{NumberOfSlots}}$$

B is the number of home buckets (which equals the number of addresses that can be

generated by the hashing function), and *S* is the bucket size (the maximum number of rows which may be stored in each bucket).

$$PD = \frac{R}{B \times S}$$

Therefore, since, we also know that we can calculate the number of buckets (*B*) required to store a particular hash table, given the number of rows *R*, and the bucket size *S* (which is a function of the page size and the row length) after choosing a packing density *PD* by solving the previous formula for *B*, which gives

$$B = \left(\frac{\left(\frac{R}{S} \right)}{PD} \right)$$

This concept is important because in SQLBase bucket size is a function of the page size (which is fixed for a specific implementation platform), the row length, and the PCTFREE specification on the CREATE TABLE statement. Most of these parameters cannot be easily changed for hashing performance reasons, but are determined by other factors. Your main parameter for tuning the performance of a hash table in SQLBase is by altering the number of buckets, *B*, in the table's storage space. The number of home buckets (*B*) is the most easily adjusted, and therefore the most important, parameter which affects packing density. By increasing the number of home buckets, packing density drops and hashing performance improves due to a decrease in the number of overflows.

Packing density itself is one of the most important parameters affecting hashing performance. As a general rule of thumb, collisions are unacceptably frequent if the packing density exceeds 70%. This rule can be greatly impacted by the bucket size, which is determined by row and page size, however. This is why you should understand how to determine a packing density figure that is appropriate for your particular table.

Hashing performance analysis

Given the choices for bucket size (*S*) and packing density (*R/N*), we can estimate the number of overflows. We must assume that the hash function is uniformly distributed, which implies two things. One is that the probabilities associated with a record being inserted into a home bucket are independent. The other is that the average probability that a record is inserted into one of *B* home buckets is $\frac{1}{B}$.

Therefore the probability that a certain set of *r* records will be inserted into a given home bucket *b* is $\left(\frac{1}{B} \right)^r$. Note that *b* is one specific occurrence of all home buckets, *B*. Also, *r* represents an arbitrary subset of all records *R*, which means that *r* may contain any number of records from 1 to *B*.

Next we wish to calculate the probability of the remaining $(R-r)$ records being inserted into different home buckets other than b. Let $1 - \frac{1}{B}$ be the probability that a record will not be inserted into a particular bucket (called b). Hence, $\left(1 - \frac{1}{B}\right)^{R-r}$ is the probability that $(R-r)$ records are not inserted in bucket b.

We have now introduced the probability that r of R records will be inserted in a home bucket (b) as well as the probability that the remaining $(R-r)$ records are not inserted in bucket b. Therefore, the probability of r records being inserted into bucket b **and** the remaining $(R-r)$ records not being inserted into bucket b is the product of $\left(\frac{1}{B}\right)^r$ and $\left(1 - \frac{1}{B}\right)^{R-r}$ (or $\left(\left(\frac{1}{B}\right)^r \times \left(1 - \frac{1}{B}\right)^{R-r}\right)$).

However, there are different combinations that R records could be taken r at a time. The number of different ways is:

$$\frac{R!}{r! \times (R-r)!} .$$

Thus the probability of a given bucket b having exactly r records (where $r \le R$) is

$$p(r) = \frac{R!}{r! \times (R-r)!} \times \left(\frac{1}{B}\right)^r \times \left(1 - \frac{1}{B}\right)^{R-r} .$$

Given $p(r)$, the probability of exactly j overflow records in a specific bucket with bucket size S is $p(S+j)$. Therefore the expected number of overflows for any given bucket is :

$$p(S+1) + 2p(S+2) + 3p(S+3) + ... + (R-S)p(R) = \sum_{j=1}^{R-S} j \times p(S+j)$$

Using this formula, we see that the expected number of overflows for all B home buckets is:

$$B \times \sum_{j=1}^{R-S} j \times p(S+j) .$$

We can express the above formula as a percentage of the records in the file:

$$OverflowRatio = \frac{B \times \sum_{j=1}^{R-S} j \times p(S+j)}{R} .$$

Since we assume a uniform distribution for any given packing density ($PD = \frac{R}{B \times S}$), any bias toward certain addresses in the hash function is likely to increase the rate of overflow. Be aware that a figure produced with this model is a minimum percentage. Given the formulas above, we can derive an expected overflow against packing density for a particular bucket size. The following table shows the expected percentage of overflow for different combinations of bucket size and packing density.

The following example illustrates the use of this table in the design of a hash table.

Bucket Size	Packing Density (%)									
	50	55	60	65	70	75	80	85	90	95
2	10.27	11.90	13.56	15.25	16.94	18.65	20.35	22.04	23.71	25.37
3	5.92	7.29	8.75	10.30	11.92	13.59	15.30	17.05	18.81	20.58
5	2.44	3.36	4.45	5.68	7.07	8.58	10.21	11.93	13.73	15.60
8	0.83	1.33	2.01	2.87	3.94	5.20	6.65	8.26	10.03	11.93
12	0.24	0.47	0.84	1.38	2.13	3.11	4.33	5.79	7.48	9.36
17	0.06	0.15	0.32	0.63	1.12	1.85	2.84	4.12	5.69	7.53
23	0.01	0.04	0.12	0.28	0.58	1.08	1.86	2.96	4.40	6.18
30	<0.01	0.01	0.04	0.11	0.29	0.63	1.22	2.14	3.45	5.16

Example 3: A company's employee information is stored using the hashing technique. Each employee record is 75 bytes. Each disk block has a usable total of 938 bytes of space. The DBA therefore calculates the bucket size as 12 ($\lfloor 938/75 \rfloor = 12$). The DBA decides on a packing density of 70%. He also projects that the system must be capable of holding up to 50,000 employees' records. The file would therefore require $\lceil (50,000/12)/0.70 \rceil = 5953$ home buckets. Also, with a packing density of 70% and a bucket size of 12, table 2 predicts that at least 2.13% of the records would be expected to go to overflow buckets. In other words, there are 50,000*2.13%=1,065 overflow records requiring $\lceil 1,065/12 \rceil = 89$ overflow buckets. Therefore every $\lceil 5,953/89 \rceil = 67$ home buckets, on average, would require an overflow bucket.

SQLBase hashing implementation

In this section, the implementation of the hash storage technique in SQLBase is discussed. We will cover how the location of table rows differs for hashed tables as contrasted to non-hashed tables. We will also describe how to set an appropriate packing density in SQLBase. In particular, the desired packing density is an input to the CREATE INDEX command in the form of a specification of the number of hash buckets to be allocated.

Location of table rows

In terms of the effect on the subject table's allocation method for each of the two techniques (hashed or non-hashed tables), rows are located as follows:

- Any table which does not have a clustered hashed index placed on it (a non-hashed table) allocates space for row insert operations in a manner which is basically chronological. The first rows stored in the table are stored on the first data page, and so on until the most recently stored rows which are found on the last data page (this is not exactly true because of measures taken to avoid lock contention between different transactions). These data pages are allocated in groups of pages called extents, the size of which is determined by the EXTENSION environment variable setting.

- All of the data pages for a hashed table are pre-allocated at the time the CREATE CLUSTERED HASHED INDEX statement is executed. Rows are located within these data pages through the use of a hash function, which maps the symbolic key value of the index to a particular offset within this already fixed number of pages. As will be seen, when these pre-allocated pages fill up, no further data pages are allocated. Rather, any rows which do not fit on the page which is targeted by the hash function are stored on *overflow pages*, which will eventually degrade performance.

Hash function

SQLBase contains a pre-programmed hash function which will provide excellent results for the vast majority of symbolic keys. This function is internal to SQLBase and cannot be changed by the database designer in any way. For efficiency, SQLBase implements much of the hash function using bit operations, such as exclusive-or (XOR), and bitwise complement (1's complement). The following procedure is followed by the SQLBase hash function when transforming a symbolic key into a hash address.

How SQLBase hashes a key to a database page

The hashing function is implemented as a 3-step key-to-address mapping:

1. Transform the symbolic key into a single binary fullword. This allows for standardization of the remainder of the process. The symbolic key may be one or more numeric or character types of any length, where each byte can be represented as an ASCII code. SQLBase performs the following 3 sub-steps to convert the symbolic key into a standard fullword (4 bytes). This fullword is then treated as an unsigned binary integer for the remaining steps.

   1A. Divide the key into 4-byte units. This accomplishes the standardization of the length.

   1B. Perform a XOR operation on all of the 4-byte units, XORing them all together and giving a 4-byte result. This ensures that all of the information contained within the symbolic key is used to compute the hash key. No parts of the key are omitted from the process.

   1C. The result of 1B is a binary fullword. SQLBase then performs a bitwise complement on it. This operation, also called a one's complement, simply "flips" the value of each bit (ones become zeros and vice versa). This is done to eliminate any bias that may have survived to this point from the symbolic key due to its coding format, particularly ASCII characters (which are a fairly small range of ASCII), or SQLBase internal format numeric fields (which are stored in base 100).

   Example 4: In an employee table, an eight-character string YYMMCCCC is used to represent a hash key. The YY (or MM) denotes the year (or month) when an employee was hired (an example is "9305", which means that an employee was hired in May, 1993), and CCCC denotes a name code for the employee (such as where "EJSM" is a code for E. J. Smith). For this example we will use a symbolic key value of "9305EJSM". The ASCII representation for 9305EJSM is 57, 51, 48, 53, 69, 74, 83, and 77.

   - Step 1A. The key is divided into four-byte units.

```
Key1 = 57,51,48,53
Key2 = 69,74,83,77
```

   - Step 1B. Key1 is XORed with key2.

```
57,51,48,53 XOR 69,74,83,77 =
0011,1001,0011,0011,0011,0000,0011,0101 XOR
0100,0101,0100,1010,0101,0011,0100,1101 =
0111,1100,0111,1001,0110,0011,0111,1000
(124,121,99,120 in ASCII)
```

   - Step 1C. Now perform a bitwise complement on the result of 1B.

```
~(0111,1100,0111,1001,0110,0011,0111,1000) =
1000,0011,1000,0110,1001,1100,1000,0111
(131,134,156,135 in ASCII)
2,206,637,191 when treated as a fullword binary integer and
converted to decimal.
```

2. Transform the binary fullword into an intermediate integer that has the same range as the number of page addresses in the table's space allocation. In this step, the method used is based on performing integer division, then finding the remainder, through the use of the MOD operation.

   This integer is derived by performing a MOD of the output of step one with a number A, where A is the number of hash buckets that are allocated. SQLBase always sets A to a prime number to avoid the possibility of divisors producing a zero result from the MOD operation.

   Example 5: To continue from example 4, assume that 1,753 hash buckets have been allocated for the employee table. This means that allowable values for the results of step 2 would lie in the range of 0-1752. Dividing 2,206,637,191 by 1753 gives a result of 1,258,777 with a remainder of 1,110. Therefore, the result of step 2 is 1,110.

3. Transform the output of step two into the physical page address. This is done by adding a starting page number, which yields the final hash address of the page containing the row.

   Example 6: Suppose that the first page number used for data storage of the employee table rows is page 60. This means that 60 must be added to the target bucket obtained in example 5 in order to get the physical target page number of this row. Since $1,110 + 60 = 1,170$, the row will be stored on page 1,170. Similarly, if a SELECT ... WHERE keyname = "9305EJSM" was performed against the employee table, SQLBase would look for the row on page 1,170.

Specifying the packing density

Packing density is the single most important design parameter that you can use to tune the performance of a hashed table. This is accomplished through careful calculation of the figure used in either the SIZE "*b*" BUCKETS or the SIZE "*n*" ROWS clauses of the CREATE CLUSTERED HASHED INDEX statement (note that "*n*" is the number of rows used by SQLBase to calculate the required hash bucket allocation, whereas "*b*" is the number of hash buckets that should be allocated specified directly). Since SQLBase provides a pre-programmed hash function, and the hash bucket size is fixed with page and row size, this estimate of the CREATE INDEX statement is the only major influence that a database designer can exercise over the results of the hashing process.

When determining what number to place in this DDL statement, an optimal choice of packing density becomes critical in order to reduce the number of overflows while still utilizing disk space efficiently. While the 70% rule of thumb figure is often used, we encourage you to look in the expected overflow table explained earlier to find the packing density that is most appropriate for the bucket size that will be in effect for each of your hashed tables. By looking at this table, you can decide what packing density is appropriate in order to obtain an expected low number of overflows. This packing density can then be used as input to the calculation of the number of buckets estimate for the CREATE INDEX statement. Remember that it is when the CREATE INDEX statement (not CREATE TABLE) is executed that a fixed number of buckets will be allocated for storing the hashed table. Also, SQLBase will round up the number you specify in the BUCKETS form, or the number it calculates in the ROWS form, of the CREATE INDEX statement to the next higher prime number.

Simple hash bucket calculation

Since in this manual we assume that you wish to control the options available in SQLBase to achieve your best possible performance, we only consider calculation of the SIZE "*b*" BUCKETS option of the CREATE CLUSTERED HASHED INDEX statement. Use of this form of the statement is encouraged in order to target a particular packing density that is desired, and simplify the specification of this figure to SQLBase. The alternative form of SIZE "*n*" ROWS is more appropriate for quick implementation of hash tables that are trivial in size and do not have stringent performance requirements. Calculating the row estimate for a desired packing density is much more difficult and involved than the bucket calculation, and in the end achieves the same result. For this reason, row estimate calculation will not be covered or explained in detail. If you wish to use the row estimate option, we encourage you to make sure your estimate is sufficiently high to avoid the poor performance that could result from realizing a large number of hash collisions in a table.

We will now describe how the number of buckets for the CREATE INDEX statement is calculated given the packing density. As you may recall, the formula

$$B = \left\lceil \frac{\left(\frac{R}{S}\right)}{PD} \right\rceil$$

allows for the direct calculation of the number of buckets, *B*, given a particular packing density (*PD*), number of rows (*R*), and bucket size expressed as the number of rows which may occupy a bucket (*S*).

The *S* parameter used in the previous formula is the same as the number of rows per page which is calculated as an intermediate result when estimating the size of a database table, as explained in the next chapter of database size estimation. To summarize that formula, the rows per page of a table are calculated as follows (note that the ΣColLengths is the sum of the length of all columns in the table, for

information on how to find the length of each column, refer to Chapter 11, *Estimation Database Size*):

$$S = \left\lfloor \frac{PageSize - 86}{\left\lceil \dfrac{24 + (2 \times NumCols) + \Sigma ColLengths \times 100}{100 - PCTFREE} \right\rceil} \right\rfloor$$

Example 7: Suppose a table has 7 columns, an average data width of 66 bytes for all columns, and is projected to contain 60,000 rows. The database is running on the SQLBase NLM server, which has a page size of 1024K. The PCTFREE parameter of the CREATE TABLE statement has been allowed to default to 10%. The designer has decided on a packing density of 70%. Bucket size is

$$\left\lfloor \frac{1024 - 86}{\left\lceil \dfrac{(24 + (2 \times 7) + 66) \times 100}{100 - 10} \right\rceil} \right\rfloor = 8 .$$

From the expected overflow table we see that the corresponding expected overflow is 3.94%. Using the formula given above, we can compute the number of buckets required in the CREATE INDEX statement as $\lceil (60,000/8)/70\% \rceil = 10{,}715$ buckets. In other words, SQLBase will allocate 10,715 pages for storing hashed data. Alternatively, the designer could choose to have an 80% packing density, increasing the expected overflow to 6.65%. The number of buckets in the CREATE INDEX statement would then be $\lceil (60,000/8)/80\% \rceil = 9{,}375$ pages would be required. Note that the use of a higher packing density will save space but that the overflow rate will also increase, in this case from 3.94% to 6.65% while the space saved is 10,715 - 9,375 = 1,340 pages or approximately 1.3 megabytes.

Chapter 11

Estimating Database Size

In this chapter we:

- Explain why estimating the size of a database can be an important task for the DBA.

- Describe the use of existing spreadsheets that simplify the database size estimation process.

- Cover the sizing formulas for the following database objects in detail:

    - Columns

    - Non-hashed tables

    - Hashed tables

    - B+Tree indexes

    - SQLBase catalog tables

    - SQLBase overhead

- Perform the complete manual size calculations on an example database as an exercise.

Introduction

The purpose of this chapter is to enable a DBA to estimate the total size of a complete database. The raw data needed to perform this task are the detailed database design, including tables, columns, indexes, and views, as well as row occurrence estimates for each table in the database.

An accurate projection of a database's ultimate size is more critical than ever before, since gigabyte database systems are becoming increasingly common in today's market. A database designer can base his estimates of the growth of the database on observable and measurable business metrics such as the number of new order transactions processed daily. This information may be extrapolated to row count estimates for the various database tables involved in each transaction. Following this, the required disk capacity can be pre-determined by using the formulas provided in this chapter. The results can then be used to determine appropriate hardware configurations, both for the initial implementation of the application as well as for periodic upgrades in capacity.

The starting point for this estimation is to calculate the total size for every table. To do so, we need to (1) calculate each column width in a table; and (2) calculate the table size based on the information derived from(1). Table size is calculated differently for tables which are the subject of a clustered hashed index, due to the change this has on the table's space allocation technique:

- If the index is created by specifying the option of CLUSTERED HASHED in the CREATE INDEX statement, we estimate the size of the hashed file as the table size. This hashed file is allocated at full size immediately when the CREATE INDEX statement is executed.

- Otherwise, we need to estimate the non-hashed table size. This is the size that we estimate the table will grow to when fully populated with data. Data pages for the table will be allocated as they are needed over time, as opposed to all at once as is the case with hashed tables.

If the table is associated with one or more BTree indexes (indexes which do not specify the option of CLUSTERED HASHED), then the calculation of the table size should also consider the space required for the indexes. The total space required for the table is then the sum of the table itself and all associated BTree indexes.

The database size can then be obtained by summing up all the table sizes, the overhead of all views defined against these tables, and the overhead of the system catalog (or data dictionary) tables in the database.

Spreadsheet usage

A number of spreadsheets are available to help the DBA in estimating table and index sizes. These spreadsheets are available for download from the Centura Web site. Users of the hashed table spreadsheet, for example, can specify the percent free parameter (PCTFREE), number of rows projected to be held in the table, and packing density. Other inputs include column names, data types, average data length for each column, and declared data length of each column. The spreadsheet calculates the number of rows contained in a page, number of row pages required, and number of rows for the CREATE CLUSTERED HASHED INDEX statement. With the availability of these spreadsheets, the task of estimating the size of a database is largely automated. The following example in Figure A illustrates one of the spreadsheets.

Three specific spreadsheets are available:

- Calculation of space requirements for a standard, non-hashed table.
- Calculation of space requirements for a hashed table (one which has a CLUSTERED HASHED index placed on it). This is the example in Figure A.
- Calculation of space requirements for a BTree index.

Note that there is no spreadsheet available for calculating space requirements for a CLUSTERED HASHED index itself. This is because there is no physical space allocated uniquely to this type of index, rather the effect of the index is to change the storage allocation method used for the subject table.

Formulas for estimating database size

The database size can be calculated by summing up all the tables, indexes, views, and a fixed amount of overhead, as follows:

```
DatabaseSize = ∑RegularTables + ∑HashedTables + ∑Indexes + ∑Views
+ FixedOverhead.
```
Before the size of any table may be calculated, the width of each column must be determined. Therefore, we first introduce the formula for calculating the column width. Then, formulas for computing sizes of various tables, indexes, and views are presented.

Note that a page is fixed at different sizes for the various implementation platforms of SQLBase. In DOS, Windows (with DOS), Windows NT, OS/2, and NetWare the page size is 1,024 (or 1K) bytes. For reasons of simplicity, we will use 1,024 bytes as the page size in the examples of this chapter. You should remember to substitute the page size that is appropriate for your environment.

1	COLUMN	DATA	DECLARED	AVERAGE		
2	NAME	TYPE	LENGTH	LENGTH	Parameters	
3	EMP_ID	CHAR	6	6	PCTFREE =	25.00
4	EMP_NAME	CHAR	16	10	**Number of Columns =**	12
5	DATE_EMP	DATE	5	5	**Number of Rows =**	50,000
6	BIRTH_DATE	DATE	5	5		
7	BASE_SALARY	INTEGER	6	4	**Data Length =**	171
8	ADDRESS_1	VARCHAR	80	50	**Row Length =**	219
9	ADDRESS_2	VARCHAR	80	50	**Row Len W/Slack =**	292
10	CITY	CHAR	20	10	**Long Pages/Row =**	1
11	STATE	CHAR	2	2	**Rows/Page =**	3
12	ZIP_CODE	CHAR	10	5	Results	
13	PHONE	CHAR	12	12	**Nbr Row Pages =**	16,667
14	DESCRIPTION	LONGVARCHAR	0	100	**Nbr Long Pages =**	50,000
15	0	0	0	0	**Total Data Pages =**	66,667
16	0	0	0	0	**Total Bytes =**	68,267,008
17	0	0	0	0	Rows for	
18	0	0	0	0	**CREATE INDEX =**	33,334
19	0	0	0	0	**ADD COLUMNS**	
20	0	0	0	0		

Figure 11A - An example of a spreadsheet which calculates the estimated size of a table. Note that this table is the subject of a CLUSTERED HASHED index, therefore one of the calculations made is for the row count to be used in the CREATE INDEX statement.

Also note that the following formulas make use of the floor and ceiling functions. When denoting these, $\lfloor x \rfloor$ is a floor of x, which truncates the fraction part of x (such as, $\lfloor 3.1 \rfloor = 3$). Also, $\lceil x \rceil$ is a ceiling of x, which rounds up the fraction part of x (such as, $\lceil 3.1 \rceil = 4$).

Calculation of column width

A specific formula used to compute a particular column's width is dependent on its declared data type. The width computed is the internal width the column will have when SQLBase converts it into the internal format used by its data type. These individual column widths must then be summed in order to determine the size of the data portion of a table row.

The input to these formulas is the average external column width of each column. This is the number of characters or digits that is typically found in the column, as opposed to the declared, or maximum, size of the column. All data types are stored as variable length fields internally in SQLBase, so for the column width calculation to be accurate for the majority of the database rows, the average width must be determined and used. The determination of the average width may be

done through either a "best guess" method or through detailed analysis of actual column contents, whichever is more feasible.

Also, hashed tables will need another calculation done using the maximum width for each column in the table. This will be needed later in order to accurately calculate the packing density for the hashed file. Since SQLBase reserves space for hashed tables based on the row estimate in the CREATE CLUSTERED HASHED INDEX statement, multiplied by the maximum possible row width (which is the sum of the maximum widths of all non-LONG VARCHAR columns), the calculation of this maximum row width allows the DBA the highest level of control over the this space allocation process. This calculation does not need to be performed for non-hashed tables.

Data type	Column width
Character	Number of characters in the column (maximum 254 bytes). For example, a character type data "Atlanta" is 7-bytes long. (Note: Physically, SQLBase stores both CHAR and VARCHAR columns the same way. However, you should continue to use both data type to explicitly document the logical data type of the attribute. For example, CHAR(8) implies the attribute always contains eight characters; whereas, VARCHAR(8) means the attribute may contain up to eight characters.)
Number	$\lceil (NumberOfDigits+2)/2 \rceil +1$ bytes. For instance, for a 10 digit integer, the length is $\lceil (10+2)/2 \rceil +1=7$ bytes.
Date	5 bytes.
Date/Time	12 bytes.
Long varchar	12 bytes consisting of the first physical page number, last physical page number, and column size are stored in a base row page. These 12 bytes are considered as the column width stored in a base row page. The actual data resides in long varchar pages. SQLBase stores only one column on each long varchar page. For example, if a column is declared as LONGVAR but its length is only one byte, an entire long varchar page is allocated to store this one-byte long varchar data. A general formula for the number of long varchar pages required for the data is (for each long varchar column): Nbr of Long Pages per Long Column $= \lceil$ size of long varchar in bytes/(1024-61)$\rceil$. The number of long varchar pages is included in the calculation of a table size. (Note: SQLBase stores zero length strings defined as either CHAR or VARCHAR as NULL (i.e., requires no physical storage); on the other hand, columns defined as LONG VARCHAR require a minimum of one LONG VARCHAR page (i.e., 1024 or 2048 bytes depending on the page size.)

