ACTIAN™

Zen Programmer's Guide

Zen v16

Activate Your Data™

Contents

About This Document

Zen developers can create applications using either the MicroKernel Engine, the Relational Engine, or a combination of the two. This documentation overviews Zen database design and concepts at the transactional and relational levels. It explains the fundamentals about how to develop applications that use Zen interfaces.

This developer information is part of the broader documentation library integrated into Zen Control Center (ZenCC). Access the documentation through the ZenCC interface on the Welcome tab, in the Help menu, or by pressing F1 (Windows) or Shift F1 (Linux).

We would appreciate your comments and suggestions about this document. Your feedback can determine what we write about the use of our products and how we deliver information to you. Please post your feedback in the community forum on the Actian Zen website.

Organization of This Guide

This guide is divided into four areas:

- Database Access Methods
- Transactional Programming with the MicroKernel Engine
- Relational Programming
- Appendixes

Database Access Methods

- Overview of Zen Access Methods

  Introduces the various visual components and APIs with which you can develop Zen applications.

- Developer Quick Start

  Surveys access method options.

Transactional Programming with the MicroKernel Engine

- Developing Applications for the MicroKernel Engine

  Covers the developing and running of applications in the MicroKernel Engine environment.

- MicroKernel Engine Fundamentals

  Describes API and MicroKernel Engine features.

- Designing a Database

  Provides information about creating a data file, improving system performance, and setting up security.

- Language Interfaces Modules

  Provides language interface source modules provided in the Zen SDK installation option.

- Interface Libraries

  An overview of Zen interface libraries and the requirements for shipping Glue DLL files.

- Working with Records

  Provides information about inserting and updating records, establishing position in a record, and adding and dropping keys.

- Supporting Multiple Clients

  Describes the fundamental concepts of supporting multiple users and applications.
- Debugging Your Btrieve Application

  Offers tips on troubleshooting your application.
- Btrieve API Programming

  Provides information to help you begin developing a Zen application by making direct calls to the Btrieve API.

Relational Programming

- Creating a Database

  Explains how to create a database by creating the data dictionary and creating database tables, columns, and indexes.
- Relational Database Design

  Introduces principles of relational database design. A thorough database design throughout the development process is critical to successful database functionality and performance.
- Inserting and Deleting Data

  Explains how to add data to a database using either Zen applications or SQL statements. It also explains how to drop (delete) rows, indexes, columns, or tables from your database or drop an entire database when you no longer need it.
- Modifying Data

  Explains how to modify table definitions, column attributes, and data. You can perform these tasks by entering SQL statements using an interactive application.
- Retrieving Data

  Discusses how you can use SELECT statements to retrieve data.
- Storing Logic

  Explains how to store SQL procedures for future use and how to create SQL triggers.
- Managing Data

  Discusses defining relationships among tables, administering database security, and controlling concurrency with transactions. It also discusses atomicity in Zen databases.

Appendixes

- Sample Collations Using International Sorting Rules

  Lists sample collations of language-specific strings, using the ISR tables provided in the MicroKernel Engine.

- Sample Database Tables and Referential Integrity

  Describes the design of the tables in a Demodata sample database.

Overview of Zen Access Methods

The following table summarizes methods of programming through Zen access methods and APIs.

Access Method	Description	Used For
Btrieve (MicroKernel Engine)	Original Btrieve API	Creating Btrieve database applications
Btrieve 2 (MicroKernel Engine)	Simplified Btrieve SDK for C and C++ developers, extended with Simplified Wrapper and Interface Generator (SWIG)	Creating Btrieve database applications
ADO.NET (Microsoft IDEs)	High-level visual or code-based programming	Visual programming of transactional or relational (SQL) applications. This is the recommended programming interface for Microsoft development environments.
PDAC (Embarcadero IDEs)	Zen Direct Access Components for Delphi and C++ Builder	Replaces functionality of Embarcadero data-aware components and eliminates need for Embarcadero Database Engine.
ODBC (relational)	Microsoft Open Database Connectivity	Creating SQL-based applications
Java Class Library	Java Class Library for MicroKernel Engine data access.	Creating Java-based applications that connect to the MicroKernel Engine
JDBC	Implementation of Oracle Java database connectivity	Creating JDBC-based SQL applications using an industry-standard API.
Distributed Tuning Interface (DTI)	Zen API for monitoring and administration	Performing administrative and utility functions from applications, creating and maintaining Data Dictionary Files from applications
Distributed Tuning Objects (DTO)	Zen object-oriented programming interface for monitoring and administration	Performing administrative and utility functions from applications, creating and maintaining Data Dictionary Files from applications

Developer Quick Start

The Zen Software Development Kit (SDK) offers you the best of both worlds for database solutions: The MicroKernel Engine offers high speed data transactions, and the Relational Engine provides full featured relational data access to the same data with greater performance in reporting and decision support.

The following topics offer quick tips to help you begin building applications with Zen:

- Choosing An Access Method
- Database Connection Quick Reference
- Additional Resources for Application Developers

Choosing An Access Method

Many factors affect development strategy choices. Availability of tools on different platforms, the developer's familiarity with a given programming environment, and portability requirements often play decisive roles in the process. On the other hand, when the developer has more flexibility, various subtle factors should be considered.

Performance is always a consideration. Run-time performance, however, must be balanced against development time: is it more important to deliver the program quickly, or to have it run quickly in use?

In the context of database programming, the database interface affects both development time and run-time performance. Often the choice between SQL and Btrieve is based on these factors alone.

If you are new to Zen products, you may want to use access methods such as ADO.NET, JDBC, Zen Direct Access Components for Delphi and C++ Builder, or other third-party development tools to develop Zen applications.

If you want to directly write to the Btrieve API, see Btrieve API Programming.

The following table compares the various Zen access methods.

Access Method	Characteristics	Suitable For
Btrieve API	• DLL can be called by (almost) any Windows programming language. • Exposes the complete feature set of the database. • Minimum size. • Greatest flexibility. • Shortest code path between application and data. • Least code overhead in the relational database management system (but more application code must be devoted to database management issues). • Client-server capability. • BLOB support.	• Applications where size or run-time performance is the primary consideration.

Access Method	Characteristics	Suitable For
Java	• Thin client. • Cross-platform portability. • Internet/Intranet capability. • Supports Winsock and JNI protocols. • Machine and OS independence. Internet and Web capability. • Minimum size. • Good flexibility. • Rowset, field abstractions implemented in the interface. • Fair overall performance (Java is an interpreted language which carries a heavy code overhead). • Client-server features, version control inherent in the language	• Web applets. • Internet-based applications. • Applications which must run on various hardware and OS platforms.
ADO.NET	• Good overall performance. • Internet capability. • XML Support • Efficient scalable architecture	• Applications running in a managed environment where runtime is paramount.
SQL/ODBC	• Abstracts the application interface from the database implementation. • Most programming languages, many applications support it. • Relational access only. • Large. Generally slower than direct interface to the native DBMS API. Provides a "generic" interface to an application. • Full relational implementation. • Subset of native functionality. • Standard interface supported by nearly all Windows programming environments and many off-the-shelf applications.	• Applications that require heterogeneous access to different data stores, or which must be independent of any particular data store, should consider ODBC. • Applications where maintaining a relational data store is the primary consideration, but run-time performance is still important.
Zen Direct Access Components	• Replaces Embarcadero Database Engine in Delphi and C++ Builder • Classes to access data from either a transactional or relational context.	• Application development using Embarcadero IDEs.

Database Connection Quick Reference

This section provides links to quick information on how to get connected to a Zen database.

These examples should only supplement the full documentation for each access method. For each access method, there are links to more detailed information.

Each sample references the Course table in the DEMODATA sample database, which is included with Zen.

- ADO.NET Connections
- JDBC Connections
- Java Class Library
- DSN-Less Connections

ADO.NET Connections

For complete information on ADO.NET tasks, see

- Using the Data Providers in *Data Provider for .NET Guide*
- Code examples provided when you install the downloaded ADO.NET sample headers and files

Sample ADO.NET DB Connection Code

```
"ServerDSN=Demodata;UID=test;PWD=test;ServerName=localhost;";
```

JDBC Connections

For complete information on JDBC tasks, see the following topics in *JDBC Driver Guide*:

- Introducing the Zen JDBC Driver
- Programming with the Zen JDBC 2 Driver

Sample JDBC Connection Code

```
Class.forName("com.pervasive.jdbc.v2.Driver");
    Connection con = DriverManager.getConnection("jdbc:pervasive://localhost:1583/DEMODATA");
    PreparedStatement stmt = con.prepareStatement("SELECT * FROM Course ORDER BY Name");
    ResultSet rs = stmt.executeQuery();
```

Java Class Library

For complete information on Java Class Library tasks, see

- Introduction to the Zen Java Interface
- Programming with the Java Class Library

Sample JCL Connection String

```
Session session = Driver.establishSession();
Database db = session.connectToDatabase();
db.setDictionaryLoc("c:\\PVSW\\DEMODATA");
```

DSN-Less Connections

Zen allows an application to perform a DSN-less connection (connecting to the SQL engine without using a DSN):

The following steps should be used when running locally on the server or from a remote client. This method works with Enterprise Server as well as Workgroup engines.

1. SQLAllocEnv
2. SQLAllocConnect
3. SQLDriverConnect: "Driver={Pervasive ODBC Client Interface};ServerName=<ServerName to Resolve>;dbq=@<ServerSide DBName>;"

Example

```
Driver={Pervasive ODBC Client Interface};ServerName=myserver;dbq=@DEMODATA;
```

ODBC Information

The implementation and limitations of the Zen ODBC access method is documented in the *ODBC Guide*. This documentation is shipped with Zen Enterprise Server, Cloud Server, Workgroup, and Client editions.

- For ODBC information, see Zen ODBC Reference in *ODBC Guide*.
- For supported SQL syntax, see SQL Syntax Reference in *SQL Engine Reference*.

Other SQL Access Methods

JDBC

For JDBC programming to the SQL engine, see the following topics in *JDBC Driver Guide*.

- Introducing the Zen JDBC Driver
- Programming with the Zen JDBC 2 Driver

PDAC

Zen Direct Access Components are for Delphi and C++ Builder applications. For more information, see the following topics in *Zen Direct Access Components Guide*.

- Using Direct Access Components
- Direct Access Components Reference

Additional Resources for Application Developers

The following topics provide further reading on Zen concepts.

Conceptual Information

This manual contains database conceptual information on both the MicroKernel Engine and Relational Engine.

Reference Information

Reference information for developers is contained in the various software development kit (SDK) manuals. In the Eclipse Help that can be installed with the database engine, see the topic Developer Reference.

Online Developer Resources

Get information for developers online at the Actian Zen website.

Sample Code

You can find the sample applications in your installation of the SDK component. The available samples include the following:

- Zen Direct Access Components using Delphi or C++ Builder
- Java Programming with the Zen Java Class Library or JDBC
- Distributed Tuning Interface for Visual C++ or Delphi
- Distributed Tuning Objects for Visual Basic

Developing Applications for the MicroKernel Engine

The following topics provide information needed to design your application with the Zen MicroKernel Engine:

- MicroKernel Engine Environment
- Configuration Issues for MicroKernel Engine

MicroKernel Engine Environment

Before an end user can run your MicroKernel Engine application, a version of MicroKernel Engine must be available to the end user's computer. You should provide the end user with information about any prerequisite the MicroKernel Engine software versions and configurations that your application requires.

Documentation

End users should have access to the following Zen documentation:

- *Getting Started with Zen*. Describes Zen software installation.
- *Status Codes and Messages*. Lists the status codes and system messages that Zen components can return.
- *Zen User's Guide*. Describes Zen utilities.

If you are a Zen OEM partner, you may bundle these documents with your application.

Configuration Issues for MicroKernel Engine

End users may need to know the following information about your MicroKernel Engine application. Include this information in the documentation you provide with your MicroKernel Engine application.

- The amount of memory your application requires.

  Your application may require more memory or disk space than the MicroKernel Engine requires on its own. Establish the disk space and memory requirements of your application and communicate this information to your users. For information about system requirements of the MicroKernel Engine, see *Getting Started with Zen* and also visit the Actian website.

- Whether the application requires the MicroKernel Engine configuration settings other than the defaults. In particular, consider whether end users need to change these MicroKernel Engine options:

  - Create File Version. Does your application need backward compatibility with a previous version of the MicroKernel Engine? If so, instruct your end users to set an appropriate value for this option.
  - Handles. Does your application need to use more than 60 logical file handles at one time? If so, instruct your end users to set this option to an appropriate value.
  - Index Balancing. Does your application set the Index Balancing file attribute on every file it creates? If so, your end users can use the default of Index Balancing turned off. If not, you may need to instruct your end users to turn Index Balancing on at the MicroKernel level. For more information, see Index Balancing.
  - Largest Compressed Record Size. Does your application use compressed records? If so, see Record and Page Compression in *Advanced Operations Guide*, and Choosing a Page Size, Estimating File Size, and Record Compression in this documentation.
  - System Data. Do all files in your database have unique keys? If so, the files are transaction durable. If not, your end users may want to set System Data to *If Needed* or *Always* in order to make the files transaction durable.

For descriptions of configuration options, see *Advanced Operations Guide*.

MicroKernel Engine Fundamentals

The following topics describe the features of the MicroKernel Engine:

- Overview of the MicroKernel Engine
- Pages
- File Types
- Data Types
- Key Attributes
- Database URIs
- Double-Byte Character Support
- Record Length
- Data Integrity
- Event Logging
- Performance Enhancement
- Disk Usage

Overview of the MicroKernel Engine

The Btrieve API is a low-level interface to the MicroKernel Engine that embodies functional aspects of the database design that might be otherwise transparent in higher-level interfaces such as SQL, Java, or ODBC. For example, the SQL interface operates independently of how the data is physically stored. However, the MicroKernel Engine developer must consider lower level aspects such as page size, physical and logical currency, type verification, and data validation. Despite these low-level considerations, the Btrieve API provides excellent flexibility and control over the data.

The MicroKernel Engine stores information in files, which can be up to terabytes in size for the 13.0 and 16.0 file versions, (256 GB for the 9.5 version, 128 GB for earlier 9.x versions, and 64 GB for other earlier versions). Inside each data file are *records*, which contain bytes of data. A file can contain quintillions of records.

The data in a record might represent an employee name, ID, address, phone number, rate of pay, and so on. However, the MicroKernel Engine interprets a record only as a collection of bytes; it does not recognize logically discrete pieces of information within a record. To the MicroKernel Engine, a last name, first name, employee ID, and so on do not exist inside a record.

The only discrete portions of information that the MicroKernel Engine recognizes in a record are *keys*. Keys provide both fast, direct access to records and a means of sorting records by key values. Because the MicroKernel Engine has no way of knowing the structure of the records in each file, you define each key by identifying the following:

- Number. This is the key's order in the list of keys. Version 6.0 and later files can have gaps between key numbers. That is, the MicroKernel Engine does not require keys to be numbered consecutively. When you add a key, you can specify a key number or let the MicroKernel Engine assign the lowest available key number. When you drop a key, you can leave the remaining key numbers as is or let the MicroKernel Engine renumber them consecutively.
- Position. This is the key's offset in bytes from the beginning of the record.
- Length. This is the number of bytes to use for the key.
- Type. This is the key's data type.
- Attributes. These provide additional information about how you want the MicroKernel Engine to handle the key values. The MicroKernel Engine supports these key attributes: segmentation, duplicatability, modifiability, sort order, case sensitivity, alternate collating sequence, and null value.

You can create or drop keys at any time. For each key defined in a data file, the MicroKernel Engine builds an *index*. The index is stored inside the data file itself. The index maps each key

value in the file to an offset in the actual data. Normally, when accessing or sorting data, the MicroKernel Engine does not search through all the records in the file. Instead, it searches the index and then manipulates only those records appropriate to the request.

You can create indexes when you create the data file, or any time thereafter. When you create a data file, you can define one or more keys for the MicroKernel Engine to use in building indexes.

You can also define *external indexes* after creating a file. An external index file is a standard data file that contains records sorted by the key you specify. Each record consists of the following:

- An address identifying the physical position of the record in the original data file
- A key value

Positioning rules (guidelines governing which record is current, which is next, and so on) are the same, regardless of when you create an index.

If you create an index at the same time that you create the file, the MicroKernel Engine stores duplicate key values in the chronological order in which the records are inserted into the file. If you create an index for a file that already exists, the MicroKernel Engine stores duplicate key values in the physical order of the corresponding records in the file at the time the index is created. How the MicroKernel Engine stores duplicate key values in an index also depends on whether the key is linked-duplicatable or repeating-duplicatable. For more information, see Duplicatability.

Note: The chronological ordering of records can change when you update records and change their key values, when you drop and rebuild an index, or when you rebuild the file. Therefore, you should not assume that the order of records in a file always reflects the order in which the records were inserted. If you want to track the order of record insertion, use a key of type AUTOINCREMENT.

You can delete, or drop, an index when your application no longer needs it. The space that the index used in the file is freed for data or for other index pages. (However, this free space remains allocated to the file; you will not see a reduction in physical file size after dropping an index.)

See Designing a Database for specific information about defining keys.

MicroKernel Engine Environment

Before an end user can run your MicroKernel Engine application, a version of MicroKernel Engine must be available to the end user's computer. You should provide the end user with information about any prerequisite the MicroKernel Engine software versions and configurations that your application requires.

Configuration Notes

End users may need to know the following information about your MicroKernel Engine application. Include this information in the documentation you provide with your MicroKernel Engine application.

- The amount of memory your application requires.

  Your application may require more memory or disk space than the MicroKernel Engine requires on its own. Establish the disk space and memory requirements of your application and communicate this information to your users. For information about system requirements of the MicroKernel Engine, see *Getting Started with Zen* and the Actian website.

- Whether the application requires the MicroKernel Engine configuration settings other than the defaults. In particular, consider whether end users need to change these MicroKernel Engine options:

  - Create File Version. Does your application need backward compatibility with a previous version of the MicroKernel Engine? If so, instruct your end users to set an appropriate value for this option.
  - Handles. Does your application need to use more than 60 logical file handles at one time? If so, instruct your end users to set this option to an appropriate value.
  - Index Balancing. Does your application set the Index Balancing file attribute on every file it creates? If so, your end users can use the default of Index Balancing turned off. If not, you may need to instruct your end users to turn Index Balancing on at the MicroKernel level. For more information, see Index Balancing.
  - Largest Compressed Record Size. Does your application use compressed records? If so, see Record and Page Compression in *Advanced Operations Guide*, and in this guide see Choosing a Page Size and Record Compression.
  - System Data. Do all files in your database have unique keys? If so, the files are transaction durable. If not, your end users may want to set System Data to *If Needed* or *Always* in order to make the files transaction durable.

For descriptions of configuration options, see *Advanced Operations Guide*.

Pages

The following topics cover pages and how the MicroKernel Engine handles them:

- Page Types
- Page Size

Page Types

Files consist of a series of *pages*. A page is the unit of storage that the database transfers between memory and disk. A file is composed of the following types of pages:

File Control Record (FCR)	Contains information about the file, such as the file size, page size, and other characteristics of the file. The first two pages in every 6.0 and later data file are FCR pages. At any given time, the MicroKernel Engine considers one of the FCR pages to be current. The current FCR page contains the latest file information.
Page Allocation Table (PAT)	Part of the internal implementation of the MicroKernel Engine for tracking pages in a file.
Data	Contains the fixed-length portion of records. The MicroKernel Engine does not split a single fixed-length record across two data pages. If a file does not allow variable-length records or use data compression, the file has data pages and no variable pages.
Variable	Contains the variable-length portion of records. If the variable-length portion of a record is longer than the remaining space on a variable page, the MicroKernel Engine splits the variable-length portion over multiple variable pages. If a file allows variable-length records or uses data compression, the file has both data and variable pages.
Index	Contains key values used in retrieving records.
Alternate Collating Sequence (ACS)	Contains alternate collating sequences for the keys in a file.

All 6.0 and later files have FCR and PAT pages. *Standard* files also contain data and index pages, and optionally, variable and ACS pages. Data-Only Files contain no index pages. Key-Only Files contain no data pages.

Page Size

You specify a fixed page size when you create a file. The page size you can specify, the file overhead, and so forth, depends on a variety of factors, including the file format. See Designing a Database for information on page sizes. The following sections provide an overview:

- Page Size Criteria
- Large vs. Small Page Size

Page Size Criteria

The page size you specify should satisfy the following criteria:

- Enables data pages appropriate to the record length of the file.

  Each data page contains a certain number of bytes for overhead. After that, the MicroKernel Engine stores as many records as possible in each data page, but does not break the fixed-length portion of a record across pages. For more information, see Choosing a Page Size.

  The optimum page size accommodates the most records while minimizing the amount of space left over in each data page. Larger page sizes usually result in more efficient use of disk space. If the internal record length (user data + record overhead) is small and the page size is large, the wasted space could be substantial.

- Allows index pages appropriate to the file key definitions.

  Each index page contains a certain number of bytes for overhead. After that, the file's index pages must be large enough to accommodate eight keys, plus overhead information for each key. See Optimum Page Size For Minimizing Disk Space for information about the number of bytes of overheard per your configuration.

- Allows the number of key segments that the file needs.

  As discussed in Segmentation, the page size you define for a file limits the number of key segments you can specify for that file.

- Optimizes performance.

  For optimum performance, set the page size to an even power of two, such as 512, 1024, 2048, 4096, 8192, or 16384 bytes. The internal MicroKernel Engine cache can store multiple size pages at once, but it is divided in powers of 2. Page sizes of 1536, 2560, 3072, and 3584 actually waste memory in the MicroKernel Engine cache. Page sizes that are powers of 2 result in a better use of cache.

Large vs. Small Page Size

To make the most efficient use of modern operating systems, you should choose a larger page size. The smaller page sizes were used when DOS was the prominent operating system (when a sector was 512 bytes and all I/O occurred in multiples of 512). This is no longer the case. Both 32-bit and 64-bit operating systems move data around their cache in blocks of 4096 bytes or larger. CD ROM drives are read in blocks of 2048 bytes.

The MicroKernel Engine indexes are most efficient when a page size of 4096 bytes or larger is used. The key will have more branches per node and thus will require fewer reads to find the correct record address. This is important if the application is doing random reads using a key. This is not important when an application accesses the file in a sequential manner either by key or by record.

A good reason for having smaller page sizes is to avoid contention. With fewer records in each page it becomes less likely that different engines or transactions will need the same page at the same time. If a file has relatively few records, and the records are small, you may want to choose a small page size. The larger the file, the less likely contention will happen.

Another potential problem with large page sizes is specific to version 7.0 and later files. There is a maximum of 256 records or variable-length sections that can fit on the same data page. If you have short or compressed records, or short variable-length sections, you can easily reach the limit while you still have hundreds of bytes available on every page. The result is a much larger file than needed. By knowing your record size, you can calculate how big of an issue this is.

Factors To Consider When Determining Page Size

- Keys work better with **larger** pages. There are more branches per B-tree node and thus fewer levels to the B-tree. Fewer levels means fewer disk reads and writes. Fewer disk reads means better performance.
- Concurrency works better with **smaller** pages, especially when client transactions are used. Since the MicroKernel Engine locks some pages changed during the transaction, all other clients must wait for locked pages until the transaction is ended or aborted. With a lot of clients trying to access the same pages concurrently, the less that is found on each page is better.
- Random access to pages works better with **smaller** pages since more of the stuff you actually use is in cache. If you access anything again, it is more likely to be still in cache.
- Sequential access to a large volume of records works better with **larger** pages since more is read at once. Since you are using most everything on each page read, there will definitely be fewer reads.

The database designer must choose between these conflicting needs. A reference table that is not changed very often, but is searched or scanned most of the time, should have larger page sizes. A transaction file which is inserted and updated within transactions should have smaller page sizes.

Only careful consideration of all factors can give the right answer to what the page size should be. For more information about choosing a page size, see Choosing a Page Size.

File Types

The Btrieve API supports three data file types, large files to a maximum file size of terabytes for the 13.0 and 16.0 file versions, 256 GB for 9.5 files, 128 GB for earlier 9.x versions, and 64 GB for other earlier versions, as well as long file names. The following topics cover these features:

- Standard Data Files
- Data-Only Files
- Key-Only Files
- Large Files
- Long File Names

Note: For users of Btrieve 6.x and earlier, the MicroKernel Engine can create files in 8.x and 7.x formats. These newer formats allow for enhancements and new features.

Btrieve 6.x and earlier cannot open files for versions 7.0 or later. However, later Btrieve releases can open pre-7.0 files. When it opens these files, it does *not* convert them to later file formats. Also, you can configure MicroKernel Engine compatibility to create 7.x or 6.x files if you need newly created files in those formats.

Standard Data Files

A standard 7.x or later data file contains two FCR pages followed by a number of PAT pages, index pages, data pages, and possibly variable and ACS pages. You can create a standard file for use with either fixed- or variable-length records. Because standard files contain all the index structures and data records, The MicroKernel Engine can dynamically maintain all the index information for the records in the file.

Data-Only Files

When you create a data-only file, you do not specify any key information, and Zen does not allocate index pages for the file. This results in a smaller initial file size than for standard files. You can add keys to a data-only file after creating the file.

Key-Only Files

Key-only files contain only FCR pages followed by a number of PAT pages and index pages. In addition, if you have defined referential integrity constraints on the file, the file may contain one or more variable pages.

Key-only files include only one key, and the entire record is stored with the key, so no data pages are required. Key-only files are useful when your records contain a single key and that key takes up most of each record. Another common use for a key-only file is as an external index for a standard data file.

The following restrictions apply to key-only files:

- Each file can contain only a single key.
- The maximum record length you can define is 253 bytes (255 bytes for a pre-6.0 file).
- Key-only files do not allow data compression.

Large Files

The MicroKernel Engine supports file sizes up to terabytes for the 13.0 and 16.0 file versions, 256 GB for 9.5 files, 128 GB for earlier 9.x versions, and 64 GB for other earlier versions. However, many operating systems do not support single files this large. In order to support files larger than the operating system file size limit, the MicroKernel Engine breaks up large files into smaller files that the operating system can support. A large, logical file is called an *extended file*. The smaller, physical files that comprise an extended file are called *extension files*. The *base file* is an original data file that has become too large to support as a single, physical file. Nonextended (that is, nonsegmented) files provide more efficient I/O and, therefore, increased performance.

You can choose to *not* to automatically extend 9.x format files or later at 2 GB. To change the segment operation setting access the configuration settings in the Zen Control Center (ZenCC) as described in Configuration Using ZenCC in *Advanced Operations Guide*. From there you can set the **Limit Segment Size to 2 GB** option. This setting as no effect on 13.0 and 16.0 files, which are never segmented.

If this option is unselected, PSQL 9.x files will not be segmented automatically at 2 GB. Data files from version 8.x and earlier will continue to be extended when they reach 2 GB. If your files are already extended, they will remain segmented.

Regardless of the configuration setting, all files will continue to be extended based on the file size limitations of the current operating system.

For information about backing up files, including extended files, see Backing Up Your Files.

Long File Names

The MicroKernel Engine supports long file names whose length is less than or equal to 255 bytes. The following items must conform to this upper limit:

- The localized multi-byte or single byte version of the string
- The UNC version of the file name that is created by the requesters, which is in UTF-8 UNICODE format.

The file name can contain spaces unless the Embedded Spaces client configuration option is disabled. The default setting is **On**. See *Advanced Operations Guide* (Long File Names and Embedded Spaces Support).

When the MicroKernel Engine generates new files based on an existing file name, such as with Large Files or during Archival Logging or Continuous Operations (for more information, see Logging, Backup, and Restore in *Advanced Operations Guide*), the new file name includes as much of the original file name as possible and an appropriate file extension, as in the following examples:

Original File Name	Generated File Name in Continuous Operation
LONG-NAME-WITHOUT-ANY-DOTS	LONG-NAME-WITHOUT-ANY-DOTS.^^^
VERYLONGNAME.DOT.DOT.MKD	VERYLONGNAME.DOT.DOT.^^^

Data Types

When using 7.*x* or a later file format, you can use the following data types when you define a key:

AUTOINCREMENT	BFLOAT	CURRENCY
DATE	DECIMAL	FLOAT
INTEGER	LSTRING	MONEY
NUMERIC	NUMERICSA	NUMERICSTS
TIME	TIMESTAMP	UNSIGNED BINARY
ZSTRING	WSTRING	WZSTRING
NULL INDICATOR		

If you are using the 6.x file format, you can use all the preceding types to define a key except for CURRENCY, TIMESTAMP, and WZSTRING.

If you are using a file format before 6.x, NUMERICSA and NUMERICSTS are not available as data or key types.

For more information, see Data Types in *SQL Engine Reference*.

Key Attributes

The following sections describe the attributes you can assign when you define a key:

- Key Attributes Description
- Key Specification

Key Attributes Description

These topics contains information on attributes that you can assign to keys:

- Segmentation
- Duplicatability
- Modifiability
- Sort Order
- Case Sensitivity
- Null Value
- Alternate Collating Sequences

Segmentation

Keys can consist of one or more *segments* in each record. A segment can be any set of contiguous bytes in the record. The key type and sort order can be different for each segment in the key.

The number of index segments that you may use depends on the file's page size.

Page Size (bytes)	Maximum Key Segments by File Version			
	8.x and earlier	9.0	9.5	13.0, 16.0
512	8	8	Rounded up[2]	Rounded up[2]
1024	23	23	97	Rounded up[2]
1536	24	24	Rounded up[2]	Rounded up[2]
2048	54	54	97	Rounded up[2]
2560	54	54	Rounded up[2]	Rounded up[2]
3072	54	54	Rounded up[2]	Rounded up[2]

Page Size (bytes)	Maximum Key Segments by File Version			
	8.x and earlier	9.0	9.5	13.0, 16.0
3584	54	54	Rounded up[2]	Rounded up[2]
4096	119	119	204[3]	183[3]
8192	n/a[1]	119	420[3]	378[3]
16384	n/a[1]	n/a[1]	420[3]	378[3]

[1]"n/a" stands for "not applicable"

[2]"Rounded up" means that the page size is rounded up to the next size supported by the file version. For example, 512 is rounded up to 1024, 2560 is rounded up to 4096, and so forth.

[3]A 9.5 format or later file can have more than 119 segments, but the number of indexes is limited to 119.

See status codes 26: The number of keys specified is invalid and 29: The key length is invalid for related information about index segments and the MicroKernel Engine.

The total length of a key is the sum of the length of the key segments, and the maximum length is 255 bytes for 13.0 format files and earlier, or 1024 bytes for 16.0 format files. Different key segments can overlap each other in the record.

When a segmented key is a nonduplicatable key, the combination of the segments must form a unique value; however, individual segments may contain duplicates. When you are defining this type of segmented key, each segment has `duplicates=no` as a key-level attribute even though that particular segment may have duplicates. To ensure that a particular segment is always unique, define it as a separate nonduplicatable key in addition to the segmented key definition.

When issuing a call to the MicroKernel Engine, the format of the key buffer must be able to accommodate the key specified by the key number. So, if defined `keynumber=0` and key 0 is a 4-byte integer, the key buffer parameter can be any of the following:

- A pointer to a 4-byte integer
- A pointer to a structure where the first (or only) element is a 4-byte integer
- A pointer to a 4-byte (or longer) string or byte array

Basically, the MicroKernel Engine gets a pointer to a memory location to be used as a key buffer. The MicroKernel Engine expects that memory location to have a data value corresponding to the specified key number for certain operations, such as Get Equal. In addition, the MicroKernel Engine may write data out to that location, and the data written will be a key value corresponding

to the specified key number. In this situation, the memory location must be allocated large enough to accommodate the entire key value.

To the MicroKernel Engine, a key is a single collection of data, even if it is made up of multiple segments. The segment feature allows you to combine noncontiguous bytes of data together as a single key. It also allows you to apply different sorting rules (as dictated by the supported data types) to different portions of the key data. The data type associated with a key segment is used typically as a sorting rule – it tells the MicroKernel Engine how to compare two values to determine which one is larger. Data types are not used to validate data.

The MicroKernel Engine always deals with an entire key, not a key segment. To work with any key, set up a key buffer that is large enough to hold the entire key. Some applications define a generic 255 byte buffer to use on all calls to the MicroKernel Engine; this is the maximum size of a key in 13.0 format files and earlier, and is often sufficient. When data is returned in this key buffer, the application usually copies data out of the generic buffer into an application variable or structure declared as the same type(s) as the key segment(s). Alternatively, pass a key buffer parameter (simple variable or structure variable) that directly corresponds to the key.

For example, suppose you want to read a record and only know the value of the first segment of the key, but not all segments. You can still utilize that key to find the data. However, you still have to pass in an entire key buffer corresponding to all segments. Because you only know part of the key value, you cannot use the Get Equal call. You have to use the Get Greater Or Equal call. In this case, initialize the key buffer with as many key values as you know and then specify low or null values for the unknown key segments.

For example, given a key 1 definition of three segments corresponding to data values ulElement2, ulElement3, and ulElement5, if you know what value you want for ulElement2, you would initialize your key buffer as:

```
SampleKey1.ulElement2 = <search value>;
SampleKey1.ulElement3 = 0;
SampleKey1.ulElement5 = 0;
```

and then pass &SampleKey1 as the key buffer parameter on a Get Greater Or Equal call. When the MicroKernel Engine completes the call and a record is found, status code 0 is returned, the corresponding data record is returned, and the key buffer is set to have the key value including all three segments.

Duplicatability

Zen supports two methods for handling duplicate key values: linked (the default) and repeating. With linked-duplicatable keys, the MicroKernel Engine uses a pair of pointers on the index page to identify the chronologically first and last records with the same key value. Additionally, the MicroKernel Engine uses a pair of pointers in each record on the data page to identify the

chronologically previous and next records with the same key value. The key value is stored once, and only on the index page.

With repeating-duplicatable keys, the MicroKernel Engine uses a single pointer on the index page to identify the corresponding record on the data page. The key value is stored on both the index page and the data page. For more information on duplicate keys, see Duplicatable Keys.

Modifiability

If you define a key as modifiable, the MicroKernel Engine enables you to change the value of a key even after the record is inserted. If one segment of a key is modifiable, all the segments must be modifiable.

Sort Order

By default, the MicroKernel Engine sorts key values in ascending order (lowest to highest). However, you can specify that the MicroKernel Engine order the key values in descending order (highest to lowest).

Note: Use caution when using descending keys with the MicroKernel Engine Get operations (Get Greater (8), Get Greater or Equal (9), Get Less Than (10), and Get Less Than or Equal (11)). In this context, Greater (or Less) refers to the order with respect to the key; in the case of a descending key, this order is the opposite of the corresponding ascending key.

When you perform a Get Greater (8) operation on a descending key, the MicroKernel Engine returns the record corresponding to the first key value that is lower than the key value you specify in the key buffer. For example, consider a file that has 10 records and a descending key of type INTEGER. The actual values stored in the 10 records for the descending key are the integers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. If the current record's key value is 5 and you perform a Get Greater operation, the MicroKernel Engine returns the record containing the key value 4.

Similarly, when you perform a Get Less Than (10) operation using a descending key, the MicroKernel Engine returns the record with the next higher value than the one you specify in the key buffer. Using the preceding example, if the current record's descending key has a value of 5 and you perform a Get Less Than operation, the MicroKernel Engine returns the record containing the key value 6.

Case Sensitivity

By default, the MicroKernel Engine is case sensitive when sorting string keys; that is, it sorts uppercase letters before lowercase letters. When you define a key to be case insensitive, the

MicroKernel Engine sorts values without distinguishing case. Case sensitivity does not apply if the key has an alternate collating sequence (ACS).

Null Value

Zen contains two ways of identifying a column of data as a Null value. The original type of null value, called a legacy null, has been used in the MicroKernel Engine for years. The newer type of null is the true null. This topic describes the legacy null and then details use of true nulls in the MicroKernel Engine.

Legacy Null

The original method of defining a nullable field is referred to as *pseudo-null* or *legacy-null*. It is based on the premise that if the entire field is full of a particular byte value, typically ASCII zero, then the field is considered Null. The byte value is defined in the key definition supplied when creating the index. Using the MicroKernel Engine, the only thing that the MicroKernel Engine can do with this knowledge is to decide whether or not to include the field in an index. There are no special sorting rules for Legacy Nulls since they are just values that sort just like all the other values, despite their special meaning.

If a key description contains the flag for All Segments Null (0x0008), then the key value is not put into the index if each and every segment in the key is considered to be null by having the null byte in every byte of the field. Likewise, if a key description contains the flag for Any Segment Null (0x0200), then the key value is not put into the index if one or more key segment is considered to be Null by the same rule

The Relational Engine never uses these flags in the indices that it defines, since in order to make joins between tables, all records in the table must be accessible through the index.

True Null Indexes

Starting with Pervasive.SQL 2000, a new type of Null Indicator was introduced called *true null*.

True nulls are implemented in the MicroKernel Engine by preceding a nullable field with a one byte Null Indicator Segment (NIS). This is an extra byte of data outside the normal column width that indicates whether the column is Null or not. A value of zero in this byte indicates that the column associated with it is normal, or Not Null. Any other value in this byte indicates that the column value is Null.

With true nulls, unlike legacy nulls, you can tell the difference between an integer that is zero and one that is Null. This is true for any type of number field. You can even distinguish a string field that has a zero length string in it from one that should be identified as null, if there is a need for such a distinction.

The Relational Engine can identify and use true null columns whether or not there is an index defined on them, but a basic data file only identifies the fields that are included in keys.

You can define true null fields within MicroKernel Engine keys by adding a Null Indicator Segment (NIS) before the nullable field in the key definition of a Create (14) or Create Index (31) operation. See Rules for True Null Keys for the rules regarding true null keys.

The MicroKernel Engine does not enforce any restrictions on the offset of the NIS, whereas the Relational Engine assumes that it immediately precedes the nullable field. As such, it is recommended to structure the fields within your record to make room for the NIS in the byte preceding any field that will use an NIS. This preserves your ability to access these tables through SQL should you need to do so.

Rules for True Null Keys

The following rules must be followed when using this new key type:

1. The field length must be 1.
2. The field must precede another field in the index. In other words, this must be a multi-segmented index with the NIS being defined immediately before another segment. The NIS cannot be the last or only key segment.
3. The field immediately following it is affected by the contents of the NIS. If the NIS is zero, then the following field is considered nonnull. If this field is anything other than zero, the field is considered NULL.
4. The offset of the NIS should be the byte preceding the following field. This is the way the Zen relational engine expects these fields to align. Therefore, if a data dictionary is created for this index, the NIS should be immediately preceding the field it controls. That said, there is nothing in the transactional API that makes this a requirement.

NIS Values

Any nonzero value is considered an indicator that the following segment is null. By default, the MicroKernel Engine makes no distinction between nonzero numbers. The Zen relational engine currently uses only a value of 1 in this field to indicate a null. It is possible, however, to make a distinction between different types of nulls. This can be done by using the Case Insensitive flag on the NIS. Since this key flag is normally only applicable to the various string and character fields, it is overloaded to have the special meaning of DISTINCT when used with an NIS. It means that different NIS values should be treated distinctly and sorted separately. Actian Corporation reserves the use of the first 15 values for future use. If you want to apply a special meaning to various types of nulls in your application, please use NIS values greater than 16. For example, more specific null definitions could be:

- Not applicable
- To be determined
- Cannot be determined
- Undetectable
- No value yet, but needed soon

When you add the DISTINCT flag (Case Insensitive) to the NIS, these nonzero values will be sorted separately as distinct values.

Sorting of True Null Values

A true null field has a nondeterminate value. In other words, its value cannot be known. According to this definition, no two Null values are equal to each other, nor are they equal to any other key value. Yet the MicroKernel Engine must group these Null values together and you must be able to find key values that are equal to Null. To accomplish this, the MicroKernel Engine interprets true null values as if they *are* equal to each other, depending on the purpose of the comparison. When sorting, and finding a place for the Null values in an index, true null values are grouped together as if they were equal to each other. But when trying to determine if a value already exists in a unique index, true nulls are *not* equal to each other.

Any nonzero value in the NIS means the following field is Null. The default behavior is to treat all nonzero values in the NIS as if they were the same value and interpret them to indicate that the nullable field is Null. As such, if you insert records that contain a variety of nonzero values in the NIS and a variety of values in the nullable field that follows, they will all be interpreted as the same value, and will be sorted as a collection of duplicates.

Linked Duplicate Keys and True Nulls

This section discusses the results of inserting several Null values into a Linked Duplicate key into a Linked Duplicate key.

Linked Duplicates contains a single key entry for each unique value, with two record address pointers; one for the first duplicate record and one for the last record in the duplicate chain. Each record contains 8 bytes of overhead consisting of pointers to the previous and next records in the chain. Each new duplicate value is added at the end of the chain, thus ensuring that the duplicate records are linked in the order they were inserted. All true null values are considered duplicates for the purpose of adding them to an index, so they all will be linked to the same chain in the order they were inserted. Even if each record contained different byte values in the NIS and the associated nullable field, there will only be one key entry pointing to the first and last record in this chain. If the NIS key segment is defined as descending, this key entry will occur first in the index. Otherwise, it will occur last.

Repeating Duplicate Keys and True Nulls

Repeating Duplicate Keys contain an actual key entry for each record represented in the index. There is no overhead in the record itself and for each record, there is a key entry that points to it. Duplicate values in this kind of index are sorted by the physical record address to which they point. This means that the order of duplicates is unpredictable, especially in a highly concurrent environment where random records are being inserted and deleted by many clients.

True Null values are interpreted as if they are duplicates and are sorted not by the bytes found in the nullable field, but rather by the record address. So when using repeating duplicate keys, the records containing true null values are grouped together, but in a random fashion. If the NIS segment is descending, they will occur first in the index, otherwise, they will occur last.

Unique Keys and True Nulls

In the MicroKernel Engine, if you define an index without either duplicate flag, the index must contain only unique values. But since the value of a true null field is indeterminate, they should not be considered duplicates. For this reason, the MicroKernel Engine allows multiple true null values to be entered into a unique key, assuming that once the value is assigned with an update operation, then the uniqueness of the key can be determined. But for the purposes of sorting these values in the index, the MicroKernel Engine groups them all together as if they were duplicates. So the section of the index containing the true null values resembles a Repeating Duplicate index. The nulls are sorted together according to the physical record address, the order of which cannot be predicted.

Nonmodifiable Keys and True Nulls

Once you put a value into a nonmodifiable key, it cannot be changed. But because a true null value does not have an actual value, the MicroKernel Engine allows you to insert a record with a true null value in any or all fields defined in true null indexes, and then later change those field values in an update operation from null to nonnull. But once any field has become nonnull, the nonmodifiability is enforced and it cannot be changed again, even if to establish the field as null again.

Get Operations and True Nulls

Even though true null values are indeterminate and are not considered equal to each other, it is possible to locate a record with a true null key segment.

The various Get operations can address true null keys by using this sequence:

1. Place the nonzero value in the NIS byte
2. Place the full key into the Key Buffer

3. Perform a Get operation as if true null values *are* equal to each other.

The following list shows the expected behavior from the Get operations:

- Get Equal and Get Greater Than or Equal will return the first record with a null in the forward direction.
- Get Less Than or Equal will return the last record with a null as viewed from the forward direction.
- Get Less Than will return the record before the null values
- Get Greater Than will return the record after the null values.

This is consistent with the behavior of the Get operations for normal duplicate values.

Distinct True Nulls

It is possible to distinguish between different values in the NIS byte. The default behavior, as indicated, is that all nonzero values in the NIS are considered to be the same thing, and whatever the NIS contains, if it is not zero, the nullable field is Null. The Relational Engine currently uses this default behavior on all true null index segments that it creates.

However, if you want to store different kinds of Null values in your table, then you can add the NOCASE flag (0x0400) to the key definition of the NIS segment. Hereafter, we will call this the DISTINCT flag. When you do this, the MicroKernel Engine will treat different NIS values as different or distinct from each other.

Distinct True Null segments are sorted in groups by their NIS value. The same rules apply as discussed above when building the various types of indexes. A linked duplicate key will have a single entry for each distinct NIS value with a pointer to the first and last occurrence of that type of Null. Repeating Duplicates and Unique keys will also group the null records by their distinct NIS value. Descending Keys have the highest NIS values grouped first, sorted down to the zero, or nonnull values. Ascending keys sort the nonnull records first, followed by NIS values of 1, then 2, and so on. Get operations pay attention to the value of the NIS. If you do a GetEQ using a key buffer where the NIS is 20, and all the NIS values in a Distinct True Null index are 1, then the MicroKernel Engine will not find any matching values.

Although the Relational Engine nor any Zen access method currently uses the DISTINCT flag when creating true null indexes, they might in the future. For this reason, NIS values 2 through 16 are reserved for future use, in case Zen needs to assign specific meanings to these 'types' of nulls. So if you use distinct null values for records accessed through the transactional Btrieve API, use values greater than 16.

Multi-Segmented True Null Keys

Consider a multi-segmented True Null index containing two nullable string columns. The key would actually be defined as a four segment index. The first segment is an NIS, followed by the first nullable field, then the second NIS followed by the second nullable field. Now consider what would happen if the following records were put into the file.

```
"AAA",  NULL "BBB", NULL "CCC",   NULL NULL, NULL
"AAA", "AAA" "BBB", "AAA" "CCC", "AAA" NULL, "AAA"
"AAA", "BBB" "BBB", "BBB" "CCC", "BBB" NULL, "BBB""
"AAA", "CCC" "BBB", "CCC" "CCC", "CCC" NULL, "CCC"
```

plus a couple more of these records; "BBB", NULL

The Relational Engine always creates True Null index segments such that the NULL values will occur first. It does this by adding the Descending flag (0x0040) to each NIS segment. Let's assume that the descending flag is used on each NIS and on the second nullable field, but not the first nullable field. If so, these records would be sorted like this.

```
1           NULL, NULL
2           NULL, "CCC "
3           NULL, "BBB""
4           NULL, "AAA "
5          "AAA", NULL
6          "AAA", "CCC"
7          "AAA", "BBB"
8          "AAA", "AAA"
9          "BBB", NULL
10         "BBB", NULL
11         "BBB", NULL
12         "BBB", "CCC "
13         "BBB", "BBB"
14         "BBB", "AAA "
15         "CCC", NULL
16         "CCC", "CCC "
17         "CCC", "BBB"
18         "CCC", "AAA "
```

The nulls always occur before the nonnulls since both NIS are descending. But when the NIS is zero, i.e, the fields are nonnull, the first field is sorted ascending and the second is sorted descending.

The following is what would be returned by various Get operations;

GetLT "BBB", NULL returns record 8 "AAA", "AAA"

GetLE "BBB", NULL returns record 11 "BBB", NULL

GetEQ "BBB", NULL returns record 9 "BBB", NULL

GetGE "BBB", NULL returns record 9 "BBB", NULL

GetGT "BBB", NULL returns record 12 "BBB", "CCC "

The GetLE has the implication that you are looking to traverse the file in the reverse direction, so it returns the first occurrence of a key value that "matches" in the reverse direction. GetEQ and GetGE imply that you are moving in the forward direction.

Excluding Records from an Index

As with legacy nulls, you can also apply the flag for "All Segments Null" (0x0008) or "Any Segment Null" (0x0200) to each segment of any index containing an NIS. When you insert a record, the MicroKernel Engine will determine of the nullable field is Null using the NIS. The same rules apply to determine if the key entry will be put into the index or not.

Note: Files created by the Relational Engine do not use these flags.

So you should not use these flags if you think that you might at some point want to access these files from SQL, where a goal might be to find any records "where column IS NULL". The Relational Engine will use the index to find the null records, but they will not be accessible through the index.

Use of Null Indicator Segment in Extended Operations.

Extended operations allow your application to access fields in a table even if they do not have indexes created for them. You can apply a filter to the fields in your record, defining fields on the fly, using knowledge of the record from any source. Thus it is possible to define True Null fields in an extended operation and have the MicroKernel Engine apply the same comparison rules that it would when sorting these fields into an index.

You must define the extended operations filter just like you would define a key. Include a filter segment for the NIS followed by the nullable field. You must include the nullable field in the filter even if you are searching for a null value, where the content of the nullable field does not matter. The MicroKernel Engineneeds both filter segments so that a GetNextExtended can be optimized against an index path and it enforces this with status 62, indicating that a filter expression for an NIS was not followed by a non-NIS.

The only comparison operator you can use for an NIS is EQ or NE. You will get status 62 if you try to use any of the other comparison operators GT, GE, LT, and LE.

A status 62 occurring from a badly formed extended operation descriptor adds a system error Zen Event Log. These system errors are listed under True Nulls and Extended Operations to help you identify the reason for the status 62.

If you want to treat different NIS values distinctly, then add 128 to the comparison operator on the NIS field. This is the same bias value that you would use to indicate case insensitivity. And just like when defining an index, the case insensitive flag has been overloaded for Null Indicator key

types to indicate that the nonzero values should be compared distinctly meaning that they should be distinguished from one another instead of treating them all the same.

If you are using the extended operations to get the best performance possible, you will be trying to search along a key path for specific limited ranges of key values. First, establish currency at the beginning of the range by using GetGE. Then follow that with GetNextExtended. Or, you can do a GetLE followed by GetPrevExtended. These extended operations can stop searching automatically when there is no more chance of finding any more values that match the filter. This is called extended operation optimization. If your filter can use optimization, it will be much more efficient because there may be a huge number of records that can be skipped and not read from the file. In order to create an optimized search, you need to be traversing the index in a direction where a limit exists. Also, your filter must exactly match the index, using AND instead of OR as segment connectors.

If you do a GetNextExtended on an ascending index, then an optimized filter can stop at the limit when the conditional operator is EQ, LT or LE. A search will have to look to the end of the file for values greater than a particular value going forward along an ascending index. Likewise, if it is a descending index, then it can stop at a limit when the conditional operator is EQ, GT or GE. This can get much more complicated when there are multiple fields in the search criteria. The simple way to think about it is that in order to optimize a filter, only the last segment can have any other conditional operator than EQ. This includes the NIS. If the conditional operator on an NIS is NE, the filter can only be optimized up to the previous filter segment.

Exactly matching the index means that each filter expression should follow the order of the segments in the index and have the same offset, length, key type, case sensitivity (or distinct flag), and ACS specification. Otherwise, extended operations cannot be optimized.

True Nulls and the SQL Engine

True nulls are implemented in the Relational Engine through the use of the Null Indicator key type and follow the rules described above. The MicroKernel Engine applications can also use this key type to identify the nullness of a nullable field regardless of its contents. This provides a way to identify null integers and other number data types, and fully manage these nullable fields.

True Nulls and Extended Operations

Status 62 occurring on an extended operation indicates that the descriptor is incorrect. Sometimes, it may be difficult to determine what exactly is wrong with the descriptor. The database engine adds a line in the Zen event log that can be used to determine the exact problem. The event log entry will look similar to the following:

```
05-12-2019 11:12:45 W3MKDE         0000053C zenengnsvc.exe    MY_COMPUTER       E
System Error: 301.36.0  File: D:\work\test.mkd
```

The following numbers immediately after the system error 301 through 318 identify the problem.

System Error	Description
301	The descriptor length is incorrect.
302	The descriptor ID must be either EG or UC.
303	One of the field types is not valid.
304	The NOCASE flag on the operator can be used only with string and Null Indicator types.
305	The ACS flags (0x08 and 0x20) on the operator can only be used with string types.
306	An unbiased operator is equal to zero.
307	An unbiased operator is greater than six.
308	An invalid expression connector was found. Only 0, 1, and 2 are allowed.
309	The ACS is not defined.
310	The last expression needs a terminator.
311	A terminator was found before the last expression. The filter segment count may be wrong.
312	The number of records to extract is zero.
313	One of the extractor field lengths is zero.
314	A Null Indicator Segment must be followed by another field.
315	A Null Indicator Segment must be connected to the next segment with an AND
316	A Null Indicator Segment can only be used with EQ or NE.
317	A Null Indicator Segment can not follow another NIS.
318	A field following a Null Indicator Segment can not be longer than 255 bytes.

Alternate Collating Sequences

You can use an alternate collating sequence (ACS) to sort string keys (types STRING, LSTRING, and ZSTRING) differently from the standard ASCII collating sequence. By using one or more ACS files, you can sort keys as follows:

- By your own user-defined sorting order, which may require a sorting sequence that mixes alphanumeric characters (A-Z, a-z, and 0-9) with nonalphanumeric characters (such as #).
- By an international sorting rule (ISR) that accommodates language-specific collations, including multibyte collating elements (such as *ll* in Spanish), diacritics (such as *ô* in French), and character expansions and contractions (such as *ß* expanding to *ss* in German).

Files can have a different ACS for each key in the file, but only one ACS per key. Therefore, if the key is segmented, each segment must use either the ACS specified for that key or no ACS at all. For a file in which a key has an ACS designated for some segments but not for others, the MicroKernel Engine sorts only the segments that specify the ACS.

User-Defined ACS

To create an ACS to sort string values differently from the ASCII standard, use the format shown in the following table.

Offset	Length	Description
0	1	Signature byte. Specify 0xAC.
1	8	A unique 8-byte name that identifies the ACS to the MicroKernel Engine.
9	256	A 256-byte map. Each 1-byte position in the map corresponds to the code point having the same value as the position's offset in the map. The value of the byte at that position is the collating weight assigned to the code point. For example, to force code point 0x61 (a) to sort with the same weight as code point 0x41 (A), place the same values at offsets 0x61 and 0x41.

Because ACS files are created using a hex editor or in a MicroKernel Engine application, they are useful mainly to application developers and not usually created by end users.

Here is an example of a 9-byte header and a 256-byte body that represent a collating sequence named UPPER. The header appears as follows:

```
AC 55 50 50 45 52 20 20 20
```

The 256-byte body looks like this, with the exception of the offset values in the leftmost column:

```
00: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
30: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
40: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
50: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
60: 60 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
70: 50 51 52 53 54 55 56 57 58 59 5A 7B 7C 7D 7E 7F
80: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
90: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
A0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
B0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
C0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
D0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
E0: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
F0: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

The header and body forming this ACS are shipped with Zen as the file UPPER.ALT. UPPER.ALT provides a way to sort keys without regard to case. (You can define a key to be case-insensitive; even so, UPPER provides a good example when writing your own ACS.)

Offsets 0x61 through 0x7A in the example have been altered from the standard ASCII collating sequence. For standard ASCII, offset 0x61 contains a value of 0x61 (lowercase *a*). When a key is sorted with the UPPER ACS, the MicroKernel Engine sorts lowercase *a* (0x61) with the collation weight at offset 0x61: 0x41. Thus, the lowercase *a* is sorted as if it were uppercase *A* (0x41). Therefore, for sorting purposes UPPER converts all lowercase letters to their uppercase equivalents when sorting a key.

The following 256-byte body performs the same function as the UPPER.ALT body except that ASCII characters preceding the ASCII space (0x20) are now sorted *after* all other ASCII characters:

```
00: E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
10: F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
20: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
30: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
40: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
50: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
60: 40 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
70: 30 31 32 33 34 35 36 37 38 39 3A 5B 5C 5D 5E 5F
80: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
90: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
A0: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
B0: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
C0: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
D0: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
E0: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
F0: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
```

In this body, different collating weights have been assigned so that a character's weight no longer equals its ASCII value. For example, offset 0x20, representing the ASCII space character, has a collating weight of 0x00; offset 0x41, representing the ASCII uppercase *A*, has a collating weight of 0x21.

To sort keys without regard to case, offsets 0x61 through 0x7A in the last example have been altered. As in the body for UPPER.ALT, offset 0x61 has the same collating weight as offset 0x41: 0x21. By having the same collating weight, offset 0x41 (A) sorts the same as offset 0x61 (a).

International Sort Rules

To specify an ACS that sorts string values using an ISO-defined, language-specific collating sequence, you must specify an ISR table name, as shown in the following examples:

Locale/Language	Code Page	ISR Table Name
US/English	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_ENUS00437_0 PVSW_ENUS00850_0
France/French	437 MS-DOS Latin-US 850 MS-DOS-Latin-1	PVSW_FRFR00437_0 PVSW_FRFR00850_0
Germany/German	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_DEDE00437_0 PVSW_DEDE00850_0
Spain/Spanish	437 MS-DOS Latin-US 850 MS-DOS Latin-1	PVSW_ESES00437_0 PVSW_ESES00850_0
Japan/Japanese	932 Shift-JIS	PVSW_JPJP00932_1[1]

[1]Use PVSW_JPJP00932_1 to ignore Japanese Dakuten in the sort order.

The ISR tables are installed with Zen and based on ISO-standard locale tables. ISR tables are stored in the collate.cfg file, which is installed with the Zen database engine. Data files can share a single ISR.

For examples, see Sample Collations Using International Sorting Rules.

Key Specification

When you create an index using either Create (14) or Create Index (31), you must provide the key specification structure.

Key Specification Block

The following table shows the data buffer structure of the key specification. Each key specification block is 16 bytes for BTRV type entry points or 24 bytes for BTRVEX types. Where two data types are given, but first is used with BTRV and the second with BTRVEX.

Field	Data Type[1]	Length	NIS Segment	Description
Key Position	Short Int[2]	2	Any offset in fixed-length part of record.	The relative position of the key within the record.

Field	Data Type[1]	Length	NIS Segment	Description
Key Length	Short Int[2]	2	1	The length of the key.
Key Flags	Short Int[2]	2	xxxxxxx1xxx1xxxx FEDCBA9876543210	See Key Flag Values.
Reserved (wide)	Byte	0 or 2	N/A	Absent for BTRV type entry points, and 2 bytes for BTRVEX types. Not used for Create (14). Initialize to 0 to maintain backward compatibility.
Unique Keys	Int or Long Long Int[2]	4 or 8	N/A	Four bytes for BTRV type entry points or 8 bytes for BTRVEX types. Not used for Create (14). Initialize to 0 to maintain backward compatibility.
Extended Data Type	Byte or Short Int[2]	1 or 2	255 (0xFF)	Specify one of the extended data types. A new data type is defined for NULL_INDICATOR. One byte of type Byte for BTRV type entry points or 2 bytes of type Short Int for BTRVEX types.
Null Value (non-indexing value)	Byte	1	N/A	Specify an exclusion value for the key.
Reserved	Short Int[2] or Byte	2 or 1	N/A	Two bytes for BTRV type entry points or 1 byte for BTRVEX types. Not used for Create (14). Initialize to 0 to maintain backward compatibility.
Manually Assigned Key Number	Byte or Short Int[2]	1 or 2		A key number. One byte of type Byte for BTRV type entry points or 2 bytes of type Short Int for BTRVEX types.
ACS Number	Byte	1	N/A	The Alternate Collating Sequence (ACS) number.
Reserved (wide)	Byte	0 or 1	N/A	Absent for BTRV type entry points, and 1 byte for BTRVEX types. Not used for Create (14). Initialize to 0 to maintain backward compatibility.

Field	Data Type[1]	Length	NIS Segment	Description

[1]Unless specified otherwise, all data types are unsigned.

[2]Integers must be stored in little-endian byte order, which is the low-to-high ordering of Intel-class computers.

Key Flag Values

The following table gives values for key flags in the key specification block.

Attribute	Binary	Hex	Description
Duplicate	0000 0000 0000 0001	0x0001	
Modifiable	0000 0000 0000 0010	0x0002	
Binary	0000 0000 0000 0100	0x0004	
Null Key (All Segments)	0000 0000 0000 1000	0x0008	
Segmented	0000 0000 0001 0000	0x0010	
ACS	0000 0000 0010 0000	0x0020	
Sort Order	0000 0000 0100 0000	0x0040	
Repeating Duplicates	0000 0000 1000 0000	0x0080	
Extended Data Type	0000 0001 0000 0000	0x0100	
Null Key (Any Segment)	0000 0010 0000 0000	0x0200	
Case Sensitivity (Distinct)	0000 0100 0000 0000	0x0400	
Existing ACS	0000 1000 0000 0000	0x0800	Internal Use Only
Reserved	0001 0000 0000 0000	0x1000	
Page Compression	0010 0000 0000 0000	0x2000	See Creating a File with Page Level Compression
Pending Key	1000 0000 0000 0000	0x8000	Internal Use Only

The following segment-specific key flags should be on, with the NIS: SEGMENTED (0x0010), EXTENDED DATA TYPE (0x0100).

The following flags should be off: None.

The following flags are ignored: BINARY (0x0004), ACS (0x0020).

Limitations and Side Effects

True null support comes with a few limitations:

- Referential Integrity. Current MicroKernel Engine supports only CASCADE and RESTRICT actions on delete, RESTRICT action on update. While SQL-92 defines CASCADE, RESTRICT, SET DEFAULT, and SET NULL on both delete and update.
- Limited number of segments. The number of index segments used for key indexing will increase because each nullable column occupies two segments, while the maximum number of index segments per data file is the same, as shown in the following table.

Page Size (bytes)	Maximum Key Segments by File Version			
	8.x and earlier	9.0	9.5	13.0, 16.0
512	8	8	Rounded up[2]	Rounded up[2]
1024	23	23	97	Rounded up[2]
1536	24	24	Rounded up[2]	Rounded up[2]
2048	54	54	97	Rounded up[2]
2560	54	54	Rounded up[2]	Rounded up[2]
3072	54	54	Rounded up[2]	Rounded up[2]
3584	54	54	Rounded up[2]	Rounded up[2]
4096	119	119	204[3]	183[3]
8192	n/a[1]	119	420[3]	378[3]
16384	n/a[1]	n/a[1]	420[3]	378[3]

[1]"n/a" stands for "not applicable"

[2]"Rounded up" means that the page size is rounded up to the next size supported by the file version. For example, 512 is rounded up to 1024, 2560 is rounded up to 4096, and so forth.

[3]A 9.5 format or later file can have more than 119 segments, but the number of indexes is limited to 119.

Database URIs

A key concept in using the Btrieve Login API or the implicit login functionality via the Create or Open functions is the database Uniform Resource Identifier (URI). The URI provides a syntax to describe the address of a database resource on a server.

This topic describes the syntax and semantics of the URIs used in Btrieve APIs.

Syntax

URIs use the following syntax:

```
access_method://user@host/dbname?parameters
```

The special characters separating the elements are required, even if an element is omitted. The following table lists the elements of this URI.

Element	Definition
access_method	Method used to access the database. This element is required. Currently, only **btrv** is supported.
user@	Optional user name. The password for the user is specified in *parameters* if needed. An at sign must be used to delimit the user name even if no *host* is specified.
host	Server where the database is located. The local machine is assumed if *host* is not specified. *Host* can be a machine name, an IP address, or the keyword "localhost" and can include a port. For example, the default port can be overriden using a URI with the elements btrv://*servername*:*port*/*database*. **Note**: The *host* element is required if the URI is accessing a database on Linux or Raspbian.
dbname	Optional database name, which corresponds to an entry in the DBNAMES.CFG file for the database engine. If no database name is specified, then the default database DefaultDB is assumed.

Element	Definition
parameters	Additional, optional parameters, which are delimited by an ampersand character. • table=*table* – SQL table name. The table name must exist in DDFs for the database. • dbfile=*file* – Name of a file whose location is relative to the data file location entry in DBNAMES.CFG for the current database. Since a relative location is specified, the use of drive letters, full or UNC paths is not permitted. The database engine resolves the full file name. The Zen client does not manipulate *file* in any manner. Embedded spaces are permitted and are escaped by the database engine. • file=*file* – Data file name. The client normalizes *file* and replaces the input name with the resultant fully qualified UNC name in the URI before sending the request to the database engine. Drive letters may be used and are interpreted as client-side drives. Use of a UNC path is also permitted, as are embedded spaces. • pwd=*password* – Clear text password. The client changes clear text passwords into encrypted passwords before transmission. • prompt=[yes | no] – Tells the client how the application wants to handle the login dialog box pop-up when the database engine returns status 170 (Login failed due to missing or invalid user name) or 171 (Login failed because of invalid password). If prompt=yes is specified, the requester always displays the login dialog even if the Prompt for Client Credentials setting is Off. If prompt=no is specified, the requester assumes that the application wants to receive the status 170/171 directly and does not want the requester to display the dialog. This is useful if you want your applications to handle the prompting for credentials in response to any 170 or 171 status codes. Values other than yes or no are ignored and the requester displays the login dialog based on the Prompt for Client Credentials setting. This option is ignored on Linux systems that are acting in the role of a client.

Parameter Precedence

The database engine enforces a precedence level on the parameters file, table, and dbfile when more than one of them is specified in a URI. That is, after parsing, the database engine leaves the parameter with the highest precedence. If two or more have the same precedence, the last parameter in the URI is returned after parsing.

The order of precedence from highest to lowest is file, table, and dbfile.

Precedence Examples

Initial URI String	Parsed URI String
btrv:///?file=MyFile.btr&table=MyTable&dbfile=DataFile.btr	btrv:///?file=MyFile.btr
btrv:///?table=MyTable&dbfile=DataFile.btr	btrv:///?table=MyTable
btrv:///?dbfile=DataFile.btr&file=MyFile.btr	btrv:///?file=MyFile.btr
btrv:///?dbfile=DataFile.btr	btrv:///?dbfile=DataFile.btr
btrv:///?file=FileOne&file=FileTwo	btrv:///?file=FileTwo
btrv:///?table=TableOne&table=TableTwo&file=MyFile.btr	btrv:///?file=MyFile.btr

Special Characters

As with any URI, certain nonalphanumeric characters have special significance in the URI syntax. If you wish to use one of these characters within one of the elements in the URI, you must use an *escape sequence* to identify the character as actual text rather than a special character. An escape sequence is simply another special character or character combination that represents the plain text equivalent of a special character.

The table below shows the special characters supported by the MicroKernel Engine URI syntax, and their associated escape sequences (represented by the percent sign and the hexadecimal value for the specified character).

Character	Meaning	Hexadecimal Value
/	Separates directories and subdirectories	%2F
?	Separates the base URI from its associated parameters	%3F
%	Specifies a special character	%25
#	Indicates a bookmark or anchor	%23
&	Separates the parameters in the URI	%26
" "	Indicates the entire content enclosed by the double quotes	%22
=	Separates a parameter and its value	%3D
space	No special meaning, but is reserved.	%20
:	Separates host from port (reserved, but not currently supported). The colon is also used in some IPv6 addresses. See IPv6.	%3A

Although the space character is reserved in the URI specification, it can be used without quotes and without escape sequencing because it is not used as a delimiter. The other symbols in the table above are used as delimiters and therefore must be escaped.

Examples

This section shows examples of URIs using escape sequences to identify special characters used within the field values.

URI	Meaning
btrv://Bob@myhost/demodata?pwd= This%20Is%20Bob	User "Bob" with password "This Is Bob."
btrv://Bob@myhost/demodata?pwd= This Is Bob	User "Bob" with password "This Is Bob."
btrv://myhost/mydb?file=c:/data%20files/pvsw/mydb/c.mkd	The %20 represents a space character. File to be opened is C:\data files\pvsw\mydb\c.mkd.
btrv://Bob@myhost/demodata?pwd= mypass%20Is%20%26%3f	User Bob with password "mypass Is &?"

Remarks

Note that an empty user name or password is different than no user name or password. For example, `btrv://@host/` has an empty user name, while `btrv://host/` has no user name, and `btrv://sam@host/?pwd=` has a user name sam with an empty password.

Some URIs allow the use of *user*:*password* syntax. However, the password is then transmitted as clear text. To prevent the transmission of the password as clear text, the Zen database URI ignores the password if one is provided using the *user*:*password* syntax. Use the *pwd*= parameter to provide a password, which the Zen client changes into an encrypted password before transmission.

Some URIs allow for server based naming authority with a syntax of *user@host*:*port*. The Zen database URI does support specifying a *port* element.

Examples

A URL (Uniform Resource Locator) is simply the address of a file or resource on the Internet. The database URI uses the same notion to address a database on a server. This topic gives

examples of the syntax and semantics of URIs for Zen databases, particularly using the MicroKernel Engine access.

Example	Meaning
btrv://myhost/demodata	Database demodata on server myhost. The server operating system can be any of the ones supported by Zen.
btrv:///demodata	Database demodata on the local machine. The local machine is running a Windows operating system. The *host* element is required for Linux systems (see the example above).
btrv://Bob@myhost/demodata	User Bob without a password accessing database demodata on server myhost.
btrv://Bob@myhost/mydb?pwd=a4	User Bob with password "a4" accessing database mydb on server myhost.
btrv://myhost/demodata?table=class	Unspecified user accessing database table named class in database "demodata" on server myhost.
btrv://myhost/?table=class	Unspecified user accessing database table named class in default database (DefaultDB) on server myhost.
btrv://myhost/mydb?file=f:/mydb/a.mkd	Unspecified user accessing the data file F:/mydb/a.mkd as seen by the client using the security credentials of the database mydb on server myhost. Note that the client normalizes drive F, which means that it must be mapped at the client to server myhost.
btrv://mydb?file=c:/mydb/a.mkd	Unspecified user accessing the data file C:/mydb/a.mkd under database mydb on the local machine. Drive C is a local drive on the local machine. The local machine is running a Windows operating system.
btrv://myhost/demodata?dbfile=class.mkd	Unspecified user accessing data file class.mkd within one of the data directories defined for the demodata database on server myhost. Because the file name is specified with dbfile= (and not file=) the client requester does not normalize class.mkd. Only the server engine normalizes class.mkd into a full path.

IPv6

The URI and UNC syntax does not allow certain special characters, such as colons. Since raw IPv6 addresses use colons, different methods of handling UNC paths and URI connections are available. Zen supports IPv6-literal.net Names and Bracketed IPv6 Addresses.

See Drive-based Formats in *Getting Started with Zen*.

Double-Byte Character Support

Zen accepts Shift-JIS (Japanese Industrial Standard) encoded double-byte characters in file paths. (Shift-JIS is an encoding technique commonly used for Japanese computers.) In addition, you can store Shift-JIS double-byte characters in records and sort them using the Japanese ISR table described in International Sort Rules. Other multi-byte characters can be stored in records, although ISR tables are currently not available to sort these records according to culturally correct rules. Your use of double-byte characters does not affect the operation of Zen applications.

Record Length

All records contain the record length, a fixed-length portion which must be large enough to contain all the data (including keys) for a record, plus the overhead required to store a record on a data page.

See Optimum Page Size For Minimizing Disk Space and Record Overhead in Bytes With Record Compression for how many bytes of overhead you must add to the logical record length to obtain a physical record length.

The following table lists the maximum record size for fixed-length records.

File Version	Without System Data[1]	With System Data[2]	With System Data v2[3]
7.x	4088 (4096 – 8)	4080 (4088 – 8)	
8.x	4086 (4096 – 10)	4078 (4086 – 8)	
9.0 through 9.4	8182 (8192 – 10)	8174 (8182 – 8)	
9.5	16372 (16384 – 12)	16364 (16372 – 8)	
13.0, 16.0	16364 (16384 – 20)	16356 (16364 – 8)	16348 (16364 – 16)

[1]The page overhead and the record overhead are subtracted from the maximum page size to determine the maximum record size. The per record overhead is 2 bytes for each file format.

[2]System data requires an additional overhead of 8 bytes.

[3]System data v2 requires an additional overhead of 16 bytes.

Note that the database engine turns on data compression for the file if the file uses system data and the record length exceeds the limit shown in the table above.

Optionally, the records in a file can contain a variable-length portion. A variable-length record has a fixed-length portion that is the same size in every record and a variable-length portion that can be a different size in each record. When you create a file that uses variable-length records, the fixed-length amount is the minimum length of each record; you do not define the maximum record length.

Theoretically, the maximum length of variable-length records is limited only by the MicroKernel Engine file size limit: up to terabytes for 13.0 and 16.0 version files, 256 GB for 9.5 files, 128 GB for earlier 9.x versions, and 64 GB for other earlier versions. In practice, the maximum is limited by such factors as the operating system, system resources, and the record access method. If you retrieve, update, or insert whole records, then the data buffer length parameter, a 16-bit unsigned integer, limits the record length to 65535.

A data buffer is a MicroKernel Engine function parameter that you use to transfer various information depending on the operation being performed. A data buffer can contain all or part of a record, a file specification, and so forth. See Designing a Database for more information on data buffers.

Note: The total bytes of data plus internal header information cannot exceed 64 KB (0x10000) bytes. The MicroKernel Engine reserves 1024 (0x400) bytes for internal purposes, meaning you can have 64512 (0xFC00) bytes of data.

If your file uses very large records, consider using variable-tail allocation tables (VATs) in the file. A VAT, which is implemented as a linked list, is an array of pointers to the variable-length portion of the record. VATs accelerate random access to portions of very large records. Some examples of very large records are binary large objects (BLOBs) and graphics.

For files that contain very large variable-length records, the MicroKernel Engine splits the record over many variable pages and connects the pages using a linked list called a *variable tail*. If an application uses chunk operations to access a part of a record and that part of the record begins at an offset well beyond the beginning of the record itself, the MicroKernel Engine may spend considerable time reading the variable-tail linked list to seek that offset. To limit such seek time, you can specify that the file use VATs. the MicroKernel Engine stores the VAT on variable pages. In a file containing a VAT, each record that has a variable-length portion has its own VAT.

The MicroKernel Engine uses VATs not only to accelerate random access, but also to limit the size of the compression buffer used during data compression. If your files use data compression, you may want to use VATs in the file.

Data Integrity

The following features support concurrent access while ensuring the integrity of your files in a multi-user environment:

- Record Locks
- Transactions
- Transaction Durability
- System Data
- Shadow Paging
- Backing Up Your Files

Record Locks

Applications can explicitly lock either one record at a time (single record lock) or multiple records at once (multiple record lock). When an application specifies a record lock, the application can also apply a wait or no-wait condition. When an application requests a no-wait lock on a record that is currently unavailable (either the record is already locked by another application or the whole file is locked by an exclusive transaction), the MicroKernel Engine does not grant the lock.

When an application requests a wait lock on a record that is unavailable, the MicroKernel Engine checks for a *deadlock condition*. You can configure the MicroKernel Engine to wait before returning a deadlock detection status code. Doing so improves performance in multi-user situations by allowing the MicroKernel Engine to wait internally, rather than forcing the application to retry the operation.

Transactions

If you have a number of modifications to make to a file and you must be sure that either all or none of those modifications are made, include the operations for making those modifications in a *transaction*. By defining explicit transactions, you can force the MicroKernel Engine to treat multiple operations as an atomic unit. Other users cannot see the changes made to a file until the transaction ends. The MicroKernel Engine supports two types of transactions: exclusive and concurrent.

Exclusive Transactions

In an *exclusive* transaction, the MicroKernel Engine locks the entire data file when you insert, update, or delete a record in that file. Other applications (or other instances of the same application) can open the file and read its records, but they cannot modify the file. The file remains locked until the application ends or aborts the transaction.

Concurrent Transactions

In a *concurrent* transaction, the MicroKernel Engine can lock either records or pages in the file, depending on the operation you are performing. The MicroKernel Engine enables multiple applications (or multiple instances of the same application) to perform modifications inside concurrent transactions in different parts of the same file simultaneously, as long as those modifications do not affect other previously locked portions of the file. The record or page remains locked until the application ends or aborts the transaction. Concurrent transactions are available only for 6.0 and later files.

Exclusive vs. Concurrent

Clients can still *read* records from a file even if a concurrent transaction has locked the requested record. However, these clients cannot be operating from within an exclusive transaction. Also, they cannot apply a lock bias to their read operation if the file containing the requested record is currently locked by an exclusive transaction, or if a concurrent transaction has locked the requested record.

When a client reads a record using an exclusive lock, the MicroKernel Engine locks only the individual record; the rest of the page on which the record resides remains unlocked.

Note: Simply opening a file from within a transaction does not lock any records, pages, or files. In addition, the MicroKernel Engine does not lock files that you flag as read-only or files that you open in read-only mode.

When you use exclusive transactions, the MicroKernel Engine causes other clients to implicitly wait on the locked file unless the No Wait bias is added to the Begin Transaction (19 or 1019) operation. The application seems to hang during this implicit wait state. If these exclusive transactions are short lived, you may not notice the wait time. However, the overall effect of many clients involved in implicit waits results in using a large amount of CPU time. Additionally, multiple position blocks in the same file share locks.

Exclusive transactions involved in implicit waits also waste network bandwidth. The MicroKernel Engine waits about one second before returning to the requester. The requester

recognizes a wait condition and returns the operation to the MicroKernel Engine. Thus, exclusive transactions also can cause extra network traffic.

The amount of extra CPU cycles and network traffic increase exponentially with the number of clients waiting on the locked file combined with the length of time involved in the exclusive transaction.

Transaction Durability

You can configure the MicroKernel Engine to guarantee Transaction Durability and atomicity by logging all operations to a single transaction log. Transaction durability is the assurance that the MicroKernel Engine finishes writing to the log when a client issues the End Transaction operation and before the MicroKernel Engine returns a successful status code to the client. Atomicity ensures that if a given statement does not execute to completion, then the statement does not leave partial or ambiguous effects in the database, thereby ensuring the integrity of your database by keeping it in a stable state.

If you want atomicity without the overhead of transaction durability, you can use transaction logging feature in PSQL V8 and later releases. See *Advanced Operations Guide* for more information on Transaction Logging.

In a default installation, the transaction log is in C:\ProgramData\Actian\Zen\logs. The log must exist on the same machine as the Zen engine. You can change the location using the transaction log directory configuration option in ZenCC by right-clicking the MicroKernel Engine and selecting Properties > Directories.

The MicroKernel Engine maintains the transaction log in one or more physical files, called log segments. The MicroKernel Engine starts a new log segment when the current log segment reaches a user-defined size limit, no files have pending changes, and the MicroKernel Engine has finished a system transaction.

All transaction log segments have a .log extension. The MicroKernel Engine names log segments with consecutive, 8-character hexadecimal names, such as 00000001.log, 00000002.log, and so on.

To improve performance on specific files, you can open a file in Accelerated mode. (Version 6.*x* MicroKernel Engine accepted Accelerated open requests, but interpreted them as Normal open requests.) When you open a file in Accelerated mode, the MicroKernel Engine does not perform transaction logging on the file.

Note: If a system failure occurs, there will be some log segments that are not deleted. These segments contain changes that did not get fully written to the data files. Do not delete these log

segments. You do not know which files are represented in these log segments. No action is necessary because those data files will automatically get rolled forward the next time they are opened.

System Data

Zen uses a 7.x transaction log file format. In order for the MicroKernel Engine to log transactions on a file, the file must contain a *log key*, which is a unique (non-duplicatable) key that the MicroKernel Engine can use to track the record in the log. For files that have at least one unique (non-duplicatable) key, the MicroKernel Engine uses one of the unique keys already defined in the file.

For files that do not have any unique keys, the MicroKernel Engine can include *system data* upon file creation. The MicroKernel Engine includes system data in a file only if the file uses the 7.x file format or later *and* if at the time the file is created, the MicroKernel Engine is configured to include system data in files upon creation. System data is defined as an 8-byte binary value with the key number 125. Even if a file has a unique, user-defined key, you may want to use system data (also called the system-defined log key), to protect against a user dropping an index.

The database engine turns on data compression for the file if the file uses system data and the record length exceeds the limit described under Record Length.

The MicroKernel Engine adds system data only at file creation. If you have existing files for which you want to add system data, than after you turn on the System Data configuration option, use the Rebuild tool as described in Converting Data Files.

Note: When the MicroKernel Engine adds system data, the resulting records may be too large to fit in the file's existing page size. In such cases, the MicroKernel Engine automatically increases the file's page size to the next accommodating size.

Shadow Paging

The MicroKernel Engine uses *shadow paging* to protect 6.0 and later files from corruption in case of a system failure. When a client needs to change a page (either inside or outside a transaction), the MicroKernel Engine selects a free, unused physical location in the data file itself and writes a new page image, called a shadow page, to this new location. During a single MicroKernel Engine operation, the MicroKernel Engine might create several shadow pages all the same size as the original logical pages.

When the changes are committed (either when the operation is complete or the transaction ends), the MicroKernel Engine makes the shadow pages current, and the original pages become

available for reuse. The MicroKernel Engine stores a map, which is the Page Allocation Table, in the file to keep track of the valid and reusable pages. If a system failure occurs before the changes are committed, the MicroKernel Engine does not update the PAT, thereby dropping the shadow pages and reverting to using the current pages, which are still in their original condition.

Note: This description simplifies the shadow paging process. For improved performance, the MicroKernel Engine does not commit each operation or user transaction individually, but groups them in a bundle called a system transaction.

When a client operates inside a transaction, the shadow pages corresponding to the original logical pages are visible only to that client. If other clients need to access the same logical pages, they see the original (unchanged) pages – that is, they do not see the first client's uncommitted changes. Shadow paging thus enhances reliability because the original file is always valid and internally consistent.

Backing Up Your Files

Backing up your files regularly is an important step in protecting your data.

For information on backing up your files, please see the following topic in *Advanced Operations Guide*: Logging, Backup, and Restore.

Event Logging

Event logging is a feature in Zen that uses a log file to store informational, warning, and error messages from the MicroKernel Engine, SQL interface, and utility components.

See Reviewing Message Logs in *Advanced Operations Guide* for details.

Performance Enhancement

The MicroKernel Engine provides the following features for enhancing performance:

- System Transactions
- Memory Management
- Page Preallocation
- Extended Operations

System Transactions

To gain better performance and to aid data recovery, the MicroKernel Engine includes one or more committed operations (both transactional and non-transactional) into a bundle of operations called a *system transaction*. The MicroKernel Engine creates a system transaction bundle for each file. A system transaction can contain operations and user transactions from one or more clients running on the same engine.

Note: Do not confuse system transactions with exclusive or concurrent transactions. Throughout this manual, the term *transaction* refers to an exclusive or concurrent transaction (also known as a user transaction). User transactions affect how changes become incorporated into the pages in cache, while system transactions affect how dirty pages in cache become a part of the files on disk. The MicroKernel Engine controls the initiation and process of system transactions.

Both user transactions and system transactions are atomic. In other words, they happen in such a way that either all or none of the changes occur. If a system failure occurs, the MicroKernel Engine recovers all files involved in the failed system transaction as it reopens the files. All changes made during a failed system transaction (that is, all operations to a file by all clients on that engine since the last completed system transaction) are lost; however, the file is restored to a consistent state, enabling the operations to be attempted again after resolving the cause of the system failure.

Zen guarantees *transaction durability* for all loggable files except those opened in Accelerated mode. (A file can be logged if it contains at least one unique, or non-duplicatable, key. The key can be system-defined.) Transaction durability is the assurance that before the MicroKernel Engine returns a successful status code to the client for an End Transaction it finishes writing to the transaction log. When you open a file in Accelerated mode, the MicroKernel Engine does not log the file; therefore, the MicroKernel Engine does not log entries to the file. Therefore, the MicroKernel Engine cannot guarantee transaction durability for that file.

After the MicroKernel Engine rolls back a file's system transaction, it replays the log when it next opens the file. Doing so restores those committed operations that were stored in the log but were not written to the file because of the system transaction rollback.

Each system transaction consists of two phases: preparation and writing.

Preparation Phase

During the preparation phase, the MicroKernel Engine executes all operations in the current system transaction, but writes no pages to the files. The MicroKernel Engine reads uncached pages from the files as needed and creates new page images only in cache.

Any of the following actions can trigger the end of the preparation phase, which marks the beginning of the writing phase:

- The MicroKernel Engine reaches the Operation Bundle Limit.
- The MicroKernel Engine reaches the Initiation Time Limit.
- The ratio of pages prepared for writing to the total number of cache pages reaches a system threshold.

Note: Generally, the preparation phase ends after a completed MicroKernel Engine operation. However, it is possible that the time limit or the cache threshold could be reached during an incomplete user transaction; the MicroKernel Engine switches to the writing phase, regardless.

Writing Phase

During the writing phase, the MicroKernel Engine writes to disk all pages prepared in the preparation phase. It first writes all data, index, and variable pages. These are actually shadow pages. While these are being written, the consistency of the files on disk remains the same.

However, the critical part of the system transaction occurs while the PAT pages are being written, because they point to the shadow page as the current page. To protect this phase, the MicroKernel Engine writes a flag in the FCR. When all PAT pages are written, the final FCR is written and the file is now consistent. If a system failure happens during this phase, the MicroKernel Engine recognizes it the next time the file is opened and rolls the file back to the previous state. Then, all durable user transactions in the transaction log file will be implemented in the file.

Frequency of System Transactions

Less Frequent

Doing system transactions less often provides a performance benefit to most configurations. These include client/server, single engine workstation and multi-engine workstation environments where files are opened exclusively.

When the MicroKernel Engine initiates system transactions less often, dirty pages, or pages that need to be written, stay in memory longer. If the application is doing a lot of change operations, these pages might get updated multiple times before being written to disk. This means fewer disk writes. In fact, the most efficient engine is one that writes only when it must.

There are three limits that, when reached, can cause the system transactions to be initiated: Operation Bundle Limit, Cache Size, and Initiation Time Limit. When any of these limits are reached, the MicroKernel Engine initiates a system transaction. See *Advanced Operations Guide* for more information about these settings.

The best way to do system transactions less often is to set the Operation Bundle Limit and Initiation Time Limit to higher values. You can also increase the size of the cache.

More Frequent

A disadvantage of causing the MicroKernel Engine to perform fewer system transactions is more data in machine memory at any point in time that needs to be written to disk. If a system failure occurs, such as a power outage, more data is lost. Although the MicroKernel Engine is designed to keep files in a consistent usable state, that state may not include the most recent changes. Of course, the use of user transactions with transaction durability will minimize this risk. You should carefully consider the risks of decreasing the frequency of system transactions versus the performance gains.

For example, if your application is using a workstation engine to update a remote file over a slow or non-reliable network connection, you should perform system transactions often so that changes are put onto disk at soon as possible.

Memory Management

The *cache* is an area of memory the MicroKernel Engine reserves for buffering the pages that it reads. When an application requests a record, the MicroKernel Engine first checks the cache to see if the page containing that record is already in memory. If so, the MicroKernel Engine transfers the record from the cache to the application's data buffer. If the page is not in the cache, the MicroKernel Engine reads the page from the disk into a cache buffer before transferring the

requested record to the application. The MicroKernel Engine cache is shared by local clients and used across multiple operations.

If every cache buffer is full when the MicroKernel Engine attempts to transfer a new page into memory, a least-recently-used (LRU) algorithm determines which page in the cache the MicroKernel Engine overwrites. The LRU algorithm reduces processing time by keeping the most recently referenced pages in memory.

When the application inserts or updates a record, the MicroKernel Engine first makes a shadow image of the corresponding page, modifies the page in the cache, and then writes that page to disk. The modified page remains in the cache until the LRU algorithm determines that the MicroKernel Engine can overwrite the image of that page in cache with a new page.

Generally, a larger cache improves performance because it enables more pages to be in memory at a given time. The MicroKernel Engine enables you to specify the amount of memory to reserve for the I/O cache buffers. To determine this amount, consider the application's memory requirements, the total memory installed on your computer, and the combined size of all files that all concurrent Zen applications will access. The configuration setting for this cache is "Cache Allocation Size."

In Zen V8 and later releases, a secondary dynamic L2 cache is also available. The configuration setting for this dynamic cache is "Max MicroKernel Engine Memory Usage." See *Advanced Operations Guide* for more information on configuring these settings.

Note: Increasing cache above the available physical memory can actually cause a significant performance decrease because part of the cache memory in virtual memory will be swapped out onto disk. We recommend that you set the MicroKernel Engine cache to about 60 percent of available physical memory after the operating system is loaded.

Page Preallocation

Page preallocation guarantees that disk space is available when the MicroKernel Engine needs it. You can enhance the speed of file operations if a data file occupies a contiguous area on the disk. The increase in speed is most noticeable on very large files. For more information about this feature, see Page Preallocation.

Extended Operations

Using the extended operations – Get Next Extended (36), Get Previous Extended (37), Step Next Extended (38), Step Previous Extended (39), and Insert Extended (40) – can greatly improve performance. Extended operations can reduce the number of MicroKernel Engine requests by 100

to 1 or more, depending on the application. These operations have the ability to filter the records returned so that records not needed by the application are not sent to it. This optimization technique has the best results in client-server environments that have to send data back and forth over a network.

See Multirecord Operations for detailed information on these operations.

Disk Usage

The MicroKernel Engine provides the following features for minimizing disk usage requirements:

- Free Space List
- Index Balancing
- Data Compression
- Blank Truncation

Free Space List

When you delete a record, the disk space it formerly occupied is put on a Free Space List. When you insert new records, the MicroKernel Engine uses pages on the Free Space List before creating new variable pages. The Free Space Threshold tells the MicroKernel Engine how much free space must remain on a variable page in order for that page to appear on the Free Space List.

This method of reusing free space eliminates the need to reorganize files to reclaim disk space. Also, the Free Space List provides a means of reducing the fragmentation of variable-length records across several pages. A higher Free Space Threshold reduces fragmentation at the cost of requiring more disk space for the file.

Index Balancing

When an index page becomes full, the MicroKernel Engine (by default) automatically creates a new index page and moves some of the values from the full index page to the new index page. Turning on the Index Balancing option lets you avoid creating a new index page every time an existing one becomes full. With index balancing, the MicroKernel Engine looks for available space in sibling index pages each time an index page becomes full. The MicroKernel Engine then rotates values from the full index page onto the pages that have space available.

Index balancing increases index page utilization, results in fewer pages, and produces an even distribution of keys among nodes on the same level, thus enhancing performance during read operations. However, using this feature also means that the MicroKernel Engine requires extra time to examine more index pages and may require more disk I/O during write operations. Although the exact effects of balancing indexes vary in different situations, performance on write operations typically degrades by about 5 to 10 percent if you use index balancing.

Index Balancing impacts the performance of a "steady-state" file by making the average change operation a little slower while making the average get operation faster. It does this by fitting more

keys in an average index page. A normal index page may be 50 to 65 percent full where an index balanced page is 65 to 75 percent full. This means there are less index pages to search.

If you create indexes with Create Index (31), the index pages will be nearly 100 percent full, which optimizes these files for reading not writing.

Note: You can also specify index balancing on a file-by-file basis by setting the Index Balanced File bit in the File Flag field in the file.

If you enable the Index Balancing option, the MicroKernel Engine performs index balancing on every file, regardless of the balanced file flag specification that the application may have set. For a description of how to specify the Index Balancing configuration option, see *Zen User's Guide*.

Data Compression

With data compression, the MicroKernel Engine compresses file records before inserting or updating them and uncompresses the records when it retrieves them. Because the final length of a compressed record cannot be determined until the record is written to the file, the MicroKernel Engine always designates a compressed file as a variable-record-length file. However, if you use data compression on a fixed-record-length file, the MicroKernel Engine prevents insert and update operations from producing a record that is longer than the fixed-record length specified for the data file.

Because the MicroKernel Engine stores compressed records as variable-length (even if you created the file as not allowing variable-length records), individual records may become fragmented across several data pages if you perform frequent insertions, updates, and deletions. This fragmentation can result in slower access times, because the MicroKernel Enginemay need to read multiple file pages to retrieve a single record. However, data compression can also result in a significant reduction of the disk space needed to store records that contain many repeating characters. the MicroKernel Enginecompresses five or more of the same contiguous characters into 5 bytes.

For more information on this feature, see Record Compression.

Blank Truncation

Blank truncation conserves disk space. It is applicable only to files that allow variable-length records and that do not use data compression. For more information on this feature, see Blank Truncation.

Designing a Database

The following topics provide formulas and guidelines for designing your database:

- Understanding Data Files
- Creating a Data File
- Calculating the Logical Record Length
- Choosing a Page Size
- Estimating File Size
- Optimizing Your Database
- Setting Up Security

Understanding Data Files

The MicroKernel Engine stores information in data files. Inside each data file is a collection of records and indexes. A record contains bytes of data. That data might represent an employee's name, ID, address, phone number, rate of pay, and so on. An index provides a mechanism for quickly finding a record containing a specific value for a portion of the record.

The MicroKernel Engine interprets a record only as a collection of bytes. It does not recognize logically discrete pieces, or fields, of information within a record. To the MicroKernel Engine, a last name, first name, employee ID, and so on do not exist inside a record. The record is simply a collection of bytes.

Because it is byte-oriented, the MicroKernel Engine performs no translation, type verification, or validation of the data in a record – even on keys (for which you declare a data type). The application interfacing with the data file must handle all information about the format and type of the data in that file. For example, an application might use a data structure based on the following format:

Information in Record	Length (in Bytes)	Data Type
Last name	25	Null-terminated string
First name	25	Null-terminated string
Middle initial	1	Char (byte)
Employee ID	4	Long (4-byte integer)
Phone number	13	Null-terminated string
Pay rate per month	4	Float
Total Record Length	72 bytes	

Inside the file, an employee record is stored as a collection of bytes. The following diagram shows the data for the record for Cliff Jones, as it is stored in the file. (This diagram replaces ASCII values in strings with the appropriate letter or number. Integers and other numeric values are unchanged from their normal hexadecimal representation.)

Byte Position	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Data Value	J	o	n	e	s	00	?	?	?	?	?	?	?	?	?	?

Byte Position	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Data Value	?	?	?	?	?	?	?	?	?	C	l	i	f	f	00	?

Byte Position	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Data Value	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Byte Position	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
Data Value	?	?	D	2	3	4	1	5	1	2	5	5	5	1	2	1

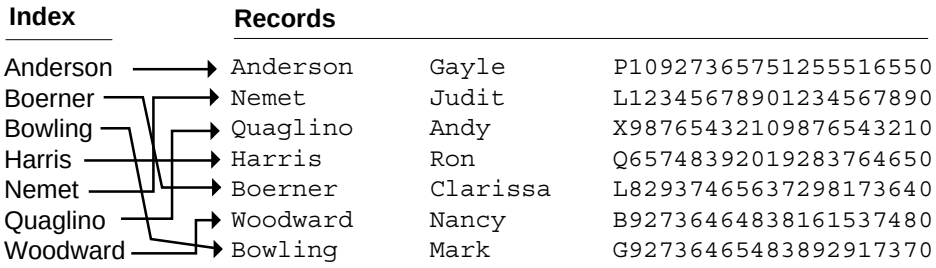
Byte Position	40	41	42	43	44	45	46	47
Data Value	2	?	?	?	3	5	0	0

The only discrete portions of information that the MicroKernel Engine recognizes in a file are *keys*. An application (or user) can designate one or more collection of bytes in a record as a key, but the bytes must be contiguous inside each *key segment*.

The MicroKernel Engine sorts records on the basis of the values in any specified key, providing direct access to return the data in a particular order. The MicroKernel Engine can also find a particular record based on a specified key value. In the preceding example, the 25 bytes that contain a last name in each record might be designated as a key in the file. An application could use the last name key to obtain a listing of all the employees named Smith, or it could obtain a listing of all employees and then display that listing, sorted by last name.

Keys also allow the MicroKernel Engine to access information quickly. For each key defined in a data file, the MicroKernel Engine builds an *index*. The index is stored inside the data file itself and contains a collection of pointers to the actual data within that file. A key value is associated with each pointer.

In the preceding example, the index for the Last Name key sorts the Last Name values and has a pointer indicating where the record is located in the data file:



Normally, when accessing or sorting information for an application, the MicroKernel Engine does not search through all the data in its data file. Instead, it goes to the index, performs the search, and then manipulates only those records that meet the application's request.

Creating a Data File

The MicroKernel Engine gives developers tremendous flexibility in optimizing database applications. In providing such flexibility, the MicroKernel Engine exposes a great deal of the inner workings of the MicroKernel Engine. If you are new to the MicroKernel Engine, the Create (14) operation may appear quite complex to you, but you do not need all of the features this operation provides to get started. This section highlights the basic requirements by stepping you through the creation of a simple, transaction-based data file. For simplification where necessary, this section uses C interface terminology.

Note: In the same directory, no two files should share the same file name and differ only in their file name extension. For example, do not name a data file invoice.btr and another one invoice.mkd in the same directory. This restriction applies because the database engine uses the file name for various areas of functionality while ignoring the file name extension. Since only the file name is used to differentiate files, files that differ only in their file name extension look identical to the database engine.

Data Layout

This section uses an example data file that stores employee records. The application will retrieve employee information by providing either a unique employee ID or the employee last name. Because more than one employee can have the same last name, the database will allow duplicate values for the last name. Based on these criteria, the following table shows the data layout for the file.

Information in Record	Data Type	Key/Index Characteristics
Last name	25 character string	Duplicatable
First name	25 character string	None
Middle initial	1 character string	None
Employee ID	4 byte integer	Unique
Phone number	13 character string	None
Pay rate per month	4 byte float	None

Now that the basic data layout is established, you can begin applying the terminology and requirements of the MicroKernel Engine. This includes determining information about the key structure and file structure before you actually create the file. You must work out these details in

advance, because Create (14) creates the file, index, and key information all at once. The following sections discuss the issues to consider in working out these details.

Key Attributes

First, determine any special functionality for the keys. The MicroKernel Engine supports a variety of key attributes you can assign, as shown in the following table.

Constant	Description
EXTTYPE_KEY	Extended Data Type. Stores a MicroKernel Engine data type other than string or unsigned binary. Use this attribute, rather than the standard binary data type. This key attribute can accommodate the standard binary and string data types, plus many others.
BIN	Standard BINARY Data Type. Supported for historical reasons. Stores an unsigned binary number. Default data type is string.
DUP	Linked Duplicates. Allows duplicate values, which are linked by pointers from the index page to the data page. For more information, see Duplicatable Keys.
REPEAT_DUPS_KEY	Repeating Duplicates. Allows duplicate values, which are stored on both the index page and the data page. For more information, see Duplicatable Keys.
MOD	Modifiable. Allows the key value to be modified after the record is inserted.
SEG	Segmented. Specifies that this key has a segment that follows the current key segment.
NUL	Null Key (All Segments). Excludes any records from the index if all segments of the key contain a specified null value. (You assign the null value when you create the file.)
MANUAL_KEY	Null Key (Any Segment). Excludes any records from the index if any segment in the key contains a specified null value. (You assign the null value when you create the file.)
DESC_KEY	Descending Sort Order. Orders the key values in descending order (highest to lowest). Default is ascending order (lowest to highest).
NOCASE_KEY	Case Insensitive. Sorts string values without distinguishing upper and lower case letters. Do not use if the key has an alternate collating sequence (ACS). In the case of a Null Indicator segment, this attribute is overloaded to indicate that nonzero null values should be treated distinctly.

Constant	Description
ALT	Alternate Collating Sequence. Uses collating sequence to sort string keys differently from the standard ASCII sequence. Different keys can use different collations. You can specify the default ACS (the first one defined in the file), a numbered ACS defined in the file, or a named ACS defined in the collate.cfg system file.
NUMBERED_ACS	
NAMED_ACS	
For simplicity these constants, defined in btrconst.h, are consistent with the C interface. Some interfaces may use other names or no constants at all. For bit masks, hexadecimal, and decimal equivalents for the key attributes, see *Btrieve API Guide*.	

You assign these key attributes for each key you define. Each key has its own key specification. If the key has multiple segments, you have to provide the specification for each segment. Some of these attributes can have different values for different segments within the same key. Using the previous example, the keys are the last name and the employee ID. Both keys use extended types; the last name is a string and the employee ID is an integer. Both are modifiable, but only the last name is duplicatable. In addition, the last name is case insensitive.

Regarding the data type you assign to a key, the MicroKernel Engine does not validate that the records you input adhere to the data types defined for the keys. For example, you could define a TIMESTAMP key in a file, but store a character string there or define a date key and store a value for February 30. Your MicroKernel Engine application would work fine, but an ODBC application that tries to access the same data might fail, because the byte format could be different and the algorithms used to generate the time stamp value could be different. For complete descriptions of the data types, see *SQL Engine Reference*.

File Attributes

Next, determine any special functionality for the file.

The MicroKernel Engine supports a variety of file attributes you can assign, as follows:

Constant	Description
VAR_RECS	Variable Length Records. Use in files that contain variable length records.
BLANK_TRUNC	Blank Truncation. Conserves disk space by dropping any trailing blanks in the variable-length portion of the record. Applicable only to files that allow variable-length records and that do not use data compression. For more information, see Blank Truncation.

Constant	Description
PRE_ALLOC	Page Preallocation. Reserves contiguous disk space for use by the file as it is populated. Can speed up file operations if a file occupies a contiguous area on the disk. The increase in speed is most noticeable on very large files. For more information, see Page Preallocation.
DATA_COMP	Data Compression. Compresses records before inserting or updating them and uncompresses records when retrieving them. For more information, see Record Compression.
KEY_ONLY	Key-Only File. Includes only one key, and the entire record is stored with the key, so no data pages are required. Key-only files are useful when your records contain a single key and that key takes up most of each record. For more information, see Key-Only Files.
BALANCED_KEYS	Index Balancing. Rotates values from full index pages onto index pages that have space available. Index balancing enhances performance during read operations, but may require extra time during write operations. For more information, see Index Balancing.
FREE_10 FREE_20 FREE_30	Free Space Threshold. Sets the threshold percentage for reusing disk space made available by deletions of variable length records, thus eliminating the need to reorganize files and reducing the fragmentation of variable-length records across several pages. A larger Free Space Threshold reduces fragmentation of the variable-length portion of records which increases performance. However, it requires more disk space. If higher performance is desired, increase the Free Space Threshold to 30 percent.
DUP_PTRS	Reserve Duplicate Pointers. Preallocates pointer space for linked duplicatable keys added in the future. If no duplicate pointers are available for creating a linked-duplicatable key, the MicroKernel Engine creates a repeating-duplicatable key.
INCLUDE_SYSTEM_DATA NO_INCLUDE_SYSTEM_DATA	System Data. Includes system data upon file creation, which allows the MicroKernel Engine to perform transaction logging on the file. This is useful in files that do not contain a unique key.

Constant	Description
SPECIFY_KEY_NUMS	Key Number. Allows you to assign a specific number to a key, rather than letting the MicroKernel Engine assign numbers automatically. Some applications may require a specific key number.
VATS_SUPPORT	Variable-tail Allocation Tables (VATs). Uses VATs (arrays of pointers to the variable-length portion of the record) to accelerate random access and to limit the size of the compression buffer used during data compression. For more information, see Variable-tail Allocation Tables.
For simplicity, these constants, defined in btrconst.h, are consistent with the C interface. Some interfaces may use other names or no constants at all. For bit masks, hexadecimal, and decimal equivalents for the file attributes, see *Btrieve API Guide*.	

The example data file does not use any of these file attributes, because the records are fixed-length records of small size.

For definitions of file attributes, see File Types. For more information about specifying file attributes during the Create operation, see *Btrieve API Guide.*

Creating File and Key Specification Structures

When you use the Create operation, you pass in a data buffer that contains file and key specification structures.

Sample Data Buffer for File and Key Specifications Using BTRV Entry Point

The following structure uses the example employee data file.

Description		Data Type[1]	Byte #	Example Value[2]
File Specification				
Logical Fixed Record Length. (Size of all fields combined: 25 + 25 + 1 + 4 + 13 + 4). For steps, see Calculating the Logical Record Length.[3]		Short Int[4]	0–1	72
Page Size.	**File Format**	Short Int	2–3	
	6.0-9.0			512
	6.0 and later			1024
	6.0-9.0			1536
	6.0 and later			2048
	6.0-9.0			3072
				3584
	6.0 and later			4096
	9.0 and later			8192
	9.5 and later			16384
	A minimum size of 4096 bytes works best for most files. If you want to fine-tune this, see Choosing a Page Size for more information. 6.0 to 8.0 file formats support page sizes of 512 times *x*, where *x* is any number up to the product 4096. 9.0 file format supports page sizes identical to previous versions except that it also supports a page size of 8192. 9.5 file format supports page sizes of 1024 times 2 $^{0 \text{ thru } 4}$. In creating 9.5 format files, if the logical page size specified is valid for the file format, the MicroKernel rounds the specified value to the next higher valid value if one exists. For all other values and file formats, the operation fails with status 24. No rounding is done for the older file formats.			
Number of Keys. (Number of keys in the file: 2)		Byte	4	2

Description	Data Type[1]	Byte #	Example Value[2]	
File Version	Byte	5	0x60	Version 6.0
			0x70	Version 7.0
			0x80	Version 8.0
			0x90	Version 9.0
			0x95	Version 9.5
			0xD0	Version 13.0
			0xD3	Version 16.0
			0x00	Use database engine default
Reserved. (Not used during a Create operation.)	Reserved	6–9	0	
File Flags. Specifies the file attributes. The example file does not use any.	Short Int	10–11	0	
Number of Extra Pointers. Sets the number of duplicate pointers to reserve for future key additions. Used if the file attributes specify Reserve Duplicate Pointers.	Byte	12	0	
Physical page size. Used when compression flag is set. Value is the number of 512-byte blocks.	Byte	13	0	
Preallocated Pages. Sets the number of pages to preallocate. Used if the file attributes specify Page Preallocation.	Short Int	14–15	0	
Key Specification for Key 0 (Last Name)				
Key Position. Provides the position of the first byte of the key within the record. The first byte in the record is 1.	Short Int	16–17	1	
Key Length. Specifies the length of the key, in bytes.	Short Int	18–19	25	
Key Flags. Specifies the key attributes.	Short Int	20–21	EXTTYPE_KEY + NOCASE_KEY + DUP + MOD	
Not Used for a Create.	Byte	22–25	0	

Description	Data Type[1]	Byte #	Example Value[2]
Extended Key Type. Used if the key flags specify Use Extended Key Type. Specifies one of the extended data types.	Byte	26	ZSTRING
Null Value (legacy nulls only). Used if the key flags specify Null Key (All Segments) or Null Key (Any Segment). Specifies an exclusion value for the key. See Null Value for more conceptual information on legacy nulls and true nulls.	Byte	27	0
Not Used for a Create.	Byte	28–29	0
Manually Assigned Key Number. Used if the file attributes specify Key Number. Assigns a key number.	Byte	30	0
ACS Number. Used if the key flags specify Use Default ACS, Use Numbered ACS in File, or Use Named ACS. Specifies the ACS number to use.	Byte	31	0
Key Specification for Key 1 (Employee ID)			
Key Position. (Employee ID starts at first byte after Middle Initial.)	Short Int	32–33	52
Key Length.	Short Int	34–35	4
Key Flags.	Short Int	36–37	EXTTYPE_KEY + MOD
Not Used for a Create.	Byte	38–41	0
Extended Key Type.	Byte	42	INTEGER
Null Value.	Byte	43	0
Not Used for a Create.	Byte	44–45	0
Manually Assigned Key Number.	Byte	46	0
ACS Number.	Byte	47	0
Key Specification for Page Compression			
Physical Page Size[5]	Char	A	512 (default value)

Description	Data Type[1]	Byte #	Example Value[2]

[1]Unless specified otherwise, all data types are unsigned.

[2]For simplification, the non-numeric example values are for C applications.

[3]For files with variable-length records, the logical record length refers only to the fixed-length portion of the record.

[4]Short Integers (Short Int) must be stored in little-endian byte order, which is the low-to-high ordering of Intel-class computers.

[5]Used only with page-level compression. Requires the Page Compression file flag (see Key Flag Values). See also Creating a File with Page Level Compression for more information.

Creating a File with Page Level Compression

For PSQL 9.5 and later, you can use Create (14) to create data files with page level compression. For earlier data files, logical pages map to physical pages, and this mapping is stored in a Page Allocation Table (PAT). A physical page is exactly the same size as a logical page.

When a file is compressed, each logical page is compressed into one or more physical page units that are smaller in size than a logical page. The physical page size is specified by the Physical Page Size attribute described under Creating File and Key Specification Structures.

The Page Compression file flag (see Key Flag Values) is used in conjunction with the Physical Page Size key specification to tell the MicroKernel to create the new data file with page level compression turned on. The logical and physical page sizes are validated as follows:

The value specified for the physical page size cannot be larger than the value specified for the logical page size. If it is then the MicroKernel will round down the value specified for the physical page size so that it is the same as the logical page size. The logical page size needs to be an exact multiple of the physical page size. If it is not then the logical page size is rounded down so that it becomes an exact multiple of the physical page size. If, as a result of these manipulations, the logical and physical values end up to be the same, then page level compression will not turned on for this file.

Calling the Create Operation

Create (14) requires the following values:

- Operation code, which is 14 for a Create.
- Data buffer containing the file and key specifications.
- Length of the data buffer.

- Key buffer containing the full path for the file.
- Key number containing a value to determine whether the MicroKernel Engine warns you if a file of the same name already exists (-1 = warning, 0 = no warning).

In C, the API call would resemble the following:

Create Operation