We define *DataLength* as the sum of all column widths in a row, which is used for the table size calculation.

DataLength=∑All Column Widths.

Table size calculations

Based on the derived *DataLength*, we can compute the estimated regular (or non-hashed) table size or the estimated hashed table size, whichever is appropriate for the table in question.

Calculation of regular table size

To compute the size for a regular table, we first obtain the *DataLength* derived from the previous discussion. We must also have an estimate of the number of rows, called *NbrOfRows* below, which will be stored in the table. From these two inputs, we can calculate the following size estimates.

Parameter	Formula
Row Length	Includes the row overhead, such as row header, and slot table entry. Row Length = 18 + (2 x number of columns) + *DataLength*.
Row Length with Slack	Considers the PCTFREE parameter specified in the CREATE TABLE statement (which defaults to 10%). Row Length with Slack = Row Length x 100 /(100 - PCTFREE)
Usable Row Page Size	In SQLBase, page overhead is 86 bytes. Usable Row Page Size = 1024 - 86 = 938 bytes.
Rows per Page	$\lfloor$(Usable Row Page Size / Row Length with Slack)$\rfloor$. If this formula evaluates to less than 1, then make it 1.
Nbr Row Pages	$\lceil$(*NbrOfRows* / Rows per Page)$\rceil$, where *NbrOfRows* is the projected number of rows in the table.
Nbr Long Pages	*NbrOfRows* x Nbr Long Pages per Long Column.
Total Data Pages	Nbr Row Pages + Nbr Long Pages

Calculation of hashed table size

Computing the size of a hashed table is similar to that for a regular table. However, we need to take into consideration the desired packing density when estimating a hashed table size. Therefore a new result parameter must be calculated as follows:

- Nbr Hashed Table Pages = Nbr Row Pages / packing density.

This calculation appears at the end of table 4, which follows.

Parameter	*Formula*
Row Length	18 + 6 + (2 x number of columns) + *DataLength*. The "6" in this equation accounts for storage of the processed hash key with the row.
Row Length with Slack	Row Length x 100 / (100 - PCTFREE)
Usable Row Page Size	1024 - 86 = 938 bytes.
Rows per Page	$\lfloor$(Usable Row Page Size / Row Length with Slack)$\rfloor$.
Nbr Row Pages	$\lceil$(*NbrOfRows* / Rows per Page)$\rceil$.
Nbr Long Pages	*NbrOfRows* x Nbr Long Pages per Long Column.
Nbr Hashed Table Pages	(See packing density discussion above for detailed explanation) Nbr Hashed Table Pages = Nbr Row Pages / packing density.
Total Data Pages	Nbr Hashed Table Pages + Nbr Long Pages

BTree Index size calculations

When a table is associated with a BTree index through the CREATE INDEX command, we need to calculate the BTree index size. To perform this computation, we first obtain the length of the index's symbolic key by performing the column width calculation for each column named as a subject of the index. We must also have an estimate of the number of rows, called *NbrOfRows* below, which will be stored in the subject table. From these two inputs, we can calculate the following size estimate for the lowest level of an index (leaf pages of the index structure) as follows:

Parameter	*Formula*
Key Length	Key Length = $\sum$ average data lengths of columns in the key.
Index Entry Length	9 + number of columns in index + Key Length.
Usable Index Page Size	(1024 - 74) x (100 - index PCTFREE) / 100 (Note that the default value for PCTFREE is 10%).
Index Entries per Page	$\lfloor$(Usable Index Page Size / Index Entry Length)$\rfloor$.
Nbr Index Pages	$\lceil$Nbr of Rows / Index Entries per Page$\rceil$.

Estimation of overhead for views

For each view there is fixed overhead and a variable overhead depending on the complexity of the view. Note that when a view references another view, the variable overhead of the referenced view is added to the view being defined. For example, if a view refers to two simple views whose variable overhead is 5 pages each, the complex view will have a body which is 5+5=10 pages plus its own variable overhead. In other words, the variable overhead of each referenced view is included in the variable overhead of the view that references it.

The size of a view can grow rapidly if it references other views which in turn reference yet other views.

Parameter	*Formula*
Fixed Overhead	12 x 1024 bytes (12 pages)
Variable Overhead	((Nbr of Tables x 150) + (Nbr of Columns x 170)) bytes
Variable Overhead of other views	The sum of the variable overhead of all views referenced by this view: Σ Variable Overhead for all views
Total View Overhead in Pages	$\lceil$(Fixed Overhead + Variable Overhead + Variable Overhead of other views) / 1024$\rceil$ pages.

Fixed amount of overhead

This calculation accounts for the overhead of the system catalog (or data dictionary) tables in the database. SQLBase maintains the skeleton system catalog tables in a file called *START.DBS*. SQLBase uses the file as a "template" for a database file. Each new database is created originally from a copy of START.DBS (the CREATE DATABASE command and the API call *sqlcre* automatically copy START.DBS to the database named in the command). The size of START.DBS therefore has to be included in the fixed overhead estimate for a database. This size may change from one release of SQLBase to another, and should be obtained from the current version when performing the calculations. For our examples, we are using the size of START.DBS from SQLBase version 5.1.4.

There is a fixed amount of overhead for various physical objects. If the object is either a regular or a hashed table, the overhead is 12 pages. For non-hash indexes, the overhead is 2 pages. Therefore, the formula for the total amount of system overhead for a database is:

```
Total Fixed Overhead Pages = (Nbr Tables x 12) + (Nbr non-hash
Indexes x 2) + ⌈(Size of START.DBS / 1024)⌉.
```

In actuality, there is also some variable overhead associated with each type of object which depends on the current size of the object. As the size of the object increases, so will this overhead rise slightly. For example, the number of bitmap pages which SQLBase uses to manage page allocation for data rows will increase with the number of rows, however this increase is very small, at approximately 1K for every 8MB of data. Because of the tiny incremental nature of this overhead, formulas are not presented here to accommodate their inclusion in the overall estimation.

Example of size estimation

We present a simple database to explain the estimation of its size using the derived formulas. Assume that a packing density of 70% is desired for the CUSTOMER table, which is the subject of a hashed index.
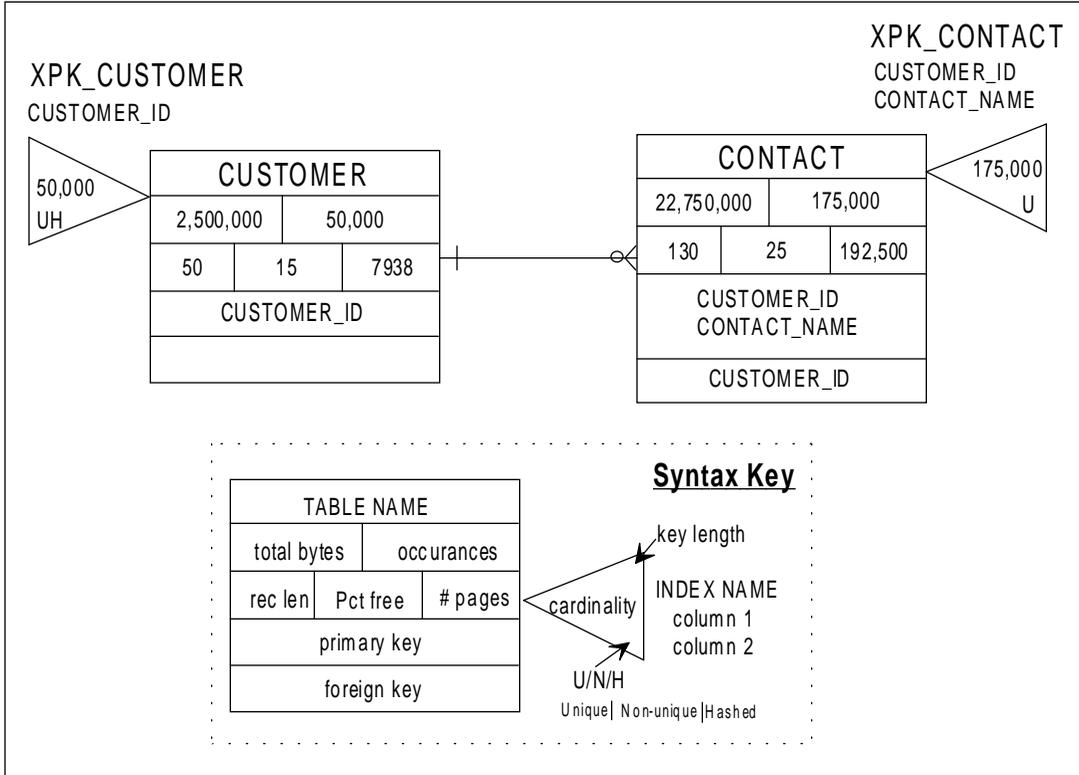
Figure 11B - Diagram of database following size calculations. All spaces concerning size information, with the exception of row occurrence estimates, would only be filled in following completion of the size estimation exercise. Note that the only data derived directly from the size estimates is the number of pages for each table. The "rec len" and "total bytes" fields reflect the external width of the

The SQLBase Data Definition Language (DDL) statements used to create this database, and associated views, follows:

```
CREATE TABLE CUSTOMER
(CUSTOMER_ID CHAR(5) NOT NULL,
CUSTOMER_NAME VARCHAR(25),
CUSTOMER_ADDR VARCHAR(50),
CUSTOMER_RATING CHARACTER(10),
PRIMARY KEY(CUSTOMER_ID))
PCTFREE 15;
CREATE TABLE CONTACT
(CUSTOMER_ID CHAR(5) NOT NULL,
CONTACT_NAME VARCHAR(25) NOT NULL,
CONTACT_PHONE DECIMAL(10,0),
CONTACT_TEXT LONG VARCHAR,
PRIMARY KEY(CUSTOMER_ID, CONTACT_NAME),
FOREIGN KEY CUSTKEY (CUSTOMER_ID) REFERENCES CUSTOMER
ON DELETE RESTRICT)
PCTFREE 25;
CREATE UNIQUE CLUSTERED HASHED INDEX XPK_CUSTOMER
ON CUSTOMER (CUSTOMER_ID)
SIZE 47628 ROWS;
CREATE UNIQUE INDEX XPK_CONTACT
ON CONTACT (CUSTOMER_ID, CONTACT_NAME)
PCTFREE 10;
CREATE VIEW BAD_CUSTOMER AS
SELECT CUSTOMER_NAME, CUSTOMER_ADDR
FROM CUSTOMER
WHERE CUSTOMER_RATING='POOR';
CREATE VIEW BAD_CONTACTS AS
SELECT C.CUSTOMER_NAME, CN.CONTACT_NAME, CN.CONTACT_PHONE
FROM BAD_CUSTOMER C, CONTACT CN
WHERE C.CUSTOMER_ID = CN.CUSTOMER_ID;
```

After an analysis of the data that will be loaded to the database was performed, the following data on average column widths was derived:

Table	Column	Maximum Width	Average Width
Customer	Customer_id	5	5
	Customer_name	25	10
	Customer_addr	50	30
	Customer_rating	10	5

Table	Column	Maximum Width	Average Width
Contact	Customer_id	5	5
	Contact_name	25	15
	Contact_phone	10	10
	Contact_text	500	100

The estimation of the database size is performed as follows:

1.   CUSTOMER:

   Data Length=∑Column Widths of Customer_ID, Customer_Name, Customer_Addr, Customer_Rating.

   Data Length=5+10+30+5=50

   Row Length=18+6+(2x4)+50=82

   Row Length with Slack=(82x100)/(100-15)=97

   Rows per Page=(1024-86)/97=9

   Nbr Row Pages=50,000/9=5,556

   Nbr Hashed Table Pages=5,556/.70=7,938.

   (Note that the .70 figure is desired packing density)

   Since there are no LONG VARCHAR columns in this table, the total data pages equals the number of hashed table pages.

2.   CONTACT:

   Data Length=∑Column Widths of Customer_ID, Contact_Name, Contact_Phone, Contact_Text.

   Data Length=5+15+(((10+2)/2)+1)+12=39

   Row Length=18+(2x4)+39=65

   Row Length with Slack=(65x100)/(100-25)=87

   Rows per Page=(1024-86)/87=10

   Nbr Row Pages=175,000/10=17,500

   Nbr Long Pages=175,000x1=175,000

   Total Data Pages=17,500+175,000=192,500

3. The following is for the BTree index calculation for XPK_CONTACT.

   Key Length=5+15=20

   Index Entry Length=9+2+20=31

   Usable Index Page Size=(1024-74)x(100-10)/100=855

   Index Entries per Page=855/31=27

   Nbr Index Pages=175,000/27=6482

4. Views:

   The first calculation is for the BAD_CUSTOMER view:

   fixed overhead=12x1024=12288 bytes

   variable overhead= (1x150)+(2x170)=490 bytes

   variable overhead of other views = 0

   total overhead of the view=(12288+490+0)/1024=13 pages

   The next calculation is for the BAD_CONTACTS view:

   fixed overhead=12x1024=12288 bytes

   variable overhead= (2x150)+(3x170)=810 bytes

   variable overhead of other views = 490

   total overhead of the view=(12288+810+490)/1024=14 pages

5. Fixed amount of overhead:

   (2x12)+(1x2)+(602,112/1024)=614 pages

6. Thus, the estimated size for the database is:

   7,938+192,500+6,482+13+14+614=207,561 pages.

   Since each page is 1024 bytes, or 1K, and one megabyte is 1024K, the size in megabytes is the number of pages divided by 1024:

   207,561 / 1024 = 203 megabytes.

Chapter 12

Result Sets

In this chapter we:

- Introduce the concept of the *result set*, which is the intermediate storage area used by SQLBase when processing a SELECT statement which retrieves multiple rows from one or more database tables.

- Cover the following detailed characteristics of result sets:

  - Structure

  - Contents

  - Life cycle

- Evaluate the implications that SQLBase's result sets have for program processing requirements, especially in the areas of locking and concurrency.

Introduction

A result set is a temporary index, possibly associated with a temporary table for certain complex queries, which contains a set of data that represents the result of a SELECT statement. Since most SELECT statement executions cause a number of rows to be returned from SQLBase, the result set typically consists of many rows. The contents of the result set's leaf pages (the bottom level in the B+ index tree structure) are pointers (ROWIDs) to the rows of the table (or tables) involved in the SELECT. Alternatively, for some complex queries these ROWIDs point to the rows of a temporary table containing the data values needed to satisfy the SELECT statement. The result set is stored on disk in a temporary file, and one or more rows are transmitted from it to the client workstation when requested via the FETCH command.

The utilization of result sets is an important feature of SQLBase. Through the use of result sets, SQLBase decreases response times for client workstation users while increasing their flexibility to position themselves within the result set when performing browse operations. SQLBase also offers a number of options relating to the processing of result sets which allow the system designer to balance the flexibility of result set processing with the concurrency requirements of his application.

Characteristics of result sets

The structure of result sets is standardized in a manner to allow their efficient use by the SQLBase server. However, the content of a result set as it exists on the server is determined in part by the nature of the SELECT statement which created it. The timing of the creation of result sets depends in part on the manner in which rows are fetched from it by the program which issued the SELECT statement originally.

Physical structure of result sets

The structure of result sets as they are stored in the server is that of a B+Tree index similar to any other SQLBase index. However, the result set is not a permanent object as user-defined indexes are and does not possess a name, nor is it owned by any database. It is owned by the transaction that created it, and is associated with the cursor used to process the SELECT statement whose results it represents. When this cursor is disconnected, or used to prepare another SQL statement, the result set is dropped from the system.

The subject of the unique index structure in a result set is the ROWID column contained within each leaf entry. To summarize chapter 8, *Database Pages*, the following ordered list of elements makes up the ROWID field, which serves to uniquely identify every row in an SQLBase database:

- Physical page number that the row resides on.

- Slot number within the page that the row occupies.

- Row serial number, which is assigned at the time of insertion and is unique among all rows. This number is never updated during the life of the row, and is never reused for any other row following the deletion of this row.

- Update serial number of the row. This number is changed each time the row is updated.

Note that the first three elements will always uniquely identify a single row, while the addition of the fourth element serves to further identify that unique row for a single state, or change level. When the row changes to a new state, through being the subject of an UPDATE command which modifies one or more columns, then the fourth element of the ROWID changes while the first three remain the same. Also, only the first two elements are needed by SQLBase to actually retrieve the row from storage. The third element serves to verify that the same row still occupies this position in the database, and that the slot has not been reused by some other row following the deletion of this row.

Logical contents of result sets

Result sets are composed of ROWIDs which point to either base table rows, or rows in a temporary table, depending on the query that was executed. Result set columns contain base table ROWIDs when the query references data columns as they appear in the actual table rows, or when a calculation or formula is to be applied to a single row column value. These ROWIDs are then used at fetch time to access the base table row where the data is located. Result set ROWIDs point to a temporary table when the column to be returned is a value calculated from more than one input, or an aggregate function, which may use one or more rows of the base table as input.

.

```
┌─────────────────────────────────────────────┐
│                    Query                      │
│    Select Acctnum, Acctname from Customer;    │
└─────────────────────────────────────────────┘
```

```
┌─────────────────────────────────┐
│       Result set leaf entries    │
├─────────────────────────────────┤
│          ROWID of row 1          │
├─────────────────────────────────┤
│          ROWID of row 2          │
├─────────────────────────────────┤
│          ROWID of row 3          │
├─────────────────────────────────┤
│                •                 │
│                •                 │
│                •                 │
├─────────────────────────────────┤
│          ROWID of row n          │
└─────────────────────────────────┘
```
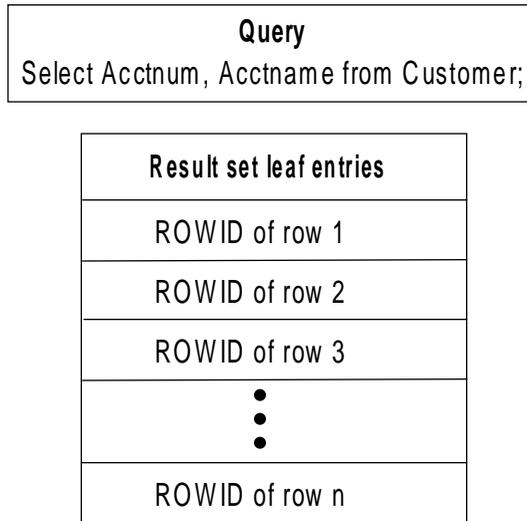
Figure 12A - Result set from a single table query with no derived data. The only entry in the leaf pages of the index tree is the ROWID of each of the Customer table's rows which qualifies for inclusion (in this case, all rows).

In Figure A the result set of a simple one-table query is shown. For SELECT statements such as this, the result set consists of the ROWIDs of the selected rows in the Customer table. When the client performs a fetch against the result set, the row or rows returned are identified by the ROWID at the cursor position. As the data rows are accessed, the columns requested by the query are retrieved and formatted for return to the client.

```
┌─────────────────────────────────────────────┐
│                   Query                       │
│                                               │
│  Select C.Acctnum, C.Acctname, I.Invoice_amt  │
│           from Customer C, Invoice I          │
│          where C.Acctnum = I.Acctnum;         │
└─────────────────────────────────────────────┘
```

Result set leaf entries	
Customer table ROWIDs	Invoice table ROWIDs
ROWID for row 1	ROWID for row 1
ROWID for row 2	ROWID for row 2
ROWID for row 3	ROWID for row 3
• • •	• • •
ROWID for row n	ROWID for row n

Figure 12B - Result set from a two table join. ROWIDs identify the qualifying rows of each table, and allow SQLBase to retrieve the appropriate columns from the selected rows of the base tables at FETCH time.

In figure B, the result set of a simple two-table join is shown. Since all of the data columns requested by the query are present in the rows of the two tables, the only columns in the result set are the ROWID pointers for the joined rows. Each ROWID pair in the result set satisfies the join condition, which is that the account number in the Customer row matches the account number in the Invoice row. When the actual rows are fetched by the client, SQLBase will use the ROWIDs to retrieve the actual data values requested by the query from both tables involved in the join. Note that this query will result in one row for each invoice a customer has on file, in other words

each customer number will be repeated in multiple rows for each invoice that customer has.

```
                          Query

Select C.Acctnum, C.Acctname, SUM(I.Invoice_amt)
       from Customer C, Invoice I
       where C.Acctnum = I.Acctnum
       group by C.Acctnum, C.Acctname;
```

Result set leaf entries		Result set temporary table		
Temporary table ROWIDs		Column 1 ___ C.Acctnum	Column 2 ___ C.Acctname	Column 3 ___ SUM of I.Invoice_amt
ROWID for row 1	→	VALUE for row 1	VALUE for row 1	VALUE for row 1
ROWID for row 2	→	VALUE for row 2	VALUE for row 2	VALUE for row 2
ROWID for row 3	→	VALUE for row 3	VALUE for row 3	VALUE for row 3
⋮		⋮	⋮	⋮
ROWID for row n	→	VALUE for row n	VALUE for row n	VALUE for row n

Figure 12C - Result set from a two table join aggregate query. Since this query contains aggregate data in the form of the SUM (I.Invoice_amt) column, a temporary table is built to hold the result set data. The result set itself points to the rows of this temporary table. This temporary table is also sometimes called a 'virtual result set.'

The aggregate query in Figure C has a significant difference from the previous queries we have examined, in that its result set contains the ROWIDs of a temporary table. This temporary table contains the actual row values to be sent back to the client in response to fetch statements. These values must be stored in a temporary table otherwise the entire aggregate set of data would have to be retrieved again in order to recalculate the result of the function. The query in figure C contrasts with the query in figure B in that it returns a single row for each customer. This row will contain the total value of all invoices for that customer, calculated by the SUM function which is applied to the invoice amount field. SQLBase creates a temporary table for any query that includes aggregate functions, complex views, or queries containing the DISTINCT, GROUP BY, HAVING, UNION, or ORDER BY clauses.

This delineation of queries that require calculations, aggregation, or other complex processing determines variations in other characteristics of result set processing. Result sets which contain the ROWIDs of base tables possess a greater flexibility for processing by SQLBase. These result sets offer some capabilities to client transactions that the other result sets do not. For the remainder of this chapter, the differentiation between these two types of result sets will be made through the term "derived". When a result set contains the ROWIDs of base tables only, where no temporary tables are built, all the data needed to satisfy the query can be retrieved from the base table rows. More complex queries, such as aggregation queries, have data derived from the processing of the query present in the temporary table, pointed to by the result set, for subsequent transmission to the client.

To summarize, the two basic types of result sets are:

- The non-derived result set, which contains only ROWIDs of data rows in the base tables.

- The derived result set, which contains a ROWID pointer to a temporary table that contains at least one column of actual data values which represent the result of some calculation or function in the query. This type of result set is also called a "virtual table".

Population of result sets

For most queries, the first part of the result set is built when the SELECT statement is the subject of an EXECUTE. When there are sufficient rows to fill one message buffer to the client, these rows are transmitted and the client may proceed. After the client has performed sufficient FETCH operations to exhaust the contents of the message buffer, a request is made to the server for another buffer of rows. The server then continues to build sufficient rows in the result set to fill another message buffer, which is transmitted to the client workstation. This process continues until all rows that are eligible for inclusion in the result set have been accessed. Note that this process assumes a sequential access of result set rows by the client workstation's program, starting at row 1 and proceeding through the result set, one row at a time, until all rows are retrieved.

Of course, the program at the client workstation may not proceed through the result set in this orderly manner. When the program explicitly repositions in the result set, SQLBase may be required to complete building the entire result set prior to sending the requested row back to the client. When this occurs, the user at the client workstation may perceive a delay if the result set that must be completed is large.

Also, some queries may require the completion of the entire result set prior to returning any rows. An example of this is a SELECT statement containing the ORDER BY clause. This clause requires that the result set rows be ordered according to the contents of one or more columns. If there is an index on the base table which

matches the sequence of the ORDER BY clause, then the result set may be built incrementally, as outlined above. However, if there is no index that can be used to satisfy the ORDER BY clause, then SQLBase will have to perform a scan of the entire table, followed by a sort operation, in order to put the result set in the correct sequence. Only following this sort may the first row of the result set be returned to the client.

Flushing of result sets

When result sets are no longer needed by the program that created them, they are *flushed* from the server. This is analogous to dropping a database object, where all traces of the object are eliminated from the system and there is no way to bring the object back. If the result set is large, the temporary file used by the server for its storage will also be freed. As mentioned previously, the result set table is not a permanent object nor does it possess a name or ownership by any database. It is owned by its creating transaction, and is associated with the cursor used to process the SELECT statement. In the absence of any special considerations taken to preserve the result set, the result set will be flushed when one of the following events occurs:

- When the owning cursor is disconnected, or used to prepare another SQL statement.
- When the program re-compiles the SELECT statement.
- When the transaction requests a COMMIT or ROLLBACK.
- When the transaction changes the current isolation mode setting.

Cursor context preservation

If the programmer has turned cursor context preservation (CCP) on, by setting the SQLPPCX flag on, then the above rules for the flushing of the result set are changed somewhat. Specifically, the program's result sets are maintained across a COMMIT, allowing the program to perform update operations and commit the results without losing its current place in the result set. Also, for programs running in the RL isolation level, and not performing DDL commands, result sets will be kept intact across the execution of a ROLLBACK command.

This flag allows programs to browse large result sets, perform updates based on the result sets, commit or rollback those updates as desired, and then continue their browsing without having to reestablish the result set. This makes these programs very effective in a multi-user environment, since the commit allows locks to be freed up which other transactions may be waiting on, while the CCP feature allows the transaction to avoid the overhead of building the same result set multiple times.

Processing result sets

You need to be aware of the implications that SQLBase's use of result sets have for transactions. While the use of result sets can significantly increase the speed of transaction execution, programs occasionally need to take steps to avoid some anomalies caused by result set usage. These aspects of result set processing may seem all the more odd because of the behavioral variations between derived and non-derived result sets, and the fact that many people may be unaware of the basic processing differences in these types of queries.

Result sets are built at a specific point in time during the execution of a program, namely when a SELECT statement is executed. However, the data contained within the result set may be perused by the client workstation user over a more extended period of time, particularly when browsing a large result set. This raises two questions relating to concurrency:

- What locks are held on the rows that were used by the SELECT to build the result set? This will affect other transactions' ability to access these rows.

- What happens if some other program updates a table row that was used to build the result set? This affects the timeliness of the data being returned to this transaction. Another way to ask this question would be: Does the information in the result set represent data at the current point in time, or at the point in time that the SELECT was processed?

Note that some transactions may not be too concerned with these issues. They may only need a result set which is representative of the data at the time of the SELECT and do not care if the data has changed since that time. These transactions should choose a high concurrency isolation mode for execution (probably RL), so that other transactions may perform updates without interference. The transaction may then browse its result set without concern for the timeliness of the data.

An example of a program that may not require a high degree of concurrency is an "orders shipped today" inquiry screen in an order processing system. A user viewing this screen would not realistically expect to see an order on it that just went out the door while he is browsing the result set of orders. The probable reason the inquiry exists is for management to identify trends and potential problems with the order shipping process and this use does not require that the screen reflect every single order in real time. The accurate representation of "orders shipped today" would be a batch report program that was run at the stroke of midnight, thereby guaranteeing that no more orders could be added to the desired information.

Lock duration on result set rows

How long locks are held by result sets is determined by the executing transaction's isolation mode. The possible isolation modes are Repeatable Read (RR), Cursor Stability (CS), Release Locks (RL), and Read Only (RO). While these are discussed in detail in chapter 13, *Concurrency Control*, their effects on result set processing are summarized below.

- In RR, S-locks remain on all pages where rows from the result set are located throughout the life of the transaction, or the next commit or rollback. While other transactions may read these rows, they will be unable to update them until the S-locks are released.

- In CS, an S-lock is only held on the current page where the most recent row FETCHed by the client is located. Since the server passes only one row at a time to the client in this mode, the page with the S-lock will be the page occupied by the row last passed in the message buffer. The S-lock will move to the page occupied by the next row in the result set when another FETCH is performed by the client. This means that a row in a page of the result set could be updated by another transaction so long as it is not the page containing the current row of the result set.

- In RL, no locks are held on result set rows permanently. During the actual execution of the SELECT statement, to build the result set, one S-lock is held on each page as it is accessed, but this lock is dropped as soon as the page is no longer needed. Once the result set is built, all locks are released. However, each row in the result set is subject to one more S-lock at the moment the fetch of that row is performed in order for SQLBase to make sure the row is not the subject of an uncommitted update from another transaction at that point in time. This S-lock is immediately released when SQLBase finishes the fetch operation for that row.

- In RO, no locks are ever held. Row images are always made current to the SELECT execution time through the use of timestamp comparisons and a RO history file which holds the prior versions of row images.

Concurrency and derived result sets

For queries which create derived result sets, or virtual tables, the remainder of the concurrency situation is fairly simple. Since the data that makes up the results of the query is actually stored in the temporary table, any updates made by other transactions after the result set is built will not be reflected, nor can these updates be detected. There is no process that can avoid this result, such as the use of FETCHTHROUGH or committing of the update. Also, committing the update with cursor context preservation on will not cause the update to be reflected in the result

set either. The only way to make the results of the update appear in the virtual table is to re-execute the SELECT statement which built it.

Concurrency and non-derived result sets

On the other hand, if the result set contains no derived columns, a transaction can detect updates to rows, or deletions of rows, made after the result set was built. For this type of query the following sections describe the techniques and procedures for dealing with these possible updates and deletions made by other transactions. Note that in the case of insertion of new rows by other transactions that would have appeared if they had been committed prior to the creation of the result set (meaning that they satisfied the predicates specified in the WHERE clause), these new ROWIDs will not appear in the result set under any conditions. The only way to include them in the result set would be to re-execute the SELECT statement in order to re-evaluate the predicates against the existing set of committed data in the table.

Detecting changes in result set rows

Transactions which execute in the RR isolation mode will not have this problem, since this mode will not allow any updates to occur to the result set. While this may be appropriate for some transactions, they should only be executed during low concurrency time periods, such as the middle of the night, otherwise a large number of time-outs and deadlocks may occur. For transactions which are processing result sets in the CS or RL isolation modes, there is a way to keep the data returned from the result set current.

SQLBase provides this capability automatically when fetching rows from the result set. Since this is done at fetch time, any rows that have already been fetched, and then updated by a different transaction at a later point in time (following the fetch), will not be detected. At the time that the fetch is performed from the result set, SQLBase uses the ROWID stored in the result set to verify the current state of the row as it is actually stored in the base table. It does this as part of the following process:

How SQLBase fetches result set rows

1. Access the ROWID stored in the result set row.

2. Retrieve the page using the page number component of the ROWID. This page could either be found in a buffer or be read from external storage.

3. Check the slot specified by the ROWID slot component to make sure the "used" bit is set (first bit in slot table entry). If this bit is off then the row has been deleted. If this has occurred, then SQL return code 00003 (FET NF: Row has not been found) is returned to the client.

4. Retrieve the row from the page using the offset specified in the slot table. This structure is explained in detail in Chapter 8, *Database Pages*.

5. Compare the row serial number component in the ROWID to the row serial number actually stored in the slot. If these are different, then the original row has been deleted and the slot has been reused by another row. If this has occurred, then SQL return code 00003 (FET NF: Row has not been found) is returned to the client.

6. Compare the update serial number component in the ROWID to the row update serial number actually stored in the row. If these are different, then the row has been updated since the SELECT statement that built the result set was processed. If this has occurred, then SQL return code 00002 (FET UPD: Row has been updated) is returned to the client, indicating the row has been updated. The newly updated row is returned to the client.

This procedure ensures that transactions processing result sets in high concurrency isolation modes can still detect modifications to the rows in their result set which occur after the result set is built.

Maximum timeliness with FETCHTHROUGH

The process just described is only effective if the row fetched by the client is not in the workstation's input message buffer. If the row is already in the client's buffer when an update takes place, then the transaction will not be able to detect it. In order to eliminate this effect, the SQLBase FETCHTHROUGH feature can allow a transaction to execute with maximum concurrency in an isolation mode such as RL, while still accessing the most recent data available. By setting the FETCHTHROUGH parameter on (via session setting or API call), the transaction can ensure that for each FETCH performed, the data will be retrieved directly from the server, who will perform the automatic recency verification described above. The alternative to this, with FETCHTHROUGH set off, would be that the row may be retrieved not from the server at all, but rather from the client's own message buffer. The contents of this buffer may have been built some time before, and no longer contain the most recent information. Because of the performance impact of using FETCHTHROUGH, however, you should only turn it on when you know that the result set rows will be updated during the life of the result set.

An example of a transaction that may require this high level of timeliness is a shipment pick transaction for a high volume warehouse. The way the transaction might work is that while a stock clerk makes his way down an aisle in the warehouse, he scans the bin numbers he passes with a bar code reader. The transaction being executed on his hand held computer has already performed a SELECT on the SQLBase server (connected via radio frequency modem) and built a list of items and quantities to be picked in the same order that the clerk will traverse the warehouse. As the clerk scans a bin, the transaction executes a FETCH of that bin's row in the Warehouse table to see how many of the bin's items should be picked by the clerk for conveyance to the packing area. Since orders are being placed constantly, the

transaction executes with FETCHTHROUGH on in order to be aware of the latest order information that may affect the quantity to pick from this bin. In this way, the clerk will be performing warehouse picks for orders which were received after he started his run through the warehouse. This thoroughness on the part of the shipping process helps to decrease the time required to turn around a customer's order, which helps the company achieve several of the strategic business goals that they purchased the system to accomplish.

The trade-off involved in guaranteeing the most recent data through use of the FETCHTHROUGH feature is an increase in network message traffic for the transaction. This is because instead of filling the message buffer with as many rows as possible to transmit to the client, the server will now place only one row at a time in the buffer. This will cause the client workstation to experience slower response times when processing large result sets. For this reason, FETCHTHROUGH should only be used on those situations where it is critical to have the most recent information, and there is reason to believe that the information is in a constant state of change. Also, anytime you know the row has been updated, such as through an UPDATE statement issued in another cursor of your program, you should set FETCHTHROUGH on to ensure that the update will be reflected in the result set.

Chapter 13

Concurrency Control

In this chapter we:

- Introduce the topic of concurrency control, which is the way a SQLBase server makes each client workstation appear as if it were in sole control of the database.

- Discuss the concept of transaction serialization, which dictates the requirements for inter-leaving transaction executions in a manner which eliminates interference between transactions and makes each transaction individually recoverable.

- Explain the various types of locks SQLBase uses to control transaction executions, and the characteristics of these locks.

- Delineate the various isolation levels available in SQLBase, which determine how long certain types of locks are held during a transaction's execution.

- Explain how to select an appropriate isolation level for a program, balancing the trade-offs of data integrity and maximum performance for all executing transactions.

Introduction

The topic of concurrency control addresses the problems faced by SQLBase in supporting a multi-user environment. When users are at their workstations using some type of client software, which could be a SQLWindows application, Quest, WinTalk, or some other program, and are connected to a SQLBase database server, they expect the results of their programs to be exactly the same as if they were executing against a local copy of the database contained on their own workstation. Users want to remain unaware that many other users are also connected to the same database and are using it in a variety of ways. Unfortunately, there are problems inherent in multi-user database access that must be considered by both SQLBase and the application developer in order to realize the users' objective. These problems are caused by the pitfalls inherent in allowing many users to access the same database concurrently. The topic of *concurrency control* addresses the software mechanisms used by SQLBase to overcome these difficulties.

The unit of work that is the subject of the concurrency problem is the *transaction (*which is discussed at length in Chapter 5, *Transaction Definition*). One of the key goals of concurrency control is that either all updates of a transaction are made permanent if its execution is successful, or it otherwise has no effect at all on the shared database. The other goal of concurrency control is that transactions not be allowed to interfere with each other. Each transaction should appear to have sole control over the shared database without concern for other transactions whose execution is being conducted simultaneously by SQLBase.

Transaction serialization

Transactions are executions of programs against the shared database which are delineated by an implicit start when the program connects to the database, and are ended through either a commit or a rollback command. Transactions which complete successfully signal this event to SQLBase through the COMMIT command, at which time the changes made by the transaction are made permanent within the database. Unsuccessful transactions terminate with the ROLLBACK command, which causes SQLBase to back-out all changes to the database that were previously performed on behalf of the transaction. If the program continues to perform database calls after the execution of a COMMIT or ROLLBACK, a new transaction is considered to have begun. A rollback of the transaction may occur for a variety of reasons besides program request (as indicated through the program issuing the ROLLBACK command) including program failure (abort), system failure (OS abend), or environmental failure (power failure). If any of these events occur, then SQLBase must ensure that all non-committed transactions are rolled back so that the database never appears to have been effected by them.

A transaction may have two possible effects as a result of its execution:

- Effects on data through the execution of INSERT, UPDATE, and DELETE DML statements.

- Effects on other transactions, where the other transactions have read data written by this transaction. These other transactions may use uncommitted data to perform further database alterations, thereby proliferating the effects of this transaction's updates.

The effects of a transaction that were made directly to the database through the use of DML statements are eliminated through the performance of a rollback by SQLBase using the record images that are kept in the log files (the details of this operation are covered in Chapter 14, *Logging and Recovery*). Eliminating the secondary effects on other transactions is where the mechanism of *database locks* comes into play. These locks are the mechanism used to ensure that transactions occur in a serialized fashion, thereby allowing their recovery without impacting the integrity of other transactions which may otherwise have been dependent on data created by an uncommitted transaction.
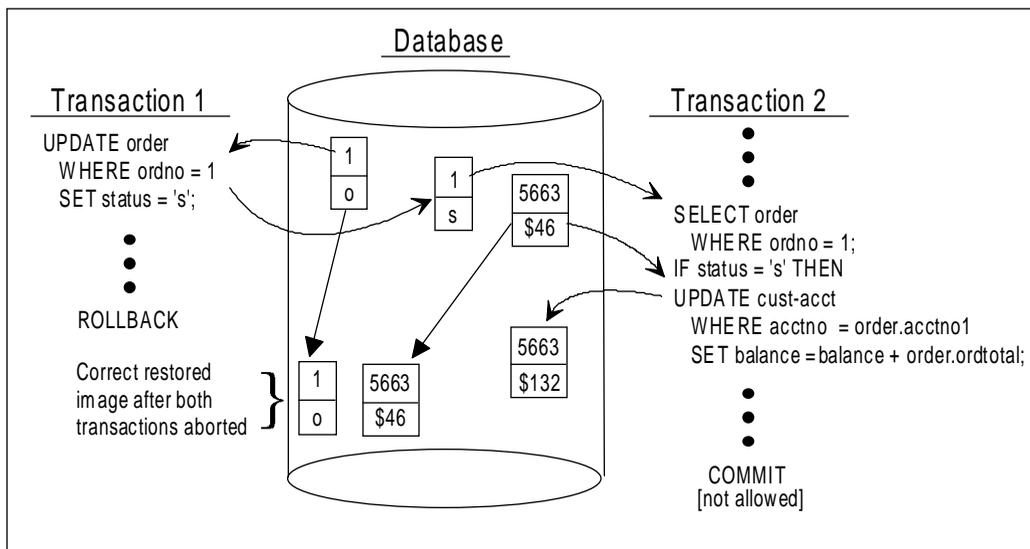
Figure 13A - Example of transaction dependency without database locking where Transaction 2 should not be allowed to commit since it depends on data supplied by the uncommitted Transaction 1. This is not a serialized execution, and is therefore only recoverable by rolling back Transaction 2 along with Transaction 1, called a "cascading abort."

An example of secondary transaction effects is shown in Figure A, where transaction 2 has read data from transaction 1 prior to transaction 1 committing. This example illustrates the impact of not having locks, which is that transaction 2 must be aborted due to the request of transaction 1 to rollback rather than commit. This *cascaded abort* of transaction 2 is required to preserve the semantic integrity of both transactions, since allowing transaction 2 to commit would propagate inaccurate data. The alternative of not processing the rollback request of transaction 1 would also allow inaccurate data to remain in the database.
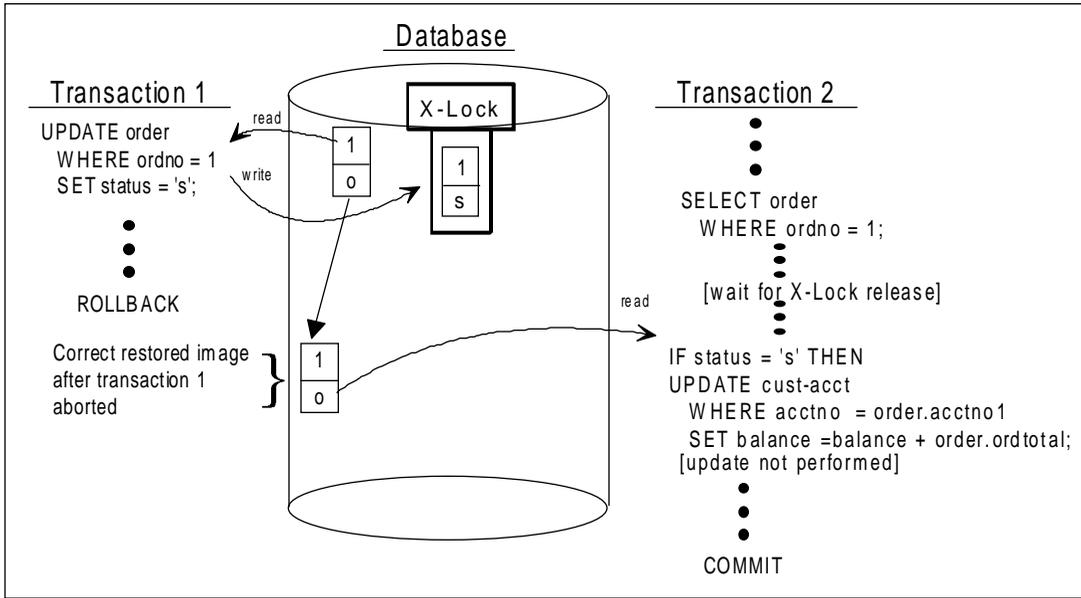
Figure 13B - The same as Figure A, except with locks. Transaction 1 places an exclusive lock on the row it updates, which causes Transaction 2 to wait until Transaction 1 completes prior to reading the record. This is a serialized execution, and both transactions are fully recoverable through rollback techniques.

Figure B shows the same two transactions executing with database locking performed. Following the update of the ORDER row by transaction 1, SQLBase places an exclusive lock on the page containing the row. This type of lock can only be held by one transaction at a time and blocks all other transactions from accessing the page (more detail on locks follows later in this chapter). When transaction 2 attempts to read the ORDER row, via a SELECT statement, it is placed into a wait state until the exclusive lock is released. During this time, transaction 1 requests a rollback, which is performed. Following the rollback, the exclusive lock is lifted on the ORDER row, and transaction 2 is allowed to continue. Notice that the row image returned to transaction 2 is not the one that existed at the time of its request, as in

Figure A, but rather the restored image which existed prior to modification by transaction 1. Since the status value is no longer 's', transaction 2 no longer modifies the CUSTOMER row as in Figure A, but terminates with a COMMIT anyway. These transactions are now serialized, they no longer execute in an inter-leaved fashion which allows them to affect each other's execution.

This example is one of a number of ways in which transactions may interfere with each other's correct execution through their interaction within a shared database. Other common problems that may occur due to lack of concurrency control include:

- Lost Updates

  Lost updates occur when two or more transactions retrieve the same record and make successive updates of that same record. If no method of locking is used, transactions can overwrite recent modifications without the benefit of seeing the newly updated record.

- Uncommitted Dependency Problem

  Rollbacks issued by one transaction can also cause another transaction to lose updated data if no locking protocol is used. If a record is updated by one transaction but not committed, then subsequently updated by a second transaction with no commit performed, a rollback by the first transaction will also rollback the second transaction's updated data.

In order to eliminate these problems, SQLBase ensures that transactions execute in a strict serialized fashion through the use of database locks. These serialized executions achieve the same effect as executing each transaction in serial fashion, with respect to all other transactions, such as would happen if they were dispatched through a FIFO (First In First Out) queue. Of course, SQLBase does not accomplish this by FIFO, since only through concurrent executions can the speed that is required for an effective RDBMS be achieved. When viewed conceptually, however, for any single transaction the effect of its execution is exactly the same as if it were executed serially, without other transactions present in the system.

Lock types and scope

SQLBase employs two mutually exclusive types of locks: *Shared Locks (S-Locks)*, and *Exclusive Locks (X-Locks)*. An alternative terminology that is sometimes used refers to S-Locks as *read locks* and to X-Locks as *write locks*. The one other type of lock which SQLBase uses on a temporary basis is the *Update Lock (U-lock)*, which is sometimes called *incremental lock* because of its transient nature. These locks are placed implicitly (that is, without being specifically requested by the transaction) by SQLBase in response to the various DML commands issued by a transaction, with one exception. This exception is the 'FOR UPDATE OF' subclause of the SELECT

command, which causes an explicit U-lock to be placed in anticipation of an update occurring later within the transaction.

S-locks

With an S-lock, more than one simultaneous transaction can lock the record; therefore, more than one transaction can access the record. S-locks are applied when a transaction reads data via a SELECT statement. When data is held by an S-lock, inserts, updates, and deletes, which require X-locks, are not allowed. Once the S-lock is removed, that is, once there is again only one transaction accessing the data, inserts, updates, and deletes are permissible (X-locks are allowed). A transaction may *upgrade* an S-lock to an X-lock only if there are no other transactions holding S-locks on the same data.

X-locks

With an X-lock, only one transaction at a time can use or lock the data. X-Locks are applied to data on inserts, updates, and deletes. While the X-lock is in effect, other transactions cannot access the data for any purpose. Transactions cannot place an X-lock on data while an S-lock on the data is in effect for some other transaction.

U-locks

Transactions place U-locks when they have retrieved data with a clear intent to update. This is usually done internally by SQLBase in order to avoid deadlock situations, such as when building a result set for a UPDATE ... WHERE command that will affect multiple rows. Only one U-lock can be held on a page at one time, similar to X-locks, but this single U-lock can coexist with S-locks, unlike X-locks. However, when the transaction holding the U-lock wishes to perform the update, which requires upgrade of the U-lock to an X-lock, then the rules for X-locks apply. This means that any S-locks that may have been coexisting (placed either before or after the U-lock was set) with the U-lock must be released prior to the upgrade to an X-lock taking place. These characteristics of U-locks make them less susceptible to deadlock situations, as will be seen later in the discussions on isolation levels and index locking.