```
/* define the data buffer structures */
/*The following three structures need to have 1-byte alignment for their fields. This can be done
either through compiler pragmas or through compiler options. */
typedef struct
{
 BTI_SINT recLength;
 BTI_SINT pageSize;
 BTI_SINT indexCount;
 BTI_CHAR reserved[4];
 BTI_SINT flags;
 BTI_BYTE dupPointers;
 BTI_BYTE notUsed;
 BTI_SINT allocations;
} FILE_SPECS;

typedef struct
{
 BTI_SINT position;
 BTI_SINT length;
 BTI_SINT flags;
 BTI_CHAR reserved[4];
 BTI_CHAR type;
 BTI_CHAR null;
 BTI_CHAR notUsed[2];
 BTI_BYTE manualKeyNumber;
 BTI_BYTE acsNumber;
} KEY_SPECS;

typedef struct
{
 FILE_SPECS fileSpecs;
 KEY_SPECS keySpecs[2];
} FILE_CREATE_BUF;
/* populate the data buffer */
FILE_CREATE_BUF dataBuf;
memset (dataBuf, 0, size of (databuf)); /* initialize databuf */
dataBuf.recLength = 72;
dataBuf.pageSize = 4096;
dataBuf.indexCount = 2;
dataBuf.keySpecs[0].position = 1;
dataBuf.keySpecs[0].length = 25;
dataBuf.keySpecs[0].flags = EXTTYPE_KEY + NOCASE_KEY + DUP + MOD;
dataBuf.keySpecs[0].type = ZSTRING;
dataBuf.keySpecs[1].position = 52;
dataBuf.keySpecs[1].length = 4;
dataBuf.keySpecs[1].flags = EXTTYPE_KEY;
dataBuf.keySpecs[1].type = INTEGER;
/* create the file */
strcpy((BTI_CHAR *)keyBuf, "c:\\sample\\sample2.mkd");
dataLen = sizeof(dataBuf);
status = BTRV(B_CREATE, posBlock, &dataBuf, &dataLen, keyBuf, 0);
```

Create Index Operation

If you create files with predefined keys, the indexes are populated with each insert, update, or delete. This is necessary for most database files. However, there is a class of database files that are fully populated before being read. These include temporary sort files and fully populated files that are delivered as part of a product.

For these files, it may be faster to build the keys with Create Index (31) after the records are written. The file should be created with no index defined so that inserting records can be accomplished faster. Then the Create Index operation will sort the keys and build the indexes in a more efficient manner.

Indexes created this way are also more efficient and provide faster access. There is no need for the MicroKernel Engine to leave empty space at the end of each page during a Create Index (31) because the index pages are loaded in key order. Each page is filled to nearly 100%. In contrast, when an Insert (2) or Update (3) operation fills an index page, the page is split in half, with half of the records being copied to a new page. This process causes the average index page to be about 50 to 65 percent full. If index balancing is used, it may be 65 to 75 percent full.

Another advantage of Create Index (31) is that all the new index pages created are located close together at the end of the file. For large files, this means less distance for the read head to cover between reads.

This technique may not be faster when files are small, such as a few thousand records. The file would also need larger index fields to benefit. Moreover, if all index pages can fit into the MicroKernel Engine cache, this method shows no improvement in speed. But if only a small percentage of the index pages are in cache at any one time, this method saves a lot of extra index page writes. This technique can greatly reduce the time needed to build a file with many records. The greater the number of index pages in the file, the faster it is to build indexes with Create Index (31) than it is one record insert at a time.

In summary, the critical thing to avoid in loading a Zen file from scratch is not enough cache memory to hold all the index pages. If this is the case, use Create (14) to create the file without indexes and use Create Index (31) when all the data records have been inserted.

See *Btrieve API Guide* for detailed information on these operations.

Calculating the Logical Record Length

You must supply the *logical record length* to Create (14). The logical record length is the number of bytes of *fixed-length* data in the file. To obtain this value, calculate how many bytes of data you need to store in the fixed-length portion of each record.

For example, the following table shows how the data bytes in the example Employees file are added together to obtain a logical record length.

Field	Length (in Bytes)
Last Name	25
First Name	25
Middle Initial	1
Employee ID	4
Phone Number	13
Pay Rate	4
Logical Record Length	72

In calculating the logical record length, you do not count variable-length data, because variable-length information is stored apart from the fixed-length record in the file (on variable pages).

The maximum logical record length depends on the file format as defined in the following table.

Zen Version	Example
13.0, 16.0	page size minus 18 minus 2 (record overhead)
9.5	page size minus 10 minus 2 (record overhead)
8.x through 9.x	page size minus 8 minus 2 (record overhead)
6.x through 7.x	page size minus 6 minus 2 (record overhead)
Pre-6.x	page size minus 6 minus 2 (record overhead)
Note: The record overhead in these examples is for a fixed-length record (not a variable record) without record compression.	

Choosing a Page Size

All pages in a data file are the same size. Therefore, when you determine the size of the pages in your file, you must answer the following questions:

- What is the optimum size for data pages, which hold the fixed-length portion of records to reduce wasted bytes?
- What is the smallest size that allows index pages to hold your largest key definition? (Even if you do not define keys for your file, the MicroKernel Engine adds a key if the Transaction Durability feature is enabled.)

The following topics guide you through answering these questions. With your answers, you can select a page size that best fits your file.

- Optimum Page Size For Minimizing Disk Space
- Minimum Page Size

Optimum Page Size For Minimizing Disk Space

Before you can determine the optimum page size for your file, you must first calculate the file's *physical record length*. The physical record length is the sum of the logical record length and the overhead required to store a record on a data page of the file. For more generalized information about page size, see Page Size.

The MicroKernel Engine always stores a minimum of 2 bytes of overhead information in every record as a usage count for that record. The MicroKernel Engine also stores an additional number of bytes in each record, depending on how you define the records and keys in your file.

Record Overhead in Bytes Without Record Compression

The following table shows how many bytes of record overhead required without record compression, depending on the characteristics of your file.

File Characteristic	File Format			
	6.x	7.x	8.x	9.0, 9.5, 13.0, 16.0
Usage Count	2	2	2	2
Duplicate Key (per key)	8	8	8	8
Variable Pointer (with variable record)	4	4	6	6
Record Length (if VATs[1] used)	4	4	4	4

File Characteristic	File Format			
	6.x	7.x	8.x	9.0, 9.5, 13.0, 16.0
Blank Truncation Use (without VATs/with VATs)	2/4	2/4	2/4	2/4
System Data	n/a[2]	8	8	8

[1]VAT: variable-tail allocation table
[2]n/a: not applicable

Record Overhead in Bytes With Record Compression

The following table shows how many bytes of record overhead required when using record compression, depending on the characteristics of your file.

File Characteristic	File Format			
	6.x	7.x	8.x	9.0, 9.5, 13.0, 16.0
Usage Count	2	2	2	2
Duplicate Key (per key)	8	8	8	8
Variable Pointer	4	4	6	6
Record Length (if VATs[1] used)	4	4	4	4
Record Compression Flag	1	1	1	1
System Data	n/a[2]	8	8	8

[1]VAT: variable-tail allocation table
[2]n/a: not applicable

Page Overhead in Bytes

The following table shows the bytes of page overhead required depending on the page type.

Page Type	File Format					
	6.x	7.x	8.x	9.0	9.5	13.0, 16.0
Data	6	6	8	8	10	18
Index	12	12	14	14	16	24

Page Type	File Format					
	6.x	7.x	8.x	9.0	9.5	13.0, 16.0
Variable	12	12	16	16	18	26

Physical Record Length Worksheet

The following table shows how many bytes of overhead you must add to the logical record length to obtain the physical record length (based on how you define the records and keys for your file). You can also find a summary of this record overhead information in other tables in this topic.

	Task Description	Example
1	Determine logical record length. For steps, see Calculating the Logical Record Length. The example file uses a logical record length of 72 bytes. For files with variable-length records, this length refers only to the fixed-length portion of the record.	72
2	Add 2 for the record usage count. For a compressed record's entry, you need to add the usage count plus the variable pointer plus the record compression flag: 6.x and 7.x: 7 bytes (2 + 4 + 1) 8.x and later: 9 bytes (2 + 6 + 1)	72 + 2 = 74
3	For each linked-duplicatable key, add 8. When calculating the number of bytes for duplicatable keys, the MicroKernel Engine does not allocate duplicate pointer space for keys defined as repeating duplicatable at creation time. By default, keys that allow duplicates created at file creation time are linked-duplicate keys. For a compressed record's entry, add 9 (nine) for pointers for duplicate keys. The example file has one linked-duplicatable key.	74 + 8 = 82
4	For each reserved duplicate pointer, add 8. The example file has no reserved duplicate pointers.	82 + 0 = 82
5	If the file allows variable-length records, add 4 for pre-8.x files and 6 for 8.x or later. The example file does not allow variable-length records.	82 + 0 = 82
6	If the file uses VATs, add 4. The example file does not use VATs.	82 + 0 = 82
7	If the file uses blank truncation, add one of the following: • 2 if the file does not use VATs • 4 if the file uses VATs. The example file does not use blank truncation.	82 + 0 = 82

Task Description		Example
8	If the file uses system data or system data v2: • System data, add 8 • System data v2, add 16 The example file does not use system data.	82 + 0 = 82
	Physical Record Length	82

Using the physical record length, you now can determine the file's optimum page size for data pages.

The MicroKernel Engine stores the fixed length portion of a data record in the data pages; however, it does not break the fixed-length portion of a record across pages. Also, in each data page, the MicroKernel Engine stores overhead information as described under Optimum Page Size For Minimizing Disk Space. You must account for this additional overhead when determining the page size.

A file contains unused space if the page size you choose minus the overhead information amount is not an exact multiple of the physical record length. You can use the formula to find an efficient page size:

Unused bytes = (Page Size minus Data Page Overhead) mod (Physical Record Length)

To optimize your file's use of disk space, select a page size that can buffer your records with the least amount of unused space. The supported page size varies with the file format. See Optimum Page Size Example. If the internal record length (user data + record overhead) is small and the page size is large, the wasted space could be substantial.

Optimum Page Size Example

Consider an example where the physical record length is 194 bytes. The following table shows how many records can be stored on a page and how many bytes of unused space remain for each possible page size.

Applicable File Format	Page Size	Records per Page	Unused Bytes	
Pre-8.x	512	2	118	(512 – 6) mod 194
8.x through 9.0			116	(512 – 8) mod 194

Applicable File Format	Page Size	Records per Page	Unused Bytes	
Pre-8.x	1024	5	48	(1024 – 6) mod 194
8.x through 9.0			46	(1024 – 8) mod 194
9.5			44	(1024 – 10) mod 194
Pre-8.x	1536	7	172	(1536 – 6) mod 194
8.x through 9.0			172	(1536 – 6) mod 194
Pre-8.x	2048	10	102	(2048 – 6) mod 194
8.x through 9.0			100	(2048 – 8) mod 194
9.5			98	(2048 – 10) mod 194
Pre-8.x	2560	13	32	(2560 – 6) mod 194
8.x through 9.0				(2560 – 6) mod 194
Pre-8.x	3072	15		(3072 – 6) mod 194
8.x through 9.0				(3072 – 6) mod 194
Pre-8.x	3584	18		(3584 – 6) mod 194
8.x through 9.0			86	(3584 – 6) mod 194
Pre-8.x	4096	21	16	(4096 – 6) mod 194
8.x through 9.0			32	(4096 – 8) mod 194
9.5			156	(4096 – 10) mod 194
13.0, 16.0			156	(4096 – 18) mod 194
9.0	8192	42	86	(8192 – 8) mod 194
9.5			34	(8192 – 10) mod 194
13.0, 16.0			26	(8192 – 18) mod 194
9.5	16384	84	78	(16384 – 10) mod 194
13.0, 16.0			70	(16384 – 18) mod 194

For planning purposes, note that page and record overhead may rise in future file formats. Given a current file format, a record size that exactly fits on a page may require a larger page size the future. Also note that the database engine automatically upgrades the page size if the record and overhead cannot fit within a specified page size. For example, if you specify a page size of 4096

for a 9.x file, but the record and overhead requirement is 4632, then the engine will use a page size of 8192.

As the table above indicates, if you select a page size of 512, only 2 records can be stored per page and 114 to 118 bytes of each page are unused depending on the file format. However, if you select a page size of 4096, 21 records can be stored per page and only 16 bytes of each page are unused. Those same 21 records would result in over 2 KB of lost space with a page size of 512.

If you have a very small physical record length, most page sizes will result in very little wasted space. However, pre-8.x file versions have a maximum limit of 256 records per page. In that case, if you have a small physical record length, and if you choose a larger page size (for example, 4096 bytes), it will result in a large amount of wasted space. For example, the following table shows the behavior of 14-byte record length for a pre-8.x file version.

Page Size	Records per Page	Unused Bytes	
512	36	2	(512 – 6) mod 14
1024	72	10	(1024 – 6) mod 14
1536	109	4	(1536 – 6) mod 14
2048	145	12	(2048 – 6) mod 14
2560	182	6	(2560 – 6) mod 14
3072	219	0	(3072 – 6) mod 14
3584	255	8	(3584 – 6) mod 14
4096	256	506	(4096 – 6) mod 14

Minimum Page Size

The page size you choose must be large enough to hold eight key values (plus overhead). To find the smallest page size allowable for your file, add the values specified in the following minimum page size worksheet, which uses a 9.5 file format as an example.

Task Description	Example
1 Determine the size of the largest key in the file, in bytes. (Using the example Employee file, the largest key is 25 bytes.) In files that do not have a unique key defined, the system-defined log key (also called system data) may be the largest key. Its size is 8 bytes.	25
2 Add one of the following: • For keys that do not allow duplicates or that use repeating duplicates, add 8 bytes. • For keys that use linked duplicates, add 12 bytes. (This example uses linked duplicates.)	25 + 12 = 37
3 Multiply the result by 8. (The MicroKernel Engine requires room for a minimum of 8 keys on a page.)	37 * 8 = 296
4 Add index page overhead for the file format: See table on page overhead in bytes for index pages under Choosing a Page Size.	296 + 16 = 312
MINIMUM PAGE SIZE	312 bytes

Select any valid page size that is equal to or greater than the result. Remember that the page size you select must accommodate the size of any keys created after file creation. The total number of key segments may dictate the minimum page size. For example, you can have only eight key segments defined in a file using a 512-page size, as shown here:

Page Size (bytes)	Number of Key Segments by File Version				
	6.x and 7.x	8.x	9.0	9.5	13.0, 16.0
512	8	8	8	Rounded up[2]	Rounded up[2]
1024	23	23	23	97	Rounded up[2]
1536	24	24	24	Rounded up[2]	Rounded up[2]
2048	54	54	54	97	Rounded up[2]
2560	54	54	54	Rounded up[2]	Rounded up[2]

Page Size (bytes)	Number of Key Segments by File Version				
	6.x and 7.x	8.x	9.0	9.5	13.0, 16.0
3072	54	54	54	Rounded up[2]	Rounded up[2]
3584	54	54	54	Rounded up[2]	Rounded up[2]
4096	119	119	119	204[3]	183[3]
8192	n/a	n/a[1]	119	420[3]	378[3]
16384	n/a	n/a[1]	n/a[1]	420[3]	378[3]

[1]"n/a" stands for "not applicable"

[2]"Rounded up" means that the page size is rounded up to the next size supported by the file version. For example, 512 is rounded up to 1024, 2560 is rounded up to 4096, and so forth.

[3]A 9.5 format or later file can have more than 119 segments, but the number of indexes is limited to 119.

Estimating File Size

You can estimate the number of pages, and therefore the number of bytes required to store a file. However, when using the formulas, consider that they only approximate file size because of the way the MicroKernel Engine dynamically manipulates pages.

Note: The following discussion and the formulas for determining file size do not apply to files that use data compression, because the record length for those files depends on the number of repeating characters in each record.

While the formulas are based on the maximum storage required, they assume that only one task is updating or inserting records into the file at a time. File size increases if more than one task updates or inserts records into the file during simultaneous concurrent transactions.

The formulas also assume that no records have been deleted yet from the file. You can delete any number of records in the file, and the file remains the same size. The MicroKernel Engine does not deallocate the pages that were occupied by the deleted records. Rather, the MicroKernel Engine reuses them as new records are inserted into the file (before allocating new pages).

If the final outcome of your calculations contains a fractional value, round the number to the next highest whole number.

Formula and Derivative Steps

The following formula is use to calculate the maximum number of bytes required to store the file. The "see step" references are to the steps following the formula that explain its individual components.

File size in bytes =

```
(page size *
   (number of data pages [see step 1] +
    number of index pages [see step 2] +
    number of variable pages [see step 3] +
    number of other pages [see step 4] +
    number of shadow pool pages [see step 5] ))
```

```
+ (special page size [see step 6] *
  (number of PAT pages [see step 7] +
   number of FCR pages + number of reserved pages [see step 8] ))
```

Determining file size requires that you account for two different categories of pages. The standard page category includes the pages when a data file is first created (see also Create (14) in *Btrieve API Guide*). In addition, the formula must account for special (nonstandard) pages as listed in the

table Page Sizes of Special Pages by File Format. The special pages are not always a multiple of the file page size.

1. Calculate the number of data pages using the following formula.

```
Number of data pages =
#r /
( (PS - DPO) / PRL )
```

   where:

   - #r is the number of records
   - PS is the page size
   - DPO is the data page overhead (see Choosing a Page Size)
   - PRL is the physical record length (see Choosing a Page Size)

2. Calculate the number of index pages for each defined key using one of the following formulas.

   For each index that does *not* allow duplicates or that allows repeating-duplicatable keys:

```
Number of index pages =
( #r /
( (PS - IPO) / (KL + 8) ) ) * 2
```

   where:

   - #r is the number of records
   - PS is the page size
   - IPO is the index page overhead (see Choosing a Page Size)
   - KL is the key length

   For each index that allows linked-duplicatable keys:

```
Number of index pages =
(  #UKV /
( (PS - IPO) / (KL + 12) ) ) * 2
```

   where:

   - #UKV is the number of unique key values
   - PS is the page size
   - IPO is the index page overhead (see Choosing a Page Size)
   - KL is the key length

The B-tree index structure guarantees at least 50 percent usage of the index pages. Therefore, the index page calculations multiply the minimum number of index pages required by 2 to account for the maximum size.

3. If your file contains variable-length records, calculate the number of variable pages using the following formula:

```
Number of variable pages =
(AVL * #r) / (1 - (FST + (VPO/PS))
```

where:

- AVL is the average length of the variable portion of a typical record
- #r is the number of records
- FST is the free space threshold specified when the file is created (see also Create (14) in *Btrieve API Guide*)
- VPO is the variable page overhead (see Choosing a Page Size)
- PS is the page size

Note: You can gain only a very rough estimate of the number of variable pages due to the difficulty in estimating the average number of records whose variable-length portion fit on the same page.

4. Calculate the number for any other regular pages:
   - 1 page for each alternate collating sequence used (if any)
   - 1 page for a referential integrity (RI) page if the file has RI constraints.

   The sum of these first four steps estimates the total number of logical pages in the file.

5. Calculate the estimated number of pages in the shadow page pool. The database engine uses a pool for shadow paging. Use the following formula to estimate its number of pages:

   Size of the shadow page pool = (number of keys + 1) * (average number of inserts, updates, and deletes) * (number of concurrent transactions)

   This formula applies if tasks execute insert, update, and delete operations outside transactions. If tasks are executing these operations *inside* transactions, multiply the average number of insert, update, and delete operations expected in the inside transactions times the outside-transactional value determined by the formula. You must further increase the estimated size of the pool of unused pages if tasks are executing simultaneous concurrent transactions.

6. Determine page size using the table Page Sizes of Special Pages by File Format at the end of these steps. Depending on the file format version, the pages sizes for FCR, reserved, and PAT pages are different from the normal pages sizes for data, index, and variable pages.

7. Calculate the number of Page Allocation Table (PAT) pages.

   Every file has a minimum of two PAT pages. To calculate the number of PAT pages in a file, use one of the following formulas:

   For pre-8.x file formats:

```
Number of PAT pages =
(((sum of pages from steps 1 through 3) * 4) /
(page size - 8 bytes for overhead)) * 2
```

   For 8.x or later file formats:

```
Number of PAT pages =
2 * (sum of pages from steps 1 through 3 /
number of PAT entries)
```

   For number of PAT entries, see the table Page Sizes of Special Pages by File Format.

8. Include 2 pages for the file control record (FCR) pages (see also File Control Record (FCR)). If you are using 8.x or later file format, also include 2 pages for the reserved pages.

Page Sizes of Special Pages by File Format

Normal Page Size	File Format v6.x and 7.x		File Format 8.x		File Format 9.0 through 9.4		File Format 9.5, 13.0, 16.0	
	Special Page Size	PAT Page Entries	Special Page Size	PAT Page Entries	Special Page Size	PAT Page Entries	Special Page Size	PAT Page Entries
512	512	n/a	2048	320	2048	320	n/a	n/a
1024	1024	n/a	2048	320	2048	320	4096	480
1536	1536	n/a	3072	480	3072	480	n/a	n/a
2048	2048	n/a	4096	640	4096	640	4096	480
2560	2560	n/a	5120	800	5120	800	n/a	n/a
3072	3072	n/a	6144	960	6144	960	n/a	n/a
3584	3584	n/a	7168	1120	7168	1120	n/a	n/a
4096	4096	n/a	8192	1280	8192	1280	8192	1280
8192	n/a	n/a	n/a	n/a	n/a	n/a	16384	16000
16384	n/a	n/a	n/a	n/a	n/a	n/a	16384	16000

"n/a" stands for "not applicable"

Optimizing Your Database

The MicroKernel Engine provides several features that allow you to conserve disk space and improve system performance. These features include the following:

- Duplicatable Keys
- Page Preallocation
- Blank Truncation
- Record Compression
- Index Balancing
- Variable-tail Allocation Tables
- Key-Only Files

Duplicatable Keys

If you define a key to be duplicatable, the MicroKernel Engine allows more than one record to have the same value for that key. Otherwise, each record must have a unique value for the key. If one segment of a segmented key is duplicatable, all the segments must be duplicatable.

Linked-Duplicatable Keys

By default in a 7.0 or later file, the MicroKernel Engine stores duplicatable keys as *linked-duplicatable* keys. When the first record with a duplicate key value is inserted into a file, the MicroKernel Engine stores the key value on an index page. The MicroKernel Engine also initializes two pointers to identify the first and last record with this key value. Additionally, the MicroKernel Engine stores a pair of pointers at the end of the record on the data page. These pointers identify the previous and next record with the same key value. When you create a file, you can reserve pointers for use in creating linked-duplicatable keys in the future.

If you anticipate adding duplicatable keys after you create a data file and you want the keys to use the linked-duplicatable method, you can preallocate space for pointers in the file.

Repeating-Duplicatable Keys

If no room is available to create a linked-duplicatable key (that is, if no duplicate pointers are available), the MicroKernel Engine creates a *repeating-duplicatable* key. The MicroKernel Engine stores every key value of a repeating-duplicatable key both on a data page and on an index

page. In other words, the key's value resides in the record on a data page and is repeated in the key entry on an index page.

Note: For users of pre-6.0 Btrieve, the term *linked-duplicatable* corresponds to *permanent*, and the term *repeating-duplicatable* corresponds to *supplemental*.

You can define a key as a repeating-duplicatable key by setting bit 7 (0x80) of the key flags in the key specification block on a Create (14) or Create Index (31) operation. Before 6.10, you could not define a key to be a repeating-duplicatable key, and bit 7 of the key flags was not user-definable. In 6.0 and later, the Stat (15) operation sets bit 7 if no room is available to create a linked-duplicatable key (and therefore, if the MicroKernel Engine has to create the key as a repeating-duplicatable key).

Files that use the 5.x format use this same key flag (called the *supplemental key attribute* in 5.x) to designate that a key was created with the 5.x Create Supplemental Index (31) operation.

Note: In pre-6.0 files, you can drop supplemental indexes only. Permanent indexes, as their name implies, cannot be dropped. In 6.0 and later files, you can drop any index.

Linked vs. Repeating

Each method has performance advantages:

- Linked-duplicatable keys provide faster lookup, because the MicroKernel Engine reads fewer pages.
- Repeating-duplicatable keys result in fewer index page conflicts when multiple users access the same page concurrently.

There are trade-offs in performance between linked-duplicate keys and repeating-duplicate keys. Generally, if the average number of duplicates of a key is two or more, a linked-duplicate key will take up less space on the disk and searches will be generally faster because there are fewer index pages. However, the opposite is true if your file stores very few records with duplicate keys and the key length is very short. This is because an entry in the linked-duplicatable tree requires 8 bytes for pointers, whereas a repeating-duplicatable key entry requires 4 bytes.

If only a small percentage of keys have any duplicates, it is advantageous to use repeating-duplicate keys to save the extra 8 bytes in each data record. There is no noticeable performance advantage to either choice when the average number of duplicates of a key is less than two.

If you expect several concurrent transactions to be active on the same file at the same time, repeating-duplicate keys will provide a greater probability that these transactions do not try to access the same pages. All pages involved in writes during a concurrent transaction have implicit locks on them. When a page is required in order to make a change during a concurrent transaction

and the page is involved in another concurrent transaction, the MicroKernel Engine might wait until the other transaction is complete. If these kind of implicit waits happen very often, the performance of the application goes down.

Neither key storage method is recommended as a shortcut for tracking chronological order. For linked-duplicatable keys, the MicroKernel Engine does maintain the chronological order of records that are inserted after the key is created, but if you rebuild the key's index, the chronological order is lost. For repeating-duplicatable keys, the MicroKernel Engine maintains the chronological order of the records only if there was no record deletion between the time the key is built and a new record is inserted. To track chronological order, use an AUTOINCREMENT data type on the key.

Page Preallocation

Preallocation guarantees that disk space is available when the MicroKernel Engine needs it. The MicroKernel Engine allows you to preallocate up to 65535 *pages* to a file when you create a data file. The following table shows the maximum number of *bytes* that the MicroKernel Engine allocates for a file of each possible page size, assuming you allocate 65535 full pages of disk space.

Page Size	Disk Space Allocated[1]
512	33,553,920
1024	67,107,840
1536	100,661,760
2048	134,215,680
2560	167,769,600
3072	201,323,520
3584	243,877,440
4096	268,431,360
8192	536,862,720
16384	1,073,725,440

[1]Value is page size multiplied by 65535.

If not enough space exists on the disk to preallocate the number of pages you specify, the MicroKernel Engine returns status code 18 (disk full) and does not create the file.

The speed of file operations can be enhanced if a data file occupies a *contiguous* area on the disk. The increase in speed is most noticeable on very large files. To preallocate contiguous disk space for a file, the device on which you are creating the file must have the required number of bytes of contiguous free space available. The MicroKernel Engine preallocates the number of pages you specify, whether or not the space on the disk is contiguous.

Use the formulas described earlier in this chapter to determine the number of data and index pages the file requires. Round any remainder from this part of the calculation to the next highest whole number.

When you preallocate pages for a file, that file actually occupies that area of the disk. No other data file can use the preallocated area of disk until you delete or replace that file.

As you insert records, the MicroKernel Engine uses the preallocated space for data and indexes. When all the preallocated space for the file is in use, the MicroKernel Engine expands the file as new records are inserted.

When you issue Stat (15), the MicroKernel Engine returns the difference between the number of pages you allocated at the time you created the file and the number of pages that the MicroKernel Engine currently has in use. This number is always less than the number of pages you specify for preallocation because the MicroKernel Engine considers a certain number of pages to be in use when a file is created, even if you have not inserted any records.

Once a file page is in use, it remains in use even if you delete all the records stored on that page. The number of unused pages that the Stat operation returns never increases. When you delete a record, the MicroKernel Engine maintains a list of free space in the file and reuses the available space when you insert new records.

Even if the number of unused pages that the Stat operation returns is zero, the file might still have free space available. The number of unused pages can be zero if one of the following is true:

- You did not preallocate any pages to the file.
- All the pages that you preallocated were in use at one time or another.

Blank Truncation

If you choose to truncate blanks, the MicroKernel Engine does not store any trailing blanks (ASCII Space code 32 or hexadecimal 0x20) in the variable-length portion of the record when it writes the record to a file. Blank truncation has no effect on the fixed-length portion of a record. The MicroKernel Engine does not remove blanks that are embedded in the data.

When you read a record that contains truncated trailing blanks, the MicroKernel Engine expands the record to its original length. The value the MicroKernel Engine returns in the Data Buffer

Length parameter includes any expanded blanks. Blank truncation adds either 2 bytes or 4 bytes of overhead to the physical size of the record (stored with the fixed-record portion): 2 if the file does not use VATs, 4 if it does.

Record Compression

When you create a file, you can specify whether you want the MicroKernel Engine to compress the data records when it stores them in the file. Record compression can result in a significant reduction of the space needed to store records that contain many repeating characters. The MicroKernel Engine compresses five or more of the same contiguous characters into 5 bytes.

Consider using record compression in the following circumstances:

- The records to be compressed are structured so that the benefits of using compression are maximized.
- The need for better disk utilization outweighs the possible increased processing and disk access times required for compressed files.
- The computer running the MicroKernel Engine can supply the extra memory that the MicroKernel Engine uses for compression buffers.

Note: The database engine automatically uses record compression on files that use system data and have a record length that exceeds the maximum length allowed. See Record Length.

When you perform record I/O on compressed files, the MicroKernel Engine uses a compression buffer to provide a block of memory for the record compression and expansion process. To ensure sufficient memory to compress or expand a record, the MicroKernel Engine requires enough buffer space to store twice the length of the longest record your task inserts into the compressed file. This requirement can have an impact on the amount of free memory remaining in the computer after the MicroKernel Engine loads. For example, if the longest record your task writes or retrieves is 64 KB long, the MicroKernel Engine requires 128 KB of memory to compress and expand that record.

Note: If your file uses VATs, the MicroKernel Engine requirement for buffer space is the product of 16 times the file's page size. For example, on a 4 KB record, 64 KB of memory are required to compress and expand the record.

Because the final length of a compressed record cannot be determined until the record is written to the file, the MicroKernel Engine always creates a compressed file as a variable-length record file. In the data page, the MicroKernel Engine stores either 7 bytes (if the file does not use VATs) or 11 bytes (if it does), plus an additional 8 bytes for each duplicate key pointer. The MicroKernel Engine then stores the record on the variable page. Because the compressed images of the records

are stored as variable-length records, individual records may become fragmented across several file pages if your task performs frequent inserts, updates, and deletes. The fragmentation can result in slower access times because the MicroKernel Engine may need to read multiple file pages to retrieve a single record.

The record compression option is most effective when each record has the potential for containing a large number of repeating characters. For example, a record may contain several fields, all of which may be initialized to blanks by your task when it inserts the record into the file. Compression is more efficient if these fields are grouped together in the record, rather than being separated by fields containing other values.

To use record compression, the file must have been created with the compression flag set. Key-only files do not allow compression.

Index Balancing

The MicroKernel Engine allows you to further conserve disk space by employing *index balancing*. By default, the MicroKernel Engine does not use index balancing, so that each time a current index page is filled, the MicroKernel Engine must create a new index page. When index balancing is active, the MicroKernel Engine can frequently avoid creating a new index page each time a current index page is filled. Index balancing forces the MicroKernel Engine to look for available space on adjoining index pages. If space is available on one of those pages, the MicroKernel Engine moves keys from the full index page to the page with free space.

The balancing process not only results in fewer index pages but also produces more densely populated indexes, better overall disk usage, and faster response on most read operations. If you add keys to the file in sorted order, index page usage increases from 50 percent to nearly 100 percent when you use index balancing. If you add keys randomly, the minimum index page usage increases from 50 percent to 66 percent.

On insert and update operations, the balancing logic requires the MicroKernel Engine to examine more pages in the file and might possibly require more disk I/O. The extra disk I/O slows down file updates. Although the exact effects of balancing indexes vary in different situations, using index balancing typically degrades performance on write operations by about 5 to 10 percent.

The MicroKernel Engine allows you to fine-tune your MicroKernel Engine environment by offering two ways to turn on index balancing: at the engine level or at the file level. If you specify the Index Balancing configuration option during setup, the MicroKernel Engine applies index balancing to every file. For a description of how to specify the Index Balancing configuration option, see *Zen User's Guide*.

You can also designate that only specific files are to be index balanced. To do so, set bit 5 (0x20) of the file's file flags at creation time. If the index Balancing configuration option is off when the MicroKernel Engine is started, the MicroKernel Engine applies index balancing only to indexes on files that have bit 5 of the file flags set.

The MicroKernel Engine ignores bit 5 in all files' file flags if the index balancing configuration option was on when the MicroKernel Engine was started. In this situation, the MicroKernel Engine applies index balancing to every file.

Files remain compatible regardless of whether index balancing is active. Also, you do not have to specify index balancing to access files that contain balanced index pages. If you turn on the MicroKernel Engine index balancing option, index pages in existing files are not affected until they become full. The MicroKernel Engine does not rebalance indexes on existing files as a result of enabling this option. Similarly, turning off the index balancing option does not affect existing indexes. Whether this option is on or off determines how the MicroKernel Engine handles full index pages.

Variable-tail Allocation Tables

Variable-tail allocation tables give the MicroKernel Engine faster access to data residing at large offsets in very large records and significantly reduce the buffer sizes the MicroKernel Engine needs when processing records in files that use data compression. Using a record's VAT, the MicroKernel Engine can divide the variable-length portion of a record into smaller portions and then store those portions in any number of variable tails. The MicroKernel Engine stores an equal amount of the record's data in each of the record's variable tails (except the final variable tail). The MicroKernel Engine calculates the amount to store in each variable tail by multiplying the file's page size by 8. The last variable tail contains any data that remains after the MicroKernel Engine divides the data among the other variable tails.

Note: The formula for finding the length of a variable tail (eight times the page size) is an implementation detail that may change in future versions of the MicroKernel Engine.

In a file that employs VATs and has a 4096-byte page size, the first variable tail stores the bytes from offset 0 through 32767 of the record's variable portion, the second tail stores offsets 32768 through 65535, and so on. The MicroKernel Engine can use the VAT to accelerate a seek to a large offset in a record because the VAT allows it to skip the variable tails containing the record's lower-offset bytes.

An application specifies whether a file will use VATs when it creates the file. If your application uses chunk operations on huge records (larger than eight times the physical page size) and accesses chunks in a random, nonsequential fashion, VATs may improve your application's performance. If your application uses whole record operations, VATs do not improve performance

because the MicroKernel Engine reads or writes the record sequentially; the MicroKernel Engine skips no bytes in the record.

If your application uses chunk operations but accesses records sequentially (for example, it reads the first 32 KB of a record, then reads the next 32 KB of the record, and so on until the end of the record), VATs do not improve performance, because the MicroKernel Engine saves your position in a record between operations, thereby eliminating the need to seek at the beginning of a chunk operation.

VATs also provide another advantage. When the MicroKernel Engine reads or writes a compressed record, it uses a buffer that must be up to twice as large as the record's uncompressed size. If the file has VATs, that buffer needs to be only as large as two variable tails (16 times the physical page size).

Key-Only Files

In a key-only file, the entire record is stored with the key, so no data pages are required. Key-only files are useful when your records contain a single key, and that key takes up most of the record. Another common use for a key-only file is as an external index for a standard file.

The following restrictions apply to key-only files:

- Each file can contain only a single key.
- The maximum record length you can define is 253 bytes.
- Key-only files cannot use data compression.
- Step operations do not function with key-only files.
- Although you can do a Get Position on a record in a key-only file, that position will change whenever the record is updated.

Key-only files contain only File Control Record pages followed by a number of PAT pages and index pages. If a key-only file has an ACS, it may also have an ACS page. If you use ODBC to define referential integrity constraints on the file, the file may contain one or more variable pages, as well.

Setting Up Security

The MicroKernel Engine provides three methods of setting up file security:

- Assign an owner name to the file
- Open the file in exclusive mode
- Use the Zen Control Center (ZenCC) security settings

In addition, the MicroKernel Engine supports the native file-level security (if available) on the server platforms.

Note: *Windows developers*: File-level security is available on the server if you installed the NTFS file system on your server. File system security is *not* available if you installed the FAT file system.

The MicroKernel Engine provides the following features for enhancing data security.

Owner Names

The MicroKernel Engine allows you to restrict access to a file by assigning an owner name using the Set Owner operation (see Set Owner (29) in *Btrieve API Guide*.) Once you assign an owner name to a file, the MicroKernel Engine requires that the name be provided to access the file. This prevents users or applications that do not provide the owner name from having unauthorized access to or changing file contents.

Likewise, you can clear the owner name from a file if you know the owner name assigned to it.

Owner names are case-sensitive and can be short or long. A short owner name can be up to 8 bytes long, and a long one up to 24 bytes. For more information, see Owner Names in *Advanced Operations Guide*.

You can restrict access to the file in these ways:

- Users can have read-only access *without* supplying an owner name. However, neither a user nor a task can change the file contents without supplying the owner name. Attempting to do so causes the MicroKernel Engine to return an error.
- Users can be required to supply an owner name for any access mode. The MicroKernel Engine restricts all access to the file unless the correct owner name is supplied.

When you assign an owner name, you can also request that the database engine encrypt the data in the disk file using the owner name as the encryption key. Encrypting the data on the disk ensures that unauthorized users cannot examine your data by using a debugger or a file dump utility.

When you use the Set Owner operation and select encryption, the encryption process begins immediately. The MicroKernel Engine has control until the entire file is encrypted, and the larger the file, the longer the encryption process takes. Because encryption requires additional processing time, you should select this option only if data security is important in your environment.

You can use the Clear Owner (30) operation to remove ownership restrictions from a file if you know the owner name assigned to it. In addition, if you use the Clear Owner operation on an encrypted file, the database engine decrypts it.

Note: Set Owner (29) and Clear Owner (30) do not process hexadecimal long owner names. Any owner name submitted to them is treated as an ASCII string.

Exclusive Mode

To limit access to a file to a single client, you can specify that the MicroKernel Engine open the file in exclusive mode. When a client opens a file in exclusive mode, no other client can open the file until the client that opened the file in exclusive mode closes it.

SQL Security

See Database URIs for information on database Uniform Resource Indicator (URI) strings. See the *Zen User Guide* for how to access the ZenCC security settings.

Language Interfaces Modules

This section covers language interface source modules provided in the Zen SDK installation option.

We provide the source code for each language interface. You can find additional information in the source modules themselves.

Visit the Actian website for online resources for developers, including articles and sample code for the various language interfaces.

Continue reading for information on specific interface modules:

- Interface Modules Overview
- C/C++
- Delphi
- DOS (Btrieve)
- Pascal
- Visual Basic

Interface Modules Overview

If your programming language does not have an interface, check if your compiler supports mixed language calls. If so, you may be able to use the C interface. The following table lists the language interface source modules for Btrieve.

Language	Compiler	Source Module
C/C++	• Most C/C++ compilers, including Embarcadero, Microsoft, and WATCOM. This interface provides multiple platform support. • Embarcadero C++ Builder	• BlobHdr.h (for Extended DOS platforms using Embarcadero or Phar Lap only) • BtiTypes.h (platform-independent data types) • BtrApi.h (Btrieve function prototypes) • BtrApi.c (MicroKernel Engine code for all platforms) • BtrConst.h (common Btrieve constants) • BtrSamp.c (sample program) • Btrvexid.h (entry point declarations) • CBBtrv.cpp • CBBtrv.mak • CBBMain.cpp • CBBMain.dfm • CBBMain.h
Delphi	• Embarcadero Delphi 1 • Embarcadero Delphi 3 and above	• BtrConst.pas (common Btrieve constants) • Btr32.dpr • Btr32.dof • BtrSam32.dfm • BtrSam32.pas (sample program) • BtrApi32.pas • BtrConst.pas (common Btrieve constants)
Pascal	• Borland Turbo Pascal 5 – 6 • Borland Pascal 7 for DOS • Extended DOS Pascal for Turbo Pascal 7 • Borland Turbo Pascal 1.5 • Borland Pascal 7 for Windows	• BtrApid.pas • BtrSampd.pas (sample program) • BtrConst.pas (common Btrieve constants) • BlobHdr.pas • BtrApiw.pas • BtrSampw.pas (sample program)
Visual Basic	• Microsoft Visual Basic for Windows NT and Windows 9X	• BtSamp32.vbp • BtrSam32.bas (sample program) • BtrFrm32.frm

The following table provides a comparison of some common data types used in data buffers for Btrieve operations, such as Create and Stat.

Assembly	C	COBOL	Delphi	Pascal	Visual Basic
quadword	long long[1]	—	—	—	—
doubleword	long[1]	PIC 9(4)	longint[1]	longint[1]	Long integer
word	short int[1]	PIC 9(2)	smallint[1]	integer[1]	Integer
byte	char	PIC X	char	char	String
byte	unsigned char	PIC X	byte	byte	Byte

[1]The value of integers depends on the environment in which you develop. In 32-bit environments, integers are the same as long integers. In 16-bit environments, integers are the same as short, or small, integers.

Programming Notes

Calling a Btrieve function always returns an INTEGER value that corresponds to a status code. After a Btrieve call, your application should always check the value of this status code. A status code 0 indicates a successful operation. Your application must be able to recognize and resolve a nonzero status.

Although you must provide all parameters on every call, the MicroKernel does not use every parameter for every operation. See the *Btrieve API Guide* for a more detailed description of the parameters that are relevant for each operation.

C/C++

This section provides C/C++ module information for the Btrieve API.

The C/C++ interface facilitates writing platform-independent applications. This interface supports development on DOS, Windows, and Linux. These modules also are documented under BTITYPES.H.

Interface Modules

This topic describes in detail the modules that comprise the C language interface.

BTRAPI.C

The file BTRAPI.C is the actual implementation of the C application interface. It provides support for all applications that call BTRV and BTRVID. When making a Btrieve call with either of these functions, compile BTRAPI.C and link its object with the other modules in your application.

The BTRAPI.C file contains `#include` directives that instruct your compiler to include BTRAPI.H, BTRCONST.H, BLOBHDR.H, and BTITYPES.H. By including these files, BTRAPI.C takes advantage of the data types that provide the platform independence associated with the interface.

BTRAPI.H

The file BTRAPI.H contains the prototypes of the Btrieve functions. The prototype definitions use the platform-independent data types defined in the file BTITYPES.H. BTRAPI.H provides support for all applications calling the BTRV and BTRVID functions.

BTRVEXID.H

This file contains the declarations of the Btrieve entry point functions for BTRVEX and BTRVEXID.

BTRCONST.H

The file BTRCONST.H contains useful constants specific to Btrieve. These constants can help you standardize references to Btrieve operation codes, status codes, file specification flags, key specification flags, and many more items.

You can use the C application interface without taking advantage of BTRCONST.H, however, including the file may simplify your programming effort.

BTITYPES.H

The file BTITYPES.H defines the platform-independent data types. By using the data types in BTITYPES.H on your Btrieve function calls, your application ports among operating systems.

BTITYPES.H also describes the switches you must use to indicate the operating system on which your application runs. The following table lists these operating system switches.

Operating System	Application Type	Switch
DOS	16-bit 32-bit with Tenberry Extender and BStub.exe 32-bit with Phar Lap 6 32-bit with Embarcadero PowerPack	BTI_DOS BTI_DOS_32R BTI_DOS_32P BTI_DOS_32B
Linux	32-bit	BTI_LINUX
Linux	64-bit	BTI_LINUX_64
Win32	32-bit Windows	BTI_WIN_32
Win64	64-bit Windows	BTI_WIN_64

BTRSAMP.C

The source file BTRSAMP.C is a sample Btrieve program that you can compile, link, and run on any of the operating systems listed under BTITYPES.H.

Programming Requirements

If you use the C application interface to make your application platform independent, you must use the data types described in BTITYPES.H. To see how these data types are used, see the file BTRSAMP.C.

Note: You must also specify a directive that identifies the operating system on which the program executes. The available values for the directive are listed in the header file BTITYPES.H. Specify the directive using the appropriate command line option for your compiler.

Delphi

The Btrieve Delphi modules are documented under BTITYPES.H.

DOS (Btrieve)

This section explains how a DOS application can use the Btrieve API.

Interface Modules

The following modules comprise the language interface for DOS applications using the Btrieve API.

BTRAPI.C

The file BTRAPI.C is the implementation of the C application interface. This file also contains the DOS interface:

#if defined(BTI_DOS)

BTI_API BTRVID(

BTI_WORD operation,

BTI_VOID_PTR posBlock,

BTI_VOID_PTR dataBuffer,

BTI_WORD_PTR dataLength,

BTI_VOID_PTR keyBuffer,

BTI_SINT keyNumber,

BTI_BUFFER_PTR clientID)

BTRAPI.C provides support for all applications that call BTRV and BTRVID. When making a Btrieve call with either of these functions, compile BTRAPI.C and link its object with the other modules in your application.

The BTRAPI.C file contains `#include` directives that instruct your compiler to include BTRAPI.H, BTRCONST.H, BLOBHDR.H, and BTITYPES.H. By including these files, BTRAPI.C takes advantage of the data types that provide the platform independence associated with the interface.

BTRAPI.H

The file BTRAPI.H contains the prototypes of the Btrieve functions. The prototype definitions use the platform-independent data types defined in the file BTITYPES.H. BTRAPI.H provides support for all applications calling the BTRV and BTRVID functions.

BTRCONST.H

The file BTRCONST.H contains useful constants specific to Btrieve. These constants can help you standardize references to Btrieve operation codes, status codes, file specification flags, key specification flags, and many more items.

You can use the C application interface without taking advantage of BTRCONST.H, however, including the file may simplify your programming effort.

BTITYPES.H

The file BTITYPES.H defines the platform-independent data types. By using the data types in BTITYPES.H on your Btrieve function calls, your application ports among operating systems.

BTITYPES.H also describes the switches you must use to indicate the DOS operating system on which your application runs. The following table lists these switches.

Operating System	Application Type	Switch
DOS	16-bit 32-bit with Tenberry Extender and BStub.exe[1] 32-bit with Phar Lap 6 32-bit with Embarcadero PowerPack	BTI_DOS BTI_DOS_32R BTI_DOS_32P BTI_DOS_32B

Pascal

The Btrieve API source modules for Pascal are described in the following topics.

Source Modules

The Pascal interface is comprised of the following source modules:

- BTRAPID.PAS – Btrieve functions interface unit for DOS.
- BTRCONST.PAS – Common Btrieve constants unit.
- BTRSAMPD.PAS – Sample Btrieve program for DOS.

BBTRAPID.PAS

BTRAPID.PAS contains the source code implementation of the Pascal application interface for DOS. This file provides support for applications calling Btrieve functions.

In order for Turbo Pascal to properly compile and link the MicroKernel Engine with the other modules in your application, you can compile BTRAPID.PAS to create a Turbo Pascal unit which you then list in the `uses` clause of your application source code.

BTRCONST.PAS

The file BTRCONST.PAS contains useful constants specific to Btrieve. These constants can help you standardize references to Btrieve operation codes, status codes, file specification flags, key specification flags, and many more items.

To use BTRCONST.PAS, you can compile it to create a Turbo Pascal unit which you then list in the `uses` clause of your application source code.

You can use the Pascal application interface without taking advantage of BTRCONST.PAS; however, using the file may simplify your programming effort.

BTRSAMPD.PAS

The source file BTRSAMPD.PAS is a sample Btrieve program that you can compile, link, and run.

Note: If your application uses Pascal record structures that contain variant strings, consider that odd-length elements in a Pascal record may require an extra byte of storage (even if the record is not packed). This is an important consideration when you define the record length for the Create (14) operation. See your Pascal reference manual for more information on record types.

Visual Basic

This section describes the Visual Basic source modules for the Btrieve API.

Visual Basic, when compiling a 32-bit application, aligns members of a UDT (user-defined data type) on 8-, 16-, or 32-bit boundaries, depending on the size of that particular member. Unlike structures, database rows are packed, meaning there is no unused space between fields. Because there is no way to turn alignment off, there must be some method to pack and unpack structures so that Visual Basic applications can access a database. The Zen Btrieve Alignment DLL, PALN32.DLL, is designed to handle this alignment issue.

In the case of Visual Basic, this language aligns elements at various multiples of bits. The following table provides various data types and shows how Visual Basic handles them:

Visual Basic Type	Type Constant	Typical Size (in bytes)	Boundary
Byte	FLD_BYTE	Any	1 byte (none)
String	FLD_STRING	Any	1 byte (none)
Boolean	FLD_LOGICAL	2	2 bytes
Integer	FLD_INTEGER	2	2 bytes
Currency	FLD_MONEY	4	4 bytes
Long	FLD_INTEGER	4	4 bytes
Single	FLD_IEEE	4	4 bytes
Double	FLD_IEEE	8	4 bytes

Programs access Btrieve calls in Visual Basic by calling the BTRCALL function. Access to this function is accomplished by including the BTRAPI.BAS module in your project. The rest of the functions required reside in PALN32.DLL.

To include PALN32.DLL in your project

- Select **Project** > **References** and select the **Zen Btrieve Alignment Library** module. If it is not shown, add it to the list first by selecting the browse button and locating the file.

The following table includes each function and the specific module it requires.

Function	Purpose	Location	Parameters	Returns
BTRCALL	To perform Btrieve operations	BTRAPI.BAS	• OP As Integer The Btrieve operation number as listed in the *Btrieve API Guide*. • Pb$ As String Stores the position block in a string used to retrieve or store records or to pass structures to Btrieve. • Db As Any Data buffer. This parameter is used to retrieve or store records or to pass structures to Btrieve. • DL As Long Length of data buffer. • Kb As Any Key buffer. • Kl As Integer Length of key buffer. • Kn As Integer Key number.	Integer The Btrieve status code returned by the operation. See *Status Codes and Messages* for more information about a specific code.
RowToStruct	Converts a row of bytes into a Visual Base UDT.	PALN32.DLL	• row (1 to n) As Byte Input array to retrieve packed data. • fld (1 to n) As FieldMap A FieldMap array used to determine data types of individual fields. • udt As Any The UDT to store the data. • udtSize As Long The size of the UDT. Generate this value using LenB( ).	Integer 0 if okay. Otherwise an error occurred.

Function	Purpose	Location	Parameters	Returns
SetFieldMap	Sets the members of a FieldMap element.	PALN32.DLL	• map As FieldMap An element of a FieldMap array. • dataType As Integer The field type, passes on the following constants: - FLD_STRING - FLD_INTEGER - FLD_IEEE - FLD_MONEY - FLD_LOGICAL - FLD_BYTE - FLD_UNICODE[1] • size As Long Size of the field, in bytes, as stored in the database.	Nothing
SetFieldMap FromDDF	Sets all the members of an array of the FieldMap type.	PALN32.DLL	• path As String A full path name to the data source. • table As String Name of the table. • userName As String Reserved, pass a null string (e.g. " "). • passwd As String Reserved, pass a null string (e.g. " "). • map (1 to n) As FieldMap The destination FieldMap array to fill. It must contain the exact number of elements as the number of fields in the record. • unicode As Integer 0 if strings are stored in record as ASCII. Otherwise, they are stored as Unicode.	Integer 0 if okay. Otherwise, an error occurred.

Function	Purpose	Location	Parameters	Returns
StructToRow	Converts a Visual Basic UDT to a row of bytes	PALN32.DLL	• row (1 to n) As Byte Output array to store packed data. • fld (1 to n) As FieldMap A FieldMap array used to determine data types of individual fields. • udt As Any The UDT from which to retrieve data. • udtSize As Long The size of the UDT. Generate this value using LenB( ).	Integer 0 if okay. Otherwise, an error occurred.

[1]The field type FLD_UNICODE is used to specify a field of Visual Basic Type 'String' that is to be stored in UNICODE both within the database row (packed structure) as well as within the UDT (user-defined data type). If the field type FLD_STRING is used, it will be converted into the default ANSI code page character set of the system in the database row, although UNICODE will be used in the UDT (user-defined data type). In short, if you wish to store the string field in UNICODE in your database, choose field type FLD_UNICODE. If you wish to store the string field in the database in the default ANSI code page character set of the system, choose FLD_STRING.

Interface Libraries

Interface libraries are covered in the following topics:

- Overview of Interface Libraries
- Distributing Zen Applications

Overview of Interface Libraries

The appropriate way to access the MicroKernel Engine from your Windows application is to link to a library that references the Btrieve Glue DLL when you compile. The Glue DLL is responsible for "glueing" your application to the Interface DLL. Like traditional glue (adhesive material), the Glue DLL is a thin layer between your application and the Interface DLL. The Glue DLL is responsible for successfully performing the following actions:

1. Loads the Interface DLL.
2. Binds to (that is, imports symbols from) the Interface DLL.

If the Glue DLL encounters a failure condition at any step, it issues an appropriate status code that your application can use to alert the user of the failure.

The following table shows the link libraries that your application can link with and the DLLs to load.

Operating System and Compiler[1]	Glue DLL	Link Library
Windows 32-bit (Microsoft Visual C++, Watcom, Embarcadero)	W3BTRV7.DLL	W3BTRV7.LIB
Windows 64-bit	W64BTRV.DLL	W64BTRV.LIB

[1]Compiler-specific libraries are in different subdirectories. To link Win32 applications, use the \Win32 directory if you use the Microsoft compiler; use the \Win32x directory if you use the Embarcadero or Watcom compiler.To link applications for UWP platforms such as Nano Server or Windows IoT Core, use the \winuwp directory.

Linux

Linux has no glue components. The application directly links against the shared library that implements the interface. The MicroKernel Engine link library is libpsqlmif for both Linux 32-bit and Linux 64-bit applications.

Distributing Zen Applications

If you plan to develop an application using a Zen database engine, you need to be aware of the following requirements for distributing your applications:

- Distribution Rules for Zen
- Installing Zen with your Application

Distribution Rules for Zen

After you have developed an application with Zen, you must be aware of the licensing agreement you have with Actian Corporation to distribute your product. If you have any questions regarding your distribution rights, please contact your sales representative.

Installing Zen with your Application

See *Installation Toolkit Handbook* for information on customizing the Zen installation.

Working with Records

The following topics cover working with records:

Sequence of Operations

Some Btrieve operations can be issued at any time, such as Create (14), Reset (28), and Version (26). However, most Btrieve operations require that you open a file using Open (0). Then, you must establish a position, or *currency*, in the file before you can operate on any records.

You can establish logical currency based on a key or physical currency by using Step operations.

- Use one of the following operations to establish physical currency:
  - Step First (33)
  - Step Last (34)
- Use one of the following operations to establish logical currency:
  - Get By Percentage (44)
  - Get Direct/Record (23)
  - Get Equal (5)
  - Get First (12)
  - Get Greater Than (8)
  - Get Greater Than or Equal (9)
  - Get Last (13)
  - Get Less Than (10)
  - Get Less Than or Equal (11)

After you establish currency, you can issue appropriate file I/O operations, such as Insert (2), Update (3), and Delete (4).

Note: Always use Btrieve operations to perform I/O on Btrieve files; never perform standard I/O on a Btrieve file.

Based on your currency, you can move through the file as follows:

- If you have established position based on physical currency, use the following operations: Step Next (24), Step Next Extended (38), Step Previous (35), or Step Previous Extended (39). The Step operations are useful for traversing a file quickly if your application does not need to retrieve the records in a specific order. The Extended operations are useful for working on a number of records at one time.
- If you have established position based on logical currency, use the following operations: Get Next (6), Get Next Extended (36), Get Previous (7), or Get Previous Extended (37). The Get

operations are useful for traversing a file in a specific order. The Extended operations are useful for working on a number of records at one time.

You cannot establish physical currency with one operation and then follow with an operation that requires logical currency. For example, you cannot issue a Step First operation and then a Get Next operation.

In data-only files, the MicroKernel does not maintain or create any index pages. You can access the records using only the Step operations and Get Direct/Record (23), all of which use the physical location to find records.

In key-only files, the MicroKernel does not maintain or create any data pages. You can access records using only the Get operations, which use logical currency to find records.

When you have finished working with a file, use Close (1) to close it. When your application is ready to terminate, issue a Stop (25).

Note: Failure to perform a Stop operation prevents the MicroKernel from returning its resources to the operating system. This failure eventually results in unpredictable system behavior, including the possibility of crashing the computer on which the application is running.

Accessing Records

Btrieve provides both physical and logical access to your data. With physical access, Btrieve retrieves records based on the physical record address within the file. With logical access, Btrieve retrieves records based on a key value contained in the record. In addition, Btrieve also allows you to access "chunks" of data within a record.

Accessing Records by Physical Location

Record accessing by physical location is faster for the following reasons:

- The MicroKernel does not have to use index pages.
- The next or previous physical record is usually already in the MicroKernel memory cache because the page on which it resides is probably in cache.

Physical Currency

Physical currency is the effect on positioning when accessing records by physical location. When you insert a record, the MicroKernel writes that record into the first free space available in the file, regardless of any key values contained in the record. This location is referred to as the physical location, or address, of the record. The record remains in this location until you delete it from the file. The Btrieve Step operations use the physical location to access records.

The record accessed last is the *current* physical record. The *next* physical record is the record with the immediately higher address relative to the current physical record. The *previous* physical record is the record with the immediately lower address. There is no physical record previous to the first physical record; likewise, there is no physical record next to the last physical record.

Together, the current, next, and previous physical locations form the physical currency within a file.

Step Operations

Your application can use the Step operations to access records based on their physical location within a file. For example, the Step First operation (33) retrieves the record that is stored in the first, or lowest, physical location in the file.

Note: You cannot perform Step operations on key-only files.

The Step Next (24) operation retrieves the record stored in the next higher physical location. Step Previous (35) retrieves the record stored in the next lower physical location in the file. Step Last (34) retrieves the record that is stored in the last, or highest, physical location in the file.

Step Next Extended (38) and Step Previous Extended (39) retrieve one or more records from the physical location following or preceding the current record.

Note: Each of the Step operations reestablishes the physical currency but destroys the logical currency, even if one existed before.

Accessing Records by Key Value

Accessing records by key value allows you to retrieve records based on their values for a specified key.

Logical Currency

Logical currency is the effect on positioning when accessing records by key value. When you insert a record into a file, the MicroKernel updates each B-tree index for which the appropriate key in the record has a nonnull value. Each key of a file determines a logical ordering of the records. The ordering is determined by the key's defined sort order or ACS.

The record accessed last is the *current* logical record. (This record is not necessarily the last record retrieved. The last record could have been retrieved by Get Direct/Chunk (23), which does not change the logical currency.) The *next* logical record is the record that is immediately next in the defined logical sequence. The *previous* logical record is the record that is immediately previous in the defined logical sequence. There is no logical record previous to the first logical record; likewise, there is no logical record next to the last logical record.

Together, the current, next, and previous logical records form the logical currency within a file.

The current logical record is also the current physical record, except when you perform an operation that uses the no-currency-change (NCC) option or when you operate on a record with a null key value. For example, you can perform an NCC Insert (2) operation and have the same logical position in the file as you had prior to the insert. The physical currency is updated.

NCC operations are useful when you must preserve your logical currency in order to perform another operation. For example, you may want to insert or update a record and then use a Get Next (6) operation based on your original logical currency.

NCC Insert Operation

```
status = BTRV( B_GET_FIRST, posBlock, dataBuf, &dataLen, keyBuf, keyNum); /* gets first record in key
```

```
path */

for (i = 0; i < numRecords; i++)
{ status = BTRV( B_INSERT, posBlock, dataBuf, &dataLen, keyBuf, -1); /* -1 for key num indicates no
currency change */
} /* inserts several records */

status = BTRV( B_GET_NEXT, posBlock, dataBuf, &dataLen, keyBuf, keyNum); /* gets next record after
first record in key path */
```

Note: When you use an NCC operation, the MicroKernel returns no information in the Key Buffer parameter. If you want to maintain logical currency, you must not change the value in the Key Buffer following an NCC operation, or your next Get operation can have unexpected results.

Get Operations

Your application can use the Get operations to retrieve records based on their values for a specified key. The appropriate Get operation can retrieve a specific record from a file or retrieve records in a certain order.

For example, the Get First (12) operation retrieves the first record by the key specified in the Key Number parameter. Likewise, Get Last (13) retrieves the last record according to the logical order based on the specified key. Some Get operations, such as Get Equal (5) or Get Less Than (10), return a record based on a key value your application specifies in the Key Buffer parameter.

Get operations establish logical currency. Your application can change from one key to another by performing the following procedure

1. Retrieve a record by issuing one of the Get operations.
2. Issue Get Position (22) to retrieve the 4-byte physical address of the record.
3. Issue Get Direct/Record (23) and pass to the MicroKernel the 4-byte physical address and the Key Number to change.

In addition to establishing logical currency, all Get operations except Get Position (22) establish the physical currency. As a result, you can continue with Step Next (24) or Step Previous (35). However, using the Step operations destroys the logical currency.

To reestablish logical currency after using a Step operation, perform the following procedure

1. Immediately after using a Step operation, issue Get Position (22) to retrieve the 4-byte physical address of the retrieved record.
2. Issue Get Direct/Record (23), passing to the MicroKernel the 4-byte position and the Key Number on which to establish logical currency.

Reading Variable-Length Records

Reading a variable-length record is the same as reading a fixed-length record in that you use the Data Buffer Length parameter to tell the MicroKernel how much room you have for the record to be returned. Set this parameter to the size of your entire Data Buffer, which can accommodate the maximum amount of data.

Note: Do not set the Data Buffer Length to a value larger than the number of bytes allocated to your Data Buffer; doing so could lead to a memory overwrite when running your application.

After a successful read operation, the Data Buffer Length parameter is changed to reflect the size of the returned record, which is the size of the fixed-length portion plus the amount of actual data in the variable portion (not the maximum size of a record). Your application should use this value to determine how much data is in the Data Buffer.

For example, suppose you have the following records in a data file:

Key 0: Owner 30-byte ZSTRING	Key 1: Account 8-byte INTEGER	Balance (Not a Key) 8 bytes	Comments (Not a Key) 1000 bytes
John Q. Smith	263512477	1024.38	Comments
Matthew Wilson	815728990	644.29	Comments
Eleanor Public	234817031	3259.78	Comments

Following are examples of Get Equal operations.

Note: While developing and debugging your application, it helps to display the Data Buffer Length just before and after the read operation to verify that it is set correctly at each point.

Get Equal Operation in C

```
/* get the record with key 1= 263512477 using B_GET_EQUAL */
memset(&dataBuf, 0, sizeof(dataBuf));
dataBufLen = sizeof(dataBuf); /* this should be 1047 */
account = 263512477;
*(BTI_LONG BTI_FAR *)&keyBuf[0] = account;
status = BTRV( B_GET_EQUAL, posBlock, &dataBuf, &dataBufLen, keyBuf, 1);
/* the dataBufLen should now be 56 */
```

Get Equal Operation in Visual BASIC

```
dataBufLen= length(dataBuf) ' this should be 1047
account% = 263512477
status = BTRV(B_GETEQUAL, PosBlock$, dataBuf, dataBufLen, account%, 1)
' the dataBufLen should now be 56
```

If the returned record is longer than the value specified by the Data Buffer Length, the MicroKernel returns as much data as it can (according to the size the Data Buffer Length was set to) and status code 22.

Accessing Records by Chunks

The Btrieve Data Buffer Length parameter, because it is a 16-bit unsigned integer, limits the record length to 65535. Chunk operations expand the record length well beyond this limit by allowing you to read or write portions of a record. A chunk is defined as an offset and length; the offset can be as large as 64 GB, but the length is limited to 65535 bytes. The limits that your operating system and the Data Buffer Length parameter impose also apply to the chunk operations; however, because the chunk operations can access any portion of a record, the limits have no effect on record length, only on the maximum size of a chunk accessible in a single operation.

For example, using the chunk operations, an application can read a 150,000 byte record by making three chunk retrieval calls. Each chunk in this example is 50,000 bytes long. The first chunk starts at offset zero, the next at offset 50,000, and the final at offset 100,000.

A chunk's offset and length do not have to correspond to any of the internal structures of a record that are known to the MicroKernel, such as key segments, the fixed-length portion of a record, or a variable tail. Also, a chunk does not have to correspond to any portion of the record that your application defines (for example, a field), although you may find it useful to update such defined portions as chunks.

Note: Chunks are defined only for the duration of the operation that defines them.

In some cases, the use of chunk operations in client/server environments allows the client Requester to use a smaller Data Buffer Length setting, which lowers the Requester memory requirement. For example, if your application used whole record operations and accessed records up to 50 KB long, your Requester would have to set its Data Buffer Length to at least 50 KB, thereby using 50 KB of RAM. But if your application used chunk operations and limited the size of each chunk to 10 KB, for example, then the Requester could set its Data Buffer Length to 10 KB, thereby saving 40 KB of RAM.

Intrarecord Currency

Intrarecord currency is relevant in chunk operations, because it tracks an offset within the current record. The current position is the offset that is one byte beyond the last byte of the chunk that was read or written. This is true even if in your last operation you attempted to read an entire record

and the MicroKernel could return only part of that record, which can happen when the Data Buffer Length is inadequate.

The exception is for an Update Chunk (53) operation that uses the Truncate subfunction. In this case, the MicroKernel defines the current position in the truncated record as the offset that is one byte beyond the end of the record itself.

By tracking intrarecord currency, the MicroKernel can do the following:

- Provide next-in-record subfunction biases for the Chunk operations.

  You specify an original offset, length and number of chunks, and the MicroKernel calculates the subsequent offsets.

- Improve performance in accessing chunks.

  Intrarecord currency can speed up any chunk operation, as long as it operates on the same record that was last accessed by the Position Block, and as long as the next chunk offset is greater than the current position in the record. That is, you do not have to access the next immediate byte in the record to benefit from intrarecord currency.

Note: The MicroKernel maintains intrarecord currency only for the current record. When you change physical or logical currency, the MicroKernel resets the intrarecord currency, as well.

Chunk Operations

You access chunks using the Get Direct/Chunk (23) operation and the Update Chunk (53) operation. To use these operations, you must define a chunk descriptor structure that defines the offset and length of the chunk. For the Get Direct/Chunk operation, the chunk descriptor structure must also specify an address to which the MicroKernel returns the chunk.

Before you can use Get Direct/Chunk (23), you must retrieve the physical address of the current record by issuing Get Position (22). You can use a single Chunk operation to retrieve or update multiple chunks in a record by using a next-in-record subfunction bias.

Inserting and Updating Records

Most of the time, inserting and updating records is a simple process: You use Insert (2) or Update (3) and pass in the record using the Data Buffer. This topic discusses some special cases involving inserts and updates.

Ensuring Reliability in Mission-Critical Inserts and Updates

While the MicroKernel is an extremely reliable data management engine, it cannot prevent system failures. System failures are more common in client-server applications, because network failures can occur. You can ensure reliability by taking advantage of these MicroKernel features:

- Transaction Durability. Transaction durability ensures that before the application receives a successful status code from an End Transaction operation, the changes are already committed to disk. Transactions are normally used to group multiple change operations that need to succeed or fail as a group. However, transaction durability can be useful even for single operations because the application has control over when the change is committed to disk.

  Consider "wrapping" individual mission-critical insert and update operations inside Begin Transaction and End Transaction operations and using the MicroKernel Transaction Durability configuration option. For more information about the Begin Transaction and End Transaction operations, see *Btrieve API Guide*. For more information about transaction durability, see Transaction Durability.

  Note: When you open a file in Accelerated mode, the MicroKernel does not perform transaction logging on the file. That is, operations performed on a file opened in Accelerated mode are not transaction durable.

- System Transaction Frequency. An alternative to using Transaction Durability is to use the MicroKernel settings Operation Bundle Limit and Initiation Time Limit to control the frequency of system transactions. For each open file, the MicroKernel bundles a set of operations into a single system transaction. If a system failure occurs, all changes made before the current system transaction completes are lost; however, the file is restored to a consistent state, enabling the operations to be attempted again after resolving the cause of the system failure.

  If you set both the Operation Bundle Limit and Initiation Time Limit to 1, the MicroKernel commits each operation as a separate system transaction. Doing so decreases performance, so this method is useful only for applications that can accept a performance decrease. One way to determine this is to measure the CPU utilization while your application runs. Applications that utilize 50 to 100 percent of the CPU are not good candidates for this approach.

Inserting Nonduplicatable Keys

If you are inserting a record with a key value that may already exist and for which you do not allow duplicates, you can proceed in one of two ways:

- Perform Insert (2). If the MicroKernel returns status code 5, then the key value does exist, and you cannot perform the insert.
- Perform a Get Equal operation with a Get Key bias (55). If the MicroKernel returns status code 4, then the key value does not already exist, and you can perform the insert.

If your Insert operation stands alone and does not depend on logical currency in the file, executing a Get Equal prior to each Insert is an additional overhead. However, for a group of inserts, the Get Equal operation facilitates any subsequent Insert operations by fetching into memory the index pages that point to the key location.

Inserting and Updating Variable-Length Records

When designing a variable-length data file, you must decide the maximum size of the variable-length portion of the record your application will support. You should set up a record structure that accommodates the fixed-length portion plus the maximum size of the variable portion. Use this structure as the Data Buffer when reading, inserting, and updating your variable-length records.

When inserting or updating a variable-length record, you use the Data Buffer Length parameter to tell the MicroKernel how much data to write. Set this parameter to the size of the fixed-length portion plus the amount of real data in the variable portion. Do *not* set the Data Buffer Length to the fixed length plus the maximum size your application allows for the variable field; if you do, the MicroKernel will always write the maximum size.

For example, suppose you want to insert the following record.

Key 0: Owner 30-byte ZSTRING	Key 1: Account 8-byte INTEGER	Balance (Not a Key) 8 bytes	Comments (Not a Key) 1000 bytes
John Q. Smith	263512477	1024.38	Comments

Following are examples of Insert operations. Note that the Data Buffer Length is computed as the fixed-length portion plus the amount of data in the Comments field (8 bytes), not the maximum size of the Comments field (1000 bytes).

Insert Operation

```
#define MAX_COMMENT 1000 /* Largest variable comment size */
typedef struct
{ char owner[30];
  int  number;
  int  balance;
} FixedData;
typedef struct
{ FixedData fix;
  char variable[MAX_COMMENT];
} DataBuffer;

  DataBuffer account;
  BTI_ULONG  dataBufLen;
  BTI_SINT   status;