The U-lock is the one lock in SQLBase that can be set explicitly. Whenever a transaction issues a SELECT ... FOR UPDATE OF command, U-locks are placed on the pages of the result set. If the program later issues an UPDATE ... WHERE CURRENT OF command referring to the cursor of the SELECT, these U-locks are upgraded to X-locks for the remainder of the transaction. Any U-locks that are still in place are allowed to downgrade to S-locks when the cursor under which the SELECT command was executed is used to prepare and execute a new statement (depending on the isolation level in effect).

Lock compatibilities

The following table summarizes the capabilities of each particular type of lock to coexist with the other types.

	Lock attempted to place		
Existing lock on page	*S-lock*	*U-lock*	*X-lock*
S-lock	Allowed	Allowed*	Wait
U-lock	Allowed*	Wait	Wait
X-lock	Wait	Wait	Wait

Note: *While S-locks and U-locks may co-exist when they are placed, the S-locks will have to be released before the U-lock may be upgraded to an X-lock.

Locking level

SQLBase places locks at the *page* level. If the record being accessed fills most of the page, only that record is locked. Otherwise, a group that is composed of the record and its adjacent records, which together make up a page, is locked. A page can contain any sort of data, including indexes and system catalog data (detailed descriptions of the various SQLBase page types can be found in Chapter 8, *Database Pages*).

Locking scope

The scope of the locks implicitly placed by SQLBase cover all the data structures that the optimizer determines are required to execute each command within a transaction. When processing a SELECT statement, the optimizer will lock those index leaf pages that are required to build the result set, if an index is chosen as a component of the access path. Also, all data pages read during the processing of the SELECT statement will have S-locks placed upon them. Often, contention between transactions occurs not on the data pages but within the index structures due to the potential proliferation of X-locks within the index. When S-locks are released is a function of the transaction's isolation level, but is also affected when transactions use cursor context preservation. On the other hand, X-locks are always held until either a commit or rollback is issued, regardless of the transaction's isolation level or the use of cursor context preservation.

Cursor context preservation

Normally, all of a transaction's locks are released by a COMMIT command. If the transaction has turned cursor context preservation on (commonly abbreviated CCP) for one or more cursors, through setting the SQLPPCX flag to one, then the rules for releasing locks change somewhat. Specifically, those cursors' result sets and associated S-locks are maintained across the COMMIT command, allowing the transaction to perform update operations and commit those results without losing its current place in the result sets of open cursors. Also, for transactions running in the RL isolation level, and not performing DDL commands, result sets will be kept intact across the execution of a ROLLBACK command. This means that following the COMMIT command for a transaction with CCP on for some cursors, locks will still be held (as required by the transaction's isolation mode) for those result sets associated with SQL SELECT commands of those cursors.

The CCP flag allows programs to browse large result sets, perform updates based on the result sets, commit or rollback those updates as desired, and then continue their browsing without having to reestablish the result set. This makes these transactions very effective in a multi-user environment, since the commit allows locks (particularly X-locks) to be freed-up and which other transactions may be waiting on, while the CCP feature allows the transaction to avoid the overhead of building the same result set multiple times.

Index locking

In the case of an INSERT or DELETE, X-locks will be placed not only on the page containing the row which is the subject of the command, but also on at least one page for every index that exists on the table. This same case holds true for UPDATE commands specifying columns which are the subject of one or more indexes.

The index page that will definitely be X-locked is the page containing the leaf entry pointing to the subject record. This is the only page in the index which will have a permanent X-lock placed on it, this X-lock will not be released until the transaction commits or ends. If index structure adjustments need to be made, such as splitting index records, then additional X-locks will be taken as well. These additional locks will be taken on those nodes within the index structure that have to be updated to perform a split or the movement of a successor key up the tree. These X-locks will be released once SQLBase completes the operation on the index tree; they will not be held through the life of the transaction (in contrast to the X-lock on the affected leaf page).

SQLBase performs the following steps when performing a DML INSERT operation which must modify an index (both DELETE and UPDATE function similarly and therefore are not explained):

- As SQLBase traverses the index tree, it places a temporary U-lock on each node it accesses. This starts with the root node and progresses downward.

- When a node within the tree is traversed, it is checked to see if it is full. If it is not full, then all the temporary U-locks on nodes higher up in the tree are released, not including the current node. This is done because any index split that may be required could not effect any higher nodes than the current, non-full node, since this node can accommodate another entry without splitting.

- When SQLBase reaches the leaf node and it is not full, then no split will be required and all temporary U-locks on higher level nodes can be immediately released. If the leaf is full, then a split is required and all higher level tree nodes that need to be adjusted as a result of the operation will have their U-locks upgraded to temporary X-locks. Of course, the leaf node itself will always be the subject of a permanent X-lock.

- SQLBase performs the index updates, both to the leaf nodes as well as any nodes further up the tree that may be affected.

- The permanent X-lock on the leaf page is always held until the transaction terminates with either a COMMIT or a ROLLBACK. All temporary X-locks on higher level pages within the index structure can now be released.

The most important aspect is that some time during the performance of a DML modification statement that affects an index, there will be a temporary U-lock on the root page of the index. Hopefully this lock will be in place for only a very short time, depending on the condition of the index tree and the insertion pattern followed by the application. However, if some other transaction should come along and want to perform an equivalent operation, they will have to wait in order to place their own temporary U-lock on the root page as well. Therefore, any application environment that is very update-transaction intensive should attempt to eliminate all but the most critical indexes on those tables that are used by most or all transactions. Each index placed on a table that is inserted, deleted, or modified at a rapid rate, will have the effect of placing a single-threaded contention point into each transaction that performs these operations.

Lock implementation

The mechanism used by SQLBase to implement locks is through an in-storage hash table. This storage construct is used because it is efficient in storing and retrieving tables with a large number of entries. When a new lock is needed, a previously allocated, but currently free lock entry may be re-used. If no lock entries are available for reuse, then additional lock entries may be dynamically added to the table in groups of 100 locks at a time. These allocations can be added to the table until the maximum number of locks specified in the LOCKS keyword of the SQL.INI file is reached, or until the server has no more memory to allocate. Each lock entry in the

table represents either a single page or a range of pages, and contains the following data:

- The group number owning the page, or page range, that is locked. As explained in Chapter 8, this number uniquely identifies one object within the database, such as a table or index.

- The logical page number of the page that is locked, or the starting logical page number of a page range that is locked. Logical page numbers are assigned sequentially within a group starting from one.

- The logical page number of the end of the range which is locked.

- The type of lock that is held, either S, U, or X. This is prefixed with a 'T' if the lock is temporary, such as TS, TU, or TX.

- The transaction identifier of the owner of the lock.

Whenever SQLBase is performing a read or a write operation, the lock table is checked for the group and page on which the operation is performed. If no conflicting lock type exists in the table, then a new lock is added on behalf of the current transaction before the operation proceeds. In this way, SQLBase ensures that locks are set prior to performing the operations they control, and that no lock conflicts exist that could undermine the integrity of the system.

Isolation levels

The protocols that are used by SQLBase to perform locking operations on behalf of a transaction are called *isolation levels*. These define how long locks will be held and how SQLBase will pass data to the transaction via the message buffer. The four available isolation levels are: Read Only (RO), Repeatable Read (RR), Cursor Stability (CS), and Release Locks (RL). These are often referred to by their abbreviations. The isolation level is selected by a transaction when it first opens a cursor to SQLBase, and remains in effect for the life of the transaction. The isolation level that is selected by the transaction applies to all cursors that will be used by the transaction. Changing isolation levels in the middle of a transaction has the effect of causing an implicit COMMIT and starting a new transaction with the new isolation level. All of the transaction's cursors will now operate under this new isolation level. Also, having cursor context preservation on will not preserve the result sets of any cursors, nor their locks, when changing a transaction's isolation level in midstream.

Using isolation levels effectively

Repeatable Read and Cursor Stability deadlock avoidance

While the Repeatable Read (RR) and Cursor Stability (CS) isolation levels offer the greatest levels of data consistency available to transactions that perform updates, their low level of concurrency (especially RR) leads many application developers to avoid them. These levels may be appropriate choices, however, for those systems that demand a high level of data consistency, due to the nature of their application. This is particularly true of the CS level when used in an intensive transaction processing environment where few rows are created or updated by each transaction execution.

The use of CS for these cases allows a guarantee of consistency without severely impacting performance provided that some guidelines are used in order to minimize the occurrence of deadlock situations. One drawback in using the CS mode that should be kept in mind is the increase in network traffic that may be caused by the single-row buffering used by CS to respond to FETCH commands. The following guidelines, when used with the CS isolation mode, will minimize deadlock situations:

- All transactions should acquire their lockable resources (table rows) in the same order. An example would be to read the CUSTOMER table, then the ORDER table, then the ITEM table in that sequence. By conforming to an ordered specification for reading and updating tables, different transactions can avoid deadlocks that are caused by contending for locks between unlike transactions.

- Transactions which perform updates of rows returned from a SELECT result set should use the FOR UPDATE OF subclause of the SELECT in order to place a U-lock at the time of the SELECT. When they perform the actual UPDATE command, they then use the WHERE CURRENT OF subclause specifying the cursor that holds the result set of the SELECT. When transactions follow this protocol, they can avoid deadlocks between like transactions competing for locks on the same tables. A secondary transaction that accesses the same page as another will have to wait until the first finishes before it can place its own U-lock, since only one U-lock may exist on a page at one time.

The use of these two guidelines can greatly reduce the possibility of transactions terminating due to true deadlock situations. Depending on the processing time of the transactions, however, there may be time-outs due to waits for locks to be released by their owners.

Release Locks isolation level

Locking strategies which are "pessimistic" prevent the situation from occurring where lost updates or uncommitted dependency problems could arise. For example, as explained above, an S-lock is placed on a data base row (in actuality the data base

page) as soon as a transaction reads the data. At this point, SQLBase will not allow any other transaction to update the row so that lost updates are guaranteed not to occur. However, even though multiple transactions read the same data base row/page, only one transaction may actually *choose* to update the row. This transaction is prevented from doing so because one of the other transactions *might* update the row.

SQLBase offers an advanced "optimistic" isolation level which most multi-user applications will choose to decrease the number of SQL timeout and deadlock errors. SQLBase offers the isolation level Release Locks, which allows conditions to develop where lost updates and uncommitted dependency problems *might occur* but provides the programmer with features to detect and avoid these problems when they actually occur. Each data base row in SQLBase is assigned a unique *ROWID* when the row is initially inserted and this field is changed every time the row is updated and committed. The programmer can track the value of ROWID and determine when a row has actually been deleted or updated by another user.

Warning: Using Release Locks (RL) isolation level with programs which perform database updating requires each program using the RL feature to conform to the same method of detecting and handling cross transaction updates, such as the technique shown here. Failure of all programs to conform will likely result in lost updates and uncommitted dependency problems that will proliferate throughout the database.

For example, with the Release Locks isolation level, the programmer would issue the following command to select a row from a customer table:

```
SELECT ROWID, COL1, COL2,...COLN FROM CUSTOMER
INTO:SavedRowID, :IntoVar1, :IntoVar2, ... :IntoVarN
WHERE CUST_NAME = 'SMITH AND SONS, INC.'
```
The following SQL update command would be used when the programmer needed to update this row:

```
  UPDATE CUSTOMER SET COL1 = '<value>' ...
  WHERE ROWID = :SavedRowID
```
If another user had actually updated or deleted this row since this program selected it, the ROWID would change such that the SQL update command would fail and the data base server would return the -806 SQL error code. The programmer can then test for the -806 error and take appropriate action (such as advising the user that some other user has either updated or deleted this customer since they read it, and ask if the user wants the program to re-read the customer record in order to get a fresh copy).

Read-Only isolation level

The Read-Only (RO) isolation level in SQLBase is a special isolation level that is available for use only to transactions that perform no updates to the database. This mean no INSERT, UPDATE, or DELETE DML statements are allowed. The RO level achieves data consistency without setting any S-locks through a technique called

multiversion concurrency control. This allows any transaction running in RO to read all of its data as it existed at the time of the SELECT statement execution, without having S-locks placed on the result set of the SELECT.

The way SQLBase handles this is by maintaining a *history file* of database page images. This history file contains the "before-update" image of database pages for a certain timeframe. Each of these images is a *version* of the database page at some prior point in time. When an RO transaction retrieves a page from the database, SQLBase compares the timestamp of the last page update with the timestamp of the SELECT statement being processed. If the database page has the more recent timestamp, then SQLBase goes to the history file to retrieve a version of the page as it existed prior to the update, and passes the rows from this page back to the transaction's program.

The Read-Only isolation level offers the highest possible levels of concurrency and data consistency for read only transactions, but there is a significant overhead cost associated with the ongoing work required to maintain the history file and the multiple versions of database pages contained therein. This overhead can be as high as 30% for databases supporting OLTP type applications, with many users executing rapid transactions. The availability of the RO isolation level is controlled by the DBA through settings in each database's control page, as well as globally through a server flag. These settings are probably best turned off (RO disabled, which is the default setting) unless a significant need for this high level of data consistency exists, in which case they may be turned on only for those periods of time when RO transactions are known to be running.

Selecting isolation levels

Selecting isolation levels is an activity that should be performed by a person knowledgable of the overall application characteristics. The selection of an appropriate isolation level requires knowledge of the overall application design and usage patterns that may be beyond the scope of every project team member's knowledge. Furthermore, the use of isolation levels should be standardized across the entire application since the choice of one transaction's program can affect the other transactions with which it will be interleaved.

Comparing isolation levels

When designing multi-user database applications, two conflicting factors must be considered before choosing an isolation level:

- **Data Consistency**. Data should be consistent when accessed by multiple users.

- **User Concurrency**. Multiple users should be able to access the same data concurrently.

The highest degree of data consistency occurs at the expense of user concurrency. For example, one locking protocol may ensure that lost updates or uncommitted dependency problems will not occur. However, this may result in many timeout and deadlock errors. A SQL timeout error occurs when a given SQL command waits a program-defined number of seconds (the default is 360 seconds, which is 6 minutes) for a lock to be released by another program so that it may access the data base page. A deadlock occurs when user A is waiting for row 1 which user B has, and user B is waiting for row 2 which user A has. In this situation, SQLBase will sense this deadlock and abort one of the requests thereby allowing one of the users to continue.
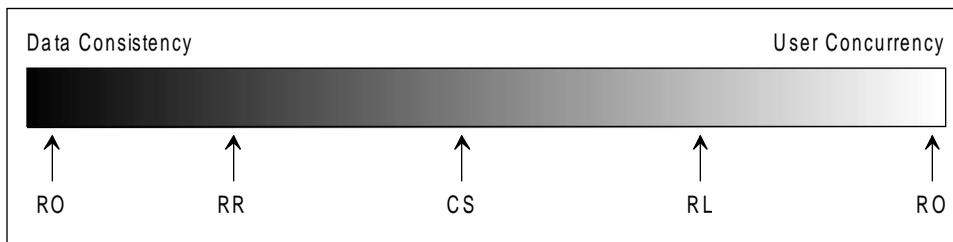
Figure 13C - An ordered approximation of isolation levels along the data consistency vs. user concurrency continuum. Note that Read Only (RO) occupies both ends of the spectrum only because it disallows any update operations and thereby avoids X-locks completely.

Criteria for choosing an isolation level

As we have seen, SQLBase provides several isolation levels. The choice of an isolation level is determined by evaluating the following criteria:

- Buffers: How critical is performance for this application? If performance is critical, and meeting performance expectations is going to be difficult, choose an isolation level that sends one buffer of rows at a time to the client (meaning it fills the entire input message buffer before sending any data). Avoid using the Cursor Stability (CS) isolation level, which sends one row at a time.

- Contention: Will there be a great deal of user contention, where many users will be attempting to access the same data simultaneously? If so, do not use an isolation level that places S-locks on all read data until a commit/rollback.

- Update: Will the users be performing inserts, updates, and/or deletes? If so, the isolation level must allow these actions.

- Consistency: Is a consistent view of data across the application important? If transactions must know about updates the moment they are made by other transactions, or must guarantee that data they have read will maintain the

same values until the transaction ends, then select an isolation level that holds read locks until commit or end.

The following table may be used to help determine which isolation level is appropriate for a specific database application based on responses to the previous criteria description.

Isolatio nLevel	Locks	Data Consistency	Criteria 1 Buffers	Criteria 2 Contentio n	Criteria 3 Update	Criteria 4 Consistenc y
RR	Locks maintained for duration of transaction, more than one page may be locked.	High data consistency, low user concurrency.	YES	NO	YES	YES
CS	S-locks held only for rows on current page.	Medium data consistency, medium user concurrency.	NO	YES	YES	NO
RO	No locks, no inserts, updates, or deletes allowed.	High data consistency, highest user concurrency.	YES	YES	NO	YES
RL	No held S-locks, S-locks released at end-of-fetch, X-locks held until commit/rollback.	Low data consistency, high user concurrency.	YES	YES	YES	NO

Common isolation level scenarios

Finally, there are some common rules-of-thumb that can be followed in the great majority of cases when selecting an isolation level for a transaction. While these represent common practice by SQLBase users, they do not substitute for knowledge of the lock process and how the various isolation levels work. These are simply a set of heuristic rules that can speed up the decision process when the transaction has no unusual characteristics that may warrant deviation from normal practice.

- For transactions which are executed in a multi-user manner throughout the day and may perform a moderate amount of update activity, the RL or CS isolation levels are the best choice. These levels provide the greatest concurrency when used in conjunction with good programming practices, such as committing frequently. The choice between these two isolation levels is often made on the following basis:

  - When the application will always be used solely with SQLBase, as opposed to another RDBMS, then the RL isolation level should be used.

- If there is a possibility that the application may be ported to another RDBMS, then the CS isolation level should be used.

  These choices are based on the fact that few RDBMSs provide such an optimistic locking protocol as RL (which makes the assumption that many rows read by a transaction will not need to be updated), but the CS isolation level is a standard feature in most RDBMSs and will usually be available. The CS protocol is the most commonly used for transaction processing systems utilizing IBM's DB2 mainframe RDBMS system.

  Examples of transactions falling into this category are order entry, confirm shipment, and receive inventory transactions in an order processing system.

- For update transactions that require their entire view of the data to be consistent throughout their execution, the RR isolation level should be used. Because of the high number of locks that may be held, these transactions should be scheduled to execute at times when the system is relatively idle, otherwise a large number of time-outs and deadlocks may result. These transactions may also include rather elaborate commit and restart logic in order to free up as many locks as possible that are not needed.

  Examples of these types of transactions, (again in an order processing application), would be processing physical inventory count, purging aged orders, or applying updated product prices.

- Transactions which meet the preceding definition of requiring a consistent view of their data throughout their definition, may also be candidates for the use of the RO isolation level, provided they do not perform any database modifications whatsoever. Since the RO mode sets no locks, the concern of causing time-outs and deadlocks is eliminated, and the transaction could be scheduled at any time regardless of system workload considerations. The downside, however, is that SQLBase must perform the work of maintaining history files for the storage of the prior versions of database rows, causing both processing overhead and more disk utilization. Of course, this drawback is not as significant if update transactions run rarely, or not at all, since the need to build rows in the history file would occur infrequently.

  An example of a transaction in this category would be a year-to-date sales report for an order processing system.

Transactions which do not easily fit into any of the preceding categories should be evaluated carefully as unique cases. Consideration should be given to the transaction's usage frequency, including transaction volume and times, the program's commit and restart logic, and the characteristics of the workload that will be executing concurrently. This analysis should yield an appropriate isolation level for any transaction. If problems are encountered when the transaction begins normal operations, the isolation level may need to be reconsidered.

Remember, when going from a restrictive isolation mode to a more relaxed mode (such as from RR to RL), there may be programming modifications required to make the transaction behave properly in the new isolation level. An example is checking ROWIDs for concurrent updates by transactions that are executing in RL mode. Application designers should be cautious when migrating to more relaxed isolation levels to ensure that data consistency is not sacrificed to the point where the application can no longer be considered reliable by its users.

Chapter 14

Logging and Recovery

In this chapter we:

- Review the various types of failures that the SQLBase server is susceptible to.

- Examine the way SQLBase uses the log files to minimize the possibility of database damage occurring during a failure.

- Evaluate the backup and recovery techniques that you can implement in order for the SQLBase safeguards to function effectively.

Introduction

There are many kinds of failures to which computer systems are susceptible. In order to protect the database against the effects of these failures, SQLBase contains a Recovery Manager (RM) software component which is responsible for ensuring that the database is always in a consistent and valid state. This consistent state is defined simply as having *all* of the effects of committed transactions applied and *none* of the effects of uncommitted transactions applied. The RM achieves its goals primarily through the use of a log file. This file contains the physical images, or history, of database changes made by transactions over time. For this reason, the log file used by SQLBase is a *physical log file*, hereafter called the log file. Since the RM component of the database server provides the recovery method for all database transactions, regardless of their program characteristics, SQLBase is said to contain *centralized recovery*.

This chapter will first discuss the broad types of failures which SQLBase protects against, then the mechanisms used to provide this protection.

Failure types

There are many possible failures that a computer system is susceptible to. Some of these failures are caused by faulty programs that place erroneous data into the database (but which commit nonetheless), or the entry of flawed data into the system by application users who are unaware of the damage they are doing. Unfortunately, these are examples of errors which cannot be recovered automatically, since any DBMS must trust that the execution of program requests which are semantically correct will place correct results in the database. As will be seen, though, some of these types of errors can be recovered from by using the same techniques as are applicable for media failures. The three basic types of system failures which the RM provides centralized recovery from are transaction failures, system failures, and media failures.

Transaction failures

Transaction failures are aborts of transactions that result in a performing rollback for the transaction. The transaction may abort at its own request (by issuing the ROLLBACK command), or due to some other cause, such as a client workstation failure or a network failure. Transaction aborts may also be issued by the SQLBase server in some situations, such as to break a deadlock situation between two transactions, one of which must be aborted in order for the other to proceed. Also, a transaction failure may affect one transaction only, or some number of transactions at once.

SQLBase provides *transaction integrity*, which prevents these types of failures from corrupting the database. The mechanism used to accomplish this is by *undo* processing from the log file. Following the undo, all database updates performed by the transaction are no longer present.

System failures

System failures cause loss of the server's volatile storage (RAM memory). The term "system" as applied to this type of failure refers to the program processing environment that is created by the operating system (OS/2 or other SQLBase-supported environment) and consists of the resources and control blocks that allow for task ownership and dispatching functions. These failures may be due to causes such as an abnormal end (abend) in the operating system environment (such as NetWare), an operator inadvertently cancelling the server program, or power failures, among others. The net effect of these types of failures is that either the SQLBase server task suddenly disappears, or it discovers that it can no longer continue processing and must request an abend. When the SQLBase server is shutdown normally (when no users are connected to the database), or a database is de-installed from the server through the DEINSTALL command, the Database Control Block (DCB) page for the database is flagged to indicate a normal shutdown took place. This is called a "graceful shutdown".

SQLBase provides for *system integrity* through the implementation of a restart feature which prevents these failures from affecting the database. Restart is invoked every time a database is first connected to after the server is started, where it examines the DCB page of the database to determine what led to the prior shutdown. If it finds that the flag in the DCB indicates no recovery is required, then it knows that the system had been terminated gracefully on its last execution. In this case, restart can be bypassed and the connection established immediately. If the flag in the DCB indicates that recovery is required, then restart must perform its processing prior to allowing the connection to the database to be established. Because of the dependency of the restart logic on the contents of checkpoint records, this process is explained in detail in the checkpoint section later in this chapter.