  strcpy(account.fix.owner, "John Q. Smith");
  account.fix.number = 263512477;
  account.fix.balance = 102438;
  strcpy (account.variable, "Comments");
  dataBufLen = sizeof(FixedData) + strlen(account.variable) +1;
    /* the +1 accommodates the null character after the data */
  status = BTRV(B_INSERT, PosBlock, &account, &dataBufLen, keyBuffer, 0);
```

Reading and Updating Fixed-length Portions

It is possible to read only the fixed-length portion of a record by setting the data buffer size to that fixed length. The MicroKernel returns only the fixed-length portion and status code 22. However, if you then use Update (3) and pass in only the fixed-length portion, the variable-length portion is lost. Instead, use Update Chunk (53), which updates any part of a record, based on a byte offset and length. Set the byte offset to 0 and the length to the length of the fixed-length portion. This operation updates the fixed-length portion and retains the variable-length portion.

Updating Nonmodifiable Keys

If you attempt to update a key value that is defined as not modifiable, the MicroKernel returns status code 10. If you want to update the key value anyway, you must first Delete (4) the record and then Insert (2) it.

No-Currency-Change (NCC) Operations

You can perform a variation on the standard insert or update, called a no-currency-change (NCC) operation, by passing in a –1 (0xFF) for the Key Number parameter. NCC operations are useful when an application must save its original logical position in a file in order to perform another operation, such Get Next (6).

To achieve the same effect without an NCC insert operation, you would have to execute these steps:

1. Get Position (22) – Obtain the 4-byte physical address for the logical current record. You would save this value for use in Step 3.
2. Insert (2) – Insert the new record. This operation establishes new logical and physical currencies.
3. Get Direct/Record (23) – Reestablish logical and physical currencies as they were in Step 1.

The NCC Insert operation has the same effect as a standard Insert in terms of logical currency, but can have a different effect in terms of physical currency. For example, executing a Get Next (6) after either procedure produces the same result, but executing a Step Next (24) might return different records.

To maintain original positioning without an NCC update operation, you would have to execute these steps:

1. Get Next (6) – Establish the next logical record.
2. Get Position (22) – Obtain the 4-byte physical address for the next logical record. You would save this value for use in Step 8.
3. Get Previous (7) – Reestablish the current logical record.
4. Get Previous (7) – Establish the previous logical record.
5. Get Position (22) – Obtain the 4-byte physical address for the previous logical record. You would save this value for use in Step 8
6. Get Next (6) – Reestablish the current logical record.
7. Update (3) – Update the affected record. If this standard update operation changes the specified key's value, it also establishes new logical currency.
8. Get Direct/Record (23) – Establish the currencies to the record that preceded or followed the record updated in Step 7 If your application is to continue searching forward, you would pass to the Get Direct/Record operation the address saved in Step 2. If the application is to continue searching backward, you would pass the address saved in Step 5.

Multirecord Operations

Zen provides a high performance mechanism for filtering and returning multiple records or portions of multiple records. The mechanism is called "extended operations" and is supported by four specific operation codes:

- Get Next Extended (36)
- Get Prev Extended (37)
- Step Next Extended (38)
- Step Prev Extended (39)

For detailed information on how to code these operations, see *Btrieve API Guide*. This section explains how to optimize your use of these operations for best performance.

Terminology

The following words can have other meanings in other contexts. For the purposes of this section, definitions are provided below.

Descriptor
Also called the Extended Expression. The whole contents of the data buffer which describes how the Btrieve extended operation should be accomplished.

Filter
A portion of the Extended Expression which describes a selection criteria to apply to the records being selected.

Condition
A portion of a filter that uses a single logical operator.

Connector
A Boolean operator that connects a condition to what follows it. Either AND, OR, or NONE

Extractor
A portion of the Extended Expression which defines what data to return.

Key
A whole index definition which may have multiple segments. Get operations require the MicroKernel to move through the data file along a single key path.

Key Segment
Compound indices, also called multisegmented keys, can have multiple segment definitions. Each segment defines an offset, length, data type, and so on.

Background

The filter evaluation mechanism for extended operations is designed to be very fast. It evaluates expressions in a straightforward manner without doing any extra processing. Because of this approach, you should be aware of some of the idiosyncracies of the interface.

- Extended operations do not establish initial positioning. They only move forward or reverse from the current position. So in order to find all records where (lastname = 'Jones' AND firstname = 'Tom' AND city = 'LasVegas') your application must perform a Get Equal operation before performing a Get Next Extended operation.
- Filter evaluation is strictly left to right. For example, your application cannot perform a single extended operations that will get all records where (Age = 32 AND Gender = "M") OR (Age = 30 AND Gender = "F").

  This kind of search must jump around the file somewhat in order to be most efficient, but extended operations do not jump around. They move along a logical key or record path one record at a time. A compound logical expression such as the one above would require an optimizer like the one in the Relational Engine that is part of Zen. Instead, the MicroKernel evaluates expressions left to right to make the filter evaluation process as fast as possible.

  To do the search above, the caller must make two Get Extended calls, each after first performing a GetEqual operation to position the cursor to the first record. If you use the four conditions from the example above in an extended operation, it would be evaluated like (Age = 32 AND (Sex = "M" OR (Age = 30 AND Sex = "F"))). In other words, for each record, the MicroKernel evaluates the first condition, then looks at the Boolean operator to determine whether it must evaluate the next condition. If the first condition is false, the AND operator means the whole expression is false.

Validation

While an extended operation can return a status 62 (Invalid Descriptor) in many ways, the following list includes some of the most common:

- Descriptor length is not long enough. This depends on the number of filter conditions, the length of each condition value and whether a collating sequence is used, and the number of fields to extract.
- Data buffer: Length must be at least long enough to contain the full descriptor.
- Each condition must have a valid data type. It must be one of the valid Btrieve key types.
- Flags used on the comparison code must be valid (FILTER_NON_CASE_SENSITIVE, FILTER_BY_ACS, FILTER_BY_NAMED_ACS), and can be used only with string type

fields (STRING_TYPE, LSTRING_TYPE, ZSTRING_TYPE, WSTRING_TYPE, WZSTRING_TYPE).

- Must be a valid comparison code, (1-6).
- Must be a valid connector, (0-2).
- Any referenced ACS or ISR must be predefined.
- The last filter condition needs a terminator (connector must be 0).
- All other filter condition can not have a terminator (connector must be 1 or 2).
- The extractor record count may not be zero.

Optimization

Optimizing an extended operation means that the MicroKernel can stop looking at records along the current key path because there is no possibility that records remaining on the key path can satisfy the filter. Starting with PSQL 2000i SP3, the MicroKernel can optimize on multiple conditions as long as they occur sequentially and match the segments of the current key.

As it evaluates each condition, the MicroKernel determines if the given segment can be optimized. In order to do that, all the following must be true:

- Must be a Get Extended operation (36 and 37), not a Step Extended operation (38 and 39).
- Optimization must not have been turned off for remaining segments because of a matching record found when evaluating a previous segment.
- An OR connector makes the current condition and any following condition not optimizable, since the expression is evaluated left to right.
- The condition must refer to the same offset in the record as the current key segment.
- The condition must do either one of the following:
  - Refer to the same field length in the record as the current key segment,
  - Be a substring of the key if the data type is one of the string types.
- The condition must refer to the same field type as the key.
- The condition must not be comparing a field with another field in the record (FILTER_BY_FIELD).
- The condition must have the same case sensitivity as the key.
- The condition must have the same ACS or ISR specification as the key if the data type is one of the string types.

- The logical operator must be EQ (=) unless the condition is being optimized to the first key segment.
- For the first key segment, the logical operator can also be LT or LE (< or <=) if the direction is forward, and GT or GE (> or >=) if the direction is reverse. For these logical operators, only one filter condition will be optimized.

The actual direction is dependent on both the direction indicated by the operation, but also on whether the current key is descending or ascending, as shown in the following table.

	Ascending Key Segment	Descending Key Segment
Get/Step Next	Ascending/Forward	Descending/Reverse
Get/Step Prev	Descending/Reverse	Ascending/Forward

Any one of the following can cause optimization to be turned off for all future segments.

- If the key has no more segments to optimize to.
- If the current condition has an OR connector.
- If the current condition IS optimized, but is a sub-string of the associated key segment.
- If the current condition IS optimized, but the logical operator is not EQ (=).
- If the current condition failed to optimize and the previous condition was optimized. This means that if a bunch of conditions are ANDed together, the first optimizing condition, the one that matched the first key segment, does NOT have to be the first condition in the filter. But once a condition is found that CAN be optimized to the first key segment, the other optimizing conditions must occur sequentially and immediately following that first optimizing condition.

Note: A defect in PSQL 2000i SP3 results in a requirement for the optimizable conditions to occur first in the filter. Thus, the ability to put not optimizable conditions before the optimizable conditions is available only after SP3. A limited ability for this was available before SP3, but only one condition could be optimized.

Examples

The following examples of multirecord operations are based on this table of sample data:

Record	Field 1	Field 2	Field 3	Field 4
1	AAA	AAA	AAA	XXX

Record	Field 1	Field 2	Field 3	Field 4
2	AAA	BBB	BBB	OOO
3	AAA	CCC	CCC	XXX
4	BBB	AAA	AAA	OOO
5	BBB	AAA	BBB	XXX
6	BBB	AAA	CCC	OOO
7	BBB	BBB	AAA	XXX
8	BBB	BBB	BBB	OOO
9	BBB	BBB	CCC	XXX
10	BBB	CCC	AAA	OOO
11	BBB	CCC	BBB	XXX
12	BBB	CCC	CCC	OOO
13	CCC	AAA	CCC	XXX
14	CCC	CCC	AAA	OOO

Consider the table above, which has a compound key on fields 1, 2 and 3 in that order. Assume that the application performs a Get First (12) operation on this file using this three-segment key, followed up with Get Next Extended (36). Notice that these examples contain parentheses where they are assumed to be. This is the only way parentheses can occur when the filter is evaluated from left to right.

Remember that in order to optimize against a key segment, the offset, length, type, case, and ACS identified in the filter condition all must be the same as the key definition.

(Field1 = AAA AND (Field2 = AAA AND (Field3 = AAA)))
The MicroKernel Engine retrieves record 1 and stops searching with status 64 (filter limit reached). The last record examined that satisfies the optimization criteria is record 1, which becomes the current record. Engines before Pervasive.SQL 2000 SP3 can optimize only one condition and thus leave the current record at record 3.

(Field1 = AAA OR (Field2 = AAA OR (Field3 = AAA)))
The MicroKernel Engine retrieves records 1, 2, 3, 4, 5, 6, 7, 10, 13, and 14 and returns with status 9 (end of file reached). No condition can be optimized since the first condition contains an OR connector. The current record becomes record 14.

(Field1 = BBB AND (Field2 = BBB OR (Field3 = BBB)))
The MicroKernel Engine would retrieve records 5, 7, 8, 9, and 11 and return with status 64. The first condition was optimized, but the second condition was not since it contained an OR connector. The last record examined that satisfies the optimization criteria is record 12 which becomes the current record.

(Field4 = OOO AND (Field2 = BBB AND (Field3 = BBB)))
The MicroKernel Engine retrieves records 2 and 8 and return with status 9. No condition can be optimized to the first key segment, so the following segments cannot be optimized. The current record becomes record 14.

(Field1 = BBB AND (first byte of Field2 = B AND (Field3 = BBB)))
The MicroKernel Engine record 8 and returns with status 64. The first two conditions can be optimized, but since the second condition is a substring, the third condition cannot be optimized. The last record examined that satisfies the optimization criteria is record 9. Engines before Pervasive.SQL 2000 SP3 can optimize only one condition and thus return with record 12 as the current record.

(Field1 = BBB AND (Field2 = Field3))
This is done by using the +64 bias on the comparison code, which indicates that the second operand is another field of the record, rather than a constant. The MicroKernel Engine retrieves records 4, 8, and 12 and returns with status 64. The first condition can be optimized, but since the second condition does not compare to a constant, it cannot be optimized. The last record examined that satisfies the optimization criteria is record 12.

(Field1<= BBB AND (Field2 <= BBB AND (Field3 <= BBB)))
The MicroKernel Engine retrieves records 1, 2, 4, 5, 7, and 8 and returns with status 64. The first condition can be optimized, but since it does not have a logical operator of EQ, the following conditions cannot be optimized. The last record examined that satisfies the optimization criteria is record 12.

(Field1= BBB AND (Field2 < BBB AND (Field3 < BBB)))
The MicroKernel Engine retrieves record 4 and returns with status 64. The first condition can be optimized, but since the second condition does not have a logical operator of EQ, it cannot be optimized. The last record examined that satisfies the optimization criteria would be record 12.

(Field1= BBB AND (Field2 = BBB AND (Field3 < BBB)))
The MicroKernel Engine retrieves record 7 and returns with status 64. The first two conditions can be optimized because they use EQ, but the third condition cannot. The last record examined that satisfies the optimization criteria is record 9. Engines prior to Pervasive.SQL 2000 SP3 can optimize on only one condition and so the current record would be 12.

(Field2>= AAA AND (Field2 <= BBB AND (Field1 >= AAA) AND (Field1 <= BBB))))
The MicroKernel Engine retrieves records 1, 2, 4, 5, 6, 7, 8, and 9 and returns with status 64. The first three conditions cannot be optimized to the first key segment, but since they are all ANDed together, the fourth condition can be used to optimize the search. The second condition would be optimizable if it occurred immediately after the fourth condition. But since it is out of position relative to the key segments, it cannot be optimized. Since only one key segment is optimized, the last record examined that satisfies the optimization criteria would be record 12. Note that there is a defect in Pervasive.SQL 2000 SP3 that prevents optimization unless the optimizable condition occurs first. So the SP3 engine would retrieve the same records, but would return status 9.

Performance Tips

This section provides some information on how to speed up your operations.

Connectors

Since extended operations evaluate logical expressions left to right, this feature is by no means a complete expression evaluator that can be used to extract whatever data you want in the most efficient manner. Extended operations are designed to be used in conjunction with a Get or Step operation to initially set the cursor in the file to the correct position. So the first suggestion is:

- Do not try to mix AND and OR connectors in the same filter. If you do, put the AND conditions up front to match the segments of the key, so that at least the engine can optimize the search to a shorter portion of the key path.

In other words, it may be appropriate to add some OR'ed conditions for fields not in the index after an optimizable condition for an indexed field. For example, let say you are searching a nationwide phone book for everyone in Texas with first name "William", "Bill", "Billy" or "Billybob". Using a key on the State field, you would use GetEqual to set the current record on the first person in Texas. Then call GetNextExtended with a filter like (State = "Texas" AND (FirstName = "William" OR (FirstName = "Bill" OR (FirstName = "Billy" OR (FirstName = "Billybob"))))). If your Extractor indicated a reject count of 10,000 and a return count of 100 records, the GetNextExtended would return after looking at about 10,000 records. But with 14 million people in Texas, you will need to keep issuing the same GetNextExtended operation over and over until you finally get to Utah and a status 64 (filter limit reached). This process would be much faster than having each record transferred to your application one at a time.

But what if a compound index existed on State and FirstName? The Get Next Extended above would still work, but it would be much faster to do a Get Equal and Get Next Extended on each of the four State and FirstName combinations, optimizing on both fields.

So you can see that having filters with OR connectors is only useful when no index is available. Priority should be placed on AND connectors for fields that match the key.

Reject Count

Another issue that you should understand is how to set the reject count. If your application is the only one being serviced by a MicroKernel engine, then using the maximum reject count is most efficient since it keeps that network traffic or interprocess communication to a minimum. However, if there are a lot of applications running in a highly concurrent environment, there can be serious consequences if you have a reject count that is too large.

The MicroKernel can handle many Btrieve requests concurrently even while doing Btrieve operations atomically on a file. So it allows any number of reader threads, but only one write thread, to access the same file. Most Btrieve operations are read operations and they do not take much time to accomplish. So if a write operation comes in, it waits until all readers are finished before it locks the file for the instant it takes to insert, update, or delete a record. This coordination works great until a read operation comes in that takes a long time to finish. That is what an extended operation with a high reject count will do if it does not find any records. It keeps reading and reading and reading. The other read operation can get done without a problem, but the write operations start to back up. After a write operation has tried to get write access to the file 100 times, it reaches what is called a frustration count. At this time, it puts a block on all new reader threads. So now all Btrieve operations on this file are hung until the extended operation is done.

- For this reason, if your application is used in a highly concurrent environment, use a reject count that is somewhere between 100 and 1000. Also try to make your extended operations optimizable so that the MicroKernel does not have to read and reject records very often.

Even with a reject count of 100 to 1000, it is better to have the MicroKernel read and reject them than it is to have the records returned to your application to reject them.

Adding and Dropping Keys

Btrieve provides two operations for adding and dropping keys in your files: Create Index (31) and Drop Index (32). The Create Index operation is useful for adding keys to a file after the file has been created. The Drop Index operation is useful for removing keys whose index pages have become damaged. After dropping a key, you can re-add it, which causes the MicroKernel to rebuild the index.

When you drop a key, the MicroKernel renumbers all higher-numbered keys, unless you specify otherwise. The MicroKernel renumbers the keys by decrementing all higher-numbered keys by 1. For example, suppose you have a file with key numbers 1, 4, and 7. If you drop key 4, the MicroKernel renumbers the keys as 1 and 6.

If you do not want the MicroKernel to automatically renumber keys, add a bias of 0x80 to the value you supply for the Key Number parameter. This allows you to leave gaps in the key numbering; consequently, you can drop a damaged key and then rebuild it without affecting the numbering of other keys in the file. You rebuild the index using Create Index (31), which allows you to specify a key number.

Note: If you dropped a key without renumbering and a user then cloned the affected file without assigning specific key numbers, the cloned file would have different key numbers than the original.

Supporting Multiple Clients

Support for multiple clients is discussed in the following topics:

- Btrieve Clients
- Passive Concurrency
- Record Locking
- User Transactions
- Examples of Multiple Concurrency Control
- Concurrency Control for Multiple Position Blocks
- Multiple Position Blocks
- ClientID Parameter

Btrieve Clients

A Btrieve client is an application-defined entity that makes Btrieve calls. Each client can make Btrieve calls and has its own resources (such as files) that are registered with the MicroKernel. In addition, the MicroKernel maintains the status of transactions (both exclusive and concurrent) on a per-client basis.

When you need to support multiple clients concurrently, use the BTRVID or BTRCALLID function, which includes a Client ID parameter. The Client ID parameter is the address of a 16-byte structure that allows the MicroKernel to differentiate among the clients on a computer. Following are examples of situations in which using a Client ID might be useful:

- You write a multi-threaded application that conducts several transactions, all in progress at the same time. For each Begin Transaction operation, the application specifies a different Client ID. The MicroKernel maintains separate transaction states for each Client ID.
- You write an application that uses two Client IDs, and for each Client ID, opens several files. Your application can execute a Reset operation using BTRVID, BTRCALLID, or BTRCALLID32, causing the MicroKernel to close the files and free the resources for a single specified Client ID.
- You write an application that allows multiple instances of itself to run simultaneously. For the integrity of your application data, all instances must appear to the MicroKernel as a single client. In this situation, your application provides the *same* Client ID parameter on each Btrieve call, regardless of which instance of the application is making that call.
- You write an application that acts as a Dynamic Data Exchange (DDE) server. Your server application, which makes Btrieve calls, must divide the returned information between the applications originating the requests to your server application. In this situation, your application can assign a *different* Client ID to each requesting application, providing a way to track information to be distributed among several clients.

The MicroKernel provides several concurrency control methods and uses several implementation tools to resolve conflicts that can occur when multiple clients attempt to access or modify records in the same file concurrently.

The concurrency control methods are as follows:

- Passive Concurrency
- Record Locking
- User Transactions

The implementation tools are as follows:

- Explicit Record Locks
- Implicit Record Locks
- Implicit Page Locks
- File Locks

The following topics discuss MicroKernel concurrency control methods in detail. While reading each topic, see the table under Conflict Codes, which summarizes the types of conflicts that can occur when two clients attempt to access or modify the same file. The table describes the actions of local clients.

Note: If your application uses the BTRVID function to define and manage multiple clients within the same application, such clients are considered local clients.

In both tables, client 1 performs an action identified by an abbreviation in the far-left column of the table, and then client 2 *attempts* to perform one of the actions identified by an abbreviation in the top row of the table.

The actions represented by the abbreviations are described in Action Codes.

Assumptions

In the table under Conflict Codes, the following assumptions are made:

- For a specific cell of the table, client 2 attempts to perform an action after client 1 starts performing an action. The first action must finish before the second action can start.
- For any cell in which client 2 performs an update or delete operation, client 2 is assumed to have read the affected record before client 1 performs its action.
- Unless explicitly stated in the action description for client 2 (as in actions MDR and MTDR), clients 1 and 2 always perform their actions on the same record, when both operations are reads or when both operations are updates or deletes.
- Unless explicitly stated in the action description for client 2 (see action ITDP), when both client 1 and client 2 perform an insert, update, or delete operation, both clients change at least one of the pages they have in common.
- When client 2 performs a modification following an insert operation by client 1, the modified record is not the inserted record, although both records share one or more data, index, or variable pages in the file.

Action Codes

RNL	Read without lock request, either nontransactional or in a concurrent transaction.
RWL	Read with lock request, either nontransactional or in a concurrent transaction.
INT	Insert, nontransactional.
ICT	Insert in a concurrent transaction.
ITDP	Insert in a concurrent transaction, changing different pages than those modified by an insert, update, or delete by client 1, who is also in a concurrent transaction.
MNT	Modify (update or delete), nontransactional.
MDR	Modify (nontransactional) a different record than the one modified by client 1.
MCT	Modify in concurrent transaction.
MTDR	Modify (in concurrent transaction) a different record than the one modified by client 1.
EXT	Read, insert, or modify in exclusive transaction.

Conflict Codes

N/A	Not applicable.
NC	No conflict or blocking between the actions of client 1 and client 2.
RB	Record-level blocking. Client 2 is blocked because of a record lock held by client 1.
PB	Page-level blocking. Client 2 is blocked because of a page lock held by client 1.
FB	File-level blocking. Client 2 is blocked because of a file lock held by client 1.
RC	Record conflict. Client 2 cannot execute the operation because the record has been modified by client 1 in the time since client 2 originally read the record. The MicroKernel returns status code 80.

For conflict codes RB, PB, and FB, the MicroKernel retries the client 2 action unless client 2 has specified a no-wait type operation (for example, a read with a no-wait lock or an insert/modify in a concurrent transaction that was started with a 500 bias). For a no-wait operation, the MicroKernel returns an error status code.

The following table shows action code combinations involving local clients.

	Client 2 Action									
	RNL	**RWL**	**INT**	**ICT**	**ITDP**	**MNT**	**MDR**	**MCT**	**MTDR**	**EXT**
Client Action										
RNL	NC	NC	NC	NC	N/A	NC	N/A	NC	N/A	NC
RWL	NC	RB	NC	NC	N/A	RB	N/A	RB	N/A	RB
INT	NC	NC	NC	NC	N/A	NC	N/A	NC	N/A	NC
ICT	NC	NC	PB	PB	NC	PB	N/A	PB	N/A	PB
MNT	NC	NC	NC	NC	N/A	RC	NC	RC	NC	NC
MCT	NC	RB	PB	PB	NC	RB	PB	RB	PB	PB
EXT	NC	FB	FB	FB	N/A	FB	FB	FB	FB	FB

Here are some examples of interpreting action code combinations:

- Combination EXT-RWL: Client 1 reads a record from the file from within an exclusive transaction. Client 2 receives status code 85 (FB, file-level blocking) when it tries to read a record from the file with a no-wait lock bias in a nontransactional mode. If client 2 specifies a wait lock bias, the MicroKernel retries the operation.
- Combination ICT-ICT: Client 1 inserts a record from within a concurrent transaction. The MicroKernel retries the operation when client 2 attempts to insert a record into the same file, because one of the pages to be modified by the operation has already been changed by the insert operation performed by client 1. If client 2 starts the concurrent transaction with a 500 bias, the MicroKernel returns status code 84. (See the Assumptions concerning this table.)
- Combination ICT-ITDP: This combination resembles ICT-ICT, except that client 2 does not change any page that has already been modified by client 1. In this case, the operation attempted by client 2 is successful (NC, no blocking, no conflict).
- Combination MCT-MTDR: Although client 1 and client 2 modify different records, client 2 is blocked by a page lock. This blockage results because the records being modified share a data page, index page, or variable page in the file. (See the Assumptions concerning this table.)

Passive Concurrency

You may choose to rely on passive concurrency for resolving update conflicts if your application performs single-record read and update operations while not inside a transaction or from within a concurrent transaction. Passive concurrency is applied automatically by the MicroKernel; it requires no explicit instructions from the application or the user.

With passive concurrency, the MicroKernel allows a client to read a record without applying any lock bias for the operation; if a second client changes the record between the time the first client reads it and the time the first client attempts to update or delete it, the MicroKernel returns status code 80. In this situation, the modification that the first client initiates is based on an outdated image of the record. Therefore, the first client must read the record again before performing the update or delete operation.

Passive concurrency allows developers to move applications directly from a single-user to a multiuser environment with only minor modifications.

This table shows how two clients interact when using passive concurrency nontransactionally.

Client 1	Client 2
1. Open file.	
	2. Open file.
3. Read record A.	
	4. Read record A.
5. Update record A.	
	6. Update record A. The MicroKernel returns status code 80.
	7. Reread record A.
	8. Update record A.

This table shows how two clients interact when using passive concurrency from within a concurrent transaction.

Client 1	Client 2
1. Begin concurrent transaction.	
	2. Begin concurrent transaction.
3. Read record A.	

Client 1	Client 2
4. Update record A.	
	5. Read record A.
6. End transaction.	
	7. Update record A. The MicroKernel returns conflict status code.
	8. Reread record A.
	9. Update record A.
	10. End transaction.

Note: Even though client 2 reads record A *after* client 1 has already executed the update operation, the MicroKernel correctly detects a conflict error in Step 7. This conflict exists because client 1 does not commit the change it made to record A until ending its transaction in Step 6. By the time client 2 attempts its update in Step 7, the image it read of record A (in Step 5) is outdated.

Record Locking

In many situations, a client might want stronger concurrency control than passive concurrency provides. Thus, the MicroKernel allows a client to ensure that the client can update or delete certain records without getting a conflict error (status code 80, an indication that another client has modified the record after this application read it). To achieve this, the client must read the record with a lock request. If the MicroKernel grants the lock, no other client can lock, update, or delete the record until the client that holds the lock releases it.

Thus, the ability to update or delete the record is guaranteed, even if in some cases the client has to wait because the operation is temporarily blocked. For example, temporary blocking can occur when another record on the same data page as the client's record is modified by another application in a concurrent transaction still in progress.

There are various kinds of record locks a client can explicitly request. For more information, see Locks.

User Transactions

Transactions reduce the possibility of lost data. If you have a number of modifications to make to a file and you must be sure that either *all* or *none* of those modifications are made, include the operations for making those modifications in a transaction. By defining explicit transactions, you can force the MicroKernel to treat multiple Btrieve operations as an atomic unit. To include a group of operations within a transaction, you enclose those operations between a Begin Transaction (19) operation and an End Transaction (20) operation.

The MicroKernel provides two types of transactions: exclusive and concurrent. The type you use depends on how severely you want to restrict access by other clients to the file you are modifying. Note that the MicroKernel does not allow other applications or clients to see the changes involved in *any* transaction (exclusive or concurrent) until the transaction ends.

When a task operates on a file inside an exclusive transaction, the MicroKernel locks the entire file for the duration of the transaction. Once a file is locked in an exclusive transaction, other nontransactional clients can read the file, but they cannot make changes to it. Another client that is also in an exclusive transaction cannot perform any operations that require a position block on the file – even standard Get or Step operations – until the first client unlocks the file by finishing the transaction.

When an application operates on a file inside a concurrent transaction, the MicroKernel locks only the affected records and pages in the file, as follows:

- The MicroKernel locks one or more records in a Get or Step operation if that operation has been specifically called with a read lock bias or if that operation has inherited a read lock bias from the Begin Transaction operation. See Locks for information about locks and lock biases.
- The MicroKernel locks only the specific records modified by an insert, update, or delete operation. If a locked record is a variable-length record, then the MicroKernel also locks all the variable pages containing portions of the record. Finally, the MicroKernel locks entries on any index pages that will be modified as a result of the insert, update, or delete operation. A small percentage of index page changes may cause entries to move from one page to another. An example is when an index page is split or combined. These changes will result in a full page lock on the index page until the transaction is completed.

As with exclusive transactions, other tasks can always read data that is locked from within a concurrent transaction. In any data file, multiple tasks can have their own concurrent transactions operating, in which they are performing insert, update, or delete operations, or in which they are performing Get or Step operations that contain read lock biases. The only restriction on these activities is that no two tasks can lock the same record or page simultaneously from their respective concurrent transactions.

The following additional features apply to concurrent transactions:

- Locked pages remain locked for the duration of the transaction.
- If the transaction simply reads a record, the MicroKernel does *not* lock either the record or the corresponding page.
- Other clients cannot see the changes to a file in a concurrent transaction until the transaction ends.

Locks

Records, pages, or even an entire file can be locked. Once locked, a record, page, or file cannot be modified by any client other than the one responsible for the lock. Similarly, locks owned by one client can prevent record, page, or file locking by another client, as explained in the rest of this section.

The MicroKernel provides two kinds of locks: explicit and implicit. When a client specifically requests the lock by including the lock request with a Btrieve operation code, that lock is called an *explicit* lock. However, even when a client does not explicitly request a lock, the MicroKernel may lock an affected record or page as the result of an action that the client performs. In this situation, the lock that the MicroKernel makes is called an *implicit* lock (see Implicit Record Locks and Implicit Locks).

Note: Unless otherwise noted, the term *record lock* refers to an *explicit* record lock.

Records can be locked implicitly or explicitly. Pages can be locked only implicitly. Files can be locked only explicitly.

The rest of this section discusses the various locks as they apply in both nontransactional and transactional environments.

Explicit Record Locks in a Nontransactional Environment

This topic discusses explicit record locks in a nontransactional environment. For information about how transactions affect the use of record locks, see Record Locks in Concurrent Transactions.

A client might not want to rely on passive concurrency. However, that same client might need to ensure that any record it reads can later be updated or deleted without receiving status code 80 (which would require the client to reread the record). A client can comply with both these requirements by requesting an explicit record lock on the record. For your application to lock a record when reading it, you can add one of the following bias values to the appropriate Btrieve Get or Step operation code:

- 100 – Single wait record lock.
- 200 – Single no-wait record lock.
- 300 – Multiple wait record lock.
- 400 – Multiple no-wait record lock.

You can only apply these lock biases to Get and Step operations. You cannot specify a lock bias on any other operations in a nontransactional environment.

Note: Single-record locks and multiple-record locks are incompatible; therefore, a client cannot hold both types of locks simultaneously on the same position block (or cursor) in a file.

Single-Record Locks

A single-record lock allows a client to lock only one record at a time. When a client successfully locks a record with a single-record lock, that lock remains in effect until the client completes one of the following events:

- Updates or deletes the locked record.
- Locks another record in the file (using a single-record lock).
- Explicitly unlocks the record using the Unlock (27) operation.
- Closes the file.
- Closes all open files by issuing a Reset (28) operation.
- Obtains a file lock during an exclusive transaction.

When a client locks a record, no other client can perform any Update (3) or Delete (4) operations on that record. However, other clients can still read the record using a Get or Step operation, as long as the Get or Step operation adheres to the following conditions:

- Contains no explicit lock bias.
- Is not performed from within a transaction that would cause the record to be locked when it is read (as in an exclusive transaction, where the MicroKernel locks the entire file, or in a concurrent transaction that was begun with a lock bias). For more information, see Record Locks in Concurrent Transactions and File Locks.

Multiple-Record Locks

A multiple-record lock allows a client to lock several records concurrently in the same file. When a client successfully locks one or more records with multiple-record locks, those locks remain in effect until the client completes one or more of the following events:

- Deletes the locked records.
- Explicitly unlocks the record(s) using the Unlock (27) operation.
- Closes the file.
- Closes all open files by issuing a Reset (28) operation.
- Obtains a file lock during an exclusive transaction.

Note: An Update operation does not release a multiple-record lock.

As with a single-record lock, when a client locks one or more records using a multiple-record lock, no other client can perform any Update (3) or Delete (4) operations on those records. Other clients can still read any of the locked records using a Get or Step operation, as described in Locks.

When a Record Has Already Been Locked

When your client requests a no-wait lock on a record that is currently not available (either the record is locked by another client or the whole file is locked by an exclusive transaction), the MicroKernel returns either status code 84 (Record/Page Locked) or status code 85 (File Locked). When your client requests a wait lock and the record is currently not available, the MicroKernel retries the operation.

Record Locks in Concurrent Transactions

Because exclusive transactions (operation 19) lock the entire file, record locks in a transaction apply only to concurrent transactions (operation 1019). (For information about transaction types, see User Transactions).

The MicroKernel allows a client to lock either single or multiple records in a file from within a concurrent transaction. The client can lock records by either of the following methods:

- Explicitly specify a record lock bias value on a Get or Step operation using one of the same bias values listed previously. Record lock bias values for concurrent transactions are the same as those for nontransactional record locks.
- Specify a record lock bias value on a Begin Concurrent Transaction (1019) operation. Again, these bias values are the same as those for nontransactional record locks, listed previously.

When you specify a record lock bias value on a Begin Concurrent Transaction operation, each operation inside that transaction – if it has no bias value of its own – inherits its bias value from the Begin Concurrent Transaction operation. For example, a Get Next (6) operation inherits the

200 bias from the preceding biased Begin Concurrent Transaction (1219) operation, causing the Get Next to be performed as a no-wait read and lock (206) operation.

As implied in the preceding paragraph, a client can still add bias values to the individual Step or Get operations that occur within the concurrent transaction. Biases added in this manner take precedence over the inherited bias.

The events that cause the release of single- and multiple-record locks in concurrent transactions are similar to those for the nontransactional environment. For single, see Single-Record Locks. For multiple, see Multiple-Record Locks with the following exceptions:

- A Close operation does *not* release explicit record locks secured from within a concurrent transaction. With version 7.0 of the MicroKernel, you can close the file within a transaction even if a record is locked.
- An End or Abort Transaction operation releases all record locks obtained from within the transaction.

Finally, when a client in a concurrent transaction reads one or more records using an unbiased Get or Step operation, and no lock bias was specified on the Begin Concurrent Transaction operation, the MicroKernel performs no locking.

Implicit Record Locks

When a client attempts to update or delete a record, either *external* to any transaction or from within a *concurrent* transaction, the MicroKernel *implicitly* tries to lock that record on behalf of the client. In an exclusive transaction, an implicit record lock is unnecessary because the MicroKernel locks the entire file prior to performing the update or delete operation. (See File Locks).

The MicroKernel can grant an implicit record lock to a client as long as no other client:

- Holds an explicit lock on the record.
- Holds an implicit lock on the record.
- Has locked the file containing the record.

Note: The MicroKernel allows any single client to hold both an explicit lock and an implicit lock on the same record.

The MicroKernel performs the specified Update or Delete operation only if it can successfully obtain the implicit record lock *and* any other locks required to secure the integrity of the file during execution of the operation. (See Implicit Locks).

If the operation is in a nontransactional environment, the MicroKernel drops the implicit record lock on completion of the update or delete operation. If the operation is in a concurrent transaction, the MicroKernel retains the lock. The lock then remains in effect until the client ends or aborts the transaction, or the client is reset (which implies an Abort Transaction operation). No explicit Unlock operation is available to release implicit record locks.

By retaining implicit locks during a transaction, the MicroKernel can prevent conflicts that occur as the result of a client explicitly locking a record (via a Get/Step operation with a lock bias value) if that record has a new uncommitted image that another client generates.

Consider what could happen if the MicroKernel did not retain implicit locks. Client 1, from within a concurrent transaction, performs an update on record A, thereby altering the image of the record. However, because client 1 has not ended its concurrent transaction, it has not committed the new image. Client 2 attempts to read and lock record A.

With no implicit locks retained, client 1 no longer has an implicit record lock on record A, meaning that client 2 can successfully read and lock the record. However, client 2 reads the *old* image of record A, because client 1 has not committed the new image. When client 1 ends its transaction (committing the changed image of record A) and client 2 attempts to update record A, the MicroKernel returns status code 80 (Conflict) because client 2's image of that record is no longer valid. See the example under Example Without Implicit Locks.

Consider a situation in which a client has locked a record (either explicitly or implicitly) or has locked the entire file containing that record. If another client attempts to update or delete the record in question from within a concurrent transaction – if it tries to implicitly lock the record – some implementations of the MicroKernel will wait, continually retrying until the client whose lock is blocking the operation releases that lock. (No version of the MicroKernel attempts any retry effort for a nontransactional update or delete.)

Supplying a bias value of 500 on the Begin Concurrent Transaction (1519) operation forces the MicroKernel not to retry the insert, update, and delete operations within a transaction.

For local clients, the MicroKernel performs deadlock detection. However, because the 500 bias suppresses retries, the MicroKernel does not need to perform deadlock detection.

This 500 bias value on the Begin Transaction operation can be combined with the record lock bias values. For example, using 1019 + 500 + 200 (1719) suppresses retries for insert, update, and delete operations *and* specifies single no-wait read locks at the same time.

Example Without Implicit Locks

The following example illustrates the usefulness of implicit locks. In the example, temporarily assume that implicit locks do not exist.

Client1	Client2
1. Begin concurrent transaction.	
2. Read record A.	
3. Update record A (locks on pages involved, but no implicit lock on record).	
	4. Read record A with single-record lock (explicit lock on record).
5. End transaction (releases page locks).	
	6. Update record A (conflict, status code 80).
	7. Reread record A with lock.
	8. Update record A.

Assuming that the MicroKernel does not apply an implicit record lock in Step 3, client 2 can successfully read and lock record A in Step 4 but cannot update that record in Step 6 because in Step 4, client 2 reads a valid image of record A. However, by the time client 2 reaches Step 6, that image is no longer valid. In Step 5, client 1 commits a new image of record A, thereby invalidating the image of the record read by client 2 in Step 4.

In reality, however, the MicroKernel implicitly locks record A in Step 3, which means that the MicroKernel returns status code 84 in Step 4, requiring client 2 to retry its read operation until client 1 performs Step 5.

Consider what would happen if Steps 3 and 4 were reversed in the preceding example. Client 2 obtains an explicit lock on record A. Client 1 is forced to wait and retry its update operation until client 2 completes its own update of record A (which releases client 2's explicit lock on that record). On client 1's next retry to update record A, the MicroKernel returns status code 80. This status indicates that client 1's image of record A was no longer valid (client 1 having read record A prior to that record being changed by client 2).

Implicit Locks

Clients have significant freedom to modify a file simultaneously because they share cache under the same MicroKernel. Nontransactional modifications (insert, update, or delete operations) never

block other nontransactional modifications or modifications in concurrent transactions by another client. Pending modifications in a concurrent transaction do not block other modifications (either nontransactional or in concurrent transactions), as long as those changes do not affect the same records.

The MicroKernel tries, on behalf of the client, to implicitly lock the *records* that are modified during execution of an insert, update, or delete operation if the modification occurs either outside of a transaction or from within a concurrent transaction. (In an exclusive transaction, an implicit record or page lock is unnecessary because the MicroKernel locks the entire file prior to performing an update or delete operation. In the case of an Insert operation, the MicroKernel requests a file lock if the client does not have one yet. See File Locks) As with implicit record locks, the MicroKernel Engine provides implicit page locks; the client does not explicitly request them.

The records on a data page being modified (or inserted) must always be locked. However, a single operation might need to lock several other records as well. For example, if the change made to a record involves one or more of the record's keys, then the MicroKernel must lock the records on the index pages containing the affected key values. The MicroKernel must also lock all index pages modified by the action of balancing the B-tree(s) during operation. If a modification affects the variable-length portion of a record, the MicroKernel must lock the variable pages as well.

If such an operation is performed in a nontransactional environment, the MicroKernel drops the implicit record locks on completion of the operation. If the operation is performed from within a concurrent transaction, the MicroKernel retains the locks, which then remain in effect until the client ends or aborts the transaction, or until the client is reset, which implies an Abort Transaction operation. No explicit Unlock operation is available to release implicit record or page locks.

If an Insert, Update, or Delete operation issued in a concurrent transaction must modify a record or page (it requires an implicit record lock), but that record or page is currently locked by another concurrent transaction (or the whole file is locked by an exclusive transaction), the MicroKernel waits, continually retrying the operation until the client whose lock is blocking the operation releases that lock. The MicroKernel does not attempt any retry effort for a nontransactional update or delete.

When a client is unsuccessful in getting an implicit record lock, the client can suppress retries on the operation by using bias value 500 on the Begin Concurrent Transaction operation.

Implicit page locks and explicit or implicit record locks have no blocking effect on each other. A client can read and lock a record on a page even if another client has implicitly locked the page containing that record (as long as the record to be locked is not the same one that has been updated, as discussed in Implicit Record Locks). Conversely, a client can update or delete a

record, thereby implicitly locking the data page that contains the affected record even if that data page contains a record already locked by another client.

File Locks

When a client touches a file for the first time in an exclusive transaction, that client tries to obtain a file lock.

Note: As the preceding statement implies, the MicroKernel does *not* lock a file when the client performs a Begin Transaction operation. The lock occurs only when the client reads or modifies a record after performing the Begin Transaction operation.

A file lock, as its name suggests, locks the entire file. A client's file locks remain in effect until that client ends or aborts the transaction, or until the client is reset (which implies performing an Abort Transaction operation).

If a client tries to lock a file in an exclusive transaction but another transaction already holds a lock on that file (a record, page, or file lock), the MicroKernel waits, continually retrying the operation until the client whose lock is blocking the operation releases that lock. In addition, if a local client blocks the operation and the MicroKernel detects a deadlock situation, the MicroKernel returns status code 78 (Deadlock Detected).

When a client is unsuccessful in getting a file lock, the client can suppress retries on the operation using a no-wait lock bias value of 200 or 400 on the Begin Exclusive Transaction (219 or 419) operation. If a client starts a transaction in this way, the MicroKernel returns status code 84 or 85 when a file lock cannot be granted.

Bias values 200 and 400 are derived historically from record locks. However, the concept of single and multiple locks from the record lock environment means nothing in an exclusive transaction environment. In effect, all records in the file are locked when the file is locked. Only the no-wait meaning of the biases is preserved in the exclusive transaction environment.

The MicroKernel accepts a wait lock bias (100 or 300) on a Begin Exclusive Transaction operation (119 or 319, respectively). However, these additional bias values have no meaning because the default mode on the Begin Transaction (19) operation is to wait.

When any part of a file is first touched in an exclusive transaction, the MicroKernel locks the entire file. Therefore, the MicroKernel ignores record lock bias values explicitly added to the operation codes for any Get or Step operations performed inside an exclusive transaction, with the following exception.

When a client performs a Begin Transaction operation in wait mode (19, 119, or 319), but the first read (Get or Step operation) in that transaction is biased by 200 or 400 (a no-wait lock bias), the

no-wait bias takes precedence over the Begin Transaction operation's wait mode. Therefore, when the client performs this biased read operation but cannot lock the file (for example, another client has already locked a record in the file), the MicroKernel does not wait (which is its default) and does not check for deadlock, because it assumes that the client retries the read operation an unlimited number of times. In this same situation, other versions of the MicroKernel that perform retries automatically recognize the no-wait bias as an indication not to retry the file lock and not to check for deadlock.

Note: The 200 and 400 bias values on a Get or Step operation performed from within an exclusive transaction have only the meaning of not waiting; they do not request an explicit record lock, as they would from within a concurrent transaction.

File locks are incompatible with both record locks and page locks; therefore, the MicroKernel does not grant a file lock to a client if another client holds a record or a page lock on that file. Conversely, the MicroKernel does not grant a record or page lock to a client if another client has already locked that file.

Examples of Multiple Concurrency Control

The following examples illustrate the use of the different concurrency control mechanisms.

Example 1

Example 1 shows the interaction between explicit and implicit record locks, implicit page locks, and passive concurrency. Assume that the two records manipulated in this example (record A and record B) reside on the same data page and that the file has only one key. For further explanation of each step, see the paragraphs following the example.

The following table shows the interactions among implicit and explicit record locks, implicit page locks, and passive concurrency.

Client1	Client2	Client3 (Nontransactional)
1. Begin Concurrent Transaction with multiple no-wait lock bias (1419)		
	2. Begin Concurrent Transaction with single wait lock bias (1119)	
3. Read Record A using Get Equal with single no-wait lock bias (205)		
	4. Read record B using Get Equal (5) (single wait lock bias inherited)	
		5. Read record B using Get Equal (5)
		6. Attempt to Delete (4) record B: MicroKernel returns status code 84, and client 3 must retry
	7. Update (3) record B	
8. Attempt to Update (3) record A: MicroKernel must retry		
	9. End Transaction (20)	
10. Retry Update (3) of record A: Successful		

Client1	Client2	Client3 (Nontransactional)
		11. Retry Delete (4) of record B: MicroKernel returns status code 80, and client 3 must reread record B
		12. Reread record B using Get Equal (5)
		13. Retry Delete (4) of record B: MicroKernel returns status code 84
14. End Transaction		
		15. Retry Delete (4) of record B: Successful

In Step 1, client 1's Begin Concurrent Transaction operation specifies a generic bias value of 400 (multiple-record no-wait locks). This bias will be inherited by each nonbiased Get or Step operation in this transaction. At this point, the MicroKernel has applied no locks to the file, its pages, or its records.

In Step 2, client 2's Begin Transaction operation specifies a generic bias value of 100 (single-record wait locks). This bias will be inherited by each nonbiased Get or Step operation in this transaction. The MicroKernel still has not applied any locks to the file, its pages, or its records.

In Step 3, client 1's Get Equal operation specifies a bias value of 200 (single-record no-wait lock). The MicroKernel accepts this bias value rather than the inherited 400 (multiple no-wait record lock), because an individual operation's specified bias value takes precedence over any inherited bias value.

In Step 4, client 2's Get Equal (5) operation does not specify any bias value of its own; therefore, it inherits the single wait lock bias value of 100 from client 2's Begin Concurrent Transaction (1119) operation. Even though record A and record B are on the same page, both lock requests (Step 3 and Step 4) are successful because a record lock request locks only the specified record; it locks neither the data page on which the record is located, nor any associated index pages.

In Step 5, client 3's nontransactional Get Equal (5) operation with no lock request is successful, because nontransactional reads are always successful (as long as the requested record exists).

In Step 6, client 3 attempts to Delete (4) record B; however, it cannot obtain the implicit record lock on record B required to delete the record, because client 2 holds an explicit lock on the

record. Consequently, the MicroKernel returns status code 84 (Record or Page Locked) to client 3. Client 3 must then relinquish control and retry deleting (if it wishes) later.

In Step 7, client 2 first successfully obtains an implicit record lock on record B. While record B is already explicitly locked by client 2 (because of Step 4's inherited single wait lock bias from Step 2), no problem exists because both the explicit lock and the implicit lock belong to the same client. At this same time, client 2 also successfully obtains page locks on the data page containing record B and on the index page containing record B's key value.

Note: The data page locked by client 2 also contains record A, which is explicitly locked by client 1. However, as explained in Implicit Locks, record locks do not block page locks.

When the MicroKernel performs the actual Update (3) operation in Step 7, it writes the new, *uncommitted* images of the modified data and index pages as shadow pages to the file. At this point, the MicroKernel releases client 2's explicit lock on record B. However, client 2 retains its implicit record lock on record B, as well as the implicit page locks it just obtained. Even after client 2 completes its Update (3) operation in Step 7, client 3 still cannot get the implicit record lock on record B, because now client 2 holds an implicit record lock on the record. Client 3 must continue its retry efforts.

If the clients are remote, client 2 puts a pending modification status on the file (in addition to the page locks, which are still necessary for concurrency control among clients local to client 2) before actually updating the file.

In Step 8, client 1 first successfully obtains an implicit record lock on record A. Even though record A's data page has already been locked by client 2, no lock conflict exists, because page locks do not block record locks (see Implicit Locks). Next, client 1 attempts to obtain an implicit page lock on the data page containing record A. This attempt fails because the data page has already been locked by client 2 in Step 7. Because the Begin Concurrent Transaction (1419) operation did not have a 500 bias specified, the MicroKernel retries the operation. The MicroKernel also performs deadlock detection if the clients are local.

Had client 1 issued its Begin Transaction operation with an additional 500 bias (1919), the MicroKernel would have returned control to the user immediately.

If the clients are remote, client 1 encounters the pending modification status set by client 2 in Step 7. Therefore, the MicroKernel retries the operation.

In Step 9, by ending its transaction, client 2 releases its implicit record lock on record B and the implicit page locks on the data index pages it locked in Step 5. At this point, the MicroKernel commits all the new page images that client 2 created during the transaction. These images now become a valid part of the file.

If the clients are remote, client 2 clears the pending modification status on the file, in addition to releasing the locks.

In Step 10, client 1's continuing update retries are finally successful, because client 2 no longer has record A's data and index pages locked.

In Step 11, despite the fact that client 2 ended its transaction in Step 9 (thereby releasing all its locks), client 3 still cannot delete record B. Now, when client 3 attempts to delete the record, the MicroKernel passive concurrency control returns status code 80 (Conflict), because client 2 has modified record B since client 3 originally read it in Step 5. At this point, client 3 must re-read the record before it can retry the Delete operation.

In Step 12, client 3 reads record B again, getting an image that reflects the changes made to the record by client 2 in Step 7 and committed in Step 9.

In Step 13, client 3 again unsuccessfully attempts to delete record B, receiving a status code 84 from the MicroKernel. This status code reflects the fact that client 1, in updating record A, has an implicit page lock on the data and index pages containing record B (assuming, as stated earlier, that the same data page contains records A and B, and that the same index page contains the key values for those records).

In Step 14, client 1 ends its transaction, committing its changes and releasing its implicit page locks.

In Step 15, client 3 is finally able to delete record B.

Example 2

Example 2 shows how file locks and passive concurrency control interact. For further explanation of each step, see the paragraphs following the example.

Client1	Client2
1. Open file 1 (0)	
2. Open file 2 (0)	
3. Open file 3 (0)	
4. Get record E, file 3 using a single record lock (105)	
	5. Open file 1 (0)
6. Begin Exclusive Transaction (119)	
7. Get record B, file 1 (5)	
	8. Get record A, file 1 (5)
	9. Update record A, file 1 (3) (status code 85, retrying)
10. Get record C, file 2 (5)	
11. Update record C, file 2 (3)	
12. Delete record B, file 1 (4)	
13. End Transaction (20)	
	14. Retry Step 9 (successful)

In Step 4, client 1 obtains an explicit record lock on record E, file 3.

In Step 6, client 1 begins an exclusive transaction. Even though client 1 has three files open, the MicroKernel has not yet locked any of those files, nor does the MicroKernel release the explicit lock on record E in file 3 as a result of performing the Begin Transaction operation.

In Step 7, client 1 obtains a file lock on file 1 as a result of touching the file. (See File Locks) Step 7 would have failed if, in a previous step (for example, between Steps 5 and 6), client 2 had read a record from file 1 using an operation with a lock bias.

In Step 8, client 2 successfully reads record A from file 1. This read is successful because it does not request any lock. However, had the Get Equal (5) operation been issued with a lock bias, the operation would have failed because client 1 currently has file 1 locked.

In Step 9, client 2 cannot obtain an implicit record lock because client 1 has the file locked. Therefore, the MicroKernel returns status code 85 (File Locked) back to client 2. Client 2 must now relinquish control and retry Step 9 until client 1 ends or aborts its transaction (which happens in Step 13). (See When a Record Has Already Been Locked)

In Step 10, client 1 obtains a file lock on file 2.

In Step 13, client 1 releases file locks on files 1 and 2.

Note: Client 1 never locked file 3 because it never touched that file in its exclusive transaction. In fact, even after Step 13, client 1 retains its explicit record lock on record E of file 3. Client 1 would have released record E only if the client had touched file 3 (thereby locking the entire file in the transaction).

In Step 14, client 2's retry of the update operation in Step 9 is finally successful.

Concurrency Control for Multiple Position Blocks

The MicroKernel supports using multiple position blocks (cursors) for the same client in the same file.

Inside either a concurrent or an exclusive transaction, multiple position blocks share the same view of modified pages. Each position block in a set of multiple position blocks sees the changes made by the set's other position blocks immediately, *even before those changes are committed*.

Multiple position blocks share all locks: explicit and implicit record locks, implicit page locks, and file locks. Consequently, for any client, no one position block's locks can prevent another position block from obtaining another lock in the same file.

When a client ends or aborts its transaction, the MicroKernel releases all that client's implicit locks and file locks. However, the MicroKernel releases a client's explicit record locks only as each position block in a file requires, regardless of whether the lock was granted from within a transaction.

For example, an Unlock (27) operation with -2 as its key value releases only the multiple-record locks belonging to the specified position block. A Close (1) operation releases only those locks obtained for the same position block that is specified when performing the Close operation. Similarly, when a client obtains a position block's first record, page, or file lock inside a transaction, the MicroKernel releases only those explicit record locks that had been obtained for that position block. In Example 2, if client 1 had opened the same file three times (instead of file 1, file 2 and file 3), and if client 1 had touched the file using only the first two position blocks, the explicit lock obtained for the third position block would have remained even after the End Transaction operation.

Multiple Position Blocks

If an application using the BTRV function has two active position blocks on the same file and issues a read with a multiple record lock for the same record from both position blocks, both receive a successful status. However, when attempting to unlock the record with either a Key Number of -1 and the position in the Data Buffer or with a Key Number of -2, the record is unlocked only if both position blocks issue the unlock calls. If only one position block makes the unlock call (it does not matter which one), another user receives a status code 84 upon trying to lock the record. After both position blocks issue the unlock, the second user can lock the record.

This behavior is also true with single record locks, although the unlock command in this case does not require a specific Key Number and position in the Data Buffer. However, both position blocks still must issue the unlock in order for another user to lock the record.

Each cursor (that is, each position block) gets a lock. The MicroKernel allows cursors of the same client to lock the same record, but each cursor must issue an unlock before the record is completely unlocked.

ClientID Parameter

When developing an application using the BTRVID function rather than BTRV, you must specify an additional parameter called a ClientID. This allows an application to assign itself more than one client identity to Btrieve and execute operations for one client without affecting the state of the other clients.

For example, assume that two applications are running on Windows and each uses three different clientIDs. This counts as six Active Clients. It does not matter if this is two instances of the same application (and the same ClientID values in each instance) or two different applications. Btrieve distinguishes between each of the six ClientIDs.

Debugging Your Btrieve Application

This chapter provides information that may be helpful in debugging your Btrieve application. It contains the following sections:

Trace Files

The MicroKernel's Trace Operations configuration option allows you to trace each Btrieve API call and save the results to a file. This is helpful in debugging applications. The following shows a sample trace file.

MicroKernel Trace File of a BUTIL STAT Call

```
MicroKernel Database Engine [Server Edition] for Windows NT trace file
Created : Wed Dec 17 18:19:09
<In> 0198   Opcode : 0026   Crs ID : 0xffffffff  Db Length : 00005   Keynum :
ff   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       00 00 00 00 00          -                           .....
KBuf:       ??                      -                           .
<Out>0198   Status : 0000   Crs ID : 0xffffffff  Db Length : 00005   Keynum :
ff   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       07 00 00 00 54          -                           ....T
KBuf:       ??                      -                           .
---------------------------------------------------------------------
<In> 0199   Opcode : 0000   Crs ID : 0xffffffff  Db Length : 00001   Keynum :
fe   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       4e 4f 54 53 48 4f 57 4e - 00                        NOTSHOWN.
KBuf:       5c 5c 4e 54 34 53 52 56 - 2d 4a 55 44 49 54 5c 43   \\NT4SRV-JUDIT\C
            24 5c 64 65 6d 6f 64 61 - 74 61 5c 74 75 69 74 69   $\demodata\tuiti

<Out>0199   Status : 0000   Crs ID : 0x00000002  Db Length : 00001   Keynum :
fe   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 4e 4f 54 53 48 4f 57 4e - 00                        NOTSHOWN.
KBuf: 5c 5c 4e 54 34 53 52 56 - 2d 4a 55 44 49 54 5c 43   \\NT4SRV-JUDIT\C
      24 5c 64 65 6d 6f 64 61 - 74 61 5c 74 75 69 74 69 $\demodata\tuiti
---------------------------------------------------------------------
<In> 0200   Opcode : 0015   Crs ID : 0x00000002  Db Length : 00028   Keynum :
fe   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 00 00 00 00 00 01 07 00 - 00 00 00 00 00 00 00 03 ................
      c3 3f 00 10 00 00 00 00 - b4 fe 36 03               .?........6.
KBuf: 00 00 00 00 1c 00 00 00 - da fe 36 03 00 00 00 00 .........6.....
      00 01 07 00 00 00 00 00 - 00 00 00 03 c3 3f 00 10.............?..

<Out>0200   Status : 0000   Crs ID : 0x00000002  Db Length : 00007   Keynum :
fe   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       03 00 0e 00 04 05 01    -                           .......
KBuf: 00 00 00 00 07 00 00 00 - da fe 36 03 03 00 0e 00 .........6.....
      04 05 01 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
---------------------------------------------------------------------
<In> 0201   Opcode : 0015   Crs ID : 0x00000002  Db Length : 33455   Keynum : ff   Clnt ID : 00 00 00
00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 2b 00 cb ff ff ff ff ff - ff ff ff ff ff ff ff 00 +...............
 00 0e 00 04 05 01 00 00 - 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 - 00 00 00 14 4e 54 34 53   ............NT4S
 52 56 2d 4a 55 44 49 54 - 5c 75 6e 6b 6e 6f 77 6e   RV-JUDIT\unknown
KBuf: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
<Out>0201   Status : 0000   Crs ID : 0x00000002  Db Length : 00064   Keynum :
ff   Clnt ID : 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 0e 00 00 10 03 70 08 00 - 00 00 01 12 00 00 00 00 .....p..........
 01 00 04 00 00 01 08 00 - 00 00 0f 00 00 00 00 00   ................
 05 00 05 00 03 05 04 00 - 00 00 0a 00 00 00 01 00 ................
 0a 00 01 00 03 01 02 00 - 00 00 00 00 00 00 02 00   ................
KBuf: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
---------------------------------------------------------------------
```

```
<In> 0202   Opcode : 0065   Crs ID : 0x00000002  Db Length : 00268   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 45 78 53 74 01 00 00 00 - 00 00 00 00 01 00 00 00 ExSt............
 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
KBuf:       ??                      -                               .
<Out>0202   Status : 0000   Crs ID : 0x00000002  Db Length : 00035   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf: 01 00 00 00 01 00 00 00 - 17 00 00 00 43 3a 5c 44 ............C:\D
 45 4d 4f 44 41 54 41 5c - 54 55 49 54 49 4f 4e 2e   EMODATA\TUITION.
 4d 4b 44                -                           MKD
KBuf:       ??                      -                               .
-----------------------------------------------------------------------
<In> 0203   Opcode : 0065   Crs ID : 0x00000002  Db Length : 00008   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       45 78 53 74 02 00 00 00 -                           ExSt....
KBuf:       ??                      -                               .

<Out>0203   Status : 0000   Crs ID : 0x00000002  Db Length : 00008   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       00 00 01 00 00 00 07 00 -                           ........
KBuf:       ??                      -                               .
-----------------------------------------------------------------------
<In> 0204   Opcode : 0001   Crs ID : 0x00000002  Db Length : 00000   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       ??                      -                               .
KBuf:       ??                      -                               .

<Out>0204   Status : 0000   Crs ID : 0x00000000  Db Length : 00000   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:       ??                      -                               .
KBuf:       ??                      -                               .

-----------------------------------------------------------------------
<In> 0205   Opcode : 0028   Crs ID : 0xffbc000c  Db Length : 00000   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:                               -
KBuf:                               -

<Out>0205   Status : 0000   Crs ID : 0xffbc000c  Db Length : 00000   Keynum :
00   Clnt ID : 00 00 00 00 00 00 00 00 00 00 85 00 50 55 00 00
DBuf:                               -
KBuf:                               -
-----------------------------------------------------------------------
```

Each operation shows values passed in to and returned by the MicroKernel. Input values are marked by `<In>` and output values are marked by `<Out>`; each is followed by a number that indicates the sequence in which the operations were processed, so that `<Out>` 0016 is the result of `<In>` 0016. The Opcode field shows the operation code performed; the Status field shows the returned status code.

`Crs ID` is the cursor ID, or handle, that the MicroKernel assigned to the request. This information can be helpful in debugging applications that support multiple clients or that support single clients with multiple cursors.

`Db Length` reflects the Data Buffer Length. `Keynum` reflects the Key Number. `Clnt ID` reflects the Client ID parameter used by the BTRVID and BTRCALLID functions. `DBuf` reflects the Data

Buffer contents. `KBuf` reflects the Key Buffer contents. The trace file truncates the Data Buffer and Key Buffer contents, depending on the MicroKernel configuration.

Note: To avoid a degradation in performance, only turn the Trace File setting on for short periods of time so that you can determine what operations are being handled by the MicroKernel.

Indirect Chunk Operations in Client-Server Environments

When attempting a Get Direct/Chunk (23) operation, your application may receive a status code 62. You may find that your application is specifying the Indirect Random Descriptor option (Subfunction 0x80000001), but with tracing enabled, you can see that the MicroKernel is actually receiving a Direct Random Descriptor option (Subfunction 0x80000000).

The indirect chunk option allows an application to specify a pointer to a data address in the application's memory block where the data should be stored after it is retrieved, rather than having the data returned in the actual Data Buffer parameter of the Btrieve call. However, since the application is running in an environment in which the MicroKernel does not have direct access to the application's memory, the Btrieve client Requester must convert indirect chunk requests into direct chunk requests before sending the request to the MicroKernel.

All applications always use interprocess communication (IPC) to communicate between the application and the MicroKernel. Since IPC is required, the MicroKernel has no access to the application's memory, so the client Requester must allocate a single contiguous block of memory and adjust all data pointers to point into that memory block. On return from the MicroKernel, the Requester converts the Data Buffer back into the appropriate format for the indirect option and moves the returned data chunk into the designated, indirect address in the application's memory block.

Engine Shutdowns and Connection Resets

If you are developing a multithreaded console application targeting either Windows 9x or Windows NT, you must set up a control handler routine to handle a potential Ctrl-C keystroke. In this control handler routine, you must clean up all of your Btrieve clients by issuing either a Reset (28) or Stop (25) operation. The cleanup process must complete before your application passes the Ctrl-C event on to the operating system.

If your application still has active clients when the system terminates the threads, the MicroKernel cannot clean up its connection with the application and is forced to allocate more system resources. This causes performance degradation and significantly increases the time needed for the engine to shut down. For more information about control handler routines, see the Microsoft documentation.

Reducing Wasted Space in Files

The MicroKernel allocates disk space as needed. If there is not enough room in the file when your application inserts new records, the MicroKernel dynamically allocates additional data and index pages to the file. The size of each allocated page is the same as the page size with which the file was created. The MicroKernel also updates directory structures to reflect the new file size.

Once the MicroKernel allocates space to a file, that space remains allocated as long as the file exists.

To reduce the space required for a file from which numerous records have been deleted, you can use the Maintenance utility as follows

1. Create a new file with the same characteristics as the original file.

   In the interactive Maintenance utility, the commands are **Open** and **Create As** from the **File** menu. In the command-line based Maintenance utility, the command is CLONE.

2. Insert records into the new file using one of the following methods:
   - Write a small application that reads the records from the original file and inserts them into the new file.
   - In the command-line Maintenance utility, use the SAVE command and then the LOAD command. Alternatively, you can use the COPY command.
   - In the interactive Maintenance utility, use the Save command and then the Load command from the Data menu. Alternatively, you can use the Copy command from the Data menu.

3. Rename the new file and then delete the original file from the disk.

Btrieve API Programming

The following topics provide information to help you begin developing a Zen application by making direct calls to the Btrieve API. The most common programming tasks are included with sample code and sample structures for Visual Basic and Delphi.

- Fundamentals of Btrieve API Programming
- Visual Basic Notes
- Delphi Notes
- Starting a Zen Application
- Btrieve API Code Samples
- Declarations of Btrieve API Functions for Visual Basic

Fundamentals of Btrieve API Programming

The following flow charts demonstrate which Btrieve operations to use to insert, update, and delete records. For more detailed information about APIs, see *Btrieve API Guide*.

Btrieve API Flow Chart

Inserting Records

1. OPEN (0)
2. INSERT (2) to add record (repeat)
3. CLOSE (1) file
4. STOP (25) to release resources

Updating Records

1. OPEN (0)
2. GET EQUAL (5) or some other single-record retrieval operation to find existing record and establish physical currency
3. Modify record
4. UPDATE (3)
5. CLOSE (1)
6. STOP (25) to release resources

Deleting Records

1. OPEN(0)
2. GET EQUAL(5) or some other single-record retrieval operation to find existing record and establish physical currency
3. DELETE(4)
4. CLOSE(1)
5. STOP (25) to release resources

Visual Basic Notes

When you develop Zen applications with Visual Basic, be aware of the following things.

Visual Basic has a known byte alignment issue with user-defined data types. You can also see the Visual Basic topic regarding the Btrieve API for information about this issue and using the PAln32.DLL, the Btrieve Alignment DLL.

Creating a record class for each type of resulting record facilitates data access as shown in the following steps:

1. Create a class named Record.
2. Create a structure defining the layout of your record:

```
Type Byte2
  field1 byte
  field2 byte
End Type
Type Rec
  Size As Byte2
  Name As String*30    'SQL Mask = x30
  Salary As String*10  'SQL Mask = zzzzzzz.99
End Type
```

3. Use iAsciiFlag = 1 and ispacing=0 to read data into an instance of Rec:

```
Dim instofRec As New Rec
```

4. Use dot notation to access data:

```
instofRec.Name="Jeff"
```

5. Use the record class to handle all the instofRec data manipulation.

Delphi Notes

The following are things to consider when you use Delphi to develop your Zen application.

- Unlike older versions of Pascal, the Delphi *string* type (without a length specifier) is dynamic and null terminated. This means that you are not guaranteed to have memory allocated to the string buffer until you assign it a value. This means when using the type *string*, you need to pad it large enough to hold the expected results. Using the StringOfChar() function, you can assign a blank string large enough to accommodate the expected return value from Btrieve, as illustrated in the following example:

```
CustKeyBuffer: string;   //long string
CustBufLen   : word;
   //MAX_KEY_LEN is 255 since BTRV() always passes 255 for the Key Length :
CustKeyBuffer := StringOfChar(' ', MAX_KEY_LEN);
CustBufLen := SizeOf(CustRec);
Status := BTRV(B_GET_FIRST, CustPosBLock, CustRec, CustBufLen, CustKeyBuffer, 0);
       {CustKeyBuffer now has the value of the key for}
   {the record retrieved}
```

- Not all of the Delphi samples included in this documentation illustrate error reporting. However, you should check return codes after every call.
- If you try the samples in this chapter, for the Fetches that use the INTERNAL_FORMAT style, the order of the fields in the query must match the order of members you fetch from the data record. When you use the FillGridHeaders() routine, you must stuff the grid in the same order as the query lists the fields.

Starting a Zen Application

When you are developing a Zen application, you must include the appropriate source module for the specific programming interface.

- BTRCONST and BTRAPI32 – source modules needed for Btrieve applications

Adding Zen Source Modules

You need to include either the Btrieve source module into the programming interface in which you are developing your applications.

To add a Btrieve source module in a Visual Basic project

1. Start a new project in Visual Basic.
2. Add an existing standard module to your project, if appropriate.
3. Add the Zen source modules.

To add a Btrieve source module in a Delphi project

1. Start a new project in Delphi.
2. Select **Options** from the **Project** menu.
3. Click the **Directories** tab.
4. Insert the location <*path*>\INTF\DELPHI in the **Search Path** data field (*path* refers to where you installed the Zen SDK components).
5. Include source modules in your USES clause.

Btrieve API Code Samples

This section shows Visual Basic, Delphi, and C/C++ code samples for the following tasks you can perform using the Btrieve API:

- Creating a File
- Inserting Records
- Updating Records
- Performing Step Operations
- Performing Get Operations
- Chunking, BLOBs, and Variable-Length Records
- Working with Segmented Indexes

Creating a File

The Create (14) operation lets you generate a file from within your application. To create a file, you need to create a structure that contains the necessary information required to build a new Btrieve file.

For specific information about this API, see *Btrieve API Guide*.

Sample Code

The following sample code illustrates how to create a file using the Create operation.

Visual Basic (Creating a File)

This subroutine creates a file called Orders.