Media failures

Media failures cause the loss of some portion of the server's non-volatile storage (disk drive). These failures may be caused by environmental effects on the disk storage environment or through defects in the storage device. Also, magnetic media are inherently subject to a limited life because eventually the bonding material holding the oxide on the platter (in the case of a disk) or the film (in the case of tape) will fail and the oxide will begin to wear away. When this happens, minor errors will begin to occur. If these errors are observed, the data may still be sufficiently readable to be copied successfully to another device. Otherwise, the oxide will eventually degrade to the point that the device will become unusable. This generally occurs to one isolated

storage device at a time. Multiple disk drives rarely fail together, unless some drastic environmental problem occurs, such as a power surge or shock damage. These potential disasters are the justification for developing and maintaining a disaster recovery plan, which involves backing critical data up onto tape or disks that may be stored off-site from the computer hardware (which is highly recommended for critical computer installations, but is beyond the scope of this chapter).

Sites can establish procedures using SQLBase's backup and recovery features which allow the impact of media failures on the database to be minimized. Through periodic backup of the database and log files the capability is provided to restore to any particular point in time that is desired. In order to recover to a specific point in time, the required backup is simply restored. Any log images that need to be processed to that restored database in order to reach the desired point in time are then processed through a SQLTalk command called ROLLFORWARD. The most important characteristic of this process is that the application of redo log images in the rollforward is an "all or nothing" choice. There is no option to bypass either some transactions on a log file, or some log files, or skip over a particular time gap. For whatever point in time is chosen to recover to, all log images prior to that point in time must be applied. SQLBase will not allow you to attempt to do otherwise, since this would probably cause severe database corruption which would most likely be irreparable.

Recovery mechanisms

The SQLBase recovery manager relies primarily on two mechanisms for accomplishing its centralized recovery objectives. The first of these is the log file, which provides the information required for undo, redo, and checkpoint processing. The other is the set of utilities which provide for backup and recovery of databases.

The log file

As mentioned in the introduction, the log file contains the physical images, or history, of database changes made by transactions over time, and is therefore called a *physical log file*. Another important characteristic of the log file is that it exists on *stable storage*, as opposed to being stored in volatile storage. Stable storage is an area that retains memory across system failures, typically a disk drive. Volatile storage is the RAM memory used by the running system, which is subject to loss of content due to a variety of types of failures. Since the log resides on stable storage, it becomes a dependable repository of historical information about the contents of the database and the way these have been altered over time by the execution of transactions. This

information is the key to the Recovery Manager's ability to perform automatic recovery from a variety of failure types.
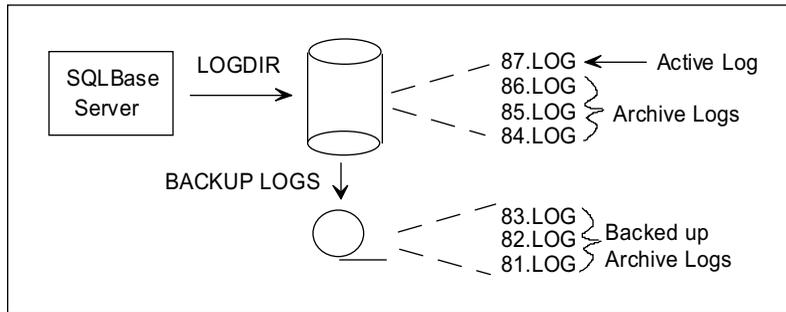
Figure 14A - There are two basic types of logs, active and archive. Archive logs may be moved to tape or other media through use of the BACKUP LOGS SQLTalk command when LOGBACKUP is on.

The log files used by SQLBase can be divided into two subtypes, the active and archive logs. Together, these two log types make up the complete database log file. The active log is the log that is currently being written to by the server; this is a disk file whose status is "open". For this reason, the active log is pinned to the disk and can only be processed by the server. Only one active log can exist at any single point in time. Whenever the server is shutdown gracefully, or a database is de-installed, all "after images" on the log that have not yet been externalized to the database are written (they are contained in dirty cache buffers), and the active log becomes an archive log. When the server restarts, or the database is re-installed, a new active log file is allocated.

There are usually many archive logs, however. These are simply old active logs that have filled up and are no longer needed by the server. When LOGBACKUP is enabled, these logs should be backed up to some stable storage media that is separate from where active log and database files are stored in order to safeguard their integrity. Of course, prior to this backup, some of the archive logs will still exist as files in the same directory or DBAREA as the active log, while waiting for the next log backup.

Note that log file names are formed by using a sequential number for the first part of the name, allowing for log files from 1 to 99,999,999 to exist. Since this number is reset to 1 whenever the database is reorganized, there is little likelihood of it ever rolling over. Appended to this number is an extension type of LOG, which identifies the file as a log file.

Log files are stored in a subdirectory of the LOGDIR directory that is named the same as the database which owns the log file. This convention allows for identification of

the owning database only so long as the log file is in a subdirectory that applies only to that database. Merging the log files of multiple databases into one subdirectory will eliminate the ability to ascertain which log files belong to which databases. If these log files are subsequently needed to ROLLFORWARD, neither the DBA nor SQLBase will be able to ascertain which log files to make available to the rollforward process. For this reason, DBAs should take care not to allow the log files of multiple database to become merged in a single directory.

Pinned log files

Some archive logs may also be pinned to the disk if they may still be needed by the server for transaction recovery or restart purposes. The reasons a log may be pinned to disk are:

- The log is active, which means the server is currently writing to it, as explained previously.

- If there are records on it from uncommitted transactions, then it may be needed for later recovery. Should one of these transactions issue a ROLLBACK, then the before row images stored on this log would be required for the undo operation.

- If it contains a system checkpoint that is either the most recent or the next to most recent checkpoint, then it may be needed for later restart.

- The log has not yet been backed up, and LOGBACKUP is in effect.
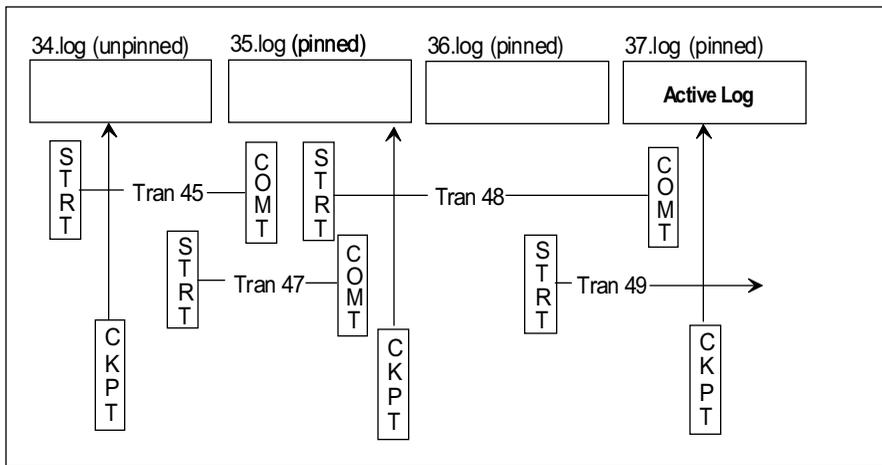
Figure 14B - The active log, 37.log in this case, is always pinned. The prior log, 36.log, is pinned because it contains log images for active task 49. The 35.log is pinned because it contains a checkpoint record that could be needed for restart. Note that 36.log would be pinned even without transaction 49, due to 35.log being pinned.

SQLBase periodically reevaluates the status of all pinned log files and deletes those that no longer meet one of these criteria. Log file status evaluations are performed whenever one of the following events occurs:

- At log rollover, usually when the current active log fills up and a new log must be created. This action can also be forced to occur even when the current log is not full through execution of the SQLTalk RELEASE LOG command. Note that some SQLBase commands issue a RELEASE LOG implicitly, such as the BACKUP SNAPSHOT command.

- When log files are backed up through the SQLTalk BACKUP LOG command, or the equivalent API call.

- When the system's "next log to backup" parameter is reset manually through the use of the SQLTalk SET NEXTLOG command, or the equivalent API call.

- When logging for the database stops because either the server was shutdown or the SQLTalk SET RECOVERY OFF command is executed.

These are the only events that cause SQLBase to consider deleting pinned log files. Performing an evaluation of the logs' pinned status whenever a transaction performs a commit or rollback would cause excessive overhead and slow down transactions, and is therefore not done by SQLBase.

Record types

The SQLBase logging manager is an integral part of the SQLBase DBMS. Whenever a database program executes, it creates before and after images for those portions of database rows that are modified in some manner. These modifications are performed through execution of the INSERT, DELETE, or UPDATE DML commands. Other commands which update the system tables in the database, including DDL and DCL commands, also cause the before and after row images of the catalog tables they update to be logged. Also written to the log at the appropriate times are control records indicating when the program connected to the database (start record), and when it issued a COMMIT or ROLLBACK command (commit and abort records). These log records allow SQLBase to maintain transaction integrity and perform point in time recovery operations. In addition, SQLBase writes a number of system control

records to the log file that allow it to maintain the integrity of the page allocation mechanism and enhance the ability to perform system restarts.
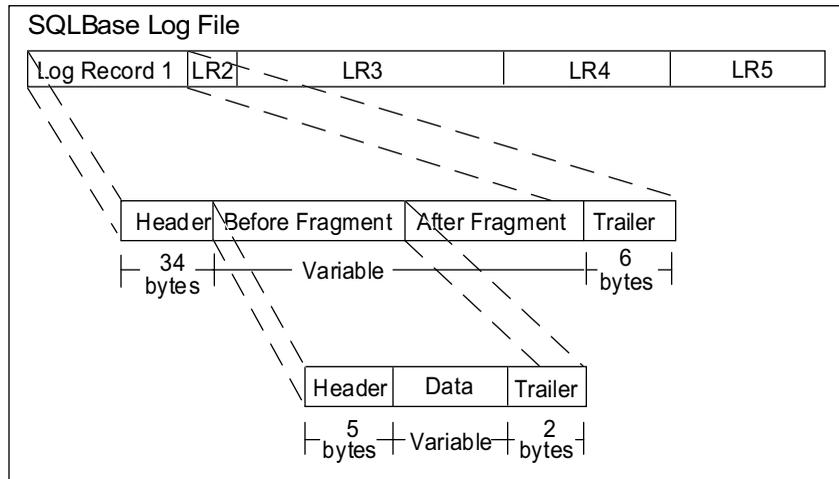
Figure 14C - The composition of the log file. Each log file contains many variable length records such as LR1 through LR5, above. Each record may be one of several types. An expansion of the image fragments for the data page log record type is shown.

As shown in figure 14C, each log file physically contains many variable length records. Each record consists of a header, a variable portion, and a trailer. The log record header is 34 bytes in length, and contains the following fields:

- Type: the format type of the log record, such as those explained below.

- Length: the total length of the log record.

- Timestamp: the point in time that the log record was written.

- Transaction: the ID of the transaction which caused the log record to be written.

- Chaining information: a pointer connection two related log records.

Each of log record is of one particular type, such as:

- Start transaction

    This log record is written whenever a program first connects to the database, thereby starting a transaction. There is no variable portion to the start transaction log record type.

- Commit transaction

  This log record is written whenever a program issues a COMMIT command. There is no variable portion to the commit transaction log record type.

- Abort transaction

  This log record is written whenever a program issues a ROLLBACK command, following completion of the undo operation. There is no variable portion to the abort transaction log record type.

- Data page modification

  Any change made to a table's data rows will cause this record to be written. Note that only those columns that actually change in the row will be logged. These changes are made through the execution of the DML commands INSERT, UPDATE, and DELETE (an exception to this case is system catalog tables, which have changes made to them through the execution of DDL commands). The variable portion of this record type has two major components:

  - Before fragment

    The before fragment is written whenever a previously existing row is modified through execution of the UPDATE or DELETE DML commands. The fragment has a 5 byte header, followed by the before images of the changed row columns, and ends with a 2 byte trailer.

  - After fragment

    The after fragment is written whenever a DML command such as INSERT or UPDATE causes changed data to remain on a database page. This fragment has a 5 byte header, followed by the after images of the changed row columns, and ends with a 2 byte trailer.

- Index page modification

  Whenever a program issues a DML command that INSERTs or DELETEs a row a table that has an index associated with it, or UPDATEs columns which are the subjects of an index, then some change will be required to be made to the index structure. These changes are logged as index page modifications, with a format similar to the database page change log record type. Each time a node of the index tree is modified, an index page modification log record is written which contains the before and after fragments of the changed portion of the index node. Note that CLUSTERED HASHED indexes are not subject to this logging activity, since they have no index tree structure.

- Checkpoint

    A checkpoint log record is written during each system checkpoint. The variable portion of the record contains a 22 byte entry for each active transaction connected to the database at the time the checkpoint was processed.

- Page allocation

    This log record type is written whenever pages are allocated from the database. There are usually three of these log records written for each page allocation action, the largest of which contains the after image of the allocated page's header and footer, which is about 60 bytes long. The other two log records record the changes made to other pages which are chained to the newly allocated page, each of these log records contains about 8 bytes of variable information.

- Long VARCHAR object

    These log records are written whenever a column whose type is LONG VARCHAR is inserted, modified, or deleted. The log record contains one page that has been allocated to store the LONG VARCHAR column in the variable portion of the log record. If the column is longer than one page, then multiple log records will be written, each containing one of the pages allocated for that column.

Undo and redo logic

The system functions that read the log and apply images from it are undo and redo. The *undo* process (also called *rollback*) is used when before images are to be restored to the database, while the *redo* (also called *rollforward*) performs the restoration of after images to the database.

The mechanism used to accomplish the restoration of before images to the database for failed transactions is by *undo* processing from the log file. This process starts at the end of the active log and reads backward from that point. As the log is read, each before image of an update is restored to the database. This continues until the start control record of the transaction is reached, at which time the abort termination record can be written at the end of the log file, and the transaction terminated. The transaction's locks are held until transaction termination to prevent other transactions from accessing the rows that are being restored to the database; these locks are finally released when the abort record is written to the log. The strict serialization that is forced upon transactions through the use of database locks assures the integrity of the undo process.

The mechanism used to accomplish the restoration of after images to the database when the updates of committed transactions must be reapplied is through *redo* processing. This process involves reading forward through the log. As the log is read,

each after image of a database change is checked against the database to ensure that it was successfully written, and if not, then the redo process writes it to the database. If the end of the log is encountered, any active transactions that remain (those for which no termination record has been found) are the subject of undo processing, as if they had been the subjects of a transaction failure.

One important characteristic of the undo and redo processes in SQLBase is that they must be capable of recovering the database to its committed state regardless of the status of updated pages in cache. Cache is the volatile system memory used as a buffer area for database pages. A large quantity of cache memory can greatly accelerate the performance of SQLBase by decreasing the number of read and write operations that must wait for external disk accesses to complete. The maintenance of the cache buffers is the responsibility of the Cache Manager (CM), which performs fetch operations to return the contents of cache to SQLBase, and flush operations to dispose of the contents of cache pages. In SQLBase, system efficiency is enhanced by allowing the CM to make the majority of decisions regarding the timing of cache operations, such as when to flush pages and when to write "dirty" pages to disk ("dirty" cache pages are those pages which have been updated in the buffer but not yet written to disk). In order for the undo and redo processes to work successfully in this environment, SQLBase must follow two rules regarding the interplay between the CM and external storage:

- Undo rule (also called the *write ahead log protocol*). This rule states that before an uncommitted value may be written to the external database (DASD), its prior value must be saved in stable storage (the log). This rule ensures that no previously committed row can be overwritten by an uncommitted row without its contents being saved in stable storage for possible undo processing. This means that the log record containing a before image of an updated row must be written prior to the CM's flushing of the dirty cache buffer containing the updated page. The CM enforces this rule by placing the log record pointer within the dirty cache buffer, and not allowing flushes of cache buffers whose associated log records have not yet been written.

- Redo rule. When processing a transaction's commit, all write operations must be completed to stable storage. This can consist of either the log file or the external database. This rule ensures that committed values are present in some form of stable storage, and not just within the contents of dirty cache buffers. SQLBase flushes log buffers at commit time to ensure compliance with this rule.

Checkpointing and restart

One problem faced during system restart is how to determine what log records need to be processed out of the voluminous contents of the total log file. The entire log file cannot be scanned and processed due to the enormous amount of time this would take. Simply examining the current log file also is inadequate, since this may take quite a while if the file is large, and some uncommitted transactions may have been active on prior log files. Because fast restarts are an important component of overall system availability time, SQLBase employs a strategy called *fuzzy checkpointing* in order to speed this process up.

In general, the activity of checkpointing is the writing of certain control information to the log on a periodic basis during normal processing in order to speed up the restart process. This information is summarized from the DBMS's internal control blocks and consists of lists of transactions and the states that they are currently in, so that the restart process can rapidly ascertain what processes need to be performed, and for which transactions. These lists, organized around the possible states of a transaction (such as active, committed, or aborted), are organized chronologically and also contain markers indicating when previous system checkpoints were taken.

When a checkpoint is actually being created, the DBMS must reach some consistent state in order for the externalized control blocks to be stable as well. During the time required to reach this consistent state, the DBMS must block the work (or suspend processing) of all active transactions, otherwise the point of time of the consistent state would always be "after this next operation" and would never actually be reached. The amount of time that transactions are blocked while this process happens depends on the amount of consistency that the checkpoint activity needs to achieve prior to writing a checkpoint record. Whatever inconsistencies the checkpoint process allows to remain must be resolved by the restart module whenever it is required to run. This is a classic trade-off situation, with higher levels of consistency taking longer to achieve at checkpoint time but yielding greater benefits in terms of simplified restart processing. Of course, since the DBMS spends much more time performing routine processing than restarting (hopefully!), getting carried away with too much consistency during checkpointing has the penalty of causing transaction delays during the creation of each checkpoint record for no subsequent benefit whatsoever.

The fuzzy checkpointing scheme used by SQLBase requires a minimum amount of system consistency to be achieved, and therefore only blocks active transactions for a very brief interval when writing the checkpoint record. The benefits during restart are still substantial, though, since most of the time only the log file records between two checkpoints and the end of the log must be traversed in order to complete the restart. These two checkpoints are the last and the second to the last (or penultimate) checkpoint in the log. The log files containing these two checkpoints will always be

pinned to disk to ensure their availability. The following tasks are performed by SQLBase when creating a checkpoint:

- Write to the log a list of transactions which are currently active, or processing.

- Flush all dirty cache buffers which have not been written since before the prior checkpoint. If the server has been busy, many of these cache buffers will have been written during the time interval since the last checkpoint because the server needs to re-use cache pages based on the least recently used (LRU) algorithm. If this is the case, then few cache pages should require flushing at this checkpoint, allowing the checkpoint to be processed faster. Alternatively, if the server has been largely inactive since the last checkpoint, there may be numerous pages containing data updated prior to that checkpoint that must now be written to disk. If this is the case, however, the server will likely still be relatively inactive and therefore the work associated with writing these buffers should not have an significant impact on current transactions. The use of this fuzzy checkpointing feature allows SQLBase to perform checkpoint processing with a low likelihood of negatively affecting performance.

When SQLBase starts, and discovers that a restart is required, the following steps are performed:

1. The next to the latest checkpoint record in the active log is located. This is called the penultimate checkpoint record.

2. Redo processes all transactions following the penultimate checkpoint marker.

3. During the redo processing, the last checkpoint record in the log will be encountered. This checkpoint contains a list of all active transactions at that point in time. Redo uses this to track all transactions from this point on in the log.

4. When a transaction ends during redo processing, it is removed from the list of active transactions.

5. Eventually, redo will encounter the end of the log file.

6. Undo now reads the log backwards from the end to process all entries in the active transaction list. Since these transactions had not been committed prior to the end of the log, they must be rolled back.

7. When undo encounters the start control record of the last transaction entry in the active list, recover is complete and the database is available for new connections.

This process ensures that the database is returned to a consistent state, both in terms of containing data only from committed transactions and not losing the contents of any write-delayed cache buffers.

Backup and recovery

SQLBase's backup and recovery features allow sites to establish procedures which minimize the impact of media failures on the database. This is made possible by "point in time" recovery, which allows the database to be restored to any prior point in time, thereby eliminating any updates or failure events that occurred between that selected point in time and the present. This also allows recovery from database corruption caused by program logic errors, bad data, or faulty recovery processes. By periodically backing up the database and log files and saving them, the backup and recovery capability is provided to restore to any particular point in time that is desired, so long as two conditions are met:

- There must be an existing database backup that was created either at, or prior to, the point in time where the database will be restored to.

- There must be a continual series of log files (either backed up or still on disk) from the database backup point in time up until the point in time being recovered to.

The tools necessary to perform these tasks are provided with SQLBase. Other backup programs and devices may be substituted if higher capacity or quicker backup is desired through the use of products such as high capacity tape devices.

Since the recovery process requires identification and coordination of the required backup and log files, sites should implement some form of labeling and tracking procedures for whatever media they decide to store these files on. The recovery process depends on reliable and rapid identification of these files in order to be completed in a minimum amount of time. Also, the frequency of these backups and log off-loads plays a key part in the site's recovery procedures.

Considerations for partitioned databases

There are special considerations that must be allowed for when performing backup and recovery operations on partitioned databases. This is because the extent allocation information for a partitioned database resides in the SQLBase server's MAIN database. Without the MAIN database, the server will be unable to access the information in the partitioned database. Therefore, the MAIN database must always be kept in synchronization with the partitioned database it controls. Whenever activity against a partitioned database is controlled by SQLBase, this level of synchronization is guaranteed. Only when using utilities external to the SQLBase software must the DBA be concerned about ensuring that the state of the MAIN database and the site's partitioned database reflect the same point in time.

Since partitioning is usually performed only for large databases, off-line backup techniques are often appropriate. The SQLBase on-line backup utilities may be used with a partitioned database, but either adequate disk space will have to be available

for creation of the backup image file or the operating system has to provide a technique to redirect the output directly to a tape device.

When using off-line backup techniques to backup partitioned databases, extra care must be taken to insure that the MAIN database remains synchronized with the partitioned database. This means that during backup, the server must be shut down, and the partitioned database and the MAIN database must both be backed up prior to restarting the server. When performing a restore made through this scenario, the server must be shut down, and both the partitioned and MAIN databases restored to the matching set of backups. Failure to maintain this parallelism between off-line backup and restore techniques will cause SQLBase to be unable to determine the proper location of the database suballocations within the DBAREAs assigned to the partitioned database, which will result in unrecoverable database corruption.

When database backup and recovery procedures are implemented using the SQLTalk BACKUP, RESTORE, and ROLLFORWARD commands, SQLBase modifies the contents of the MAIN database as necessary to ensure that they accurately reflect the current state of the suballocations within the partitioned database's DBAREAs. Therefore, the DBA can be unconcerned about explicitly managing the backup and recovery of the MAIN database in this situation. Of course, backing up the MAIN database might still be considered as a part of a disaster recovery plan that allows a site to restore server operations on another machine that does not have an initial installation of SQLBase software.

Chapter 15

Introduction to Query Optimization

In this chapter we:

- Follow the evolution of data access languages from their procedural roots to their modern declarative implementation represented by the SQL language used by SQLBase.

- Examine the various categories of query optimizers that exist in relational database management systems.

Data access languages

Before we can address query optimization, it is necessary to review data access languages. You use a *data access language* to store data into and retrieve data from a database in a database system. Two fundamental different types of data access languages exist today: procedural (or navigational) and declarative.