```
Sub CreateOrdersFile(OrdFileLocation As String)

        'The following syntax sets up file specifications.
  OrdFixedRecSize = Len(OrdRecBuf)
  FileDefinitionBuffer.RecLen = OrdFixedRecSize
  FileDefinitionBuffer.PageSize = 4096
  FileDefinitionBuffer.IndxCnt = 2
  FileDefinitionBuffer.FileFlags = VARIABLELENGTH

       'Key 0, by order number.
  FileDefinitionBuffer.KeyBuf0.KeyPos = 1
  FileDefinitionBuffer.KeyBuf0.KeyLen = 4
  FileDefinitionBuffer.KeyBuf0.KeyFlags = EXTTYPE + MODIFIABLE
  FileDefinitionBuffer.KeyBuf0.KeyType = Chr$(BAUTOINC)

       'Key 1, by contact number.
  FileDefinitionBuffer.KeyBuf1.KeyPos = 5
```

```
  FileDefinitionBuffer.KeyBuf1.KeyLen = 4
  FileDefinitionBuffer.KeyBuf1.KeyFlags = EXTTYPE + MODIFIABLE + DUP
  FileDefinitionBuffer.KeyBuf1.KeyType = Chr$(BUNSGBIN)

  BufLen = Len(FileDefinitionBuffer)
  OrdFileLocation = OrdFileLocation & " "
  KeyBufLen = Len(OrdFileLocation)
  CopyMemory OrdKeyBuffer, OrdFileLocation, Len(OrdFileLocation)
  giStatus = BTRCALL(BCREATE, OrdPosBlock, FileDefinitionBuffer, BufLen, _
                     ByValOrdFileLocation, KeyBufLen, 0)
End Sub
```

Delphi (Creating a File)

The following routine creates a variable-length file called Customer.

```
function CreateCustomerFile(FileName: String): SmallInt;
var
  CustRec : CustomerRecordType;              //User defined record structure
  CustBufLen : word;
  CustPosBlock : TPositionBlock;       //array[1..128] of byte
  CustFileLocation : String[255];
  CustFixedRecSize : LongInt;
  FileDefinitionBuffer : FileCreateBuffer; //Structure type for creating a file
  FilebufLen : Word;
  KeyNum : ShortInt;

begin
    {The following syntax defines file specifications.}
  {calculate the size of just the fixed portion of the record.}
  CustFixedRecSize                          := SizeOf(CustRec) - SizeOf(CustRec.Notes);
  FileDefinitionBuffer.fileSpec.recLen    := CustFixedRecSize;
  FileDefinitionBuffer.fileSpec.PageSize  := 4096;
  FileDefinitionBuffer.fileSpec.IndexCount:= 4;
  FileDefinitionBuffer.fileSpec.FileFlags := VARIABLELENGTH;

    {Define key specifications, Key 0 by contact number}
  FileDefinitionBuffer.keyspecArray[0].Position := 1;
  FileDefinitionBuffer.keyspecArray[0].Length   := 4;{4 byte integer}
  FileDefinitionBuffer.keyspecArray[0].Flags    := KFLG_EXTTYPE_KEY +  KFLG_MODX;
  FileDefinitionBuffer.keyspecArray[0].KeyType  := AUTOINCREMENT_TYPE;

  {Key 1, by contact name.}
  FileDefinitionBuffer.keyspecArray[1].Position := 5;
  FileDefinitionBuffer.keyspecArray[1].Length   := 30;
  FileDefinitionBuffer.keyspecArray[1].Flags    := KFLG_EXTTYPE_KEY +KFLG_MODX +
                                                   KFLG_DUP;
  FileDefinitionBuffer.keyspecArray[1].KeyType  := STRING_TYPE;

  {Key 2, by company name.}
  FileDefinitionBuffer.keyspecArray[2].Position := 35;
  FileDefinitionBuffer.keyspecArray[2].Length   := 60;
  FileDefinitionBuffer.keyspecArray[2].Flags    := KFLG_EXTTYPE_KEY + KFLG_MODX +
                                                   KFLG_DUP;
  FileDefinitionBuffer.keyspecArray[2].KeyType  := STRING_TYPE;

    {Key 3 by sales representative, by next contact date.}
  FileDefinitionBuffer.keyspecArray[3].Position := 220;
  FileDefinitionBuffer.keyspecArray[3].Length   := 4;
  FileDefinitionBuffer.keyspecArray[3].Flags    := KFLG_EXTTYPE_KEY + KFLG_MODX +
                                                   KFLG_DUP + KFLG_SEG;
  FileDefinitionBuffer.keyspecArray[3].KeyType  := LSTRING_TYPE;
  FileDefinitionBuffer.keyspecArray[4].Position := 223;
```

```
  FileDefinitionBuffer.keyspecArray[4].Length   := 4;
  FileDefinitionBuffer.keyspecArray[4].Flags    := KFLG_EXTTYPE_KEY + KFLG_MODX +
                                                   KFLG_DUP;
  FileDefinitionBuffer.keyspecArray[4].KeyType  := DATE_TYPE;

  CustFileLocation   := FileName + #0;    {path and file name of file to create}
  FilebufLen         := sizeof(FileDefinitionBuffer);
  KeyNum             := 0;
  FillChar(CustPosBlock, SizeOf(CustPosBlock), #0);

  Result := BTRV(B_CREATE,              //OpCode 14
                 CustPosBLock,          //Position Block (the "cursor" or "handle")
                 FileDefinitionBuffer, //Definition of new file
                 FileBufLen,            //Length of the definition
                 CustFileLocation[1],   //Path and file name
                 keyNum);               //0 (zero) means Overwrite any existing file
end;   {CreateCustomerFile}
```

C/C++ (Creating a File)

```
BTI_SINT CreateCustomerFile(LPCTSTR szCustomerFileName)
{
  Customer_Record_Type  CustRec;                //User defined record structure
  char                  CustPosBlock[POS_BLOCK_SIZE]; //"Cursor" into customer file
  char                  CustFileLocation[255];
  size_t                CustFixedRecSize;
  FileDescriptionType   FileDefBuf;             //Structure type for creating a file
  BTI_WORD              FilebufLen;
  char                  KeyNum;                 //1 byte signed int
  BTI_SINT              iStatus;

     /* calculate the size of just the fixed portion of the record: */
  CustFixedRecSize       = sizeof(CustRec) - sizeof(CustRec.Notes);
  FileDefBuf.RecLen      = CustFixedRecSize;
  FileDefBuf.PageSize    = 4096;
  FileDefBuf.IndxCnt     = 4;
  FileDefBuf.DupPointers = 4;
  FileDefBuf.FileFlags   = VAR_RECS | BALANCED_KEYS;
    /* DEFINE KEY SPECIFICATIONS KEY 0 - BY CONTACT NUMBER */
  FileDefBuf.KeyBuf[0].KeyPos   = 1;
  FileDefBuf.KeyBuf[0].KeyLen   = 4;
  FileDefBuf.KeyBuf[0].KeyFlags = EXTTYPE_KEY | MOD;
  FileDefBuf.KeyBuf[0].KeyType  = AUTOINCREMENT_TYPE;
    /* KEY 1 - BY CONTACT NAME */
  FileDefBuf.KeyBuf[1].KeyPos   = 5;
  FileDefBuf.KeyBuf[1].KeyLen   = 30;
  FileDefBuf.KeyBuf[1].KeyFlags = EXTTYPE_KEY | DUP | MOD ;
  FileDefBuf.KeyBuf[1].KeyType  = STRING_TYPE;
    /* KEY 2 - BY COMPANY NAME */
  FileDefBuf.KeyBuf[2].KeyPos   = 35;
  FileDefBuf.KeyBuf[2].KeyLen   = 60;
  FileDefBuf.KeyBuf[2].KeyFlags = EXTTYPE_KEY | DUP | MOD;
  FileDefBuf.KeyBuf[2].KeyType  = STRING_TYPE;
    /* KEY 3 - BY SALES REP BY NEXT CONTACT DATE */
  FileDefBuf.KeyBuf[3].KeyPos   = 220;
  FileDefBuf.KeyBuf[3].KeyLen   = 3;
  FileDefBuf.KeyBuf[3].KeyFlags = EXTTYPE_KEY | DUP | MOD | SEG;
  FileDefBuf.KeyBuf[3].KeyType  = STRING_TYPE;

  FileDefBuf.KeyBuf[4].KeyPos   = 223;
  FileDefBuf.KeyBuf[4].KeyLen   = 4;
  FileDefBuf.KeyBuf[4].KeyFlags = EXTTYPE_KEY   | DUP | MOD;
  FileDefBuf.KeyBuf[4].KeyType  = DATE_TYPE;
```

```
  //
  //-------------------------------------------------------------------------
  FilebufLen = sizeof(FileDefBuf);
  KeyNum = 0; // Overwrite trans
  strcpy(CustFileLocation, szCustomerFileName);
  iStatus = BTRV(B_CREATE,
                 CustPosBlock,
                 &FileDefBuf,
                 &FilebufLen,
                 CustFileLocation,
                 KeyNum);
  return(iStatus);
}   //CreateCustomerFile()
```

Sample Structures (Creating a File)

These are the sample structures used in the previous code sample for Visual Basic, Delphi, and C/C++, respectively.

Visual Basic (Creating a File) – Sample Structure

```
Declare Function BTRCALL Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, _
  Db As Any, DL As Long, Kb As Any, ByVal Kl, _
  ByVal Kn) As Integer

Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" _
  (hpvDest As Any, hpvSource As Any, ByVal cbCopy As Long)

Type OrderRecordBufferType
  OrderNumberAs typ_byte4        '4 byte unsigned
  ContactNumber As typ_byte4     '4 byte unsigned
  OrderDateAs DateType
  OrderTotal As typ_byte4        '4 byte real
  NotUsed As String * 64
End Type

Type OrderKeyBufferType
  BufferValue(255) As Byte
  OrderNumber As typ_byte4
  CustNumber As typ_byte4
  NotUsed As String * 64
End Type

Type FileSpec
  RecLenAs Integer
  PageSize As Integer
  IndxCnt As Integer
  NotUsedAs String * 4
  FileFlags As Integer
  Reserved As String * 2
  Allocation As Integer
  KeyBuf0 As KeySpec
  KeyBuf1 As KeySpec
  KeyBuf2 As KeySpec
  KeyBuf3 As KeySpec
  KeyBuf4 As KeySpec
  KeyBuf5 As KeySpec
End Type

Global FileDefinitionBuffer As FileSpec
```

```
      {The following are the Order table variables.}
Global OrdPosBlock              As Byte(0 to 127)
Global OrdRecPos                As typ_byte4
Global OrdRecBuf                As OrderRecordBufferType
Global OrdKeyBuffer             As OrderKeyBufferType
Global OrdFixedRecSize          As Long
Global OrdFileLocation          As String
```

Delphi (Creating a File) – Sample Structure

```
type
  CustomerRecordType = packed record
    Anything you want…
  end;    //CustomerRecordType

type
  TPositionBlock = array[0..127] of byte;

type
  BTI_KEY_DESC = packed record
    Position  : BTI_SINT;
    Length    : BTI_SINT;
    KeyFlags  : BTI_SINT;
    NumUnique : BTI_LONG;
    ExtKeyType: BTI_BYTE;
    NullVal   : BTI_BYTE;
    Reserv    : array [0..1] of BTI_CHAR;
    KeyNumber : BTI_UBYTE;
    ACSNumber : BTI_UBYTE;
  end;   {BTI_KEY_DESC}

  BTI_KEY_ARRAY = array [0..MAX_KEY_SEG - 1] of BTI_KEY_DESC;
  BTI_ACS       = array [0..ACS_SIZE - 1] of BTI_CHAR;

type
  FileCreateBuffer = packed record
    RecLen     : BTI_SINT;
    PageSize   : BTI_SINT;
    NumKeys    : BTI_SINT;
    Reserved1  : BTI_LONG;
    FileFlags  : BTI_SINT;
    DupPointers: BTI_BYTE;
    Reserved2  : BTI_BYTE;
    Alloc      : BTI_SINT;
    Keys       : BTI_KEY_ARRAY;
    ACS        : BTI_ACS;
  end;   {BTI_FILE_DESC}
```

Note that the alternate collating sequence (ACS) is placed after the entire Keys array for ease of definition. Because Btrieve expects the ACS to immediately follow the last key segment, the ACS must be moved down to that position within the structure.

C/C++ (Creating a File)– Sample Structure

```
struct CustRec
{
  anything you want…
}  //CustRec
struct date_type
{
  BTI_BYTE  day;
```

```
  BTI_BYTE  month;
  BTI_SINT  year;
};  //date_type

struct KeySpec
{
  BTI_SINT  KeyPos;
  BTI_SINT  KeyLen;
  BTI_SINT  KeyFlags;
  BTI_LONG  KeyTot;
  BTI_CHAR  KeyType;
  BTI_CHAR  NulValue;
  BTI_CHAR  NotUsed[2];
  BTI_BYTE  KeyNumber;
  BTI_BYTE  ACSNum;
};  //KeySpec
struct FileDescriptionType
{
  BTI_SINT  RecLen;
  BTI_SINT  PageSize;
  BTI_SINT  IndxCnt;
  BTI_LONG  RecTot;
  BTI_SINT  FileFlags;
  BTI_BYTE  DupPointers;
  BTI_BYTE  NotUsed;
  BTI_SINT  Allocation;
  KeySpec   KeyBuf[119];
};  //FileDescriptionType
```

Inserting Records

The Insert (2) operation inserts a record into a file. Before you can make a call to this API:

- The file must be open.
- The record to be inserted must be the proper length, and the key values must conform to the keys defined for the file.

You can insert a row by calling BINSERT with the row to be inserted in the data buffer. For specific information about this API, see *Btrieve API Guide*. The following sample code and sample structures illustrate how to perform the Insert operation in Visual Basic, Delphi, and C/C++.

Sample Code

The following sample code illustrates how to insert a record using the Insert operation.

- Visual Basic (Inserting Records)
- Delphi (Inserting Records)
- C/C++ (Inserting Records)

Visual Basic (Inserting Records)

```
FillCustBufFromCustomerEdit

InsertCustomerRecord  'BtrCallModule Procedure

Sub FillCustBufFromCustomerEdit()
  Dim tmplong As Long
  Dim StrDay As String * 2
  Dim StrMonth As String * 2
  Dim StrYear As String * 4

  tmplong = CLng(FormCustomerEdit.EdContactNumber.Text)
  CustRecBuf.ContactNumber = ToType4(tmplong)
'see this function in the Sample Structures (Inserting Records) Sample
  CustRecBuf.ContactName       = FormCustomerEdit.EdContactName.Text
  CustRecBuf.CompanyName       = FormCustomerEdit.EdCompanyName.Text
  CustRecBuf.Address1          = FormCustomerEdit.EdAddress1.Text
  CustRecBuf.Address2          = FormCustomerEdit.EdAddress2.Text
  CustRecBuf.City              = FormCustomerEdit.EdCity.Text
  CustRecBuf.State             = FormCustomerEdit.EdState.Text
  CustRecBuf.ZipCode           = FormCustomerEdit.EdZip.Text
  CustRecBuf.Country           = FormCustomerEdit.EdCountry.Text
  CustRecBuf.SalesRep          = FormCustomerEdit.EdSalesRep.Text
  StrDay                       = Mid$(FormCustomerEdit.EdContactDate.Text, 1, 2)
  StrMonth                     = Mid$(FormCustomerEdit.EdContactDate.Text, 4, 2)
  StrYear                      = Mid$(FormCustomerEdit.EdContactDate.Text, 7, 4)
  CustRecBuf.NextContact.Day   = CByte(StrDay)
  CustRecBuf.NextContact.Month = CByte(StrMonth)
  CustRecBuf.NextContact.Year  = CInt(StrYear)
  CustRecBuf.PhoneNumber       = FormCustomerEdit.EdPhone.Text
  CustRecBuf.Notes             = Trim(FormCustomerEdit.EdNotes.Text) & Chr$(0)
  FormCustomerEdit.EdRecLength = Str(CustBufLength)
End Sub

Sub InsertCustomerRecord()
  Dim lCustBufLength As Long
  Dim iKeyNum As Integer
  Dim iKeyBufLen As Integer
  lCustBufLength = Len(CustRecBuf) - MaxNoteFieldSize + _
    Len(Trim(CustRecBuf.Notes))
  ' CustBufLength = 238
  iKeyBufLen = KEY_BUF_LEN
  iKeyNum = CustKeyBuffer.CurrentKeyNumber
  giStatus = BTRCALL(BINSERT, CustPosBlock, CustRecBuf, _
                     lCustBufLength, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub
```

Delphi (Inserting Records)

```
function InsertCustomerRecord( var CustPosBlock :  TPositionBlock;
                                   CustRec      :  CustomerRecordType)
                                                :  SmallInt;
var
 CustBufLen : Word;
 KeyNum : ShortInt;
 CustKeyBuffer: String[255];
begin
      {Calculate the total size of variable-length record}
 CustBufLen := SizeOf(CustRec) - SizeOf(CustRec.Notes) +  Length(CustRec.Notes);
 KeyNum := -1; {specify "No Currency Change" during insert}
 FillChar(CustKeyBuffer, SizeOf(CustKeyBuffer), #0); {not needed going in}
 Result := BTRV(B_INSERT,           //OpCode 2
                CustPosBLock,       //Already opened position block
```

```
                CustRec,            //record to be inserted
                CustBufLen,         //Length of new record
                CustKeyBuffer[1],   //not needed for NCC insert
                KeyNum);
end;   {InsertCustomerRecord}
```

C/C++ (Inserting Records)

```
BTI_SINT InsertCustomerRecord(char                 CustPosBlock[POS_BLOCK_SIZE],
                              Customer_Record_Type CustRec)
{
  BTI_WORD  CustBufLen;
  char      KeyNum;    //signed byte
  char      CustKeyBuffer[255];
  BTI_SINT  iStatus;

       /* Calculate the total size of variable length record : */
  CustBufLen = sizeof(CustRec) - sizeof(CustRec.Notes) + strlen(CustRec.Notes);
  KeyNum = -1;   //specify "No Currency Change" during insert
  memset(CustKeyBuffer, sizeof(CustKeyBuffer), 0);   //not needed going in
  iStatus = BTRV(B_INSERT,          //OpCode 2
                 CustPosBlock,      //Already opened position block
                 &CustRec,          //record to be inserted
                 &CustBufLen,       //Length of new record
                 CustKeyBuffer,     //not needed for NCC insert
                 KeyNum);
  PrintStatus("B_INSERT: status = %d", iStatus);
  return(iStatus);
}  // InsertCustomerRecord()
```

Sample Structures (Inserting Records)

These are the sample structures used in the previous code sample for Visual Basic, Delphi, and C/C++, respectively.

Visual Basic (Inserting Records) – Sample Structure

```
Global Const BINSERT = 2

     ' The following are Customer table data structures.

Type CustomerRecordBufferType
  ContactNumber As typ_byte4
  ContactNameAs String * 30
  CompanyName As String * 60
  Address1 As String * 30
  Address2 As String * 30
  City As String * 30
  StateAs String * 2
  ZipCodeAs String * 10
  CountryAs String * 3
  PhoneNumberAs String * 20
  SalesRepAs String * 3
  NextContactAs DateType
  NotUsedAs String * 12
  Notes As String * MaxNoteFieldSize
End Type

     'The following are Customer file variables.
Global CustPosBlockAs Byte(0 to 127)
```

```
Global CustRecBuf As CustomerRecordBufferType
Global CustKeyBufferAs CustomerKeyBufferType
Global CustFixedRecSize As Long
Global CustFileLocationAs String
Global CustPositionAs typ_byte4
Global CustPosPercent As typ_byte4

Function ToInt(vValue As typ_byte4) As Long
  Dim iInt As Long
  CopyMemory iInt, vValue, 4
  ToInt = iInt
End Function

Function ToType4(vValue As Long) As typ_byte4
  Dim tmpTyp4 As typ_byte4
  CopyMemory tmpTyp4, vValue, 4
  ToType4 = tmpTyp4
End Function
```

Delphi (Inserting Records) – Sample Structure

```
type
  CustomerRecordType = packed record
    Anything you want…
  end;    //CustomerRecordType
```

C/C++(Inserting Records) – Sample Structure

```
struct CustRec
{
  anything you want…
}  //CustRec
```

Updating Records

The Update (3) operation changes the information in an existing record. To make this Btrieve call, the file must be open and have physical currency established. If you want to update a record within a transaction, you must retrieve the record within the transaction.

For more information about this API, see *Btrieve API Guide*. The following sample code and sample structures illustrate how to perform the update operation in Visual Basic, Delphi, and C/C++.

Sample Code

The following sample code illustrates modifying a file using the update operation.

- Visual Basic (Updating Records)
- Delphi (Updating Records)
- C/C++ (Updating Records)

Visual Basic (Updating Records)

```
FillCustBufFromCustomerEdit
UpdateCustomerRecord       'BtrCallModule Procedure

Sub FillCustBufFromCustomerEdit()
  Dim tmplong As Long
  Dim StrDay As String * 2
  Dim StrMonth As String * 2
  Dim StrYear As String * 4

  tmplong                      = CLng(FormCustomerEdit.EdContactNumber.Text)
  CustRecBuf.ContactNumber     = ToType4(tmplong)
  CustRecBuf.ContactName       = FormCustomerEdit.EdContactName.Text
  CustRecBuf.CompanyName       = FormCustomerEdit.EdCompanyName.Text
  CustRecBuf.Address1          = FormCustomerEdit.EdAddress1.Text
  CustRecBuf.Address2          = FormCustomerEdit.EdAddress2.Text
  CustRecBuf.City              = FormCustomerEdit.EdCity.Text
  CustRecBuf.State             = FormCustomerEdit.EdState.Text
  CustRecBuf.ZipCode           = FormCustomerEdit.EdZip.Text
  CustRecBuf.Country           = FormCustomerEdit.EdCountry.Text
  CustRecBuf.SalesRep          = FormCustomerEdit.EdSalesRep.Text
  StrDay                       = Mid$(FormCustomerEdit.EdContactDate.Text, 1, 2)
  StrMonth                     = Mid$(FormCustomerEdit.EdContactDate.Text, 4, 2)
  StrYear                      = Mid$(FormCustomerEdit.EdContactDate.Text, 7, 4)
  CustRecBuf.NextContact.Day   = CByte(StrDay)
  CustRecBuf.NextContact.Month = CByte(StrMonth)
  CustRecBuf.NextContact.Year  = CInt(StrYear)
  CustRecBuf.PhoneNumber       = FormCustomerEdit.EdPhone.Text
  CustRecBuf.Notes             = Trim(FormCustomerEdit.EdNotes.Text) & Chr$(0)
  FormCustomerEdit.EdRecLength = Str(CustBufLength)
End Sub

Sub UpdateCustomerRecord()
  Dim lCustBufLength As Long
  Dim iKeyBufLen As Integer
  Dim iKeyNum As Integer

      ' The following syntax updates a customer record.

lCustBufLength = Len(CustRecBuf) - MaxNoteFieldSize + _
  Len(Trim(CustRecBuf.Notes))
iKeyBufLen = KEY_BUF_LEN
iKeyNum = CustKeyBuffer.CurrentKeyNumber
giStatus = BTRCALL(bUpdate, CustPosBlock, CustRecBuf, _
                   lCustBufLength, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub
```

Delphi (Updating Records)

```
function UpdateCustomerRecord( var CustPosBlock: TPositionBlock;
                                   CustRec     : CustomerRecordType)
                                               : SmallInt;
var
 CustBufLen : Word;
 KeyNum : ShortInt;
 CustKeyBuffer: String[255];
begin
  { Calculate the total size of variable length record : }
 CustBufLen := SizeOf(CustRec) - SizeOf(CustRec.Notes) +  Length(CustRec.Notes);
 KeyNum := -1; {specify "No Currency Change" during update}
 FillChar(CustKeyBuffer, SizeOf(CustKeyBuffer), #0); {not needed going in}
 Result := BTRV(B_UPDATE,           //OpCode 3
                CustPosBLock,       //Already opened position block
```

```
                CustRec,            //new record
                CustBufLen,         //Length of new record
                CustKeyBuffer[1],   //not needed for NCC update
                KeyNum);
end;   {UpdateCustomerRecord}
```

C/C++ (Updating Records)

```
BTI_SINT UpdateCustomerRecord(char                 CustPosBlock[POS_BLOCK_SIZE],
                              Customer_Record_Type CustRec)
{
  BTI_WORD  CustBufLen;
  char      KeyNum;    //signed byte
  char      CustKeyBuffer[255];
  BTI_SINT  iStatus;
       /* Calculate the total size of variable length record : */
  CustBufLen = sizeof(CustRec) - sizeof(CustRec.Notes) + strlen(CustRec.Notes);
  KeyNum = -1;   //specify "No Currency Change" during update
  memset(CustKeyBuffer, sizeof(CustKeyBuffer), 0);   //not needed going in
  iStatus = BTRV(B_UPDATE,           //OpCode 3
                 CustPosBlock,       //Already opened position block
                 &CustRec,           //record to be inserted
                 &CustBufLen,        //Length of new record
                 CustKeyBuffer,      //not needed for NCC insert
                 KeyNum);
  PrintStatus("B_UPDATE: status = %d", iStatus);
  return(iStatus);
}  //UpdateCustomerRecord()
```

Sample Structures (Updating Records)

These are the sample structures used in the previous code sample for Visual Basic, Delphi, and C/C++, respectively.

Visual Basic (Updating Records) – Sample Structure

```
Global Const bUpdate = 3
```

See the Sample Structures (Inserting Records) for the Insert operation.

Delphi (Updating Records) – Sample Structure

```
type
  CustomerRecordType = packed record
    Anything you want…
  end;    //CustomerRecordType
```

C/C++ (Updating Records) – Sample Structure

```
struct CustRec
{
  anything you want…
}  //CustRec
```

Performing Step Operations

The Step operations (Step First, Step Next, Step Last, Step Previous) allow you to retrieve a record into the data buffer. The MicroKernel does not use a key path to retrieve the record. For more detailed information about these APIs, see *Btrieve API Guide*.

The following sample code and sample structures illustrate how to perform the Step operations in Delphi and C/C++.

Note: You should never depend on the order in which the records are returned. The MicroKernel may move a record within the file at any time. Use Get Operations if you need the records in a specific order.

Sample Code

The following sample code illustrates how to retrieve a record using the Step operations.

Delphi (Step Operations)

The following code example returns the record in the first physical location in the file.

```
   { Get First physical record from file : }
CustBufLen := SizeOf(CustRec);       //Maximum size of the data record
Status     := BTRV(B_STEP_FIRST,     //OpCode 33
                   CustPosBLock,     //Already opened position block
                   CustRec,          //Buffer for record to be returned in
                   CustBufLen,       //Maximum length to be returned
                   CustKeyBuffer[1], //Not needed in Steps
                   CustKeyNumber);   //Not needed in Steps

    {Get Second record in file: (no guaranteed order) }
CustBufLen := SizeOf(CustRec);      //Reset - previous Step may have changed it.
Status     := BTRV(B_STEP_NEXT,     //OpCode 24
                   CustPosBLock,
                   CustRec,
                   CustBufLen,
                   CustKeyBuffer[1]
                   CustKeyNumber);

     { Back to the First record : }
CustBufLen := SizeOf(CustRec);     //Reset - previous Step may have changed it.
Status     := BTRV(B_STEP_PREV,    //OpCode 35
                   CustPosBLock,
                   CustRec,
                   CustBufLen,
                   CustKeyBuffer[1],
                   CustKeyNumber);
```

C/C++ (Step Operations)

```
    /* Get First physical record from file : */
  CustBufLen = sizeof(CustRec);     //Maximum size of the data record
  iStatus = BTRV(B_STEP_FIRST,      //OpCode 33
                 CustPosBLock,      //Already opened position block
```

```
                &CustRec,          //Buffer for record to be returned in
                &CustBufLen,       //Maximum length to be returned
                CustKeyBuffer,     //Not needed in Steps
                KeyNum);           //Not needed in Steps
    /* Get Second record in file: (no guaranteed order) */
  CustBufLen = sizeof(CustRec);    //Reset - previous Step may have changed it.
  iStatus = BTRV(B_STEP_NEXT,      //OpCode 24
                CustPosBLock,
                &CustRec,
                &CustBufLen,
                CustKeyBuffer,
                KeyNum);
      /* Back to the First record : */
  CustBufLen = sizeof(CustRec);    //Reset - previous Step may have changed it.
  iStatus = BTRV(B_STEP_PREVIOUS,  //OpCode 35
                CustPosBLock,
                &CustRec,
                &CustBufLen,
                CustKeyBuffer,
                KeyNum);
```

Sample Structures

These are the sample structures used in the previous code sample for Delphi and C/C++, respectively.

Delphi (Step Operations) – Sample Structure

```
type
  CustomerRecordType = packed record
    Anything you want…
  end;   //CustomerRecordType
```

C/C++ (Step Operations) – Sample Structure

```
struct CustRec
{
  anything you want…
}  //CustRec
```

Performing Get Operations

The Get operations allow you to retrieve a record. These operations require the key buffer parameter to specify which row (or rows) to return. For more detailed information about these APIs, see *Btrieve API Guide*.

The following sample code and sample structures illustrate how to perform some Get operations in Visual Basic, Delphi, and C/C++.

Sample Code

The following sample code illustrates how to retrieve a file using the Get operations.

Visual Basic (Get Operations)

```
Sub LoadContactBox(RecPosition As typ_byte4)
  FormBrowseCustomers.lstContact.Clear
  GetDirectCustomerRecord 'BtrCallModule Procedure
  If giStatus = 0 Then

    'The following syntax writes the contact list box string.
  FormatContListBoxString
  If giStatus = 0 Then
    PosIndex = 0
    PosArray(PosIndex) = RecPosition
    FirstRecPosition = RecPosition
  End If
  Else
    FormBrowseCustomers.lblMsg.Caption = "didn't get record"
  End If

  'The following syntax fills the rest of list box.
  While giStatus = 0 And PosIndex < CustMaxNumRows - 1
  GetNextCustomerRecord 'BtrCallModule Procedure
  If giStatus = 0 Then
    'write contact listbox string
    FormatContListBoxString

    'The following syntax returns the record position.
    GetPositionCustomerRecord'BtrCallModule Procedure
    If giStatus = 0 Then
      PosIndex = PosIndex + 1
      PosArray(PosIndex) = RecPosition

    'The following syntax configures the pointers to the array of the record    'positions.

      Select Case PosIndex
        Case 1
          SecondRecPosition = RecPosition
        Case 10
          SecToLastRecPosition = RecPosition
        Case 11
          LastRecPosition = RecPosition
      End Select
    End If
  End If
  Wend
  If FormBrowseCustomers.lstContact.ListCount <> 0 Then
  FormBrowseCustomers.lstContact.ListIndex = 0
  End If
End Sub

Sub GetDirectCustomerRecord()
  Dim iKeyBufLen As Integer
  Dim iKeyNum As Integer

    'The following syntax retrieves the direct record by the Record Position.

  BufLen = Len(CustRecBuf)
  iKeyBufLen = MaxKeyBufLen
  iKeyNum = CustKeyBuffer.CurrentKeyNumber

    'The following syntax places the address in the data buffer.

  CustRecBuf.Notes = "" 'clear variable length area before retrieve
  LSet CustRecBuf = RecPosition
  giStatus = BTRCALL(BGETDIRECT, CustPosBlock, _
  CustRecBuf, BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
```

```
  DBLen = BufLen
End Sub

Sub GetNextCustomerRecord()
  Dim iKeyNum As Integer
  Dim iKeyBufLen As Integer

    'The following syntax returns the next customer record.

  BufLen = Len(CustRecBuf)
  iKeyBufLen = KEY_BUF_LEN
  iKeyNum = CustKeyBuffer.CurrentKeyNumber
  giStatus = BTRCALL(BGETNEXT, CustPosBlock, CustRecBuf, _
                     BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub

Sub GetPositionCustomerRecord()
  Dim iKeyBufLen As Integer
  Dim iKeyNum As Integer

'The following syntax returns the record position.

  BufLen = MaxDataBufLen
  iKeyBufLen = KEY_BUF_LEN
  iKeyNum = CustKeyBuffer.CurrentKeyNumber
  giStatus = BTRCALL(BGETPOS, CustPosBlock, RecPosition, _
                     BufLen, CustKeyBuffer, iKeyBufLen, iKeyNum)
End Sub
```

Delphi (Get Operations)

```
var
 CustKeyBuffer: LongInt;
begin
 CustBufLen := SizeOf(CustRec);
 CustKeyNumber := 0; {In order by Contact ID}

    {The following syntax returns the first record from the file using the specified}    {sort order.}
 CustBufLen := SizeOf(CustRec);      //Maximum size of the data record
 Status     := BTRV(B_GET_FIRST,     //OpCode 12
                    CustPosBLock,    //Already opened position block
                    CustRec,         //Buffer for record to be returned in
                    CustBufLen,        //Maximum length to be returned
                    CustKeyBuffer,     //Returns the key value extracted from the record
                    CustKeyNumber);  //Index order to use for retrieval

     {The following syntax returns the second record in a file in the specified sort}     {order.}
 CustBufLen := SizeOf(CustRec);   //Reset - previous Get may have changed it.
 Status     := BTRV(B_GET_NEXT,   //OpCode 6
                    CustPosBLock,
                    CustRec,
                    CustBufLen,
                    CustKeyBuffer[1],
                    CustKeyNumber);

  {The following syntax returns the previous record in the file.}
 CustBufLen := SizeOf(CustRec);   //Reset - previous Step may have changed it.
 Status     := BTRV(B_GET_PREV,   //OpCode 7
                    CustPosBLock,
                    CustRec,
                    CustBufLen,
                    CustKeyBuffer[1],
                    CustKeyNumber);
```

C/C++ (Get Operations)

```
    /* Get First logical record from file : */
  CustBufLen = sizeof(CustRec);     //Maximum size of the data record
  iStatus = BTRV(B_GET_FIRST,       //OpCode 12
                 CustPosBLock,      //Already opened position block
                 &CustRec,          //Buffer for record to be returned in
                 &CustBufLen,       //Maximum length to be returned
                    CustKeyBuffer,     //Returns the key value extracted from the record
                 CustKeyNumber);    //Index order to use for retrieval
    /* Get Second record in file: in order by chosen key */
  CustBufLen = sizeof(CustRec);     //Reset - previous Get may have changed it.
  iStatus = BTRV(B_GET_NEXT,        //OpCode 6
                 CustPosBLock,
                 &CustRec,
                 &CustBufLen,
                 CustKeyBuffer,
                 CustKeyNumber);
      /* Back to the First record : */
  CustBufLen = sizeof(CustRec);     //Reset - previous Get may have changed it.
  iStatus = BTRV(B_GET_PREVIOUS,    //OpCode 7
                 CustPosBLock,
                 &CustRec,
                 &CustBufLen,
                 CustKeyBuffer,
                 CustKeyNumber);
```

Sample Structures (Get Operations)

These are the sample structures used in the previous code sample for Visual Basic and Delphi, respectively.

Visual Basic (Get Operations) – Sample Structure

```
Global Const BGETNEXT = 6
Global Const BGETDIRECT = 23
Global Const BGETPOS = 22
```

Delphi (Get Operations) – Sample Structure

```
type
  CustomerRecordType = packed record
    Anything you want…
  end;    //CustomerRecordType
```

C/C++ (Get Operations) – Sample Structure

```
struct CustRec
{
  anything you want…
}  //CustRec
```

Chunking, BLOBs, and Variable-Length Records

Btrieve chunk operations allow you to read or write portions of variable-length records and BLOBs. The maximum record length is 64 GB; however, the maximum fixed-record length is 64 KB (65535 bytes). Use chunking when you want to access portions of a record beyond the first 65535 bytes.

Sample Code

The following sample code illustrates how to handle chunking, binary large objects (BLOBs), and variable-length records.

Visual Basic (Chunking/BLOBs/Variable-Length Records)

```
Private Sub LoadImageFromBtrieve()

'The following syntax returns the image stored in Btrieve into the file described
'in the output image text box.

  Dim lBytes As Long
  Dim lBytesread As Long
  Dim sLine As String
  Dim lBytesToRead As Long
  Dim iKey As Integer
  Dim lAddressMode As Long
  Dim lNumberOfChunks As Long
  Dim lChunkOffset As Long
  Dim lChunkAddress As Long
  Dim lChunkLength As Long
  Dim iNumChunksRead As Integer

  GetEqualGraphicRecord 'gets the record and part of the blob
  On Error GoTo FileNotFound

  FormCustomerGraphic.MousePointer = 11
  lNumberOfChunks = 0
  On Error GoTo BMPOpenError
  Open txtOutputImage.Text For Binary Access Write As #1
  lBytesread = (BufLen - 68)     'saves the number of bytes read - fixed length of
                'graphic record.fixed length = 68 bytes on first
                'chunk of graphic record (GetEqualGraphicRecord
                'above).

  sLine = Right(ChunkReadBuffer.ChunkArray, lBytesread)
  Put #1, , sLine          'write the first chunk to the bmp file.
  iNumChunksRead = 1
  If giStatus = 22 And (BufLen = MaxChunkSize) Then
    GetPositionGraphicRecord     'you have to have the position of the current record
                'before you can do a get chunk
 Do
  lNumberOfChunks       = 1
  lChunkOffset         = 0
  lChunkAddress         = 0
  lChunkLength         = MaxChunkSize
  iNumChunksRead        = iNumChunksRead + 1
  ChunkGetBuffer.RecordAddress = GrphPosition

 'H80000000 (Get random chunk)
```

```
    'H40000000 (Next in record bias) causes use of intrarecord currency.

   ChunkGetBuffer.AddressMode   = ToType4((&H80000000 + &H40000000))
   ChunkGetBuffer.NumberOfChunks = ToType4(lNumberOfChunks)
   ChunkGetBuffer.ChunkOffset   = ToType4(lChunkOffset)
   ChunkGetBuffer.ChunkAddress  = ToType4(lChunkAddress)
   ChunkGetBuffer.ChunkLength   = ToType4(lChunkLength)

'The previous syntax uses the read buffer. Subsequent get chunks use the entire
'buffer because the fixed length of the record was read with the first get chunk
'GetEqualGraphicRecord

'The following syntax loads the read buffer with the get buffer.

 CopyMemory ChunkReadBuffer, ChunkGetBuffer, Len (ChunkGetBuffer)
 GetGraphicChunk
 If giStatus = 0 Then    'you should get a 103 if you read past the end of the
                 'record
    If MaxChunkSize <> BufLen Then
     sLine = Left(ChunkReadBuffer.ChunkArray, BufLen)
     lBytesread = lBytesread + (BufLen)
     Else
     sLine = ChunkReadBuffer.ChunkArray
     Bytesread = lBytesread + MaxChunkSize
    End If
     If Len(sLine) > 0 Then
     Put #1, , sLine
    End If
   End If
  Loop While (giStatus = 0)
 End If
 Close #1
 On Error Resume Next
 Set Image1.Picture = LoadPicture(txtOutputImage.Text)
  FormCustomerGraphic.MousePointer = 0
  NumChunks.Text = iNumChunksRead
  NumBytes.Text = lBytesread
  LastStatus.Text = giStatus
  On Error GoTo 0
Exit Sub

   'InvalidPicture:

 MsgBox Err.Number & ": " & Err.Description & vbCrLf & "Load from disk and save", vbOKOnly, "Invalid
Picture in Graphic file"
 Resume Next

FileNotFound:
 MsgBox Err.Number & ": " & Err.Description, vbOKOnly, "Graphic Load Error"
 FormCustomerGraphic.MousePointer = 0
 On Error GoTo 0

BMPOpenError:
 MsgBox "Directory for temporary imaging work does not exist. " & vbCrLf & _

  "Please select a valid directory for image out.", vbOKOnly, "User path error"

 Screen.MousePointer = vbDefault
 On Error GoTo 0
End Sub

Sub GetGraphicChunk()
 Dim sKeyBuffer As String
 Dim iKeyBufLen As Integer
```

```
 BufLen = Len(ChunkReadBuffer)
 sKeyBuffer = Space$(KEY_BUF_LEN)
 iKeyBufLen = KEY_BUF_LEN

  {In the following syntax, the key number must be set to -2 for chunking mode.}

 giStatus = BTRCALL(BGETDIRECT, GrphPosBlock, ChunkReadBuffer, _ BufLen, ByVal
           sKeyBuffer, iKeyBufLen, -2)
End Sub
```

Sample Structures (Chunking/BLOBs/Variable-Length Records)

These are the sample structures used in the previous code sample for Visual Basic.

Visual Basic (Chunking/BLOBs/Variable-Length Records) – Sample Structure

```
Type GraphicRecordBufferType
 ContactNumber As typ_byte4
 NotUsed As String * 64
End Type

Type GraphicKeyBufferType
 BufferValue(255)As Byte
 CurrentKeyNumberAs Integer
 ContactNumber As typ_byte4
 NotUsed As String * 64
End Type

Type ChunkReadDescriptorNext
 ChunkArray As String * MaxChunkSize
End Type

Type ChunkGetDescriptor
 RecordAddressAs typ_byte4
 AddressModeAs typ_byte4
 NumberOfChunks As typ_byte4
 ChunkOffsetAs typ_byte4
 ChunkLengthAs typ_byte4
 ChunkAddress As typ_byte4
End Type

Global ChunkGetBuffer As ChunkGetDescriptor
Global ChunkReadBuffer As ChunkReadDescriptorNext

    ' Graphic Table Variables
Global GrphPosBlock As Byte(0 to 127)
Global GrphRecBuf As GraphicRecordBufferType
Global GrphKeyBuffer As GraphicKeyBufferType
Global GrphFixedRecSize As Long
Global GrphFileLocation As String
Global GrphKeyNumber As Integer
Global GrphPosition As typ_byte4
```

Working with Segmented Indexes

The following sample code illustrates how to handle segmented indexes.

Sample Code

Visual Basic (Segmented Indexes)

```
Sub CreateCustomerFile(CustFileLocation As String)

    'The following syntax creates the customer file and configures the file
  'specifications.

 CustFixedRecSize = Len(CustRecBuf) - Len(CustRecBuf.Notes)
 FileDefinitionBuffer.RecLen = CustFixedRecSize
 FileDefinitionBuffer.PageSize = 4096
 FileDefinitionBuffer.IndxCnt = 4
 FileDefinitionBuffer.FileFlags = VARIABLELENGTH

   'The following defines key specifications.

   'Key 0, by contact number.
 FileDefinitionBuffer.KeyBuf0.KeyPos = 1
 FileDefinitionBuffer.KeyBuf0.KeyLen = 4
 FileDefinitionBuffer.KeyBuf0.KeyFlags = EXTTYPE + MODIFIABLE
 FileDefinitionBuffer.KeyBuf0.KeyType = Chr$(BAUTOINC)

     'Key 1, by contact name.
 FileDefinitionBuffer.KeyBuf1.KeyPos = 5
 FileDefinitionBuffer.KeyBuf1.KeyLen = 30
 FileDefinitionBuffer.KeyBuf1.KeyFlags = EXTTYPE + MODIFIABLE + DUP
 FileDefinitionBuffer.KeyBuf1.KeyType = Chr$(BSTRING)

    'Key 2, by contact name.
 FileDefinitionBuffer.KeyBuf2.KeyPos = 35
 FileDefinitionBuffer.KeyBuf2.KeyLen = 60
 FileDefinitionBuffer.KeyBuf2.KeyFlags = EXTTYPE + MODIFIABLE + DUP
 FileDefinitionBuffer.KeyBuf2.KeyType = Chr$(BSTRING)

   'Key 3, by sales representative, by next contact date.

   'This is a segmented key:
 FileDefinitionBuffer.KeyBuf3.KeyPos = 220
 FileDefinitionBuffer.KeyBuf3.KeyLen = 3
 FileDefinitionBuffer.KeyBuf3.KeyFlags = EXTTYPE + _ MODIFIABLE + DUP + SEGMENT
 FileDefinitionBuffer.KeyBuf3.KeyType = Chr$(BSTRING)
 FileDefinitionBuffer.KeyBuf4.KeyPos = 223
 FileDefinitionBuffer.KeyBuf4.KeyLen = 4
 FileDefinitionBuffer.KeyBuf4.KeyFlags = EXTTYPE + MODIFIABLE + DUP
 FileDefinitionBuffer.KeyBuf4.KeyType = Chr$(BDATE)

 BufLen = Len(FileDefinitionBuffer)
 CustFileLocation = CustFileLocation & " "
 KeyBufLen = Len(CustFileLocation)
 giStatus = BTRCALL(BCREATE, CustPosBlock, FileDefinitionBuffer, _
  BufLen, ByVal CustFileLocation, KeyBufLen, 0)
End Sub
```

Delphi (Segmented Indexes)

To see code on creating segmented indexes, see "Key 3" in the Delphi (Creating a File) code sample.

```
var
  CustKeyBuffer : record                    //Segmented key buffer
```

```
    SalesRep     : array[0..2] of Char;
    NextContact  : DateType;              //Btrieve Date structure
  end;

  CustBufLen: Word;
  CustKeyNumber: ShortInt;
begin
  CustKeyNumber := 3;                       //In order by SalesRep/Date
  CustKeyBuffer.SalesRep := 'TO';           //Look for person with initials TO
  CustKeyBuffer.NextContact.Day := 9;       //and NextContact of 9/9/98
  CustKeyBuffer.NextContact.Month := 9;
  CustKeyBuffer.NextContact.Year := 1998;

  CustBufLen := SizeOf (CustRec);

  {The following syntax gets the first record from the file using the specified}
  {sort order (KeyNum) :}
  Status := BTRV(B_GET_EQUAL,      //OpCode 5
                 CustPosBLock,     //Already opened position block
                 CustRec,          //Buffer for record to be returned in
                 CustBufLen,       //Maximum length to be returned
                 CustKeyBuffer,    //Returns the key value extracted from the record
                 CustKeyNumber);   //Index order to use for retrieval
```

C/C++ (Segmented Indexes)

```
struct   //Segmented key buffer
  {
    char      SalesRep[3];
    date_type NextContact;   //Btrieve Date structure
  } CustKeyBuffer;
  BTI_WORD CustBufLen;
  char     CustKeyNumber;
  CustKeyNumber = 3;                      //In order by SalesRep/Date
  CustKeyBuffer.SalesRep = "TO";          //Look for person with initials TO
  CustKeyBuffer.NextContact.Day = 9;      // and NextContact of 9/9/98
  CustKeyBuffer.NextContact.Month = 9;
  CustKeyBuffer.NextContact.Year = 1998;
  CustBufLen = sizeof(CustRec);
        /* Get First record from file using specified sort order (KeyNum): */
  iStatus = BTRV(B_GET_EQUAL,        //OpCode 5
                 CustPosBLock,       //Already opened position block
                 &CustRec,           //Buffer for record to be returned in
                 &CustBufLen,        //Maximum length to be returned
                 CustKeyBuffer,      //Specifies the record to look for
                 CustKeyNumber);     //Index order to use for retrieval
```

Declarations of Btrieve API Functions for Visual Basic

The following are declarations of the Btrieve API functions for Visual Basic.

```
Declare Function BTRCALL Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, Db As Any, DL As Long, Kb As Any,
ByVal Kl, ByVal Kn) As Integer

Declare Function BTRCALLID Lib "w3btrv7.dll" (ByVal OP, ByVal Pb$, Db As Any, DL As Long, Kb As Any,
ByVal Kl, ByVal Kn, ByVal ID) As Integer

Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" (hpvDest As Any, hpvSource As Any, ByVal
cbCopy As Long)
```

Creating a Database

A Zen database consists of two basic parts:

- Data files to store the data physically
- A data dictionary to describe the data

The following topics explain named databases, bound databases, and how to create a database and use a data dictionary to manage data files as tables, columns, and indexes:

- Named Databases
- Bound Databases
- Creating Database Components
- Naming Conventions,
- Creating a Data Dictionary
- Creating Tables
- Creating Columns
- Creating Indexes

Named Databases

A named database has a logical name that allows users to identify it without knowing its actual location. When you name a database, you associate that name with a particular dictionary directory path and one or more data file paths. When you log in to Zen using a database name, Zen uses the name to find the database dictionary and data files. Your database must be named before you can do the following:

- Define triggers
- Define primary and foreign keys
- Bind a database
- Suspend database integrity constraints

You use Zen Control Center to name existing, unbound databases and to create new, bound databases. See *Zen User's Guide* for more information.

Bound Databases

Binding a database ensures that the MicroKernel enforces database security, referential integrity (RI), and triggers, regardless of the method used to access the data. The MicroKernel enforces these integrity controls as follows:

- When you define security on a *bound* database, Btrieve users cannot access it.
- When you define security on an *unbound* database, Btrieve users can access it.
- When no security is defined on a *bound* database, Btrieve users can access the data files as shown in the following table.

Bound File Constraint	Level of Access Using Btrieve
RI constraints defined	Users can access and do anything within RI constraints
INSERT, UPDATE, or DELETE triggers defined	Corresponding Btrieve operations will cause triggers to execute.

Note: Even if you do not bind your database, Zen automatically stamps a data file as bound if it has a trigger, has a foreign key, or has a primary key that is referenced by a foreign key. Thus, a data file may be part of an unbound database, but be bound. In such cases, the MicroKernel enforces integrity constraints on the file as if it were part of a bound database.

The dictionary and data files in a bound database cannot be referenced by other named databases. Also, bound data files cannot be referenced by other tables in the database.

When you create a bound database or bind an existing database, Zen stamps every dictionary and data file with the name of the bound database. Also, Zen stamps every data file with the name of the table associated with that data file. In addition, when you add new tables or dictionary files to the database, Zen automatically binds them.

Creating Database Components

Use Zen Control Center to create databases. See *Zen User's Guide*.

To create tables in the database, use Zen Control Center or the CREATE TABLE syntax defined in *SQL Engine Reference*. When you issue a CREATE TABLE statement, you must include commands that define columns. In addition, you can include commands that define referential integrity (RI) constraints.

Naming Conventions

When you create a database, Zen allows you to assign a descriptive name to each database component. Users and applications refer to the components of the database using these names. This topic outlines the conventions to which you must adhere when naming database components.

For more information, see Identifier Restrictions in *Advanced Operations Guide*.

Unique Names

The following database components must have unique names within a dictionary:

- Tables
- Views
- Indexes
- Keys
- User names
- Group names
- Stored procedures
- Triggers
- Column names within a single table

Names for parameters and variables must be unique within a SQL statement. Because Zen keywords are reserved words, you cannot use them for naming database components or in parameter names and variables. See SQL Reserved Words in *SQL Engine Reference* for a list of reserved keywords.

When a column name is repeated in different tables, you can *qualify* it in each table by preceding it with the relevant table name or alias name. For example, you can refer to the ID column in the Student table as Student.ID. This is a *fully qualified* column name, and the table name (Student) is the *column qualifier*.

Valid Characters

Following are the valid characters for the names of database components at the SQL level, and for variables and parameter names:

- a through z

- A through Z
- 0 through 9
- _ (underscore)
- ^ (caret)
- ~ (tilde)
- $ (dollar sign)

Note: The name of a database component must begin with a letter. If you specify the name of a database component or a parameter name that does not follow these conventions, specify the name in double quotes (such as "name").

Maximum Name Lengths

Zen restricts the maximum length of database component names in a dictionary. See Identifier Restrictions in *Advanced Operations Guide* and Relational Engine Limits in *SQL Engine Reference*.

Case Sensitivity

Zen is case-sensitive when you are defining database component names. If you create a table named TaBLe1, Zen stores the table name in the dictionary as TaBLe1. With the exception of user names, user group names, and passwords, Zen is case-insensitive after you define the component name. After defining the table TaBLe1, you can refer to it as table1.

User names, user group names, and passwords are case-sensitive in Zen. For example, when you log in as the master user, you must specify the user name as Master.

When retrieving data, Zen displays names for tables, views, aliases, and columns based on the case in which they were created.