Procedural data access languages

Before the introduction of SQL database systems, most database systems (for example, IDMS, IMS and dBase) were based on procedural, or navigational, data access languages. Procedural data access languages require the programmer to code the programming logic necessary to navigate the physical data structures to identify and access the required data. For example, with IDMS, programmers must write navigational logic to *FIND CALC* a specific record and *OBTAIN NEXT* within specific sets. Similarly, in dBase, programmers *USE* a specific index.

By referring to the physical data structures in the application programs, the application programs become dependent upon them. Application programs then require updating if changes occur to the physical structures. For example, if an index is deleted in dBase, all application programs that use that index must be modified.

This dependency between the application programs and the physical data structures significantly increases the cost of application development and maintenance. During the development of large complex computer systems, it is quite common to discover inadequacies in the physical database design during the programming phase of the project. To remedy these inadequacies, the database administrator must implement changes to the physical structures.

These *schema changes* in turn impact all existing programs which reference the altered physical structures. These programs must be updated to reflect the new schema. If the program relied on some altered physical structure to navigate the database and other programming logic was based on this navigation, a significant amount of re-coding may be required. Likewise, maintenance of existing systems often results in schema changes and therefore this dependency also drives up the cost of maintaining an application.

In addition, procedural data access languages are often proprietary. Consequently, application programs become closely bound to the particular database systems on which they were developed. This binding of application programs to particular database systems severely limits the ability to migrate these application programs to improved and lower cost database systems.

Declarative data access languages

With the introduction of SQL and relational database systems, it became possible to sever the link between application programs and the physical data structures. Declarative access languages only specify what data is needed by the application program, leaving it to the database system to determine how to navigate the physical structures to access the required data. SQL is an example of a declarative data access language.

Decoupling the application programs and the physical data structures results in several significant benefits:

- **You can respond to changing data requirements without impacting existing application programs**. For example, if existing indexes become obsolete, you are free to drop existing indexes and create new ones without impacting existing programs. The new indexes created may result in either an increase or decrease in program performance; however, you are assured that the existing programs will execute without error (assuming they PREPARE their SQL prior to execution, some RDBMSs such as SQLBase may also automatically recompile stored SQL access plans as well). This is not true with procedural data access languages that abort program execution whenever physical structures are either not found or have been altered.

- **Application program complexity is reduced.** The database system, not the programmer, determines how to navigate the physical structures. This frees the programmer for other tasks, since this aspect of programming is often the most complex aspect of the program's logic.

- **Application program error is reduced**. The complexities of data access often result in program error unless the programmer is highly skilled and extremely careful. The primary advantage of a computer is the ability to execute simple instructions at a high rate of speed and without error. Consequently, database systems, in general, out-perform programmers when determining how to navigate physical structures to access required data.

The component of SQL database systems that determines how to navigate the physical structures to access the required data is called the *query optimizer*. The navigational logic to access the required data is the *access path* and the process used by the query optimizer to determine the access path is called *query optimization*.

Query optimization

The query optimization process determines the access path for all types of SQL DML statements. However, the SQL SELECT statement (a query) presents the biggest challenge to access path selection; therefore, the process is commonly referred to as *query* optimization rather than data access optimization. Further, it is worth noting that the term query optimization is a misnomer in that there is no guarantee that the query optimization process will produce an *optimal* access path. A more appropriate term might be query improvement. For example, the best possible access path given the known costs. However, we use the standard term to conform to convention.

Early relational database systems processed queries in a simple and direct manner without attempting to optimize either the query itself or the access path. This resulted in unacceptably long query processing times, and led to the belief among some application programmers that relational database systems were not practical for real world applications. To speed query processing times, a great deal of research and testing was carried out to discover ways to increase the efficiency of query processing. Query optimization can thus be defined as the sum of all techniques that are employed to improve the efficiency of processing queries.

Syntax optimization

The first progress in optimizing queries was in finding ways to restate the query so that it produced the same result, but was more efficient to process. For example, consider the following query:

```
SELECT VENDOR_CODE, PRODUCT_CODE, PRODUCT_DESC
FROM VENDOR, PRODUCT
WHERE VENDOR.VENDOR.CODE = PRODUCT.VENDOR_CODE AND
VENDOR.VENDOR_CODE = "100"
```

The most direct way to process this query is:

1. Form the Cartesian product of the vendor and product tables.

2. Restrict the resulting table to rows that meet the WHERE condition.

3. Project the three columns named in the SELECT clause.

If the vendor table has 50 rows and the product table has 1000 rows, then forming the Cartesian product would require 50,050 rows to be read and written. Restricting the result would require 50,000 more rows to be read from the resulting product, and if 20 rows meet the WHERE condition, 20 rows would be written. Projecting the result would then require 20 additional reads and writes. Processing the query in this manner would require 100,090 rows to be read and written.

However, one of the algebraic rules that apply to SQL is that

```
(table_a JOIN table_b) WHERE restriction on table_a
```
is equivalent to:

```
(table_a WHERE restriction on table_a) JOIN table_b
```
This means that restrictions on a table can be performed as early as possible to reduce the number of rows that must be processed further. Applying this rule to the example above, we can process the query as follows:

1. Restrict the rows in the part table to those for which VENDOR_CODE = "100". This would require 1000 reads and 20 writes.

2. Perform the join of the result to the vendor table in step 1. This would require 20 reads of the intermediate result above, which presumably could be kept in memory. Joining the intermediate result with the vendor table would then require 100 additional reads, and 20 writes.

Processing the query would require only 1120 reads and 40 writes to produce an equivalent result. The transformation described in this example is called *syntax optimization*. There are many formulas for making transformations, and all modern query optimizers use transformation rules to restate a query into a more efficient form. The important point to realize is that these automatic transformations free you from having to try many equivalent forms of the same query in search of the one form that yields the best performance, because the optimizer is usually transforming all of these queries into the same final form.

Rule-based optimization

As progress was being made in improving the processing of queries, other efforts to improve table access were taking place. These improvements included indexing techniques and hashing. However, these improvements added complexity to query processing. For example, if a table had indexes on three different columns, then any one of these indexes could be used to access the table (along with sequential table access).

In addition, many new algorithms to join tables began to appear. The two most basic join algorithms are:

- *Nested Loop Join* - A row is read from the first table, called the outer table, and then each row in the second table, called the inner table, is read as a candidate for the join. Then the second row in the first table is read, and again each of the rows in the inner table is read again, and so on until all rows in the first table are read. If the first table has M rows, and the second table has N rows, then M x N rows are read.

- *Merge Join* - This join method assumes that the two tables are sorted (or indexed) so that rows are read in order of the column (or columns) on which they will be joined. This allows the join to be performed by reading rows from each table and comparing the join column values until a match is found. In this way, the join is completed with one pass through each table.

Further, join operations are both commutative and associative; consequently, it is theoretically possible to perform joins in any order. For example, all of the following statements are equivalent:

```
(table_a JOIN table_b) JOIN table_c
table_a JOIN (table_b JOIN table_c)
(table_a JOIN table_c) JOIN table_b
```

However, different access paths, join algorithms, and join orders may result in significantly different performance. Consequently, when joining several tables each with multiple indexes, there exists literally hundreds of possible combinations of join orders, join algorithms and access paths to select from. Each produce the same result but with varying performance characteristics.

The first method of dealing with this complexity was to establish heuristic rules for selecting between access paths and join methods, which is called *rule-based optimization*. With this approach, weights and preferences are assigned to the alternatives based on principles that are generally true. Using these weights and preferences, the query optimizer works through the possible execution plans until it arrives at the best plan based on its rules. Some of the rules used by this type of optimizer are based on the placement of variable tokens such as table or column names within the query's syntax structure, when these names are moved a dramatic difference in the performance of the query can sometimes be realized. For this reason, rule-base optimizers are said to be syntax sensitive, and one of the methods for tuning this type of database system involves moving tokens to different positions inside a statement.

Rule-based optimization provides satisfactory system performance in those situations when the heuristics are accurate. However, often the general rules are inaccurate. To detect these situations, the query optimizer must consider the characteristics of the data such as:

- The number of rows in the table
- The range and distribution of values for a given column
- The row width and consequently the number of rows per physical page on disk
- The height of an index
- The number of leaf pages in an index

These data characteristics can greatly affect the efficiency of processing a query. This resulted in the next type of optimization.

Cost-based optimization

Cost-based optimization is similar to rule-based optimization, except that cost-based optimizers use statistical information to select the most efficient execution plan. The cost of each alternative execution plan is estimated using statistics such as the number of rows in a table and the number and distribution of the values in a table column. The cost formulas typically consider the amount of I/O and CPU instructions required to perform the execution plan. The statistics are stored in the system catalog and maintained by the database system.

To understand how statistics can make a difference, consider the following query:

```
SELECT CUST_NBR, CUST_NAME
FROM CUSTOMER
WHERE STATE = "FL";
```
If an index exists on the STATE column, a rule-based optimizer would use this index to process the query. However, if ninety percent of the rows in the customer table have FL in the STATE column, then using the index is actually slower than simply processing the table sequentially. A cost-based optimizer, on the other hand, would detect that using the index would not be advantageous.

The cost-based optimization approach today represents the state-of-the-art in query optimization techniques. Most high quality relational database systems, including SQLBase, employ a cost-based optimizer.

Query optimization steps

While the query optimizers of modern relational database systems differ in sophistication and complexity, they all follow the same basic steps in performing query optimization:

1. *Parsing* - The optimizer first breaks the query up into its component parts, checks for syntax errors, and then converts the query to an internal representation for further processing.

2. *Conversion* - The optimizer next applies rules for converting the query into a format that is syntactically optimal.

3. *Develop Alternatives* - Once the query has undergone syntax optimization, the optimizer next develops alternatives for processing the query.

4. *Create Execution Plan* - Finally, the optimizer selects the best execution plan for the query, either by following a set of rules (rule-based optimization) or by computing the costs for each alternative (cost-based optimization).

Because steps 1 and 2 take place without regard to the actual data in the tables used, it is not necessary to repeat these steps if the query needs to be recompiled. Therefore, most optimizers will store the result of step 2 and use it again when it reoptimizes the query at a later time.

Chapter 16

Overview of the SQLBase Query Optimizer

In this chapter we:

- Review basic relational operators implemented within the SQL data access language.

- Examine the physical operations used by SQLBase to perform the relational operators.

- Learn how to view SQL statements as query trees, which simplifies the processing steps required for their execution.

Introduction

In the next three chapters we examine the query optimizer contained within SQLBase. The SQLBase query optimizer (often simply called the optimizer) is a specific implementation of a cost-based optimizer, meaning its decisions on access path selections are based on the estimates of potential costs associated with executing a particular access plan.

The basis of these cost estimates are the statistics contained within the SQLBase catalog and control areas within the database. The advantages of this approach are that the access plan decisions made by SQLBase can be made with very recent information concerning the actual database contents. Just how recent and accurate this information is depends on the procedures you follow in administering your databases.

This chapter reviews the basic relational operators implemented by SQL followed by an overview of the techniques that SQLBase may use when actually executing these operations. This set of techniques, called *physical operators,* is the set of actions the optimizer considers when building access plans.

While we summarize the actions taken to perform each physical operator, we will also briefly summarize the major factors comprising the cost of the operator (which is usually I/O). Finally, the structure of an access plan is explained, with an emphasis on the heuristics used by the optimizer when building a set of access plans on which to perform cost analysis.

Throughout the next three chapters, examples that illustrate optimizer operations generally use the SQL SELECT statement. You should keep in mind, however, that all SQL Data Manipulation Statements (DML) are processed by the optimizer. This includes not just the SELECT statement, but the INSERT, UPDATE, and DELETE statements as well.

The steps involved in optimizing these other statements are virtually identical to those of the SELECT, the only difference being operations used to perform locking and logging during update operations. These exceptions are covered in *Concurrency Control* and *Logging and Recovery* chapters. The benefits of using the SELECT statement for examples is that the SELECT offers many more relational operations, such as a join, which exercise a much wider variety of optimizer processes than other DML statements.

A secondary advantage of using SELECT statements for examples is that the execution of a SELECT statement creates a result set which visually shows the final output of SQLBase's processing. This contrasts to the other DML statements, which change the state of the database, but these changes are not visible externally except through the return of error codes or other environmental variables. Only a subsequent

SELECT statement will reveal the changes which were made to the contents of the database by the execution of one of these DML commands.

Relational operations

The basic operators of the relational algebra were defined by E. F. Codd when he published his landmark paper delineating the relational model access language in 1972. The operations defined in this relational algebra form the logical operators that must be processed by the optimizer. These logical operators are implicit in the syntax of SQL DML statements submitted to SQLBase for processing.

The following relational algebra operations have one or more relations as inputs, and one relation as output. A relation is simply a table of some degree *n*, where n is the number of attributes (or columns) in the table. These operations are broken down into two classes: traditional set operations and special relational operators. While the traditional set operations are sufficient to process most multiple relation queries, the special relational operators specify the desired results more efficiently. This last class of operators is the basis of the SQL language, and is examined in more detail throughout the remainder of this chapter. The traditional set operations are defined here to provide background information.

- Traditional set operations

    An important distinction of the traditional set operators is that (with the exception of the Cartesian product) the relations used as input to these operations must be of the same number of columns, with each matching attribute defined on the same domain. This means that each table must have the same number of columns, and that each column pair of any position (for example, column 1 of each table) must be defined with the exact same data type and scale.

    - Union

        The union of two relations is all rows belong to either or both relations. It can be formed by appending one table to the other and eliminating duplicate rows.

    - Intersection

        The intersection of two relations consists of the rows which appear in both relations. Any row which appears in only one table, but not the other, is not a member of the intersection set.

- Difference

  The difference between two relations is all rows which belong to the first
  table but not to the second. Note that this operation is not commutative
  like the other traditional set operators, meaning that $A - B \neq B - A$.

- Cartesian product

  The Cartesian product of two relations is the table resulting from
  concatenating each row of one table with every row of the other table.
  The table resulting from this operation contains as many rows as the
  product of the rows in the input tables. This means that the Cartesian
  product of a 15 row table and a 50 row table contains 750 rows. As
  mentioned previously, this is the only traditional set operation where the
  format of the two tables can differ.

- Special relational operations

  SQL generally uses these operations more than the traditional set operations:

  - Projection

    The projection operation limits the columns of a table that are referenced
    in a SQL statement.

  - *Selection*(restriction)

  - A select limits the result set to only those rows that meet the
    qualifications. The selection criteria compares one or more columns of
    the table to either one or a set of constants or expressions.

  - Join

    The join operation creates a result set by concatenating rows together
    from multiple tables. These concatenated rows must also meet the
    specified join criteria, which is a comparison between one or more
    columns of the tables involved. These comparisons differ from the select
    operation in that they compare columns of different tables.

Because of the central role played by these special relational operators in SQL
statements, we will now examine them in more detail.

Projection

When a SELECT references specific columns of one or more tables, only those columns are returned:

```
SELECT NAME, PHONE FROM CUSTOMER;
```
The alternative to performing a projection is to use the asterisk (*) wild card notation which causes all columns to be included:

```
SELECT * FROM CUSTOMER;
```
Note that many sites have standards prohibiting this notation from being used in code, since changing the number of columns in the table can have an adverse affect on the program containing the statement.

Selection

The select is the row equivalent of the column projection operation. In SQL, the WHERE clause specifies a selection by naming columns in comparison expressions with either constants, that consist of one or a set of values, or expressions that evaluate to a single value or set of values. The following are examples of the select operation:

```
SELECT * FROM CUSTOMER
WHERE NAME = 'JONES';
SELECT * FROM PRODUCT
WHERE STOCK_QTY <= REORDER_QTY;
SELECT * FROM ORDERS
WHERE (STATUS IN ('C','P','S')) AND
(TOTAL_AMT > 1000);
```
Each of the comparisons contained within the WHERE clause is called a *predicate*. Some SQL statements, such as the last example shown here, contain more than one predicate. When the operation specified in a predicate is performed on a row, it is called *applying the predicate*. If the predicate evaluates to TRUE, it is said to be *satisfied*, otherwise it is *not satisfied*. When a statement has more than one predicate, they must be connected by one of the Boolean operators *AND* or *OR*. When all of the predicates of a statement are connected by AND, they are said to be *AND-connected*, or conjunctive. When they are all connected by OR, they are *OR-connected*, or d*isjunctive*. This terminology of predicates plays an important role in how they are used by the query optimizer, as we shall see later.

Note that the form of the WHERE clause that compares columns of two different tables is not a select operation, but rather a specification for a join operation, which is covered next.

Join

A join creates its result set from two or more tables in a similar manner as the Cartesian product mentioned earlier; it concatenates each row of one table with every row of the other table. The difference between the Cartesian product and the join is that only rows meeting some specified join criteria are eligible for inclusion in the result set. This is logically equivalent to building the Cartesian product, then performing a select operation against it. However, the join allows for a more efficient operation with most query optimizers because they can eliminate rows from the result set without first forming the entire Cartesian product as an intermediate result.

To perform a join, use the SQL FROM and WHERE clauses. The FROM clause must name each input table to the join. The WHERE clause specifies the join criteria, by comparing one or more columns of one table with columns of another table. The predicates used as join criteria determine how many rows of the Cartesian product will be bypassed in processing the join. The more restrictive the join criteria are, the more efficient the join will be when compared to the Cartesian product (note that the Cartesian product is formed by not specifying any join criteria at all). Here are some examples of joins:

```
SELECT NAME, AMOUNT
FROM CUSTOMER, RECEIVABLE
WHERE CUSTOMER.CUST_NO = RECEIVABLE.CUST_NO;
SELECT NAME, QTY, DESC
FROM CUSTOMER C, ORDER O, PRODUCT P
WHERE (C.CUST_NO = O.CUST_NO)
AND (O.PROD_NO = P.PROD_NO);
```
In the second example, the 'C', 'O', and 'P' letters allow you to qualify column names with their correct tables without having to spell out the entire table name. In SQL, these are called *correlation variables.* Since most column names need to be qualified they are often used in joins due to the presence of multiple table names in the FROM clause.

Two important characteristics of the join operation are that it is both commutative and associative in nature:

```
A JOIN B = B JOIN A (commutative)
A JOIN (B JOIN C) = (A JOIN B) JOIN C (associative)
(A JOIN B) JOIN C = B JOIN (A JOIN C) (both)
```
The effect of these properties is that when processing a join of more than two tables, the optimizer may select any sequence of two table joins to complete the operation. This allows the optimizer to treat a join of any number of tables as a series of two table joins, which avoids having to have special physical operations available to join any specific number of tables.

The type of comparison performed is often used to further refine the name of the join. These sub-classifications of the join are briefly covered below.

Equijoin

The most common example of the join is the equijoin, which uses the '=' operator. This version of the join operation is used whenever business relationships are used to combine information from two tables. These joins occur by setting a primary key in the parent table equal to the foreign key in the child table.

```
SELECT C.CUST_NO, C.CUST_NAME, O.ITEM_NO, I.DESC
FROM CUST C, ORDER O, ITEM I
WHERE (C.CUST_NO = O.CUST_NO) AND
(O.ITEM_NO = I.ITEM_NO);
```

Natural join

This is a particular form of equijoin where all columns of the two tables are included, with the exception of one set of the key columns, which are duplicated across the tables. This is a full equijoin between two tables without any projection except for the elimination of the duplication of the key columns.

Semijoin

The semijoin operation is equivalent to an equijoin with a subsequent projection that only includes the columns of one of the join tables. This is useful when rows of a table are needed that meet a selection criteria that is defined in terms of membership within another table's foreign key column. An example would be the set of all products which have been ordered during January of 1993:

```
SELECT P.PROD_NO, P.PROD_DESC
FROM PRODUCT P, ORDER O
WHERE (O.PROD_NO = P.PROD_NO) AND
(O.ORD_DATE BETWEEN JAN-1-1993 AND JAN-31-1993);
```
The :equijoin is also useful in a distributed environment when the two tables to be joined are at different nodes on the network.

Outerjoin

The outerjoin allows the non-participating rows of one of the joined tables to be included in the result set. The reason these rows are termed non-participating is because they contain key values which are not referenced by any rows of the other table. For example, a row in the product set which contains a product number that has never been ordered by any customer would be a non-participating row in a join of the product table with the order table. In SQLBase, these rows could be included in the result set anyway through the specification of the outerjoin operator, '+', on the order table key as in the following example:

```
SELECT *
FROM PRODUCT P, ORDER O
WHERE P.PROD_NO = O.PROD_NO(+);
```

Selfjoin

A selfjoin is an equijoin of a table with itself. This is also called a 'recursive join'. Note that in this case, you must assign correlation variables in order to avoid ambiguous column references, as in the following example which lists the names of all employees and their assigned managers:

```
SELECT E.NAME, M.NAME
FROM EMPLOYEE E, EMPLOYEE M
WHERE E.MNGR_NO = M.EMPLOYEE_NO;
```

Aggregation

While aggregation was not originally specified as a relational operation, its inclusion as a standardized capability in SQL served to make it a commonplace operation. The purpose of aggregation is to represent a table of information through some derived statistic, such as the sum or mean (average) of a set of numbers.

SQL supports two types of aggregation, the simplest is scalar aggregation, which derives a single output from a column of a relation. Scalar aggregation in SQL is performed through the inclusion of an aggregate function in a query that does not have any GROUP BY clause. Examples of scalar aggregation include:

```
SELECT SUM(SALARY) FROM EMPLOYEE;
SELECT COUNT(*)
FROM EMPLOYEE
WHERE NAME = 'DAVE';
```
The first query produces the total salary paid to all employees, while the second query tells how many employees are named 'DAVE'. The distinguishing characteristic of these queries is the presence of some aggregate function in the SELECT list which is allowed to range over the entire result set of the table access (after performing the project and select operations).

The second type of aggregation supported by SQL is function aggregation. This type of aggregation uses the same aggregate functions as scalar aggregation. The difference between the two is that function aggregation always creates another table as output. This contrasts with scalar aggregation, which simply returns a single value as output.

The SQL syntax of a statement performing function aggregation always includes the GROUP BY clause. The columns named in the GROUP BY clause become, in effect, the key to the new table that is output from the statement. Examples of function aggregation are:

```
SELECT DEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DEPT;
SELECT @MONTH(BIRTH_DATE), COUNT(*)
```

```
FROM EMPLOYEE
GROUP BY @MONTH(BIRTH_DATE);
```
Here the first query lists the department and average salary for each department, while the second lists the birth month and number of employee birthdays in each month throughout the year. In each of these statements, the aggregation function only ranges over the rows of the input table that have equal values in the columns specified in the GROUP BY clause.

SQLBase physical operators

When SQLBase executes a query plan, it reads the plan and performs each specified step in turn. Each of these steps tells SQLBase what operation to perform at that step, and what inputs and outputs are required. When executing query plans, SQLBase uses *physical operators*. These are different than *logical operators* such as SQL statements which specify relational operations that must be performed. For every possibly logical operator, there is at least one, and possible many, physical operators that allow SQLBase to execute the operation in an efficient manner.

Every physical operator has either one or two inputs, depending on the nature of the operation. It also has one output table (which of course may consist of only a single row, or even no rows). This output table may be a result set which is the final output of the query, or it may be an intermediate table which will be used as an input table to an operation later in the query plan. One query plan can contain many of these intermediate tables. They can be stored within SQLBase's cache memory, or on disk in a temporary file. When a step in the query plan is finished executing, the temporary input tables it uses are either freed from memory or deleted from disk.

Sort

The sort operation is performed whenever a table must be placed into a specific sequence according to one or more columns, which are collectively called the *sort key*. This physical operator has one table as input, and one table as output. The basic technique SQLBase uses for sorting is the postman sort, which is a radix sort.

Aggregation

The aggregation operator also has one table as input, and one table as output. The SQLBase physical aggregation operator performs a serial scan of the qualifying data rows, with the aggregate function calculation being performed for each row. When SQLBase performs simple scalar aggregation, the input rows can be in any sequence. However, when aggregate functions are being evaluated through SQL statements containing GROUP BY clauses, the input table to the aggregation operation must be in sequence by the columns specified in the GROUP BY. This prerequisite is satisfied by the query optimizer through the placement of either a sort operator prior to the

aggregation, or through the use of a table access technique which returns table rows in a specific sequence (discussed later in this chapter).

Disk access operation

This group of physical operators is responsible for returning rows from a single table. These rows can then either be processed by other steps in the query plan, or constitute the final result set, depending on the SQL statement being processed. Disk access operators are entirely responsible for the access of data from disk, however. Therefore, one of these operators always appears as the first step of a query plan, in order to perform the access of data to whatever table the optimizer decided should be processed first.

Table scan

This is the simplest technique for accessing a base table. Each data page associated with the table is read in succession. For each page, the rows on that page are extracted for processing. Note that this extraction may require accessing additional pages in the form of either extent pages or overflow pages (in the case of hashed tables). In the absence of any extent or overflow pages, the number of disk accesses required for a table scan is equal to the number of data pages assigned to the table, including pages required for any LONG VARCHAR columns that may be specified.

Index leaf scan

This access technique is used whenever a specific sequence is desired for the rows of a table, and an index matches this sequence, but there are no predicates specified for the table that are present in the index. In other words, this access technique can substitute for a sort if the index sequence matches the desired table sequence.

The basis for this technique is the B+Tree modification to the basic B-Tree, which links together the leaf pages of an index. This list of linked leaf pages is called the *sequence set* of the index. It is traversed in key sequence, with each index node's rows accessed as the node is processed. This means that the worst case I/O count is the number of leaf pages in the index plus the number of rows in the table. This is sometimes improved on, however, by the table being either fully or partially clustered in the index sequence. We will see how the optimizer accounts for this possibility in the next chapter.

Matching index scan

The matching index scan uses the full capabilities of SQLBase's B+Tree index structures. This technique is useful whenever a SQL statement only requires a portion of a table to be processed, based on predicates that match the columns present in the index. Since the generic capabilities of B+Tree indexes are exploited by SQLBase, this access technique can use any index which has predicate columns in the leftmost key positions. The optimizer generally uses this technique if the predicates are

restrictive enough to make the extra I/Os for the index worthwhile. Also, if an inequality predicate is used for some index column, this technique uses the index tree to find the starting leaf, and then scans the sequence set forward or backward from that point.

The number of I/Os incurred by this access technique is one for each level in the index, plus one for each index node that is accessed, plus one for every row that must be retrieved. The optimizer estimates costs associated with this access technique using an index statistic called the selectivity factor, which is explained in detail in the next chapter. Briefly, the selectivity factor describes the expected number of rows in the table that would satisfy a predicate.

Hash access

A table which has a clustered hashed index placed on it may be accessed via this index whenever all of the columns contained within the index are used in equality predicates in an SQL statement. As explained in Chapter 10, *Hashed Indexes*, the hashing technique lacks any generic or scanning capabilities. This is why the hashed index key must be fully specified with the equality predicate in the SQL statement.

Whenever the restrictive conditions placed on the use of the hash access technique are met, this is probably the quickest possible alternative for table access and will likely be selected by the optimizer. The I/O cost of this technique is one I/O for each row, or set of rows, which has the key specified in the predicates. Extra I/O costs may be incurred for searching a chain of overflow pages if the hashed table has experienced a large number of collisions, and this condition may cause the optimizer to reject the hash access technique.

Join operations

The relational join is one of the most frequently executed operations of any relational database. It can also be extremely time consuming due to its data intensive nature. This is because the most simplistic implementation of a join requires every row of the one table to be compared to every row of another table. In order to execute joins as quickly and efficiently as possible, SQLBase selects from among a number of alternative techniques. All of these operators accept two tables as input, and produce a single table as output. These input and output tables can be any combination of database tables, temporary tables, or final result sets.

Terminology that is common to all join algorithms is the naming of the two tables involved. Since one row of a table must be compared to all rows of another table, the table for which a row is held is called the *outer table*. The table which is scanned while a row is held from the outer table is called the *inner table*. These terms first originated with the loop join algorithm, where they are especially appropriate, as we shall see next.

Simple loop join

This is the most straightforward method for joining two tables, and can be used for any join criteria. It can also be the worst performing join operation, especially if both input tables are large. One advantage of the simple loop join is that it can be used in any case, regardless of the type of join criteria specified or the absence of index structures. Also, there are cases where the simple loop join can be the best performing technique, particularly if one of the input tables can fit entirely into cache memory.

The loop join algorithm obtains a row from the outer table, then scans the entire inner table to find the rows which meet the join criteria. Whenever the criteria are met, the rows are appended and output. When the end of the inner table is reached, a new row is obtained from the outer table, and the scan of the inner table is started again.

Due to the nested loop structure of this technique, the worst case for the number of I/O operations is the number of data pages in the inner table multiplied by the number of data pages in the outer table. This can be improved on dramatically if the inner table is small enough to fit into cache memory, when the number of I/Os drops to the sum of the data pages in the two tables. For this reason, SQLBase always chooses the smaller of the two tables to be the inner table when cache is sufficient to hold the table, at which point this join technique may become less costly than other alternatives.

The following two alternative access methods are enhancements to the basic loop join. They could be used whenever a suitable index is present, in which case they can speed up the join process greatly over the simple loop join.

Loop join with index

This variation on the nested loop join can be used when there is an index on one of the tables that has the join columns in the high order positions of the key. When this is the case, the optimizer uses the table with the index as inner table of the loop algorithm, and uses the index to limit the search for rows matching the current row being held from the outer table. This greatly decreases the number of I/Os needed to process the inner table. The best case is when the inner table is clustered, and an equijoin is being performed, in which case the number I/Os is equal to the sum of the number of data pages in the two tables, plus the height of the index tree, plus the number of rows in the outer table. The last two cost components account for the overhead of accessing the index structure when joining row occurrences.

Loop join with hash index

This enhancement of the loop join technique can be applied when one of the tables to be joined has a hash index placed on it. The other two prerequisites for this technique are that the hash index must be placed on all of the columns of the join criteria (and no others), and the join must be an equijoin. When these conditions are met, this join technique can be very efficient. The worst case is when the number of I/Os is equal to

the number of data pages in the outer table plus the number of rows in the inner table. This only occurs if every row in the inner table is referenced by the outer table. When not all rows of the inner table are referenced in the outer table, there is a reduction in the number of I/Os.

Index merge join

This technique can be used when there are indexes present on the join criteria for both tables. The basic algorithm is to scan the sequence set of the two indexes, and merge rows from the table based on their join criteria being satisfied. For instance, when processing an equijoin, first an entry is read from the sequence set of the outer table. Then, an entry is read from the inner table's sequence set. If the inner table's join column is less than the outer table, the scan of the inner table's sequence set continues. If the inner table's join column is greater than the outer table's, the outer table's sequence set is scanned until either equal or greater values are found. When the join columns in the two indexes satisfy the join criteria, the rows of the tables are read from the data pages, appended to each other, and written to the output table.

The I/O cost of this technique depends on how many rows of the tables meet the join criteria. The best case is on an equijoin of primary keys, when the cost are equal to the number of leaf nodes in the two sequence sets, plus the number of rows in the two tables. If the tables are also clustered by the sequence sets, the number of I/Os to fetch rows is reduced to the number of data pages allocated to the two tables plus the sum of leaf pages on the two indexes.

Hash join

This technique can be used for equijoins only, and does not require any indexes on the join criteria for the two tables. The algorithm first performs the setup phase, when it scans the outer table and places each row into a hash table at a location determined by applying a hash function to the join criteria. The smaller table is always chosen as the outer table, in order to minimize the memory demands of the hash table.

In the second phase, called the *probe*, the inner table is scanned and the same hash function used in the setup is applied to the join columns in order to locate a possible match. Any rows found in the hash bucket from the first table are then examined. The rows that satisfy the join criteria are appended to the inner table row and written to the output table.

This can be a very rapid join technique, especially when the smaller table can be hashed into a table fitting in memory. When this is the case, the I/O required is the sum of the data pages of the two tables. Otherwise, additional I/O is required to partition the hash table into segments, and manage those segments in a temporary file.

Imbedded operators

The two most common relational operators, select and project, are implemented within the other physical operators. This allows the optimizer to perform these operations at the earliest possible time in an access plan, which immediately reduces the amount of data that the remainder of the plan must handle. The implementation of the project operator is further enhanced by performing *index-only* access when appropriate, as explained in the following paragraphs.

Select

The select operation restricts access to rows that do not satisfy the predicate logic of the query. These predicates are evaluated as early as possible when executing the query, which is usually the first time the row containing the column named in the predicate is accessed. Each file access physical operator in SQLBase accepts predicates to perform this row filtering when accessing a table. If the predicates in the query are conjunctive, or *and*-connected, a form of logic called *short-circuit* testing may be performed. This technique allows the row to be discarded as soon as the first unsatisfied predicate is found.

When the SQLBase optimizer estimates the cost of a query, the effect of the predicates used for the select operation is taken into account through a statistic called the *selectivity factor*. This figure represents the proportion of rows in a table that would be expected to satisfy a particular predicate. The optimizer makes its estimate of selectivity factor based on statistics associated with the table's indexes and certain heuristics concerning the predicate's comparison operator. We will examine this process in more detail in the next chapter.

Project

The project operation is implemented within other physical operators by passing the list of columns to be retrieved to any file access operator. That operator then only includes those columns in its output table. Performing the project at the earliest possible time in the query plan reduces the width of intermediate tables and result sets, thereby saving space both in memory and temporary disk files. Note that one side effect of performing the project at page access time is a possible reduction in I/Os due to avoiding access to extent or LONG VARCHAR pages which do not contain needed columns.

Index-only table access

Another benefit of building the project operation into the other physical operators is the ability to perform index-only access. This occurs whenever the columns that are specified for projection in the query are all present in one of the table's indexes. That index must also perform whatever predicate logic is required for the select operation. When this happens, SQLBase's optimizer realizes that the queries data requirements

can be satisfied without access to the table's data pages. This can result in significant I/O savings, depending on the number of rows which satisfy the predicates of the query.

Query plan structure

When the optimizer groups a set of physical operators together to execute a SQL statement, the resulting specification is called a *query plan*.

An analogy to the programming task makes the role of the query plan easier to understand. A programmer codes instructions in some high level computer language, such as C, which is then compiled to produce an executable form of the programmer's code. This executable form uses machine instructions to talk to the CPU that runs the program. When the CPU executes the program, it processes machine instructions rather than the programmer's original code.

Similarly, a programmer codes SQL statements, which are then processed by the optimizer to produce the executable form of the programmer's SQL, which is the query plan. When SQLBase executes the SQL statement, it actually processes the query plan, not the original SQL. Just as the C code has been translated into a more rapidly executable form by the C compiler, the SQL statements are converted into a form which the SQLBase engine can process quickly and efficiently.

Query trees

One way to view a query that provides for easier understanding is called the *query tree*. This is a tree structure, where the final result of the query is at the top of the tree (the root), and the database tables involved in the query are at the leaves. The intermediate nodes of the tree, between the leaves and the root, represent physical operations that are performed during the query execution. When the query plan is executed, the sequence of node traversal is from bottom to top, and from left to right. This is the sequence in which the operations are printed when the 'SET PLANONLY' option of SQLTalk is executed to print out a query plan for a given SQL statement.

The following example shows a query tree for a SQL statement that gets the names of authors who published books in 1993:

```
SELECT A.ANAME
FROM AUTHORS A, BOOKS B
WHERE (B.DATE_PUBLISHED = '1993') AND
```

```
(A.ANAME = B.ANAME);
```
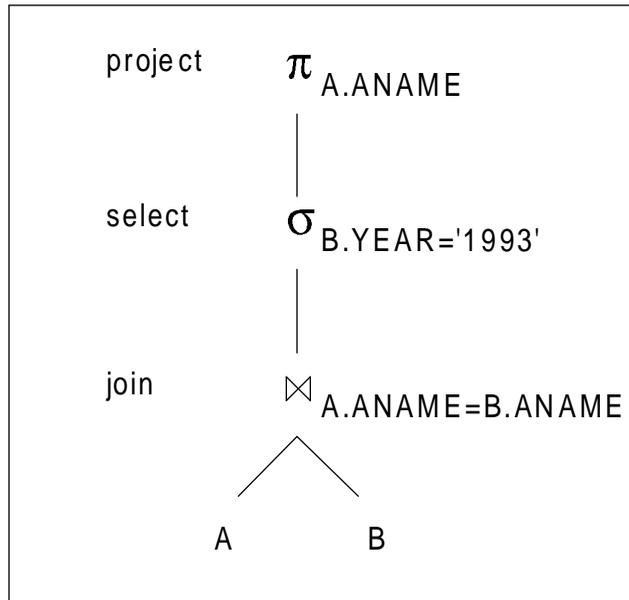

Figure 16A - A query tree for: Get names of authors who published books in 1993.

Building query trees

When the optimizer builds query trees for cost estimation and possible final selection, it uses some simple rules that determine the placement of various nodes in the tree. These rules allow the optimizer to limit the number of query trees that are built for further consideration by the cost estimation task. By limiting the possible access plan alternatives in this way, the optimization process can execute more efficiently.

One of the important heuristics is to apply restrictions such as the select and project operators as early as possible. This is because select and project operations can reduce the size of the result table. In figure B, the optimizer improves the query tree

of Figure A by applying a select first to reduce the number of rows that are input to the equijoin operation.
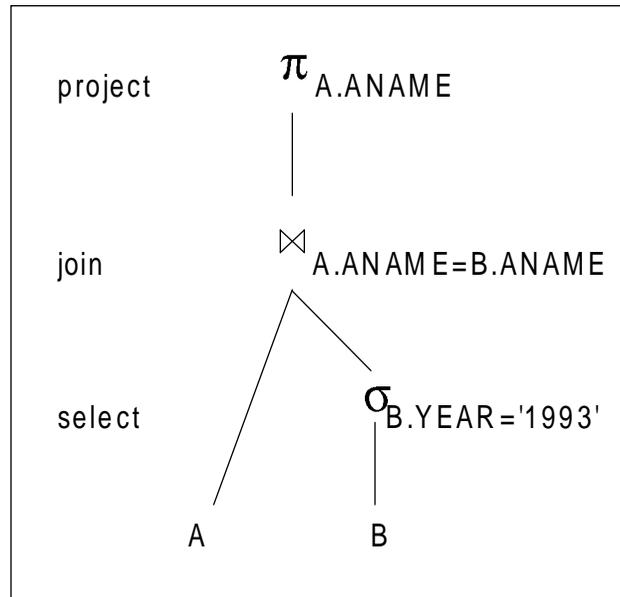
Figure 16B - Using a heuristic rule to move the SELECT operator down in the query tree in Figure A.

Furthermore, when multiple select operations are performed in a query against a table, the ones with the lowest selectivity factor are performed lower in the tree, in order to reduce the size of the intermediate results as rapidly as possible. Of course, when predicates can be evaluated depends on when all the data referenced by the predicate is available. SQLBase converts some predicates in order to evaluate them as rapidly as possible, as we shall now see.

Predicate logic conversion

Whenever multiple predicates are specified using the relational select operation in a query, an important heuristic is to convert them into an equivalent condition in *conjunctive form*, which is a regrouping created by ANDing a set of predicates where each predicate contains only OR operators. For instance, the clause:

```
WHERE COL1>5 OR (COL2<500 AND COL3>150)
```
can be rewritten as:

```
WHERE (COL1>5 OR COL2<500) AND (COL1>5 OR COL3>150)
```

The conjunctive form is preferable since it can be evaluated with short-circuit techniques. A set of predicates in conjunctive form evaluates to TRUE only if every OR grouping evaluates to TRUE. It evaluates to FALSE if any grouping evaluates to FALSE. In the short-circuit technique, if the result of evaluating any single OR grouping is FALSE, there is no need to evaluate other conditions at all. Since the number of comparison operations that are performed in a query have a direct impact on the amount of CPU time needed, the short-circuit evaluation of predicates in conjunctive form can save CPU cycles.

Chapter 17

SQLBase Optimizer Implementation

In this chapter we:

- Discuss the details of how the SQLBase query optimizer performs access plan analysis.

- Summarize the statistical information available to the SQLBase query optimizer to develop access plan cost estimates.

- Explain how the query optimizer identifies possible access plans for processing an SQL statement.

- Review the two cost components SQLBase estimates for each access plan using statistics about the database:

  - Total number of I/O operations.

  - Total CPU time required.

Database statistics

When the SQLBase query optimizer performs access plan selection, it uses statistics about the database to estimate the costs associated with each candidate access plan. These statistics, which are stored in the SQLBase system catalog tables, are associated with the various tables and indexes contained within the database. Some of these statistics are maintained dynamically, meaning that SQLBase keeps them internally updated during normal server operations. The majority of statistics, however, are maintained statically, meaning that they are only calculated and stored when specific events occur.

The event that usually causes these statistics to be refreshed is the 'UPDATE STATISTICS' SQL statement. This command examines a database's tables and indexes, calculates the required statistics, and then stores them in the appropriate tables. Statistics are also created for an index or a table when it is initially created. If no rows are present at this time, the statistics are set to default values.

Starting with SQLBase version 6.0, statistics can now also be updated directly by the DBA, using the user modifiable statistics feature. In this feature, the access paths chosen by the optimizer reflect the cost calculations that would be performed if the database was populated in a manner that would yield the statistics placed in the catalog by the DBA. In this way, the DBA can influence the optimizer into making decisions that are based not on the current database contents, but rather on the database as the DBA projects it will be populated at some point in the future. In order to accurately calculate and store these statistics, the DBA must have a thorough understanding of what these statistics mean.

Statistics that are dynamically maintained by SQLBase are used by the optimizer whenever the DBA has not overridden the catalog statistics using the user modifiable statistics feature. When you override the catalog statistics with your own calculated values, the optimizer will them ignore the dynamic internal statistics in favor of your catalog statistics.

Table statistics

The statistics associated with database tables are maintained in two separate system catalog tables, each of which is described in the following paragraphs. All columns not specifically mentioned as being maintained dynamically are static, and are populated by running UPDATE STATISTICS or at CREATE time.

Statistics in SYSTABLES

The following statistical information is maintained in these columns of the
SYSADM.SYSTABLES table in the system catalog:

- ROWCOUNT

  This is the number of rows in the table. This statistic is maintained
  dynamically in the table's control page, but is only externalized to the
  SYSTABLES column when the UPDATE STATISTICS command is
  executed.

- PAGECOUNT

  This is the total number of data pages in the table. This is also a dynamic
  statistic that is externalized when UPDATE STATISTICS is run. Also, when
  this column is maintained by the system, it will always be the sum of the next
  two columns, ROWPAGECOUNT and LONGPAGECOUNT. If, on the
  other hand, the DBA has set these statistics explicitly, then this relationship
  may not hold true.

- ROWPAGECOUNT

  This is the number of base row pages occupied by the table, plus any extent
  pages that may also be allocated. This is another dynamic statistic which is
  only externalized on command.

- LONGPAGECOUNT

  This is the number of pages allocated to the table for the storage column
  defined with the LONG VARCHAR data type. This is a dynamic statistic
  which is only externalized on command.

- EXTENT_PAGECOUNT

  This is the average contribution of data stored in base and extent row pages.
  It excludes contribution from long pages.

- AVGROWLEN

  This is the actual average length of rows in the table. This can differ
  significantly from the defined row length since SQLBase stores all columns
  as variable data regardless of the data type used in the column definition. Note
  that this row length is only the length of the row as stored on base table and
  extent pages, and therefore excludes all LONG VARCHAR columns.

- AVGROWLONGLEN

  This is the actual average length of all LONG VARCHAR columns stored in
  the table. This column will contain zero if there are no columns of this data
  type stored.

Statistics in SYSCOLUMNS

The following statistical information is maintained in these columns of the SYSADM.SYSCOLUMNS table in the system catalog:

- AVGCOLLEN

    This is the actual average length of this column across all rows in the table. This can differ from the defined column length since SQLBase stores all columns as variable data. A base row page stores "a long data description" for every non-NULL long column.

- AVGCOLLONGLEN

    This is the actual average length of a LONG VARCHAR column across all rows of the table. This value is zero if this is not a LONG VARCHAR column.

Index statistics

The statistics associated with indexes on tables are also maintained in two separate system catalog tables, as detailed in the following paragraphs. All columns (except one, the HEIGHT column in the SYSINDEXES table) are maintained statically, and are populated when UPDATE STATISTICS is run or the index is initially created.

Statistics in SYSINDEXES

The following statistical information is maintained in these columns of the SYSADM.SYSINDEXES table in the system catalog:

- HEIGHT

    The height of the index tree is the number of nodes that have to be read from the root to the leaf level inclusive. Also, it is called the depth of the tree. The statistic is also maintained dynamically in the index's control page. This field is null for a hash index.

- LEAFCOUNT

    The total number of nodes in the bottom level (leaves) of the index is the leafcount. Also, it is the number of pages in the index's sequence set. This field is null for a hash index.

- CLUSTERCOUNT

    The total number of page changes that would occur if the entire table was read through the index's sequence set. For a perfectly clustered index, this column is equal to the number of base data pages in the table. At the other extreme, the highest value for a totally unclustered index is equal to the number of rows in the table. This field is null for a hash index.

- PRIMPAGECOUNT

  The number of primary pages that have been allocated in the base table for the subject table of a hashed index. It is the same as the number of hash slots available for distribution of the table's rows (see Chapter 10, *Hashed Indexes*, for more information on the significance of this figure). This field is null for a B+Tree index.

- OVFLPAGECOUNT

  This is the number of overflow pages that SQLBase allocates for the subject table of a hashed index. When the table and index are initially created, this number reflects the number of overflow pages that SQL Base pre-allocates. After that, this number increases as additional overflow pages are required to handle hashing collisions. This field is null for a B+Tree index.

- AVGKEYLEN

  For B+Tree indexes, this is the average length of the key across all entries in the index. This number is needed because SQLBase maintains all index entries as variable length, minimum prefix only fields (see Chapter 9, *B-Tree Indexes*, for more information on these index entries). This field is null for a hash index.

Statistics in SYSKEYS

The following statistical information is maintained in this column of the SYSADM.SYSKEYS table in the system catalog:

- DISTINCTCOUNT

  This is the number of distinct keys in the index for that portion of the key from the first column up to the COLNO of this entry. If every possible combination of values for a composite key exists, this number is the product of each successive column's *cardinality*. Cardinality is the number of distinct values that a column has in a table. The simplest example is a column called SEX_CODE, which has a cardinality of two, for the values 'M' and 'F'. Consider the following index:

  ```
  CREATE INDEX XNKSALS
  ON EMPLOYEE (DEPT,
  SALARY,
  YEARS_SERVICE);
  ```

  The employee table contains 250 rows, one for every employee in the company. Also, the cardinality characteristics of the columns are as follows:

  - DEPT - There are 10 departments.

- SALARY - No two employees in the company have the exact same salary; in other words, there are 250 discrete salary values.

- YEARS_SERVICE - The company experienced three distinct hiring periods, and no one has ever left. Therefore, there are only three distinct values for this column.