```
SELECT *
FROM Course#
```

Zen returns the column names as follows:

```
"Name", "Description", "Credit_Hours", "Dept_Name"
```

Creating a Data Dictionary

Zen uses the dictionary to store information about the database. The dictionary consists of several system tables that describe the tables and views of your database.

The system tables contain several types of database information, including table and index definitions, column characteristics, and integrity and security information. The following table describes the system tables Zen creates. See also System Tables in *SQL Engine Reference*.

Operation	Resulting Table
Create a data dictionary	X$File, X$Field, X$Index
Specify column attributes	X$Attrib
Create stored SQL procedures	X$Proc
Define database security	X$User, X$Rights
Define referential constraints	X$Relate
Define views	X$View
Define triggers	X$Trigger, X$Depend

Because the system tables are part of the database, you can query them to determine their contents. If you have the appropriate rights, you can also create system tables or change their contents.

Note: Zen does not display all of the data in the system tables. For example, information about stored views and procedures, other than their names, is available only to Zen. In addition, some data (such as user passwords) is displayed as encrypted characters.

For a complete reference to the contents of each system table, see *SQL Engine Reference*.

Once you have created a dictionary, you can add tables, columns, and indexes to your database.

To create a named database

Note: You must have named databases in order to use some features, such as referential integrity and triggers.

1. Create a directory in which to store the new dictionary tables.
2. Use Zen Control Center to add a Named Database. See To create a new database in *Zen User's Guide*.

To create a dictionary for an unnamed database

1. Run DDF Builder.
2. Follow the steps in *DDF Builder User's Guide* for creating data dictionary files (DDFs). See Getting Started with DDF Builder.

Creating Tables

When you create a table, you must name it. Each table name must be unique within a database. For more information about rules for naming tables, see Naming Conventions.

When you are deciding which tables to create in your database, consider that different users can look at data in different combinations using *views*. A view looks like a table and can be treated as a table for most purposes (such as retrieving, updating, and deleting data). However, a view is not necessarily associated with a single table; it can combine information from multiple tables. For more information, see Retrieving Data.

You can create a table using ZenCC. See To start Table Editor for a new table in *Zen User's Guide*.

Aliases

You can assign aliases (also called *alias names*) to table names in the following elements of statements:

- FROM clause of a SELECT or DELETE statement.
- INTO clause of an INSERT statement.
- List of tables in an UPDATE statement.

Note: Aliases apply only to the statement in which you use them. Zen does not store them in the data dictionary.

An alias can be any combination of up to 20 characters. Always separate the table name from the alias with a blank. Separate the alias and the column name with a period (.). Once you specify an alias for a particular table, you can use it elsewhere in the statement to qualify column names from that table.

The following example specifies the alias name *s* for the table Student and *e* for the table Enrolls.

```
SELECT s.ID, e.Grade
FROM Student s, Enrolls e
WHERE s.ID = e.Student_ID#
```

You can use an alias to do the following:

- Replace long table names.

  When you are working interactively, using aliases can save typing time, especially when you need to qualify column names. For example, the following statement assigns *s* as the alias for

the Student table, *e* for the Enrolls table, and *c1* for the Class table. This example uses aliases to distinguish the source of each column in the selection list and in the WHERE conditions.

```
SELECT s.ID, e.Grade, c1.ID
   FROM Student s, Enrolls e, Class c1
   WHERE (s.ID = e.Student_ID) AND
         e.Class_ID = c1.ID)#
```

- Make a statement more readable. Even in statements with only a single table name, an alias can make the statement easier to read.
- Use the table in the outer query in a correlated subquery:

```
SELECT s.ID, e.Grade, c1.ID
   FROM Student s, Enrolls e, Class c1
   WHERE (s.ID = e.Student_ID) AND
      (e.Class_ID = c1.ID) AND
      e.Grade >=
         (SELECT MAX (e2.Grade)
         FROM Enrolls e2
         WHERE e2.Class_ID = e.Class_ID)#
```

Creating Columns

You create columns when you create a table using a CREATE TABLE statement, or you can add columns to an existing table using an ALTER TABLE statement. In either case, you must specify the following characteristics:

- Column name – identifies the column. Each column name must be unique within a table and the column name cannot exceed 20 characters. Zen is case-sensitive when defining database column names, but case-insensitive after you define the column name. For example, if you create a column named *ColuMN1*, the name is stored in the dictionary as *ColuMN1*; subsequently, you can refer to it as *column1*. For more information about rules for naming columns, see Naming Conventions.
- Data type – identifies the kind of data to expect, such as a string of characters or a number, and how much disk storage space to allocate.

For more information about data types, see the *Btrieve API Guide*.

Creating Indexes

Indexes optimize operations that either search for or order by specific values. Define indexes for any columns on which you frequently perform either of these operations. Indexes provide a fast retrieval method for a specific row or group of rows in query optimization. Zen also uses indexes with referential integrity (RI). Indexes improve performance on joins and help to optimize queries. For more information about RI, see *Zen User's Guide*.

In Zen databases, the MicroKernel Engine creates and maintains indexes as part of the physical file for which they are defined. The MicroKernel Engine performs all maintenance for insert, update, or delete operations. These activities are transparent to any Zen application.

To create an index, use a CREATE INDEX statement. This method creates a named index. You can delete named indexes after you create them. For more information about dropping indexes, see Inserting and Deleting Data.

While indexes allow you to sort rows and retrieve individual rows quickly, they increase the disk storage requirements for a database and decrease performance somewhat on insert, update, and delete operations. You should consider these trade-offs when defining indexes.

The next example uses a CREATE INDEX statement to add an index to a table that already exists:

```
CREATE INDEX DeptHours ON Course
(Dept_Name, Credit_Hours)#
```

Note: Be aware that if you use the CREATE INDEX statement on files that contain a lot of data, execution could take some time to complete, and other users may not be able to access data in that file in the meantime.

For detailed information about the CREATE TABLE and CREATE INDEX statements, see *SQL Engine Reference*.

Index Segments

You can create an index on any single column or group of columns in the same table. An index that includes more than one column is called a *segmented index*, in which each column is called an *index segment*.

For example, the Person table in the Demodata sample database has the following three indexes:

- A segmented index consisting of the Last Name column and the First Name column
- The Perm_State + Perm_City column
- The ID column

The number of *index segments* is affected by the page size of the data file. See *Btrieve API Guide* for more information on how to use the PAGESIZE keyword. The maximum number of indexes you can create for a table depends on the page size of its data file and the number of segments in each index. As the following table shows, data files with page sizes smaller than 4096 bytes cannot contain as many index segments as a data file with a page size of 4096. The number of index segments that you may use depends on the page size of the file.

Page Size (bytes)	Maximum Key Segments by File Version			
	8.x and earlier	9.0	9.5	13.0, 16.0
512	8	8	Rounded up[2]	Rounded up[2]
1024	23	23	97	Rounded up[2]
1536	24	24	Rounded up[2]	Rounded up[2]
2048	54	54	97	Rounded up[2]
2560	54	54	Rounded up[2]	Rounded up[2]
3072	54	54	Rounded up[2]	Rounded up[2]
3584	54	54	Rounded up[2]	Rounded up[2]
4096	119	119	204[3]	183[3]
8192	n/a[1]	119	420[3]	378[3]
16384	n/a[1]	n/a[1]	420[3]	378[3]

[1]"n/a" stands for "not applicable"

[2]"Rounded up" means that the page size is rounded up to the next size supported by the file version. For example, 512 is rounded up to 1024, 2560 is rounded up to 4096, and so forth.

[3]A 9.5 format or later file can have more than 119 segments, but the number of indexes is limited to 119.

See also the status codes 26: The number of keys specified is invalid and 29: The key length is invalid, both in *Status Codes and Messages*.

Using the page size and fixed record length, you can calculate the efficiency with which data is being stored (such as the number of wasted bytes per page). By having fewer records per page, you can improve concurrency where page-level locking is concerned.

By default, Zen creates all tables with a page size of 4096 bytes. However, you can specify a different page size using the PAGESIZE keyword in a CREATE TABLE statement, or you can

create a table using the MicroKernel Database Engine and specify a different page size for that table.

When calculating the total number of index segments defined for a table, a nonsegmented index counts as one index segment. For example, if your table has three indexes defined, one of which has two segments, the total number of index segments is four.

You can use Zen Control Center to display the number of defined index segments and the page size of a data file. For information about this utility, see *Zen User's Guide*.

Index Attributes

When you create an index, you can assign to it a set of qualities, or *attributes*. Index attributes determine the modifiability of the index and how Zen sorts the indexes you define for a table. You can include parameters specifying index attributes anytime you create or alter an index definition.

Indexes can have the following attributes:

Case-sensitivity	Determines how Zen evaluates uppercase and lowercase letters during sorting. By default, Zen creates case-sensitive indexes. To create a case-insensitive index, specify the CASE keyword when you create the index.
Sort order	Determines how Zen sorts index column values. By default, Zen sorts index column values in ascending order (from smallest to largest). To create an index that sorts in descending order, specify the DESC keyword when you create the index.
Uniqueness	Determines whether Zen allows multiple rows to have the same index column value. By default, Zen creates non-unique indexes. To create an index that requires unique values, specify the UNIQUE keyword when you create the index.
Modifiability	Determines whether you can modify index column values after Zen stores the corresponding row. By default, Zen does not allow changes to index column values once Zen stores the row. To create a modifiable index, specify the MOD keyword when you create the index.

Segmentation	Indicates whether the index is segmented (whether it consists of a group of columns combined into a single index). By default, Zen creates indexes that are not segmented. To create a segmented index using the CREATE TABLE statement, specify the SEG keyword for each index segment you create, except the last segment in the index. (The SEG keyword indicates that the next column specified is a segment of the index you are creating.) Because you can create only one index at a time with the CREATE INDEX command, you do not need to use the SEG keyword to specify a segmented index. If you specify more than one column, Zen creates a segmented index using the columns in the order in which you specify them.
Partial	Indicates whether Zen uses a portion of a CHAR or VARCHAR column, designated as t he only or last index column, when the total size of the column(s) plus overhead is equal to or greater than 255 bytes. By default, Zen does not create partial indexes. To create a partial index using the CREATE INDEX statement, specify the PARTIAL keyword.

Uniqueness and modifiability apply only to entire indexes. You cannot apply uniqueness or modifiability to a single index segment without applying it to the entire index. For example, if you create a segmented index and specify the MOD keyword for one of the index segments, you must specify the MOD keyword for every segment.

In contrast, you can apply case-sensitivity, sort order, and segmentation to individual index segments without affecting the entire index. For example, you can create a case-insensitive index segment in an otherwise case-sensitive index.

Partial Indexes apply only to the last column defined in the index, as long as that column meets the following criteria:

- The column is the only column defined in the index, or it is the last column defined in the index.
- The last index column is of data type CHAR or VARCHAR.
- The total size of the index, including column overhead, is equal to or greater than 255 bytes.

For more information on indexes and their attributes, see CREATE INDEX in *SQL Engine Reference*.

Relational Database Design

This section includes the following topics:

Overview of Database Design

This section introduces the fundamental principles of relational database design. A thorough database design supports an effective development process and is critical to successful database functionality and performance.

The Demodata sample database is provided with installation and is frequently used in the documentation to illustrate database concepts and techniques.

Stages of Design

Once you understand the basic structure of a relational database, you can begin the database design process. Designing a database is a process that involves developing and refining a database structure based on the requirements of your business.

Database design includes the following three stages:

1. Conceptual Database Design
2. Logical Database Design
3. Physical Database Design

Conceptual Design

The first step in the database design cycle is to define the data requirements for your business. Answering these types of questions helps you define the conceptual design:

- What types of information does my business currently use?
- What types of information does my business need?
- What kind of information do I want from this system?
- What are the assumptions on which my business runs?
- What are the restrictions of my business?
- What kind of reports do I need to generate?
- What will I do with this information?
- What kind of security does this system require?
- What kinds of information are likely to expand?

Identifying the goals of your business and gathering information from the different sources who will use the database is an essential process. With this information you can effectively define your tables and columns.

Logical Design

Logical database design helps you further define and assess your business' information requirements. Logical database design involves describing the information you need to track and the relationships among those pieces of information.

Once you create a logical design, you can verify with the users of the database that the design is complete and accurate. They can determine if the design contains all of the information that must be tracked and that it reflects the relationships necessary to comply with the rules of your business.

Creating a logical database design includes the following steps:

1. Define the tables you need based on the information your business requires (as determined in the conceptual design).
2. Determine the relationships between the tables. (See the section Table Relationships for more information.)
3. Determine the contents (columns) of each table.
4. Normalize the tables to at least the third normal form. (See the section Normalization for more information.)
5. Determine the primary keys. (See the section Keys for more information.)
6. Determine the values for each column.

Table Relationships

In a relational database, tables relate to one another by sharing a common column. This column, existing in two or more tables, allows you to join the tables. There are three types of table relationships: one-to-one, one-to-many, and many-to-many.

A *one-to-one relationship* exists when each row in one table has only one related row in a second table. For example, a university may decide to assign one faculty member to one room. Thus, one room can only have one instructor assigned to it at a given time. The university may also decide that a department can only have one Dean. Thus, only one individual can be the head of a department.

A *one-to-many relationship* exists when each row in one table has many related rows in another table. For example, one instructor can teach many classes.

A *many-to-many relationship* exists when a row in one table has many related rows in a second table. Likewise, those related rows have many rows in the first table. A student can enroll in many courses, and courses can contain many students.

Normalization

Normalization is a process that reduces redundancy and increases stability in your database. Normalization involves determining in which table a particular piece of data belongs and its relationship to other data. Your database design results in a data-driven, rather than process or application-driven, design which provides a more stable database implementation.

When you normalize your database, you eliminate the following columns:

- Columns that contain more than one non-atomic value.
- Columns that duplicate or repeat.
- Columns that do not describe the table.
- Columns that contain redundant data.
- Columns that can be derived from other columns.

First Normal Form

Columns in the first normal form have the following characteristics:

- They contain only one atomic value.
- They do not repeat.

The first rule of normalization is that you must remove duplicate columns or columns that contain more than one value to a new table.

Tables normalized to the first normal form have several advantages. For example, in the Billing table of the sample database, first normal form does the following:

- Allows you to create any number of transactions for each student without having to add new columns.
- Allows you to query and sort data for transactions quickly because you search only one column (transaction number).
- Uses disk space more efficiently because no empty columns are stored.

Second Normal Form

A table is in the second normal form when it is in the first normal form and only contains columns that provide information about the key of the table.

In order to enforce the second rule of normalization, you must move those columns that do not depend on the primary key of the current table to a new table.

A table violates second normal form if it contains redundant data. This may result in inconsistent data which causes your database to lack integrity. For example, if a student changes her address, you must then update all existing rows to reflect the new address. Any rows with the old address result in inconsistent data.

To resolve these differences, identify data that remains the same when you add a transaction. Columns like Student Name or Street do not pertain to the transaction and do not depend on the primary key, Student ID. Therefore, store this information in the Student table, not in the transaction table.

Tables normalized to the second normal form also have several advantages. For example, in the Billing table of the sample database, second normal form allows you to do the following:

- Update student information in just one row.
- Delete student transactions without eliminating necessary student information.
- Use disk space more efficiently since no repeating or redundant data is stored.

Third Normal Form

A table is in the third normal form when it contains only independent columns.

The third rule of normalization is that you must remove columns that can be derived from existing columns. For example, for a student, you do not have to include an Age column if you already have a Date of Birth column, because you can calculate age from a date of birth.

A table that is in third normal form contains only the necessary columns, so it uses disk space more efficiently since no unnecessary data is stored.

In summary, the rules for the first, second, and third normal forms state that each column value must be a fact about the primary key in its entirety, and nothing else.

Keys

An ODBC key is a column or group of columns on which a table's referential integrity (RI) constraints are defined. In other words, a key or combination of keys acts as an identifier for the data in a row.

For more information about referential integrity and keys, see *Advanced Operations Guide*.

Physical Design

The physical database design is a refinement of the logical design; it maps the logical design to a relational database management system. In this phase, you examine how the user accesses the

database. This step of the database design cycle involves determining the following types of information:

- Data you will commonly use.
- Columns requiring indexes for data access.
- Areas needing flexibility or room for growth.
- Whether denormalizing the database will improve performance. (To denormalize your database, you reintroduce redundancy to meet performance.) For more information, see Normalization.

Inserting and Deleting Data

This chapter includes the following sections:

Overview of Inserting and Deleting Data

After creating a data dictionary, tables, and columns, you can add data to the database using SQL Data Manager. SQL statements allow you to do the following:

- Specify literal values to insert.
- Select data from other tables and insert the resulting values into entire rows or specified columns.

When you insert a literal value, it must conform to the specified column's data type and length.

You can drop (delete) rows, indexes, columns, or tables from your database. In addition, you can drop an entire database when you no longer need it.

Inserting Values

You can use a VALUES clause in an INSERT statement to specify literal values to insert into a table. The following example inserts a new row into the Course table of the sample database:

```
INSERT INTO Course
VALUES ('ART 103', 'Principles of Color', 3, 'Art');
```

In this example, listing the columns Name, Description, Credit_Hours, and Dept_Name is optional because the statement inserts a value for each column in the table, in order. However, a column list is required if the statement inserted data only into selected columns instead of the entire row, or if the statement inserted data into the columns in a different order than is defined in the table.

For complete information on the INSERT statement, see INSERT in *SQL Engine Reference*.

Transaction Processing

When you attempt to insert data into a table, Zen returns an error if the data is invalid. Any data inserted before the error occurred is rolled back. This enables your database to remain in a consistent state.

You can use transaction processing in a Zen database to group a set of logically related statements together. Within a transaction, you can use savepoints to effectively nest transactions; if a statement in a nesting level fails, then the set of statements in that nesting level is rolled back to the savepoint. See the following topics in *SQL Engine Reference* for information about transaction processing and savepoints:

- START TRANSACTION
- COMMIT
- ROLLBACK
- SAVEPOINT
- RELEASE SAVEPOINT

Deleting Data

There are two types of DELETE statements: positioned and searched.

You can use a DELETE statement to delete one or more rows from a table or an updatable view. To specify specific rows for Zen to delete, use a WHERE clause in a DELETE statement.

```
DELETE FROM Class
WHERE ID = 005#
```

The positioned DELETE statement deletes the current row of a view associated with an open SQL cursor.

```
DELETE WHERE CURRENT OF mycursor;
```

For more information, see DELETE and DELETE (positioned) in *SQL Engine Reference*.

Dropping Indexes

If you find that you no longer need a named index, use a DROP INDEX statement to drop it.

```
DROP INDEX DeptHours#
```

For more information, see DROP INDEX in *SQL Engine Reference*.

Dropping Columns

To drop a column from a table, use an ALTER TABLE statement.

```
ALTER TABLE Faculty
DROP Rsch_Grant_Amount#
```

This example drops the Rsch_Grant_Amount column from the Faculty table and deletes the column definition from the data dictionary.

For more information, see ALTER TABLE in *SQL Engine Reference*.

Note: Be aware that if you use the ALTER TABLE statement on files that contain a lot of data, execution could take some time to complete, and other users may not be able to access data in that file in the meantime.

Dropping Tables

To drop a table from the database, use a DROP TABLE statement.

```
DROP TABLE Student#
```

This example drops the InactiveStudents table definition from the data dictionary and deletes its corresponding data file (INACT.MKD).

For more information, see DROP TABLE in *SQL Engine Reference*.

Note: You cannot drop system tables. See *SQL Engine Reference* for a complete listing of system tables.

Dropping an Entire Database

When you no longer need a particular database, you can delete it using the SQL Data Manager in the Zen Control Center. See the *Zen User's Guide* for more information.

Modifying Data

The following topics cover data modifications:

- Overview of Modifying Data
- Modifying Tables
- Setting Default Values
- Using UPDATE

Overview of Modifying Data

After creating a database, you can modify it as follows:

- After creating tables, you can modify the table definitions.
- After creating columns, you can set optional column attributes.
- After adding data to the database, you can modify the data.

You can perform these tasks using the SQL Data Manager. For information about interactive applications, see *Zen User's Guide*. For more information about SQL statements, see *SQL Engine Reference*.

Modifying Tables

You can use an ALTER TABLE statement to modify a table definition after creating the table. ALTER TABLE statements allow you to add or drop columns; add or drop primary and foreign keys; and change the pathname of a table's data file.

The following example adds a numeric column called Emergency_Phone to the Tuition table in the sample database.

```
ALTER TABLE Tuition ADD Emergency_Phone NUMERIC(10,0)#
```

For more information about columns, see Inserting and Deleting Data. For more information about primary and foreign keys, see Managing Data

Setting Default Values

Zen inserts a default value if you insert a row but do not provide a value for that column. Default values ensure that each row contains a valid value for the column.

In the Person table of the sample database, all students live in a state. Setting a default value such as TX for the State column ensures that the most probable value is always entered for that column.

To set a default value for a column, use a DEFAULT statement in the CREATE TABLE statement:

```
CREATE TABLE MyTable(c1 CHAR(3) DEFAULT 'TX', ID INTEGER)#
SELECT * FROM MyTable#
```

Result of SELECT Statement:

```
"c1", "ID"
"TX", "1234"
1 row fetched from 2 columns
```

Using UPDATE

You can use an UPDATE statement to change the data in a row that is already in a table. UPDATE statements let you modify specific columns in a row. Also, you can use a WHERE clause in an UPDATE statement to specify which rows for Zen to change. This is referred to as a searched update. Using SQL declared cursors and the Positioned UPDATE statement, you can update the current row of a declared cursor from which you are fetching data.

```
UPDATE Course
SET Credit_Hours = 4
WHERE Course.Name = 'Math'#
```

This example instructs Zen to find the row that contains the course name Math and change the Credit Hours column value to 4.

As shown in the previous example, you can use a constant to update a column by placing it on the right hand side of a SET clause in an UPDATE statement.

Retrieving Data

The following topics discuss the use of SELECT statements for data retrieval:

- Overview of Retrieving Data
- Views
- Selection Lists
- Sorted and Grouped Rows
- Joins
- Subqueries
- Restriction Clauses
- Functions

Overview of Retrieving Data

Once your database contains data, you can retrieve and view that data using a SELECT statement. Zen returns the data you request in a *result table*. Using SQL statements, you can do the following:

- Create temporary views or permanent (*stored)* views.
- Specify a *selection list* that lists the columns to retrieve from one or more tables in your database.
- Specify how to sort the rows.
- Specify criteria by which to group the rows into subsets.
- Assign a temporary name (alias) to a table.
- Retrieve data from one or more tables and present the data in a single result table (a *join*).
- Specify a *subquery* within a SELECT statement.
- Specify a *restriction clause* to restrict the rows Zen selects.

Views

A view is the mechanism for examining the data in your database. A view can combine data from multiple tables or can include only certain columns from a single table. Although a view looks like a table, it consists of a selected set of columns or calculations based on those columns from tables in your database. Thus, a view may contain data from columns in more than one table or data that is not actually in any table at all (for example, SELECT COUNT (*) FROM Person).

Features of Views

Following are some of the features of views:

- You can arrange the columns of a view in any order except that the variable-length column must be last. You can specify only one variable-length column.
- You can use a restriction clause to specify the set of rows that Zen returns in a view. The restriction clause lists criteria that the data must satisfy to be included in the view. For more information, see the section Restriction Clauses.
- You can design and customize views for each user and application that accesses the database. You can store these view definitions within the data dictionary for later recall.
- You can include any number of stored view names in a table list when retrieving, updating, or deleting data unless the view is a read-only view. In a read-only view, you can only retrieve data.
- In a stored view, you must provide headings for the view's computed columns and constants and use those names in a list of column names when you retrieve data from the view.

Temporary and Stored Views

You can use SELECT statements to create temporary views or stored views. You use a *temporary view* only once and then release it. Zen places the definition of a *stored view* in the data dictionary (X$Proc) so you can recall the view later. You use CREATE VIEW statements to create and name stored views.

Each view name must be unique within a database and cannot exceed 20 characters. For more information about rules for naming views, see Inserting and Deleting Data.

Zen is case-sensitive when defining database element names. If you create a stored view named PhoNE, Zen stores the view name in the data dictionary as PhoNE. Zen is case-insensitive after you define the view name. After defining the stored view PhoNE, you can refer to it as phone.

Using stored views provides the following features:

- You can store frequently executed queries and name them for later use. The following example creates a stored view named Phones based on the Department table.

```
CREATE VIEW Phones (PName, PPhone)
AS SELECT Name, Phone_Number
FROM Department#
```

- You can specify the name of the stored view in table lists when retrieving, updating, and deleting data. The stored view behaves as though it is a table in the database, but it is actually reconstructed internally by the Zen engine each time it is used. The following example updates the phone number for the History Department in the Department table by referring to the stored view Phones.

```
UPDATE Phones
SET PPhone = '5125552426'
WHERE PName = 'History'#
```

- You can specify *headings*. A heading specifies a column name that is different from the name you defined for the column in the dictionary. The following example specifies the headings Department and Telephone for the stored view Phones.

```
CREATE VIEW Dept_Phones (Department, Telephone)
AS SELECT Name, Phone_Number
FROM Department#
```

  You can use the headings in subsequent queries on the view, as in the following example.

```
SELECT Telephone
FROM Dept_Phones#
```

  If the selection list contains simple column names and you do not provide headings, Zen uses the column name as the column heading.

  You must use headings to name constants and computed columns that you include in the view. The following example creates the headings Student and Total.

```
CREATE VIEW Accounts (Student, Total)
AS SELECT Student_ID, SUM (Amount_Paid)
FROM Billing
GROUP BY Student_ID#
```

  You must also use headings if you specify SELECT * from multiple tables that have any duplicate column names.

- You can create customized views for each user or application that accesses the database. You can store these view definitions within the data dictionary for later recall.

Read-Only Tables in Views

You *cannot* insert, update, or delete rows from views that contain read-only tables. (Here the term *update* refers to insert, update, and delete; if a table is read-only, you cannot update it.) Some tables are read-only whether or not they are in views that are specified as such; such tables are

intrinsically read-only, and you cannot update them. A table is read-only if it meets one of the following criteria:

- The database has security enabled, and the current user or user group has only SELECT rights defined for the database or the table.
- The data files have been flagged read-only at the physical file level (for example, using the ATTRIB command under DOS or Windows or the **chmod** command in Linux or Raspbian).
- You execute a SELECT clause that creates a view and contains any of the following items:
  - An aggregate function in the selection list.
  - A GROUP BY or HAVING clause.
  - A UNION.
  - The DISTINCT keyword.
- You execute a SELECT statement that creates a view, and the table contains any of the following characteristics:
  - It appears in a nonmergeable view that is in the SELECT statement's FROM clause.
  - It is a system table. System tables are always opened as read-only in a view, even if this overrides the view's open mode.
  - A column from the table appears in a computed column or a scalar function in the selection list.
  - The table appears in the FROM clause of a subquery that is not correlated to the outermost query. A subquery can be directly or indirectly correlated to the outermost query. A subquery is directly correlated with the outermost query if it contains a reference to a column from a table and its specific occurrence in the outermost query's FROM clause. A subquery is indirectly correlated to the outermost query if it is correlated to a subquery that is in turn directly or indirectly correlated to the outermost query.
  - The open mode is read-only.
  - You execute a Positioned UPDATE statement with any of the following keywords, without specifying FOR UPDATE:

    ORDER BY

    SCROLL

Mergeable Views

A view is mergeable if you can rewrite the SELECT statement using only base tables and columns.

For example, if you want to know how many students are in a class, you can define a view to calculate that. The view NumberPerClass is defined as follows:

```
CREATE VIEW NumberPerClass (Class_Name, Number_of_Students)
AS SELECT Name, COUNT(Last_Name)
FROM Person, Class, Enrolls
WHERE Person.ID = Enrolls.Student_ID
AND Class.ID = Enrolls.Class_ID

GROUP BY Name#
```

The view NumberPerClass is defined as follows:

```
SELECT *
FROM NumberPerClass#
```

The view NumberPerClass is then mergeable because we can rewrite the SELECT statement as follows:

```
SELECT Name, COUNT(Last_Name)
FROM Person, Class, Enrolls
WHERE Person.ID = Enrolls.Student_ID
AND Class.ID = Enrolls.Class_ID

GROUP BY NAME#
```

The view NumberPerClass is nonmergeable if you want to write a SELECT statement as follows:

```
SELECT COUNT(Name)
FROM NumberPerClass
WHERE Number_of_Students > 50#
```

This statement is invalid for the view NumberPerClass. You cannot rewrite it using only base tables and base columns.

A view is mergeable if it does *not* contain any of the following characteristics:

- It refers to a nonmergeable view.
- It has an aggregate function in its selection list or a DISTINCT keyword, and it appears in the FROM clause of a SELECT statement that has an aggregate in its selection list.
- It has a DISTINCT keyword and appears in the FROM clause of a SELECT statement that has more than one item in its FROM clause, does not have an aggregate in its selection list, and does not have a DISTINCT keyword.
- It has an aggregate in its selection list and appears in the FROM clause of a SELECT statement with either more than one item in its FROM clause or a WHERE clause restriction.

Selection Lists

When you use a SELECT statement to retrieve data, you specify a list of columns (a selection list) to include in the result table. To retrieve all the columns in a table or tables, you can use an asterisk (*) instead of a list of columns.

Note: Avoid using * in place of the list. Using * can expose an application to potential problems if the number of columns or column sizes in a table changes. Also, it typically returns unnecessary data.

The following example selects three columns from the Class table.

```
SELECT Name, Section, Max_Size
FROM Class;
```

The following example selects all columns from the Class table.

```
SELECT * FROM Class;
```

When retrieving data, Zen displays column names based on how you specify the names in the query.

- If you explicitly specify a column name, Zen returns it as you entered it. The following example specifies column names in all lowercase.

  ```
  SELECT name, section, max_size FROM Class#
  ```

  Zen returns the column names as follows:

  ```
  "Name", "Section", "Max_Size"
  ```

  These column names are headings for the returned data; they are not data themselves.

  The following example defines aliases for the tables Department and Faculty.

  ```
  SELECT d.Name, f.ID FROM Department d, Faculty f;
  ```

  Zen returns the column names as follows:

  ```
  "Name", "ID"
  ```

- If you use * to specify column names, they appear in all uppercase, as in the following example.

  ```
  SELECT * FROM Department;
  ```

  Zen returns the column names as follows:

  ```
  "Name", "Phone_Number", "Building_Name", "Room_Number", "Head_Of_Dept"
  ```

  The following example defines aliases for the tables Department and Faculty.

  ```
  SELECT * FROM Department d, Faculty f;
  ```

Zen returns the column names as follows:

```
"Name"
"Phone_Number"
"Building_Name"
"Room_Number"
"Head_Of_Dept"
"ID"
"Dept_Name"
"Designation"
"Salary"
"Building_Name"
"Room_Number"
"Rsch_Grant_Amount"
```

Sorted and Grouped Rows

Once you have decided what data to include in your result table, you can specify how to order the data. You can use the ORDER BY clause to sort the data, or you can use a GROUP BY clause to group rows by a certain column. When you group the data, you can also use aggregate functions to summarize data by group. For more information about aggregate functions, see Aggregate Functions.

The following example orders all rows by last name in the Person table of the sample database.

```
SELECT *
FROM Person
ORDER BY Last_Name#
```

The following example groups the results by the Building Name column in the Room table. This example also uses two aggregate functions, COUNT and SUM.

```
SELECT Building_Name, COUNT(Number), SUM(Capacity)
FROM Room
GROUP BY Building_Name;
```

Joins

A join results from a statement that combines columns from two or more tables into a single view. From this view, you can retrieve, insert, update, or delete data, provided it is not read-only.

Note: This section primarily discusses joining tables using SELECT statements. However, you can also create joins with INSERT, UPDATE, and DELETE statements by applying a single statement to more than one table. *SQL Engine Reference* describes these SQL statements and how to optimize joined views.

You can retrieve data from tables by listing each table or view name in a FROM clause. Use a WHERE clause to specify one or more *join conditions*. A join condition compares an expression that references a column value from one table to an expression that references a column value from another table.

When data is properly normalized, most joins associate values based on some specified key value. This allows you to extract data in terms of referential integrity relationships. For example, if you want to know which professor teaches each class, you can create a join based on the Faculty ID, which is a foreign key in the Class table and a primary key in the Person table:

```
SELECT DISTINCT Class.Name, Person.Last_Name
FROM Class, Person, Faculty
WHERE Class.Faculty_ID = Person.ID
   AND Class.Faculty_ID = Faculty.ID;
```

This example joins two tables on the basis of common values in a common column: Faculty ID.

You can also join tables by making numeric comparisons between columns of like data types. For example, you can compare columns using <, >, or =. The following self-join on the Faculty table identifies all faculty members whose salary was higher than each faculty member (this would produce considerably more records than the faculty table contains):

```
SELECT A.ID, A.Salary, B.ID, B.Salary
FROM Faculty A, Faculty B
WHERE B.Salary > A.Salary;
```

Similar comparisons of dates, times, and so forth can produce many useful and meaningful results.

When joining columns, choose columns that are of the same data type when possible. For example, comparing two NUMERIC columns is more efficient than comparing a NUMERIC column with an INTEGER column. If the columns are not of the same data type but are both numeric or strings, Zen scans both the tables and applies the join condition as a restriction to the results.

When you use string type columns in a WHERE clause, one column in the join condition can be a computed string column. This allows you to concatenate two or more strings and use a join condition to compare them to a single string from another table.

The way in which Zen handles a join depends on whether the join condition contains an index column.

- *If the join condition contains a column that is defined as an index*, performance improves. Using the index to sort rows in the corresponding table, Zen selects only rows that meet the restriction clause condition.
- *If the join condition does not contain a column that is defined as an index*, performance is less efficient. Zen reads each row in each table to select rows that meet the restriction clause condition. To enhance performance, you can create an index in one of the tables before executing the join. This is especially helpful if the query is one that you perform often.

Joining Tables with Other Tables

To specify a join using a SELECT statement, use a FROM clause to list the relevant tables and a WHERE clause to specify the join condition and the restriction. The following example also uses aliases to simplify the statement.

```
SELECT Student_ID, Class_ID, Name
FROM Enrolls e, Class cl
WHERE e.Class_ID = cl.ID;
```

The next example joins three tables:

```
SELECT p.ID, Last_Name, Name
FROM Person p, Enrolls e, Class cl
WHERE p.ID = e.Student_ID AND e.Class_ID = cl.ID;
```

The next example retrieves a list of students who received a grade lower than a 3.0 in English.

```
SELECT First_Name, p.Last_Name
FROM Person p, Student s, Enrolls e, Class cl
WHERE s.ID = e.Student_ID
 AND e.Class_ID = cl.ID
 AND s.ID = p.ID
 AND cl.Name = 'ACC 101'
 AND e.Grade < 3.0;
```

In this example, the first three conditions in the WHERE clause specify the join between the four tables. The next two conditions are restriction clauses connected by the Boolean operator AND.

Joining Views with Tables

To join a view with one or more tables, include a view name in the FROM clause. The view you specify can include columns from a single table or from several joined tables.

Types of Joins

Zen supports equal joins, nonequal joins, null joins, Cartesian product joins, self joins, and left, right, and full outer joins.

For more information on the syntax of joins, see the following topics in *SQL Engine Reference*:

- SELECT
- JOIN

Equal Joins

An equal join occurs when you define the two join columns as equal. The following statement defines an equal join.

```
SELECT First_Name, Last_Name, Degree, Residency
FROM Person p, Student s, Tuition t
WHERE p.ID = s.ID AND s.Tuition_ID = t.ID;
```

Nonequal Joins

You can join tables based on a comparison operation. You can use the following operators in nonequal joins:

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

The following WHERE clause illustrates a join that uses a greater than or equal operator.

```
SELECT Name, Section, Max_Size, Capacity, r.Building_Name, Number
FROM Class cl, Room r
WHERE Capacity >= Max_Size;
```

Cartesian Product Joins

A Cartesian product join associates each row in one table with each row in another table. Zen reads every row in one table once for each row in the other table.

On large tables, a Cartesian product join can take a significant amount of time to complete since Zen must read the following number of rows to complete this type of join:

```
(# of rows in one table) * (# of rows in another table)
```

For example, if one table contains 600 rows and the other contains 30, Zen reads 18,000 rows to create the Cartesian product join of the tables.

The following statement produces a Cartesian product join on the Person and Course tables in the sample database:

```
SELECT s.ID, Major, t.ID, Degree, Residency, Cost_Per_Credit
FROM Student s, Tuition t#
```

Self Joins

In a self join, you can specify a table name in the FROM clause more than once. When you specify a self join, you must assign aliases to each instance of the table name so that Zen can distinguish between each occurrence of the table in the join.

The following example lists all the people who have a permanent address in the same state as the person named Jason Knibb. The query returns the ID, first name, last name, current phone number, and e-mail address.

```
SELECT p2.ID, p2.First_Name, p2.Last_Name, p2.Phone, p2.EMail_Address
FROM Person p1, Person p2
WHERE p1.First_Name = 'Jason' AND p1.Last_Name = 'Knibb' and p1.Perm_State = p2.Perm_State
```

Left, Right, Full Outer Joins

Information about outer joins can be found in *SQL Engine Reference*. See SELECT and JOIN.

Subqueries

A subquery (also known as a *nested query*) is a SELECT statement contained within one of the following:

- The WHERE clause or HAVING clause of another SELECT statement.
- The WHERE clause of an UPDATE or DELETE statement.

A subquery allows you to base the result of a SELECT, UPDATE, or DELETE statement on the output of the nested SELECT statement.

Except in correlated subqueries, when you issue a subquery Zen parses the entire statement and executes the innermost subquery first. It uses the result of the innermost subquery as input for the next level subquery, and so forth.

For more information about expressions you can use with subqueries, see *SQL Engine Reference*.

Subquery Limitations

A subquery in a WHERE clause becomes part of the search criteria. The following limits apply to using subqueries in SELECT, UPDATE, and DELETE statements:

- You must enclose the subquery in parentheses.
- The subquery cannot contain a UNION clause.
- Unless you use an ANY, ALL, EXISTS, or NOT EXISTS keyword in the WHERE clause of the outer query, the selection list of the subquery can contain only one column name expression.
- Either TOP or LIMIT is allowed within a single query or subquery, but not both.

You can nest several levels of subqueries in a statement. The number of subqueries you can nest is determined by the amount of memory available to Zen.

Correlated Subqueries

A correlated subquery contains a WHERE or HAVING clause that references a column from a table in the outer query's FROM clause; this column is called a *correlated column*. To test the results from a subquery against the results from the outer query, or to test for a particular value in a query, you must use a correlated subquery.

Since the correlated column comes from the outer query, its value changes each time a row in the outer query is fetched. Zen then evaluates the expressions in the inner query based on this changing value.

The following example shows the names of courses that provide more credit hours than time actually spent in the class room.

```
SELECT c.Name, c.Credit_Hours
   FROM Course c
   WHERE c.Name IN
      (SELECT c1.Name
      FROM Class cl
```

```
      WHERE c.Name = cl.Name AND c.Credit_Hours >
         (HOUR (Finish_Time - Start_Time) + 1))#
```

To improve performance, you could rephrase the previous statement as a simple query.c.

```
SELECT c.Name, c.Credit_Hours
   FROM Class c1, Course c
   WHERE c1.Name = c.Name AND c.Credit_Hours >
      (HOUR (Finish_Time - Start_Time) + 1)#
```

Restriction Clauses

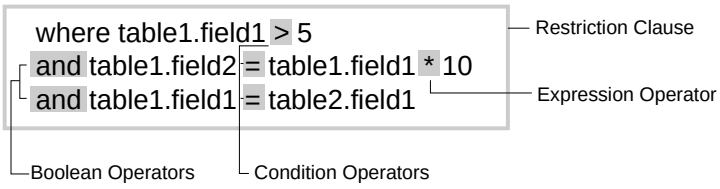
A *restriction clause* is an ASCII text string of operators and expressions. A *restriction clause* specifies selection criteria for the values in the columns of a view, limiting the number of rows the view contains. The syntax of certain clauses (such as WHERE or HAVING) requires using a restriction clause. A restriction clause can specify these conditions:

- Restriction condition – Compares an expression that references a column value to either a constant or another expression that references a column value in the same table.
- Join condition – Compares an expression that references a column value from one table to an expression that references a column value from another table.

A restriction clause can contain multiple conditions. It can also contain a SELECT subquery that bases search criteria on the contents of other tables in the database. The condition containing the subquery can contain the EXISTS, NOT EXISTS, ALL, ANY, and SOME keywords, or the IN range operator.

You can specify a restriction clause using a WHERE or HAVING clause in a SELECT, UPDATE, or DELETE statement.

The following restriction clause example illustrates restriction clause elements.



Restriction Clause Operators

Restriction clauses can use three types of operators:

- Boolean operators – Connect conditions in a restriction clause.
- Condition operators – Connect expressions to form a condition. A condition operator can be a relational or a range operator.
- Expression operators – Connect two expressions to form another expression. An expression operator can be an arithmetic or string operator.

Boolean Operators

Boolean operators specify logical conditions:

Operator	Description
AND	If all search conditions connected with AND are true, the restriction passes.
OR	If at least one of the conditions connected by OR is true, the restriction passes.
NOT	If the condition is false, the restriction passes.

Condition Operators

A condition operator can be a relational or a range operator.

- Relational operator – Compares a column value with either another column value or a constant. If the value of the column is true, Zen selects the row.
- Range operator – Compares a column value with a specified range of values for the column. If the value of the column is true, the restriction passes, and Zen selects the row.

The following table lists the relational operators.

Operator	Description	Operator	Description
<	Less than	>=	Greater than or equal to
>	Greater than	!=	Not equal to
=	Equal to	<>	Not equal to
<=	Less than or equal to		

The following table lists the condition operators.

Operator	Description
IN	Value exists in specified list.
NOT IN	Value does not exist in specified list.
BETWEEN	Value exists within specified range.
NOT BETWEEN	Value does not exist within specified range.
IS NULL	Value is the defined NULL value for the column.
IS NOT NULL	Value is not the defined NULL value for the column.

Operator	Description
LIKE	Value matches specified string. You can substitute two wildcard characters for actual characters. The percent sign (%) represents any sequence of *n* characters (where *n* can be zero). The underscore (_) represents a single character.
NOT LIKE	Value does not match specified string.

With the IN and NOT IN operators, the second expression can be a subquery instead of a column name or constant.

Expression Operators

Expression operators allow you to create expressions for computed columns using arithmetic or string operators. For more information, see Functions.

Restriction Clause Examples

The following examples demonstrate some of the restriction clause operators.

OR and Equal To (=)

The following example uses the relational EQUAL TO and boolean OR operators. It selects all rows in which the value of the State column is Texas or New Mexico.

```
SELECT Last_Name, First_Name, State
FROM Person
WHERE State = 'TX' OR State = 'NM'#
```

IN

The following example uses the IN operator. It selects the records from the Person table where the first names are Bill and Roosevelt.

```
SELECT * FROM Person WHERE First_name IN
('Roosevelt', 'Bill')#
```

LIKE

The following example uses the LIKE operator:

```
SELECT ID, First_Name, Last_Name, Zip
FROM Person
WHERE Zip LIKE '787%';
```

This example retrieves records in the Person table where the zip code begins with '787'.

Functions

Once your database contains data, you can use functions on the data to return a result for a set of column values (using aggregate functions) or accept one or more parameters as input and return a single value (using scalar functions).

Aggregate Functions

An aggregate function is a function that returns a single result for a given set of column values. Zen supports the aggregate functions shown in the following table.

Function	Description
AVG	Determines the average of a group of values. If the operand is *not* a DECIMAL, then AVG returns an 8-byte FLOAT. If the operand is a DECIMAL, AVG returns a 10-byte DECIMAL.
COUNT	Counts the number of rows in a specified group. COUNT always returns a 4-byte INTEGER.
DISTINCT	Include the DISTINCT keyword in your SELECT statement to direct the engine to remove duplicate values from the result. By using DISTINCT, you can retrieve all unique rows that match the SELECT statement's specifications.
MAX	Returns the maximum value of a group of values. MAX returns the same data type and size as the operand.
MIN	Returns the minimum value of a group of values. MIN returns the same data type and size as the operand.
SUM	Determines the sum of a group of values. If the operand is *not* a DECIMAL, then SUM returns an 8-byte FLOAT. If the operand is a DECIMAL, SUM returns a 10-byte DECIMAL.

For more information about each of these functions, see *SQL Engine Reference*.

Arguments to Aggregate Functions

For AVG and SUM functions, the argument to the function must be the name of a numeric column. The COUNT, MIN, and MAX functions can provide results on numeric or nonnumeric columns.

You cannot nest aggregate function references. For example, the following reference is *not* valid:

```
SUM(AVG(Cost_Per_Credit))
```

You can use aggregate functions in an expression, as in the following example:

```
AVG(Cost_Per_Credit) + 20
```

You can also use an expression as an argument to a group aggregate function. For example, the following expression *is* valid:

```
AVG(Cost_Per_Credit + 20)
```

The aggregate functions treat null column values as significant. For example, on a table that contains 40 rows of data and 5 rows of null values, the COUNT function returns 45.

You can use the DISTINCT keyword to force Zen to treat all null column values as a single value. The following example calculates the average column value in the Grade column:

```
AVG(DISTINCT Grade)
```

The DISTINCT keyword affects the AVG, COUNT, and SUM functions. It has no effect on the MIN and MAX functions.

Aggregate Function Rules

You can use aggregate functions in a SELECT statement as follows:

- As items in the selection list.
- In a HAVING clause.

Generally, you use aggregate functions in a SELECT statement that contains a GROUP BY clause to determine aggregate values for certain groups of rows. However, if the SELECT statement does not contain a GROUP BY clause and you want to use aggregate functions in it, all the items in the selection list must be aggregate functions.

If the SELECT statement *does* contain a GROUP BY clause, the column or columns specified in the GROUP BY clause must be select terms that are single columns, *not* aggregate functions. All the select terms that are not also listed in the GROUP BY clause, however, must be aggregate functions.

The following example returns a result table that allows you to determine the amount each student has paid.