Given these conditions, the SysKeys rows which describe these index columns have the following values for DISTCOUNT:

- DEPT - 10

  One for each department in the company.

- SALARY - 250

  Since each of the employees has a discrete salary value, the number of keys at this point in the index equals the number of rows in the table. Alternatively, if there are only five salaries paid by the company, and each department has someone at each of the five salary levels, then the DISTCOUNT for this column is 50 (10 times 5).

- YEARS_SERVICE - 250

  Since the salary field had already raised the cardinality of the index keys to be equal to the number of rows in the table, this field cannot raise it any higher. It is impossible for the number of index entries at any level to exceed the number of table rows. However, in the alternate scenario where there are only five salary levels, the value of DISTCOUNT at this column is 150 (3 times 50), assuming that every department had people at each salary level who were hired in each of the three hiring waves.

Selectivity factor

The selectivity factor is a statistic which is computed by the optimizer for predicates used by the relational select operator (recall that the select operation is implemented internally in all SQLBase file access operators). In addition, the selectivity factor is used by the optimizer to determine if an index access path is more efficient than a table scan. Consequently, SQLBase only calculates the selectivity factor for those predicates which contain attributes for which an appropriate index exists.

The selectivity factor estimates the number of rows which are returned following the application of the select predicates on a table of known size. This is critical when the optimizer is evaluating an index in an access plan. The index will be more efficient than a table scan only if the selectivity factor multiplied by the number of rows in the table are less than the number of database pages occupied by the table.

In other words, if most of the table rows have to be retrieved anyway, using the index only incurs the overhead of reading through the index structure without saving the

need to read the same number of data pages. Since this is the case, the selectivity factor appears in virtually every I/O costing formula associated with an index access operation that is used by the optimizer.

Numerically, the selectivity factor is a probability, which means it ranges from zero to one. Multiplying the number of rows in a table by the selectivity factor associated with that table's predicates will directly yield the number of rows which would survive the select operation, based on the assumption that the values of the table are evenly distributed among its rows.

Single predicates

Determining the selectivity factor of a single predicate, such as EMPLOYEE_NO = 65, depends on what operator is used in the predicate. The operator affects the selectivity factor because it determines the relationship between the rows that satisfy the predicate and the other operand.

The more restrictive the operator, the lower the selectivity factor of the predicate. The most restrictive operator is equality (=), since only a single column value can satisfy the predicate. In this case, the selectivity factor is simply the inverse of the cardinality of the column, which is stored in the DISTINCTCOUNT column of the SYSADM.SYSINDEXES table entry for this index's key.

However, it is not as easy to calculate the selectivity factor for inequality operators. Consider the following two predicates:

```
ORDER_DATE > JAN-01-1900
ORDER_DATE > SYSDATE - 1 DAY
```

Obviously, the first predicate could be expected to return many more rows than the second when applied to the ORDER table. But how can the SQLBase optimizer determine this? It turns out that the optimizer calculates the selectivity factor by accessing the upper levels of the index structure which contains the column in question (in this example, an index on ORDER_DATE). It then estimates the number of rows that satisfies the predicate by extrapolating the proportion of index keys at this index level. This technique, called the *BTree scan*, allows the SQLBase optimizer to make reasonably accurate estimations about the results of inequality predicates.

The other factor that plays a role in determining the selectivity factor of a single inequality predicate is whether the column is compared to a constant or a bind variable. This is not critical for equality predicates because the optimizer can calculate the selectivity factor from the DISTINCTCOUNT column with the assumption of equal value distribution throughout the table. This assumption allows the optimizer to be insensitive to the actual value of the operand on the other side of the equality operator. However, for the optimizer to use the BTree scan technique, it must be able to find a value for the other operand. When the operand is a bind variable, the optimizer must fall back to making an assumption that is actually a hard-coded value.

The following table summarizes the determination of the selectivity factor of a single predicate, depending on the operator and whether a constant or bind variable is used as the other operand. Note that where 'transformation' is listed, the predicate is eliminated from the query through semantic transformation to another form.

Predicate Type	Operator Constant	Operator Not Constant
=	1/cardinality	1/cardinality
!=, <>	1 - 1/cardinality	1/3
>	btree scan	1/3
!>	btree scan	1/3
<	btree scan	1/3
!<	btree scan	1/3
>=	btree scan	1/3
<=	btree scan	1/3
between	btree scan	1/3
null	treated as a zero length constant	not applicable
exists	transformation	transformation
like	btree scan	indeterminate
in	transformation	transformation

Multiple predicates

When multiple predicates are placed on a single table, the SQLBase optimizer follows some fixed rules when combining their selectivity factors into a total selectivity factor for the entire relational select operation. The first rule regards how many predicates are considered when calculating a total selectivity factor for predicates containing a mix of equality and inequality comparison operators. SQLBase follows these rules in determining which predicates to evaluate a selectivity factor for:

- When the high order columns of the index key are contained within equality predicates, the optimizer can find the selectivity factor for the multiple columns by reading the DISTINCTCOUNT column of the lowest order key's SYSKEYS row. When the first inequality predicate is encountered, it is ignored along with any remaining predicates containing columns that appear in the index. An example of this is:

```
EMPLOYEE_NO = 45 AND
DEPT = 50 AND
    SALARY > 25000
```

Assume the index is defined on a composite key consisting of EMPLOYEE_NO, then DEPT, followed by SALARY. The combined selectivity factor for both the EMPLOYEE_NO and DEPT columns, which appears in the DISTINCTCOUNT column of the DEPT row of the SYSKEYS catalog table, is used by the optimizer to estimate the number of rows that will be returned from the query. The effect of the inequality predicate on the SALARY column is ignored. For an example of how the combined DISTINCTCOUNT is calculated for multiple columns, see the database statistics section earlier in this chapter.

- If the high order column of the index key is the subject of an inequality predicate, only the selectivity factor of this predicate is evaluated, and any remaining predicates are ignored (regardless of whether they are inequality or equality predicates). An example of this is:

```
SALARY > 25000 AND
DEPT = 50 AND
YEARS_SERVICE > 3
```

Assume the index is defined on a composite key consisting of SALARY, then DEPT, followed by YEARS_SERVICE. The optimizer determines the selectivity factor by looking at the root page of the index to determine what proportion of table rows have a SALARY > 25000. This proportion becomes the selectivity factor for the entire group of predicates. The DEPT and YEARS_SERVICE predicates are ignored.

Use of statistics by the optimizer

The first step the SQLBase query optimizer performs in preparing a statement for execution is to build a list of candidate access paths that could be executed to satisfy the data requirements of the SQL statement. Because the optimizer is cost-based, database statistics are used as the basis for projecting an estimate of the work required to execute each possible access plan considered by the optimizer as a candidate for possible execution.

After the SQLBase query optimizer builds a group of alternative access plans, each plan is assigned an estimated execution cost so that the most inexpensive plan may be chosen for subsequent execution. The two components of query execution cost evaluated by SQLBase are I/O and CPU cost. These costs are then combined in order to assign the plan an overall cost. Once all alternative plans are assigned a cost, the smallest is selected and the remainder are discarded.

Chapter 18

Working with the SQLBase Query Optimizer

In this chapter we:

- Discuss the specific actions that may be taken in order to improve the performance of a query which is causing its transaction to fail to meet response time objectives.

- Specify a seven step procedure that may be applied to tune the performance of a query in a systematic manner.

Introduction

Section 1, *Database Design,* presents a methodology for physical database design. This database design method compares general principles and heuristics to create an internal schema that exhibits good performance characteristics for most SQL queries. However, the physical design method also acknowledges that some individual queries may not meet their performance requirements without further tuning (see chapter 7, *Physical Design Control*).

A myriad of tuning options faces database designers for increasing the performance of individual queries. One of the primary ingredients for success is knowing which option to use in which situation and when. However, one primary option is working with the SQLBase query optimizer to ensure that the correct access paths are available to the query optimizer. This chapter addresses this issue.

When working with the SQLBase optimizer to increase the performance of a specific SQL SELECT statement, the database designer performs three distinct activities: update statistics, determine the optimal index set, and override the query plan(s) chosen by the SQLBase optimizer.

Update statistics

As described earlier in this section, SQLBase maintains several statistics regarding tables and indexes. The query optimizer uses these statistics to calculate the cost of various access paths. Therefore, the first aspect of working with the query optimizer, improving the performance of a given query, is to make sure these statistics are correct and up-to-date.

Optimal index set

Another aspect of working with the query optimizer involves adding indexes to increase the speed of restrictions and joins, and to eliminate sorts for GROUP BY and ORDER BY clauses. If a SELECT command is performing poorly, it is probably either joining, sorting, or reading large tables. Indexes can increase the performance of all these operations.

As discussed in chapter 6, a common indexing guideline is to create indexes on all primary and foreign keys. This is because in most systems these are the columns that appear most frequently in WHERE clause predicates and ORDER BY and GROUP BY clauses. This initial index set (the collection of all indexes in a database) provides indexes for restrictions and eliminates sorts across primary and foreign keys. Furthermore, joins are frequently the most time-consuming aspect of a specific query. The fastest join algorithm for large tables is the index merge. The initial index set ensures that the index merge access path is available to the optimizer as long as the join columns are either primary or foreign keys. For all other joins, the optimizer is

limited to the slower join algorithms, the hash join or one of the nested loop techniques. However, for most SELECT commands, the access paths available to the optimizer with the initial index set provide adequate performance.

On the other hand, the initial index set may not meet the performance requirements of all queries. For example, some queries containing join columns that are neither primary nor foreign keys may perform poorly. For those queries not meeting their performance criteria, you must add additional indexes to increase the speed of the join. However, you do not want to create one or more specific indexes for every SELECT command performing poorly since:

- Indexes slow update DML commands, specifically all INSERT and DELETE statements, as well as any UPDATE statements that modify a column that is contained within one or more indexes.

- Index management is both time-consuming and costly.

- Indexes take additional disk space.

- Indexes can also cause locking contention because many rows may be stored on a single index node page.

Consequently, the general design goal is to create an index set that contains the fewest possible indexes which meets the performance requirements of all database transactions. This is the optimal index set.

Tuning a SELECT statement

This section presents a procedure for working with the SQLBase optimizer to tune a query that is performing poorly. The procedure iterates to the optimal index set. We also assume you are familiar with the protocol described in chapter 7 for physical design control. The procedure consists of seven discrete steps, each of which are described in the following paragraphs.

Step 1: Update statistics

Before adding indexes, verify that the database statistics stored in the system tables are correct. If you execute the query against an actual production database, you should update the statistics for all the tables listed in the FROM clause using the UPDATE STATISTICS command. On the other hand, if you are using a small test database, you may have to manually calculate and override the statistics calculated by SQLBase for both the tables listed in the FROM clause and their indexes.

Once you have updated the statistics, you should compile the SQL command with the PLANONLY parameter set ON. Compare the new query plan to the old plan before updating the statistics to determine if the plan has changed. (Sometimes it requires several minutes or hours to execute a query. By comparing the plans, you can avoid executing a query only to find that the performance is identical to the last time the

query was executed). If so, execute the query to determine if performance has improved significantly.

Step 2: Simplify the SELECT command

Before adding indexes or forcing plans, you should simplify the query. The objective is to make the SQL for the SELECT statement as simple as possible in order to eliminate as many variables as possible. Once you have simplified the query, compile the SQL command with the PLANONLY parameter set ON. Compare the new query plan to the plan before simplifying the command to determine if the plan has changed. If so, execute the query to determine if performance has improved significantly.

To simplify the SELECT command:

- Eliminate unnecessary predicates or clauses
- Add parentheses to arithmetic and Boolean expressions
- Convert bind variables to constants

Eliminate unnecessary predicates or clauses

It is common to include more SQL clauses in a SELECT command than is absolutely necessary. This is usually done to ensure the correct answer is returned and stems from a lack of confidence in one's understanding of the SQL syntax. Before attempting to tune the query, the database designer should remove all unnecessary clauses and predicates. Clauses which are commonly included but not needed are:

- **ORDER BY clauses**. Frequently, an ORDER BY clause is included even though an exact ordering of the result set is not required by the application program or end-user.
- **WHERE clause predicates**. Frequently, WHERE clauses contain redundant restriction predicates. For example, the predicates in the following WHERE clause are redundant, since DEPT_NO is the primary key, and therefore will uniquely identify only one row:

```
WHERE DEPT_NO = 10 AND DEPT_NAME = 'OPERATIONS'
```

Add parentheses to arithmetic and Boolean expressions

The SQLBase SQL language syntax includes a specific precedence for evaluating arithmetic and Boolean expressions. However, it is easy to make a mistake in applying the precedence rules. Therefore, we recommend you add parentheses to these expressions to explicitly state the precedence. You may think the optimizer is evaluating an expression in one manner when it is actually using another.

Convert bind variables to constants

Bind variables are general place holders used when compiling an SQL command. It is often desirable to use stored commands to eliminate the overhead of compilation when one SQL statement will be executed many times. These stored commands are pre-compiled, and their access plan is saved in the system catalog along with their source text. Also, programs sometimes issue one prepare for a statement that will be executed many times throughout the program with little modification. In these situations, bind variables are used to allow the values of some variable tokens in the command to change between executions without having to re-compile. However, when bind variables are used, the query optimizer does not know what values are bound to the program variables when the SQL command is executed. This causes the query optimizer to make some gross assumptions when calculating the selectivity factor. These assumptions may lead to a high selectivity factor which can prevent the optimizer from using a specific index. Consequently, when tuning queries, you should convert all bind variables to constants so that the query optimizer has specific values to work with.

For example, when bind variables are used in restriction predicates, the query optimizer is forced to use a default selectivity factor of 1/3. Consider the following predicates:

```
Amount > :BindVariable
Amount > 1000
```
The advantage of the first predicate is that the SQL command can be compiled once and executed many times using many values in the bind variable. The disadvantage is the query optimizer has less information available to it at compile time. The optimizer does not know whether 10 or 10,000,000 will be bound to the program variable. Hence, the optimizer is unable to calculate the selectivity factor. Consequently, in these situations the optimizer simply sets the selectivity factor to a constant that may not reflect the true situation (this constant is currently 1/3 for SQLBase 6.0, but is subject to change in future releases). Since this 1/3 figure is relatively high (in the absence of other more restrictive predicates) the optimizer may not use any index associated with the column contained in these predicates.

On the other hand, with the second predicate, the optimizer knows the comparison value at the time of the compile and is therefore able to calculate a more accurate selectivity factor.

Bind variables also cause problems when used in LIKE predicates. The LIKE operator can include wild card characters. For example, the following query finds all customers whose name begins with the letter A:

```
select * from customer where name like 'A%'
```
The wild card characters can occur in the beginning (for example, '%A'), middle (for example, 'A%A'), and terminal positions. The nature of BTree indexes is such that

these indexes are only helpful in queries involving the LIKE operator when the wild card characters do not occur in the beginning. Therefore, the SQLBase optimizer does not use an index if a LIKE operator is used and a wild card is in the beginning position.

Furthermore, the SQLBase optimizer also does not use an index when bind variables are used with the LIKE operator. This is because the optimizer does not know at runtime where the wild card will occur. Therefore, the optimizer chooses an access path (i.e., table scan) which satisfies the query with the wild card in any position, resulting in significantly decreased performance. Hence, convert all bind variables to constants in LIKE predicates.

Step 3: Review the query plan

Execute the query with the PLANONLY parameter ON to display the query plan which the optimizer has chosen. Familiarize yourself with the query plan since you must understand the plan in order to use it. Several items to especially look for are:

- ***Conversion of subqueries to joins***: The optimizer converts most subqueries to joins. Make sure you understand where these transformations are taking place; you need to know where these transformations occur in some of the following steps.

- ***Temporary tables being created***: If a temporary table is created, this can indicate that the optimizer is sorting intermediate results. If this is occurring, you may want to add an index in one of the following steps to avoid the sort.

- ***Slower join techniques***: The hash join and nested loop techniques are not as fast as the index merge technique for large tables. If these techniques are being used, you may want to add indexes in steps 5 or 6 so that these joins use the index merge technique. Of course, for some situations the hash join may represent the best possible method when large amounts of data have to be processed.

Step 4: Locate the bottleneck

The query which is performing poorly can contain many clauses and predicates. If so, you must determine which clauses or predicates are resulting in the poor performance. Generally if you remove one or two clauses or predicates, query performance improves significantly. These clauses are the "bottleneck."

Experiment with the query; iteratively remove (comment out) clauses and/or predicates until the query exhibits substantially superior performance. Locate the bottleneck clauses and predicates. For example:

- If the query contains an ORDER BY clause, comment it out and see if the query plan changes. If the query plan changes, execute the query to determine if performance improves significantly.

- If the query contains several joins, locate the one that is taking the longest to perform. Successively comment out all but one join and execute the query. Determine if one join is the bottleneck.

- Eliminate any AT (@) functions (such as @UPPER) being performed on WHERE clause predicates and see if performance increases. It may be that an index is not being used because of a @function. If this is the case, the index may be altered to contain the function if the query and function are sufficiently critical. Of course, this alternative may then make the index unusable for other queries whose performance had been acceptable before the modification.

Step 5: Create single column indexes for the bottlenecks

If you complete steps 1 through 4 and the query is still not meeting its performance requirements, you must create indexes specifically for the query to increase performance. In general, single column indexes are preferable to multi-column indexes because they are more likely to be useful to other queries. Try to increase the performance of the query with single column indexes first before resorting to multi-column indexes.

Locate the following table columns used in the query which do not have an index (either a single attribute index or a right-hand subset of a multi-column index):

- ***Join columns***: This includes subqueries that the optimizer converts to joins. If the primary key of the table is a composite key, the join is specified across multiple columns and a multi-column index is needed to increase the performance of this join. This index should already exist.

- ***GROUP BY columns***: If the GROUP BY clause contains more than one column, a multi-column index is needed to increase the performance of this GROUP BY clause. Skip these columns until step 6.

- ***ORDER BY columns***: If the ORDER BY clause contains more than one column, a multi-column index is needed to increase the performance of this ORDER BY clause. Skip these columns until step 6.

- ***Low cost restrictions***: These are the table columns referenced in the WHERE clause restriction predicate with the lowest selectivity factor for each table reference in the FROM clause.

Compare these table columns to the bottlenecks discovered in step 4. One of the table columns identified above should match one of the bottleneck clauses or predicates identified in step 4. If so, create a single column index on one of these columns. Determine if the addition of this index changes the query plan by running the query

with the PLANONLY parameter ON. If it does, execute the query to determine if adding this index significantly increases the performance of the query.

If adding the index does not increase performance significantly, create another index on another table column identified above. Continue until you have tried all columns identified above. If you are unable to increase the performance of the query by adding a single column index, proceed to the next step.

Step 6: Create multi-column indexes

As a last resort, you may have to create one or more multi-column indexes specifically for this query to achieve the desired performance. The procedure for doing so is identical to the one described in the previous section for single column indexes:

1. Create a multi-column index.

2. Determine if the query plan changes. If it does, execute the query to determine if performance increases.

3. If performance does not improve, create another index. Continue creating indexes until all possible multi-column indexes have been created.

The types of multi-column indexes to create are listed below:

- **Multi-column GROUP BY clause**: If the query has a GROUP BY clause containing more than one column, create an index on all of the columns listed in the GROUP BY clause. Be sure to specify the columns in the same order as listed in the GROUP BY clause.

- **Multi-column ORDER BY clause**: If the query has an ORDER BY clause containing more than one column, create an index on all of the columns listed in the ORDER BY clause. Be sure to specify the columns in the same order as listed in the ORDER BY clause. Also, be sure to specify the same sequencing (ascending or descending) for each column as specified in the ORDER BY clause.

- **Join column(s) plus the low cost restriction**: For each table listed in the FROM clause that has at least one restriction specified in the WHERE clause (in descending order from the largest table to the smallest), create a multi-column index on the join column(s) and the column in the restriction predicate with the lowest selectivity factor.

- **Join column(s) plus all restrictions:** For each table listed in the FROM clause that has at least one restriction specified in the WHERE clause (in descending order from the largest table to the smallest), create a multi-column index on the join column(s) and all columns included in any restrictions involving that table. The order of the columns in the index is critical to whether the index will be selected for use by the optimizer. In

general, list equality columns before other columns, and list all equality predicate columns in increasing selectivity factor order. After listing all equality predicates in the CREATE INDEX statement, list the inequality predicate which has the lowest selectivity factor as the last column in the index. Any other inequality predicates present in the WHERE clause will probably not benefit the performance of the statement, nor will they be considered by the SQLBase query optimizer when selecting access paths.

Step 7: Drop all indexes not used by the query plan

As discussed in the previous section, indexes slow down UPDATE DML commands and index management is time-consuming and costly. Therefore, you should keep track of the indexes created during steps 5 and 6 and drop all indexes that are not used by the query.

Index

examples 16-5
selectivity factor 6-11—6-12, 6-16, 7-12, 16-14, 17-6, 18-5
    for inequality operators 17-7
    total 17-8
semantic validation 2-11
sequencing node properties 9-5
sequential reads 8-2
server names 2-4
set operations 16-3
slot
    byte offset 8-6
    row header 8-5
    table 8-4—8-9
SMALLINT 3-4
sort key 16-9
splitting tables 6-2—6-6
    horizontal splitting 6-2, 6-4
    vertical splitting 6-2
SQLBase catalog 16-2
sqlexp function 7-6
SQLPPCX flag 13-8
standardized templates 2-3
static data 7-7
statistics
    AVGCOLLEN 17-4
    AVGCOLLONGLEN 17-4
    AVGKEYLEN 17-5
    AVGROWLEN 17-3
    AVGROWLONGLEN 17-3
    CLUSTERCOUNT 17-4
    DISTCOUNT 17-5
    DISTINCTCOUNT 17-7
    HEIGHT 17-4
    LEAFCOUNT 17-4
    LONGPAGECOUNT 17-3
    OVFLPAGECOUNT 17-5
    PAGECOUNT 17-3
    PRIMPAGECOUNT 17-5
    ROWCOUNT 17-3
    ROWPAGECOUNT 17-3
    UPDATE STATISTICS 7-6, 17-2, 17-4, 18-3
    user modifiable 17-2
strategic data plan 2-4, 2-6
strategic management 2-4
system catalog tables 17-2
    SYSADM.SYSCOLUMNS 17-4
    SYSADM.SYSINDEXES 17-4

    SYSADM.SYSKEYS 17-5
    SYSADM.SYSTABLES 17-3
system development 2-2

T

table names 2-3
table scan 8-2, 16-10
table size calculation
    hashed 11-6
    regular 11-6
temporary table 12-6, 18-6
TIME 3-4
TIMESTAMP 3-4
transaction
    batch 5-4
    complexity 5-4, 5-5
    data manipulation language (DML) 1-4
    database 5-2
    defining 5-2, 5-3
    definitions 5-2—5-6
    description 5-3
    high complexity 5-4, 5-5
    low complexity 5-4, 5-5
    medium complexity 5-5
    name 5-3
    number 5-3
    online 5-4
    performance 5-2, 6-4
    performance requirements 5-5
    relative priority 5-5
    secondary effects example 13-4
    SQL statements 5-6
    type 5-4
    volumes 5-4
transformation rules 15-5
tree
    multiway 9-2, 9-4
tuning the schema 7-6

U

uncommitted data
    dependency problem 13-5
    effect on other transactions 13-3
union 16-3
update serial number 8-5—8-7
user names 2-3

V

VARCHAR 3-4, 6-4−6-5
view administration 1-5
view names 2-3
virtual table 12-7

W

WHERE 6-7, 6-8, 6-12−6-13, 16-6
workstation names 2-4
write locks(X-Locks) 13-5