```
SELECT Student_ID, SUM(Amount_Paid)
FROM Billing
GROUP BY Student_ID;
```

You can also include aggregate functions in HAVING clauses used with a GROUP BY clause. Using the HAVING clause with a GROUP BY clause restricts the groups of rows Zen returns. Zen performs the aggregate function on the column of each group of rows specified in the

GROUP BY clause, and returns a single result for each set of rows that has the same value for the grouping column.

In the following example, Zen returns row groups only for students currently enrolled with more than 15 credit hours:

```
SELECT Student_ID, SUM(Credit_Hours)
FROM Enrolls e, Class cl, Course c
WHERE e.Class_ID = cl.ID AND cl.Name = c.Name
GROUP BY Student_ID
HAVING SUM(Credit_Hours) > 15;
```

Scalar Functions

Scalar functions such as CONCAT and CURDATE accept one or more parameters as input and return a single value. For example, the LENGTH function returns the length of a string column value. You can use scalar functions in Zen statements that allow computed columns in expressions.

The type of expression operator you can use depends on the type of result the function returns. For example, if the function returns a numeric value, you can use arithmetic operators. If the function returns a string value, you can use string operators.

You can nest scalar functions, but each nested function must return a result that is an appropriate parameter to the next level scalar function, as in the following example:

```
SELECT RIGHT (LEFT (Last_Name, 3), 1)
FROM Person;
```

Zen executes the LEFT function first. If the value in the Last Name column is Baldwin, the string resulting from the LEFT function is Bal. This string is the parameter of the RIGHT function, which returns 'l' as the rightmost character of the string.

You can use scalar functions that return a numeric result within a computed column that calculates a numeric value. You can also use scalar functions that return a string value as an expression to another string function, but the total length of the string result cannot exceed 255 bytes.

In *SQL Engine Reference*, see Bitwise Operators for information on scalar functions you can use with Zen.

Storing Logic

The following topics explain how to store SQL procedures for future use and how to create triggers. For information about stored views, see Retrieving Data.

- Stored Procedures
- SQL Variable Statements
- SQL Control Statements
- SQL Triggers

Stored Procedures

Using stored procedures, you can group logically associated programming steps into a general process and then invoke that process with one statement. You can also execute this process using different values by passing parameters.

Once invoked, SQL stored procedures are executed in their entirety without internal communication between a host language program and the SQL engine. You can invoke them independently, and they can be invoked as part of the body of other procedures or triggers. For more information about triggers, see SQL Triggers.

You can use SQL variable statements within stored procedures to store values internally from statement to statement. See SQL Variable Statements for more information about these statements.

You can use SQL control statements in stored procedures to control the execution flow of the procedure. For more information about these statements, see SQL Control Statements.

Stored Procedure and Positioned Update

The following is an example of a stored procedure and positioned update:

```
DROP PROCEDURE curs1
CREATE PROCEDURE curs1 (in :Arg1 char(4)) AS
BEGIN
DECLARE :alpha char(10) DEFAULT 'BA';
DECLARE :beta INTEGER DEFAULT 100;

DECLARE degdel CURSOR FOR
SELECT degree, cost_per_credit FROM tuition WHERE Degree = :Arg1 AND cost_per_credit = 100
FOR UPDATE;
OPEN degdel;
FETCH NEXT FROM degdel INTO :alpha,:beta
```

```
DELETE WHERE CURRENT OF degdel;
CLOSE degdel;
END

CALL curs1('BA')
```

Declaring Stored Procedures

To define a stored procedure, use the CREATE PROCEDURE statement.

```
CREATE PROCEDURE EnrollStudent (in :Stud_id integer, in :Class_Id integer);

BEGIN
INSERT INTO Enrolls VALUES (:Stud_id, :Class_Id, 0.0);
END
```

The maximum size for a stored procedure name is 30 characters. Parentheses are required around the parameter list, and the parameter name may be any valid SQL identifier.

Stored procedures must have unique names in the dictionary.

For information about the syntax of the CREATE PROCEDURE statement, see CREATE PROCEDURE in *SQL Engine Reference*.

Invoking Stored Procedures

To invoke a stored procedure, use the CALL statement.

```
CALL EnrollStudent (274410958, 50);
```

You must define a value for every parameter. You can assign a value to a parameter using the associated argument in the CALL statement or with the associated default clause in the CREATE PROCEDURE statement. An argument value for a parameter in a CALL statement overrides any associated default value.

You can specify calling values in a CALL statement using either of the following two ways:

- Positional arguments – Allow you to specify parameter values implicitly based on the ordinal position of the parameters in the list when the procedure was created.
- Keyword arguments – Allow you to specify parameter values explicitly by using the name of the parameter whose value is being assigned.

You cannot assign a parameter value twice in the argument list (either positional or keyword). If you use both positional arguments and keyword arguments in the same call, the keyword arguments must not refer to a parameter that receives its value through the positional arguments. When using keyword arguments, the same parameter name must not occur twice.

For syntax information, see CALL in *SQL Engine Reference*.

Deleting Stored Procedures

To delete a stored procedure, use the DROP PROCEDURE statement.

```
DROP PROCEDURE EnrollStudent;
```

For syntax information, see DROP PROCEDURE in *SQL Engine Reference*.

SQL Variable Statements

SQL variables use SET to assign values that are accessible from statement to statement. You can use SET statements inside stored procedures. These statements are called assignment statements.

Procedure-Owned Variables

An SQL variable you define inside a stored procedure is a *procedure-owned variable*. Its scope is the procedure in which it is declared, so you can reference it only within that procedure. If a procedure calls another procedure, the procedure-owned variable of the calling procedure cannot be directly used in the called procedure and instead must be passed in a parameter. You cannot declare a procedure-owned variable more than once in the same stored procedure.

If a compound statement is the body of a stored procedure, then no SQL variable name declared in that procedure can be identical to a parameter name in the parameter list of that procedure. For more information, see Compound Statement.

Assignment Statements

An assignment statement initializes or changes the values of SQL variables. The value may be a computed expression involving constants, operators, and this or other SQL variables.

```
SET :CourseName = 'HIS305';
```

The value expression may also be a SELECT statement.

```
SET :MaxEnrollment = (SELECT Max_Size FROM Class
WHERE ID = classId);
```

For syntax information, see SET in *SQL Engine Reference*.

SQL Control Statements

You can only use control statements in the body of a stored procedure. These statements control the execution of the procedure. The control statements include the following:

- Compound statement (BEGIN...END)
- IF statement (IF...THEN...ELSE)
- LEAVE statement
- Loop statements (LOOP and WHILE)

Compound Statement

A compound statement groups other statements together.

```
BEGIN
DECLARE :NumEnrolled INTEGER;
DECLARE :MaxEnrollment INTEGER;

DECLARE :failEnrollment CONDITION
FOR SQLSTATE '09000';

SET :NumEnrolled = (SELECT COUNT (*)
FROM Enrolls
WHERE Class_ID = classId);

SET :MaxEnrollment = (SELECT Max_Size
FROM Class
WHERE ID = classId);

IF (:NumEnrolled >= :MaxEnrollment) THEN
SIGNAL :failEnrollment ELSE
SET :NumEnrolled = :NumEnrolled + 1;
END IF;
END
```

You can use a compound statement in the body of a stored procedure or a trigger. For more information, see SQL Triggers.

Although you can nest compound statements within other compound statements, only the outermost compound statement can contain DECLARE statements.

For information about compound statement syntax, see BEGIN [ATOMIC] in *SQL Engine Reference*.

IF Statement

An IF statement provides conditional execution based on the truth value of a condition.

```
IF (:counter = :NumRooms) THEN
LEAVE Fetch_Loop;
END IF;
```

For syntax information, see IF in *SQL Engine Reference*.

LEAVE Statement

A LEAVE statement continues execution by leaving a compound statement or loop statement.

```
LEAVE Fetch_Loop
```

A LEAVE statement must appear inside a labeled compound statement or a labeled loop statement. The statement label from the LEAVE statement must be identical to the label of a labeled statement containing LEAVE. This label is called the *corresponding label*.

Note: A compound statement can contain a loop statement; therefore, since you can embed loop statements, the statement label in a LEAVE statement can match the label of any of the embedded loops or the label of the body of the stored procedure.

For syntax information, see LEAVE in *SQL Engine Reference*.

LOOP Statement

A LOOP statement repeats the execution of a block of statements.

```
FETCH_LOOP:
LOOP
FETCH NEXT cRooms INTO CurrentCapacity;

IF (:counter = :NumRooms) THEN
LEAVE FETCH_LOOP;
END IF;

SET :counter = :counter + 1;
SET :TotalCapacity = :TotalCapacity + :CurrentCapacity;
END LOOP;
```

If each statement in the SQL statement list executes without error and Zen does not encounter a LEAVE statement or invoke a handler, then execution of the LOOP statement repeats. A LOOP statement is similar to a WHILE statement in that execution continues while a given condition is true.

If a LOOP statement has a beginning label, it is called a *labeled LOOP statement*. If you specify the ending label, then it must be identical to the beginning label.

For syntax information, see LOOP in *SQL Engine Reference*.

WHILE Statement

A WHILE statement repeats the execution of a block of statements while a specified condition is true.

```
FETCH_LOOP:
WHILE (:counter < :NumRooms) DO
   FETCH NEXT cRooms INTO :CurrentCapacity;
   IF (SQLSTATE = '02000') THEN
     LEAVE FETCH_LOOP;
   END IF;

   SET :counter = :counter + 1;
   SET :TotalCapacity = :TotalCapacity + :CurrentCapacity;
END WHILE;
```

Zen evaluates the Boolean value expression. If it is true, then Zen executes the SQL statement list. If each statement in the SQL statement list executes without error and no LEAVE statement is encountered, then execution of the loop statement repeats. If the Boolean value expression is false or unknown, Zen terminates execution of the loop statement.

If a WHILE statement has a beginning label, it is called a *labeled WHILE statement*. If you specify an ending label, it must be identical to the beginning label. For syntax information, see WHILE in *SQL Engine Reference*.

SQL Triggers

Triggers are actions defined on a table that you can use to enforce consistency rules for a database. They are dictionary objects that identify the appropriate action for the DBMS to perform when a user executes a SQL data modification statement on that table.

To declare a trigger, use the CREATE TRIGGER statement.

```
CREATE TRIGGER CheckCourseLimit;
```

The maximum size for a trigger name is 30 characters.

To delete a trigger, use the DROP TRIGGER statement.

```
DROP TRIGGER CheckCourseLimit;
```

You cannot invoke a trigger directly; they are invoked as a consequence of an INSERT, UPDATE, or DELETE action on a table with an associated trigger. For syntax information, see the following topics in *SQL Engine Reference*:

- CREATE TRIGGER
- DROP TRIGGER
- INSERT
- UPDATE
- DELETE

Note: In order to prevent circumvention of triggers, Zen stamps the data file containing a trigger as a bound data file. A bound file enforces the database constraints for Btrieve users. For more information, see *SQL Engine Reference*.

Timing and Ordering of Triggers

Since triggers execute automatically for a given event, it is important to be able to specify when and in what order the trigger or triggers should execute. You specify time and order when you create the trigger.

Specifying the Triggered Action Time

When an event that is associated with a trigger occurs, the trigger must execute either before the event or after the event. For example, if an INSERT statement invokes a trigger, the trigger must execute either before the INSERT statement executes or after the INSERT statement executes.

```
CREATE TABLE Tuitionidtable (primary key(id), id ubigint)#
```

```
CREATE TRIGGER  InsTrig
BEFORE INSERT ON Tuition
REFERENCING NEW AS Indata
FOR EACH ROW
INSERT INTO Tuitionidtable VALUES(Indata.ID);
```

You must specify either BEFORE or AFTER as the triggered action time. The triggered action executes once for each row. If you specify BEFORE, the trigger executes before the row operation; if you specify AFTER, the trigger executes after the row operation.

Note: Zen does not invoke a trigger by enforcing an RI constraint. Also, a table may not have a DELETE trigger defined if an RI constraint may also cause the system to perform cascaded deletes on that table.

Specifying Trigger Order

You may have situations in which an event invokes more than one trigger for the same specified time. For example, an INSERT statement may invoke two or more triggers that are defined to execute after the INSERT statement executes. Since these triggers cannot execute simultaneously, you must specify an order of execution for them.

Since the following CREATE TRIGGER statement specifies an order of 1, any subsequent BEFORE INSERT triggers that you define for the table must have a unique order number greater than 1.

```
CREATE TRIGGER CheckCourseLimit
BEFORE INSERT
ON Enrolls
ORDER 1
```

You designate the order value with an unsigned integer, which must be unique for that table, time, and event. If you anticipate inserting new triggers within the current order, leave gaps in the numbering to accommodate this.

If you do not designate an order for a trigger, then the trigger is created with a unique order value that is higher than that of any trigger currently defined for that table, time, and event.

Defining the Trigger Action

The trigger action executes once for each row. The syntax for the trigger action is as follows:

```
CREATE TRIGGER  InsTrig
BEFORE INSERT ON Tuition
REFERENCING NEW AS Indata
FOR EACH ROW
INSERT INTO Tuitionidtable VALUES(Indata.ID);
```

If the triggered action contains a WHEN clause, then the triggered SQL statement executes if the Boolean expression is true. If the expression is not true, then the triggered SQL statement does not execute. If no WHEN clause is present, then the triggered SQL statement executes unconditionally.

The triggered SQL statement can be either a single SQL statement, including a stored procedure call (CALL *procedure_name*), or a compound statement (BEGIN...END).

Note: The triggered action must not change the subject table of the trigger.

When you need to reference a column of the old row image (in the case of DELETE or UPDATE) or a column of the new row image (in the case of INSERT or UPDATE) in the triggered action, you must add a REFERENCING clause to the trigger declaration, as follows:

```
REFERENCING NEW AS N
```

The REFERENCING clause allows you to maintain information about the data that the trigger modifies.

Managing Data

The following topics cover data management:

Overview of Managing Data

This section covers the following tasks:

- Defining relationships among tables
- Administering database security
- Controlling concurrency
- Atomicity in SQL databases

In most cases, you can use SQL statements to perform these database management tasks.

You can also enter the SQL statements using the SQL Data Manager. For more information about using SQL Data Manager, see *Zen User's Guide*.

Defining Relationships Among Tables

You can use referential integrity (RI) with Zen to define how each table is related to other tables in the database. RI assures that when a column (or group of columns) in one table refers to a column (or group of columns) in another table, changes to those columns are synchronized. RI provides a set of rules that define the relationships between tables. These rules are known as *referential constraints*. (Referential constraints are also informally referred to as *relationships*.)

When you define referential constraints for tables in a database, the MicroKernel Engine enforces the constraints across all applications that access those tables. This frees the applications from checking table references independently each time an application changes a table.

You must name your database in order to use RI. Once you have defined referential constraints, each affected data file contains the database name. When someone attempts to update a file, the MicroKernel Engine uses the database name to locate the data dictionary containing the applicable RI definitions and checks the update against those RI constraints. This prevents Zen applications from compromising RI, since the MicroKernel Engine blocks updates that do not meet referential constraints.

To define referential constraints on the tables in a database, use CREATE TABLE and ALTER TABLE statements. For syntax information, see the following topics in *SQL Engine Reference*:

- CREATE TABLE
- ALTER TABLE

Referential Integrity Definitions

The following definitions are useful in understanding referential integrity.

- A *parent table* is a table that contains a primary key referenced by a foreign key.
- A *parent row* is a row in a parent table whose primary key value matches a foreign key value.
- A *delete-connected table* occurs if your deletion of rows in one table causes the deletion of rows in a second table. The following conditions determine whether tables are delete-connected:
  - A self-referencing table is delete-connected to itself.
  - Dependent tables are always delete-connected to their parents, regardless of the delete rule.
  - A table is delete-connected to its grandparents when the delete rules between the parent and grandparents is CASCADE.

- A *dependent table* is a table that contains one or more foreign keys. Each of these foreign keys can reference a primary key in either the same or a different table. A dependent table can contain multiple foreign keys.

  Every foreign key value in a dependent table must have a matching primary key value in the associated parent table. In other words, if a foreign key contains a particular value, the primary key of one of the rows in the foreign key's parent table must also contain that value.

  Attempting to insert a row into a dependent table fails if the parent table for each referential constraint does not have a matching primary key value for the foreign key value in the dependent table row being inserted. Attempting to delete a row in a parent table to which foreign keys currently refer either fails or causes the dependent rows to be deleted as well, depending on how you have defined the referential constraints.
- A *dependent row* is a row in a dependent table; its foreign key value depends on a matching primary key value in the associated parent row.
- An *orphan row* is a row in a dependent table that has a foreign key value that does not exist in the index corresponding to the parent table's primary key. The dependent key value does not have a corresponding parent key value.
- A *reference* is a foreign key that refers to a primary key.
- A *reference path* is a particular set of references between dependent and parent tables.
- A *descendant* is a dependent table on a reference path. It may be one or more references removed from the path's original parent table.
- A *self-referencing table* is a table that is its own parent table; the table contains a foreign key that references its primary key.
- A *cycle* is a reference path in which the parent table is its own descendant.

Keys

To use referential integrity, you must define keys. Keys can be either primary and foreign.

A *primary key* is a column or group of columns whose value uniquely identifies each row in a table. Because the key value is always unique, you can use it to detect and prevent duplicate rows.

A *foreign key* is a column or set of columns that is common to the dependent and parent tables in a table relationship. The parent table must have a matching column or set of columns that is defined as the primary key. Foreign keys reference primary keys in a parent table. It is this relationship of a column in one table to a column in another table that provides the MicroKernel Engine with its ability to enforce referential constraints.

Primary Keys

A good primary key has these characteristics:

- It is mandatory; it must store nonnull values.
- It is unique. For example, the ID column in a Student or Faculty table is a good key because it uniquely identifies each individual. It is less practical to use a person's name because more than one person might have the same name. Also, databases do not detect variations in names as duplicates (for example, Andy for Andrew or Jen for Jennifer).
- It is stable. The ID of a student is a good key not only because it uniquely identifies each individual, but it is also unlikely to change, while a person's name might change.
- It is short; it has few characters. Smaller columns occupy less storage space, database searches are faster, and entries are less prone to mistakes. For example, an ID column of 9 digits is easier to access than a name column of 30 characters.

Creating Primary Keys

You create a referential constraint by creating a foreign key on a table. However, before creating the foreign key, you must create a primary key on the parent table to which the foreign key refers.

A table can have only one primary key. You can create a primary key using either of the following:

- A PRIMARY KEY clause in a CREATE TABLE statement.
- An ADD PRIMARY KEY clause in an ALTER TABLE statement.

The following example creates the primary key ID on the Person table in the sample database:

```
ALTER TABLE Person
ADD PRIMARY KEY (ID);
```

When creating a primary key, remember that Zen implements the primary key on the table using a unique, nonnull, nonmodifiable index. If one does not exist for the specified columns, then Zen adds a nonnamed index with these attributes containing the columns specified in the primary key definition.

Dropping Primary Keys

You can delete a primary key only after you have dropped all foreign keys that depend on it. To drop a primary key from a table, use a DROP PRIMARY KEY clause in an ALTER TABLE statement. Since a table can have only one primary key, you do not have to specify the column name when you drop the primary key, as the following example illustrates:

```
ALTER TABLE Person
DROP PRIMARY KEY;
```

Changing Primary Keys

To change a table's primary key, follow these steps:

1. Drop the existing primary key using a DROP PRIMARY KEY clause in an ALTER TABLE statement.

Note: Doing so does not remove the column, or the index used by the primary key; it only removes the primary key definition. To remove the primary key, there must be no foreign key referencing the primary key.

2. Create a new primary key using an ADD PRIMARY KEY clause in an ALTER TABLE statement.

Foreign Keys

A *foreign key* is a column or set of columns that is common to the dependent and parent tables in a table relationship. The parent table must have a matching column or set of columns that is defined as the primary key. When you create a foreign key, you are creating a referential constraint, or a data link, between a dependent table and its parent table. This referential constraint can include rules for deleting or updating dependent rows in the parent table.

The foreign key name is optional. If you do not specify a foreign key name, Zen tries to create a foreign key using the name of the first column in the foreign key definition. For more information about naming conventions for foreign keys and other database elements, see Naming Conventions.

Because Zen keywords are reserved words, you cannot use them in naming database elements. For a list of keywords, see SQL Reserved Words in *SQL Engine Reference*.

Creating Foreign Keys in Existing Tables

To create a foreign key in an existing table, follow these steps:

1. Ensure that a primary key exists in the parent table you are referencing.

   All columns in the primary and foreign key must be of the same data type and length, and the set of columns must be in the same order in both definitions.

2. Zen creates a nonnull index for the column or group of columns specified in the foreign key definition. If the table definition already has such an index, Zen uses that index; otherwise, Zen creates a nonnamed index with the nonnull, non-unique, and modifiable index attributes.

3. Create the foreign key using an ADD CONSTRAINT clause in an ALTER TABLE statement.

   For example, the following statement creates a foreign key called Faculty_Dept on the column Dept_Name in the Faculty table of the sample database. The foreign key references the primary key created in the Department table and specifies the delete restrict rule.

```
ALTER TABLE Faculty
ADD CONSTRAINT Faculty_Dept FOREIGN KEY  (Dept_Name)
REFERENCES Department
ON DELETE RESTRICT;
```

Creating Foreign Keys When Creating a Table

To create a foreign key when creating the table, follow these steps:

1. Ensure that a primary key exists in the parent table you are referencing.

   All columns in the primary and foreign key must be of the same data type and length, and the set of columns must be in the same order in both definitions.

2. Zen creates a nonnull index for the column or group of columns specified in the foreign key definition. If the table definition already has such an index, Zen uses that index; otherwise, Zen creates a nonnamed index with the nonnull, non-unique, and modifiable index attributes.

3. Create the table using a CREATE TABLE statement and include a FOREIGN KEY clause.

   For example, the following statement creates a foreign key called Course_in_Dept on the column Dept_Name in a table called Course.

```
CREATE TABLE Course
   (Name CHAR(7) CASE,
   Description CHAR(50) CASE,
   Credit_Hours USMALLINT,
```

```
      Dept_Name CHAR(20) CASE)#

   ALTER TABLE Course
      ADD CONSTRAINT Course_in_Dept
      FOREIGN KEY (Dept_Name)
      REFERENCES DEPARTMENT(Name)
      ON DELETE RESTRICT
```

Dropping Foreign Keys

To delete a foreign key from a table, use a DROP CONSTRAINT clause in an ALTER TABLE statement. You must specify the foreign key name since a table can have more than one foreign key.

```
ALTER TABLE Course
DROP CONSTRAINT Course_in_Dept;
```

Referential Constraints

Databases on which you define referential constraints must meet the following requirements:

- The database must have a database name.
- The database must reside on a single workstation drive or a single mapped network drive.
- The data files must be in 6.x or later MicroKernel Engine format.

  For information about converting 5.x or later data files to 6.x or 7.x format, see *Advanced Operations Guide*.

In order for a database to support referential integrity it must also support the concept of foreign keys. A foreign key is a column or set of columns in one table (called the dependent table) that is used to reference a primary key in another table (called the parent table). The RI rule requires all foreign key values to reference valid primary key values. For example, a student cannot enroll in a nonexistent course.

You can use a CREATE TABLE or ALTER TABLE statement to define keys on a table in a named database. The following sections explain how to create and modify keys. These sections also provide examples of referential constraints.

After you define referential constraints on a database, applications that do not perform data updates according to referential rules may fail. For example, if an application tries to insert a row into a dependent table before inserting the corresponding parent row into the parent table, the insertion fails. See Referential Integrity Rules for more information.

Note: If a file has referential constraints defined, it is a bound data file. If a user tries to access it with Btrieve, then the Btrieve user can access the file, but can only perform actions within RI constraints. For more information about bound data files, see Understanding Database Rights.

Referential Integrity Rules

Certain rules apply to inserting and updating rows in dependent tables and updating and deleting rows in parent tables when you define referential constraints on database tables. Zen supports the restrict and cascade rules as follows:

- Insert into dependent table – The parent table for each foreign key definition must have a corresponding primary key value for the foreign key value being inserted. If any parent table does not have a corresponding value, then the insert operation fails.
- Update in the dependent table – The parent table for each foreign key definition must have a corresponding primary key value for the foreign key value (the new value for the foreign key). If any parent table does not have a corresponding value, then the update operation fails.
- Update in the parent table – This is not allowed. You cannot update primary key values. To perform a similar operation, delete the row you want to update, then insert the same row with the new primary key value.
- Delete in the parent table – You can specify either the cascade or restrict rule for this operation. *Cascade* means that if a dependent table contains a foreign key value that matches the primary key value being deleted, then all rows containing that matching value are deleted from the dependent table also.

  Restrict means that if a dependent table contains a foreign key value that matches the primary key value being deleted, then the Delete operation on the parent table fails. The cascade operation is recursive; if the dependent table has a primary key that is the parent table of a cascade foreign key, then the process is repeated for that set of data.

Insert Rule

The insert rule is a *restrict rule*. For each foreign key in the row being inserted, the foreign key value must be equivalent to a primary key value in the parent table. The parent table must contain a parent row for the foreign key in the row you are inserting; otherwise, the insertion fails. Zen causes the MicroKernel Engine to automatically enforce the insert rule on dependent tables.

Update Rule

The update rule is also a *restrict rule*. A foreign key value must be updated to an equivalent primary key value in the parent table. If the parent table does not contain a parent row for the foreign key value, the update fails.

You can explicitly specify the update rule as restrict when you define a foreign key on a table; however, Zen causes the MicroKernel Engine to enforce the rule by default if you do not specify it.

Delete Rule

You can explicitly specify the delete rule as either *restrict* or *cascade* when you define a foreign key. If you do not specify the delete rule explicitly, Zen assumes a default of restrict for the delete rule.

- If you specify restrict as the delete rule, Zen causes the MicroKernel Engine to check each row you attempt to delete from a parent table to see if that row is a parent row for a foreign key in another table. If it is a parent row, Zen returns a status code and does not delete the row. You must first delete all corresponding rows in the referenced table or tables before you can delete the parent row.
- If you specify cascade as the delete rule, Zen causes the MicroKernel Engine to check each row you attempt to delete from a parent table to see if that row is a parent row for a foreign key in another table. The MicroKernel Engine then checks the delete rule for each descendant of that table. *If any descendant has restrict* as the delete rule, the attempted deletion fails. *If all descendants have cascade* as the delete rule, Zen deletes all dependent rows on the reference path to the original parent table.

The following guidelines govern the delete rule for foreign keys:

- A cycle with two or more tables cannot be delete-connected to itself. Consequently, the delete rule for at least two of the dependent tables in the cycle must not be cascade.
- The last delete rule in all paths from one table to another must be the same.
- If the delete rule for the foreign key is cascade, then the table containing the foreign key may not have a delete trigger defined on it.
- If the table containing the foreign key has a delete trigger defined on it, then the delete rule must be restrict.

Zen enforces these guidelines on databases that have referential constraints defined. If you attempt to declare delete rules that violate these guidelines, Zen returns a status code to indicate an error occurred.

Zen enforces the delete rule guidelines to avoid certain anomalies that might otherwise occur when you delete dependent rows from tables. Following are examples of anomalies that might occur without these guidelines.

Anomaly on Delete-Connected Cycles

A cycle with two or more tables cannot be delete-connected to itself. Consequently, the delete rule for at least two of the dependent tables in the cycle must be restrict.

Assume you want to execute the following statement.

```
DELETE FROM Faculty
```

Because of the relationships between the Faculty and Department tables, deleting a row from Faculty first deletes a row from Faculty, then from Department, where the cascaded delete stops because of the restrict rule on the name of the department.

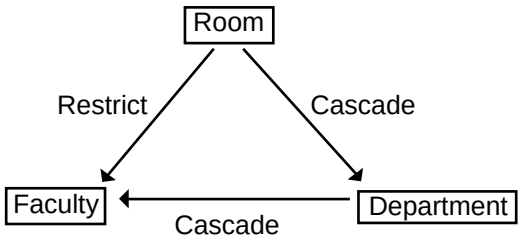
The results could be inconsistent, depending on the order in which Zen deletes rows from the Faculty table. If it attempts to delete the row in which the ID is 181831941, the delete operation fails. The restrict rule on the Department name prevents Zen from deleting the first row in the department table in which the primary key value equals Mathematics, since the second row in Faculty continues to reference this row's primary key.

If instead, Zen deletes the Faculty rows in which the primary keys equal 179321805 and 310082269 first (in either order), all the rows in Faculty and Department are deleted.

Since the result of the example DELETE statement is consistent, no rows are deleted.

Anomaly on Multiple Paths

Delete rules from multiple delete-connected paths must be the same. The following figure shows an example of an anomaly that might occur without this guideline. In the figure, the arrows point to the dependent tables.



Faculty is delete-connected to Room through multiple delete-connected paths with different delete rules. Assume you want to execute the following statement.

```
DELETE FROM Room
WHERE Building_Name = 'Bhargava Building'
AND Number = 302;
```

The success of the operation depends on the order in which Zen accesses Faculty and Department to enforce their delete rules.

- If it accesses Faculty first, the delete operation fails because the delete rule for the relationship between Room and Faculty is restrict.
- If it accesses Department first, the delete operation succeeds, cascading to both Department and Faculty.

To avoid problems, Zen insures that the delete rules for both paths that lead to Faculty are the same.

Referential Integrity in the Sample Database

This section demonstrates the table and referential constraint definitions on the sample database.

Creating the Course Table

The following statement creates the Course table.

```
CREATE TABLE Course
(Name CHAR(7) CASE,
Description CHAR(50) CASE,
Credit_Hours USMALLINT,
Dept_Name CHAR(20))
```

Adding a Primary Key to Course

The following statement adds a primary key (Name) to the Course table.

```
ALTER TABLE Course
ADD PRIMARY KEY (Name);
```

Creating the Student Table with Referential Constraints

The following statement creates the Student table and defines its referential constraints.

```
CREATE TABLE Student
(ID UBIGINT,
PRIMARY KEY (ID),
Cumulative_GPA NUMERICSTS(5,3),
Tuition_ID INTEGER,
Transfer_Credits NUMERICSA(4,0),
Major CHAR(20) CASE,
Minor CHAR(20) CASE,
Scholarship_Amount DECIMAL(10,2),
Cumulative_Hours INTEGER)

CREATE UNIQUE INDEX Tuition_ID ON Student(ID)

ALTER TABLE Student ADD CONSTRAINT
S_Tuition
FOREIGN KEY (Tuition_ID)
REFERENCES Tuition
ON DELETE RESTRICT
```

Administering Database Security

Zen security capabilities allow you to protect data by limiting operations on data columns to particular users. These limits may range from allowing a user to see only certain columns in a table, to allowing them to see all the columns in a table, but not update them. Zen makes no assumptions about database authorization based on operating system rights and permissions. By default, all users accessing a database through Zen have complete read-write access to the data. You must enable and define database security to limit this access and protect the database from unauthorized update or access through Zen.

Zen security statements allow you to perform the following actions to limit access to your database:

- Enable security for the database.
- Identify users and groups of users and assign passwords to them.
- Grant rights to users and user groups.
- Revoke rights from users and user groups.
- Disable security for the database.
- Retrieve information about security defined for a database.

Understanding Database Rights

The following table shows the rights you can grant to users and user groups.

Right	Description
Login	Allows a user to log in to a database. You assign this right when you create a user and a password. The Login right does not give users access to data, however. You must assign other rights to users before they can access data. You cannot assign the Login right to a user group.
Create Table	Enables a user to create new table definitions. The user automatically has full access rights to the tables he or she creates at the time of creation, but the Master User can later revoke read, write, and alter rights for the table. The Create Table right is also referred to as a global right, because it applies to the entire data dictionary.
Select	Allows a user to query tables for information. You can grant the Select right for specific columns or for a whole table.
Update	Gives a user the right to update information in specified columns or tables. You can grant the Update right for specific columns or for a whole table.

Right	Description
Insert	Allows a user to add new rows to tables. You can grant the Insert right only at the table level.
Delete	Allows a user to delete information from tables. You can grant the Delete right only at the table level.
Alter	Allows a user to change the definition of a table. You can grant the Alter right only at the table level.
References	Allows a user to create foreign key references that refer to a table. The References right is necessary for defining referential constraints.
All	Includes Select, Update, Insert, Delete, Alter, and References rights.

You can assign certain types of rights over the whole database or for a particular database element. For example, when you assign the Update right to a user or user group, you can limit it to certain tables or to certain columns in tables. In contrast, when you assign the Create Table right to a user or user group, that user or user group has the Create Table right for the entire database. You cannot apply the Create Table right to a single table or column.

While the Create Table and Login rights apply to the entire database, all other rights apply to tables. In addition, you can apply Select and Update rights to individual columns in tables.

Establishing Database Security

The following nine steps describe the general procedure for establishing security for a database.

1. Log in to the database for which you want to establish security.

   For more information about logging in to a database, see *Zen User's Guide*.

2. Enable security for the database by creating the master user and specifying the master password with the SET SECURITY statement.

   After you have enabled security as Master, the name of the master user is *Master* (case-sensitive), the password you specified when you enabled security becomes the master password (also case-sensitive). For more information, see Enabling Security.

3. *Optional*: Define a minimal set of rights for the PUBLIC group.

   All users automatically belong to the PUBLIC group. For more information, see Granting Rights to the PUBLIC Group.

4. *Optional*: Create user groups with the CREATE GROUP statement.

You can create as many groups as you need for your system. However, a user can belong to only one group other than PUBLIC. For more information, see Creating User Groups.

5. *Optional*: Grant rights to each user group with the GRANT CREATETAB and GRANT (access rights) statements. For more information, see Granting Rights to User Groups.
6. Grant login privileges to users by specifying their user names and passwords with the GRANT LOGIN statement, and if you choose, assign each user to a user group. For more information, see Creating Users.
7. Grant rights to the users you have created who are not members of a user group using the GRANT CREATETAB and GRANT (access rights) statements. For more information, see Granting Rights to Users.
8. *Optional*: To protect your files from unauthorized Btrieve access, make the database a bound database. For more information about bound databases, see Understanding Database Rights.

Enabling Security

You can use a SET SECURITY statement to enable security. In response, Zen creates the master user, who has complete read-write access to the database. The password you specify with a SET SECURITY statement becomes the master password for the database.

The following example enables security for a database and specifies the password for the master user as Secure:

```
SET SECURITY = Secure;
```

Passwords are case-sensitive.

When you enable security, Zen creates the system tables X$User and X$Rights. Enabling security excludes all users except the master user from accessing the database until you explicitly create other users and grant them login rights.

Creating User Groups and Users

After you enable security, your database has one user (Master) and one user group (PUBLIC). To provide other users access to the database, log in to the database as the master user and create users by name and password. You can also organize the users in user groups.

User names are case-sensitive in Zen. Therefore, when you log in as the master user, you must specify the user name as *Master*.

Creating User Groups

To simplify security administration, you can organize users in user groups. You can create as many groups as you need, each with different rights and permissions, but each user in the database can belong to only one group. Once the user is in a group, that login inherits only the rights of that group plus the rights in the PUBLIC group, and any individual rights granted to that user are ignored. The rights of all users in a group are the same. To give a user unique rights, create a special group just for that user.

To create a user group, use a CREATE GROUP statement.

```
CREATE GROUP Accounting;
```

You can also create multiple user groups at once.

```
CREATE GROUP Accounting, Registrar, Payroll;
```

User group names are case-sensitive, cannot exceed 30 characters, and must be unique to the database. For more information about rules for naming user groups, see *Advanced Operations Guide*.

Creating Users

When you create a user for a database, Zen enters the corresponding user name and password into the database security tables. To create a user, use a GRANT LOGIN TO statement. The following example creates the user Cathy and assigns Passwd as her password.

```
GRANT LOGIN TO Cathy:Passwd;
```

Note: Zen stores passwords in encrypted form. Therefore, you cannot query the X$User table to view user passwords.

You can also assign a user to a user group when you create the user. For example, to assign the user Cathy to the Accounting group, use the following statement:

```
GRANT LOGIN TO Cathy : Passwd
IN GROUP Accounting;
```

User names and passwords are case-sensitive. See Identifier Restrictions in *Advanced Operations Guide* for permissible user name and password lengths and characters.

Granting Rights

This topic explains how to grant rights to user groups and individual users.

Granting Rights to the PUBLIC Group

All users automatically belong to the PUBLIC group, a special user group used to define the minimum set of rights for all users of a particular database. No user can have fewer rights than those assigned to the PUBLIC group. You cannot drop a user from the PUBLIC group, and you cannot revoke rights from a user if those rights are granted to the PUBLIC group.

By default, the PUBLIC group has no rights. To change the rights of the PUBLIC group, use a GRANT (access rights) statement. For example, the following statement allows all users of the sample database to query the Department, Course, and Class tables in the database:

```
GRANT SELECT ON Department, Course, Class
TO PUBLIC;
```

After granting rights to the PUBLIC group, you can create other groups to define higher levels of access. You can also give individual users additional rights that differ from any other user or group, provided the user is not part of a group.

Granting Rights to User Groups

You can assign rights to a user group and add user names and passwords to the group. Doing so eliminates assigning rights for each user individually. Also, security is easier to maintain if you assign security rights to groups, since you can change the rights of many users by granting new rights or revoking existing rights for an entire group at once.

To grant rights to a user group, use a GRANT (access rights) statement. For example, the following statement allows all users in the Accounting group to alter the Billing table definition in the sample database.

```
GRANT ALTER ON Billing TO Accounting;
```

Note: Remember that granting the Alter right implicitly grants the rights Select, Update, Insert, and Delete.

Granting Rights to Users

After you create a user, that user can log in to the database. However, the user cannot access data until you either place the user in a user group with rights or grant rights to the user.

To grant rights to a user, use a GRANT (access rights) statement. The following example allows the user John to insert rows into the billing table in the sample database.

```
GRANT INSERT ON Billing
TO John;
```

Note: Granting the Insert right implicitly grants the rights Select, Update, and Delete.

Dropping Users and User Groups

To drop (delete) a user, use a REVOKE LOGIN statement.

```
REVOKE LOGIN FROM Bill;
```

This statement removes the user Bill from the data dictionary. After you drop a user, the user cannot access any tables in the database unless you disable security for the database.

You can also drop multiple users at once, as in the following example.

```
    REVOKE LOGIN FROM Bill, Cathy, Susan;
```

To drop a user group, follow these steps:

1. Drop all users from the group, as in the following example:

```
    REVOKE LOGIN FROM Cathy, John, Susan;
```

2. Use a DROP GROUP statement to drop the group. The following example drops the Accounting group:

```
    DROP GROUP Accounting;
```

Revoking Rights

To revoke user rights, use the REVOKE statement. The following example revokes SELECT rights for user Ron from the Billing table of the sample database.

```
REVOKE SELECT
ON Billing
FROM Ron;
```

Disabling Security

To disable security for a database, follow these steps:

1. Log in to the database as the master user.
2. Issue a SET SECURITY statement, specifying the NULL keyword, as follows:

```
    SET SECURITY = NULL;
```

When you disable security for a database, Zen removes the X$User and X$Rights system tables from the database and deletes the associated DDF files.

Note: You cannot disable security simply by deleting the USER.DDF and RIGHTS.DDF data dictionary files. If you delete these and try to access the database, Zen returns an error and denies access to the database.

Retrieving Information about Database Security

When you set up database security, Zen creates the system tables X$User and X$Rights. Because the system tables are part of the database, you can query them if you have the appropriate rights.

For a reference to the contents of each system table, see System Tables in *SQL Engine Reference*.

Concurrency Controls

The MicroKernel Engine and its automatic recovery functions handle the physical integrity of your database. Zen provides logical data integrity using the transaction and record-locking capabilities of the MicroKernel Engine. Zen, in conjunction with the MicroKernel Engine, provides the following types of concurrency controls:

- Isolation levels for transactions
- Explicit locks
- Passive control

Transaction Processing

Transaction processing lets you identify a set of logically related database modifications, either within a single table or across multiple tables, and require them to be completed as a unit. Transaction processing involves two important concepts:

- A *logical unit of work*, or *transaction*, is a set of discrete operations that must be treated as a single operation to ensure database integrity. If you make a mistake or encounter a problem during a transaction, you can issue a ROLLBACK WORK statement to undo the changes you have already made.

  For example, the Registrar might credit a student account with an amount paid in one operation, then update the amount owed in a second operation. By grouping these operations together you ensure the student's finances are accurate.
- A *locking unit* is the amount of data from which other tasks are blocked until your transaction is complete. (A task is a Zen session.) Locking prevents other tasks from changing the data you are trying to change. If other tasks can also change the data, Zen cannot roll back work to a previously consistent state. Thus, within a transaction, only one task may access a given locking unit at a time. However, multiple cursors that belong to the same task can access the locking unit at the same time.

The START TRANSACTION statement begins a transaction. When you have issued all the statements you want to complete during the transaction, issue a COMMIT WORK statement to

end the transaction. The COMMIT WORK statement saves all your changes, making them permanent.

Note: START TRANSACTION and COMMIT WORK are only supported in stored procedures. For more information on these two SQL statements, see *SQL Engine Reference*.

If an error occurs in one of the operations, you can roll back the transaction and then retry it again after correcting the error. For example, if you need to make related updates to several tables, but one of the updates is unsuccessful, you can roll back the updates you have already made so the data is not inconsistent.

Zen automatically performs the rollback operation if two tasks are sharing a login session and the task that originated the session logs out before the second task completes its transition.

Starting and Ending Transactions

To begin a transaction, issue a START TRANSACTION statement in a stored procedure. After issuing all the statements you want to complete during the transaction, issue a COMMIT WORK statement to save all your changes and end the transaction.

```
START TRANSACTION;
UPDATE Billing B
SET Amount_Owed = Amount_Owed - Amount_Paid
WHERE Student_ID IN
(SELECT DISTINCT E.Student_ID
FROM Enrolls E, Billing B
WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;
```

For more information about the START TRANSACTION statement, see *SQL Engine Reference*.

Using Savepoints to Nest Transactions

In a SQL transaction, you can define additional markers called *savepoints*. Using savepoints, you can undo changes after a savepoint in a transaction and continue with additional changes before requesting the final commit or abort of the entire transaction.

To begin a transaction, use the START TRANSACTION statement. The transaction remains active until you issue a ROLLBACK or COMMIT WORK statement.

To establish a savepoint, use the SAVEPOINT statement.

```
SAVEPOINT SP1;
```

To rollback to a savepoint, use the ROLLBACK TO SAVEPOINT statement.

```
ROLLBACK TO SAVEPOINT SP1;
```

The savepoint name must specify a currently active savepoint in the current SQL transaction. Any changes made after establishing this savepoint are cancelled.

To delete a savepoint, use the RELEASE SAVEPOINT statement.

```
RELEASE SAVEPOINT SP1;
```

You can only use this statement if a SQL transaction is active.

If you issue a COMMIT WORK statement, all savepoints defined by the current SQL transaction are destroyed, and your transaction is committed.

Note: Do not confuse ROLLBACK TO SAVEPOINT with ROLLBACK WORK. The former cancels work only to the indicated savepoint, while the latter cancels the entire outermost transaction and all savepoints established within it.

Savepoints provide a way to nest your transactions, thereby allowing the application to preserve the previous work in the transaction while it waits for a sequence of statements to complete successfully. As an example, you can use a WHILE loop for this purpose. You can set a savepoint before beginning a sequence of statements that may fail on the first attempt. Before your transaction can proceed, this sub-transaction must complete successfully. If it fails, the sub-transaction rolls back to the savepoint, where it can start again. When the sub-transaction succeeds, the rest of the transaction can continue.

A SQL transaction must be active when you issue a SAVEPOINT statement.

Note: The MicroKernel allows each transaction a total of 255 internal nesting levels. However, Zen uses some of these levels internally to enforce atomicity on INSERT, UPDATE, and DELETE statements. Therefore, a session can effectively define no more than 253 savepoints to be active at one time. This limit may be further reduced by triggers that contain additional INSERT, UPDATE, or DELETE statements. If your operation reaches this limit, you must reduce the number of savepoints or the number of atomic statements contained within it.

Work that is rolled back within a savepoint cannot be committed even if the outer transaction(s) completes successfully. However, work that is completed within a savepoint must be committed by the outermost transaction before it is physically committed to the database.

For example, in the sample database you might start a transaction to register a student for several classes. You may successfully enroll the student in the first two classes, but this may fail on the third class because it is full or it conflicts with another class for which the student has enrolled. Even though you failed to enroll the student in this class, you don’t want to undo the student's enrollment for the previous two classes.

The following stored procedure enrolls a student into a class by first establishing a savepoint, SP1, then inserting a record into the Enrolls table. It then determines the current enrollment for

the class and compares this to the maximum size for the class. If the comparison fails, it rolls back to SP1; if it succeeds, it releases savepoint SP1.

```
CREATE PROCEDURE Enroll_student( IN :student ubigint, IN :classnum integer);
BEGIN
DECLARE  :CurrentEnrollment INTEGER;
DECLARE  :MaxEnrollment INTEGER;
SAVEPOINT SP1;
INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);
SELECT COUNT(*)  INTO  :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;
SELECT  Max_size INTO  :MaxEnrollment FROM  Class WHERE  ID = :classnum;
IF :CurrentEnrollment >= :MaxEnrollment
THEN
ROLLBACK to SAVEPOINT SP1;
ELSE
RELEASE SAVEPOINT SP1;
END IF;

END;
```

Note: When working at the SQL level, transactions are controlled in different ways depending on the interface. For ODBC, transactions are controlled through the use of SQL_AUTOCOMMIT option of the *SQLSetConnectOption* API, in conjunction with the *SQLTransact* API.

For more information about the syntax of any of these statements, see the topics for these statements in *SQL Engine Reference*.

Special Considerations

Transactions do not affect the following operations:

- Operations that create or change dictionary definitions. Therefore, you cannot roll back the results of the following statements: ALTER TABLE, CREATE GROUP, CREATE INDEX, CREATE PROCEDURE, CREATE TABLE, CREATE TRIGGER, and CREATE VIEW.
- Operations that remove dictionary definitions. Therefore, you cannot roll back the results of the following statements: DROP DICTIONARY, DROP GROUP, DROP INDEX, DROP PROCEDURE, DROP TABLE, DROP TRIGGER, and DROP VIEW.
- Operations that grant or revoke security rights. Therefore, you cannot roll back the results of the following statements: CREATE GROUP, DROP GROUP, GRANT (access rights), GRANT CREATETAB, GRANT LOGIN, REVOKE (access rights), REVOKE CREATETAB, and REVOKE LOGIN.

If you attempt any of these operations within a transaction and Zen completes the statement, then you cannot roll back the results.

You cannot alter or drop a table (in other words, change its dictionary definition) during a transaction if you have previously referred to that table during the transaction. For example, if you

start a transaction, insert a record into the Student table, and then try to alter the Student table, the ALTER statement fails. You must commit the work from this transaction, and then alter the table.

Isolation Levels

An isolation level determines the scope of a transaction locking unit by allowing you to define the extent to which a transaction is isolated from other users, who may also be in a transaction. When you use isolation levels, Zen automatically locks pages or tables according to the isolation level you specify. These automatic locks, which Zen controls internally, are called *implicit locks*, or *transaction locks*. Locks that an application specifies explicitly are called *explicit locks*, formerly *record locks*.

Zen offers two isolation levels for your transactions :

- Exclusive (locks the entire data file you are accessing). Corresponds to the ODBC isolation level SQL_TXN_SERIALIZABLE
- Cursor stability (locks either the row or page you are accessing). Corresponds to the ODBC isolation level SQL_TXN_READ_COMMITTED

You set the isolation level using the ODBC API *SQLSetConnectOption.*

Exclusive Isolation Level (SQL_TXN_SERIALIZABLE)

When you use the exclusive isolation level, the locking unit is an entire data file. Once you access a file or files within an exclusive transaction, those files are locked from any similar access by any other user in a transaction. This type of locking is most effective when few applications attempt to access the same tables at the same time, or when large parts of the file must be locked in the course of a transaction.

Zen releases the lock on the file or files when you end the transaction. When you access a table during an exclusive transaction, the following conditions take effect:

- Other tasks that are in a transaction cannot read, update, delete, or insert rows in that table until you end the transaction.
- Other tasks that are *not* in a transaction can read rows in the table, but they cannot update, delete, or insert rows.
- Multiple cursors within the same task can read any row in the table. However, when you perform an update, delete, or insert operation with a particular cursor, Zen locks the entire data file for that cursor.

When you access tables through a joined view using the exclusive isolation level, Zen locks all the accessed data files in the view.

Cursor Stability Isolation Level (SQL_TXN_READ_COMMITTED)

The MicroKernel Engine maintains data files as a set of *data pages* and *index pages*. When you use the cursor stability isolation level, the locking unit is a *data page* or *index page* instead of a data file. When you read records within a cursor stability transaction, Zen locks the data pages that contain those records for possible update, but allows concurrent access to a table by multiple tasks within transactions. These read locks are released only when you read another set of records. Zen supports set level cursor stability since it allows an application to fetch multiple records at a time.

In addition, any data modifications you make to the data or index pages cause those records to remain locked for the duration of the transaction, even if you issue subsequent reads. Other users in a transaction cannot access these locked records until you commit or roll back your work. However, other applications can lock other pages from the same files within their own transactions.

When you access a file during a cursor stability transaction, Zen locks data and index pages as follows:

- You read a row, but you do not update it or delete it. Zen locks the data page for that row until your next read operation or until you end the transaction.
- You update a non-index column in a row, delete a row from a table that does not contain indexes, or insert a new row into a table that does not contain indexes. Zen locks the data page for that row for the remainder of the transaction, regardless of subsequent read operations.
- You update an indexed column in a row, delete a row from a table that contains indexes, or insert a new row into a table that contains indexes. Zen locks the affected index page(s), as well as the data page, for the remainder of the transaction, regardless of subsequent read operations.

Cursor stability ensures that the data you read remains stable, while still allowing other users access to other data pages within the same data files. Within the cursor stability isolation level, you can generally achieve greater concurrency for all tasks by limiting the number of rows you read at one time, thereby locking fewer data pages at a time. This allows other network users access to more pages of the data file, since you do not have them locked.

However, if your application is scanning or updating large numbers of rows, you increase the possibility of completely locking other users out of the affected tables. Therefore, it is best to use cursor stability for reading, writing, and committing small transactions.

Cursor stability does not lock records within a subquery. Cursor stability does not guarantee that the conditions under which a row is returned do not change, only that the actual row returned does not change.

Transactions and Isolation Levels

Whenever you access data within a transaction, Zen locks the accessed pages or files for that application. No other application can write to the locked data pages or files until the locks are released.

Using the cursor stability isolation level, when you access tables through a joined view, Zen locks all the accessed pages for all the tables in the view. Using the exclusive isolation level, when you access tables through a joined view, Zen locks all the accessed tables in the view.

Zen performs *no-wait* transactions. If you try to access a record that another task has locked within a transaction, Zen informs you that the page or table is locked or that a deadlock has been detected. In either case, roll back your transaction and begin again. Zen allows multiple cursors in the same application to access the same data file.

The following steps illustrate how two applications interact while accessing the same tables within a transaction. The steps are numbered to indicate the order in which they occur.

Task 1	Task 2
1. Activate the view.	
	2. Activate the view.
3. Begin a transaction.	
	4. Begin a transaction.
5. Fetch records.	
	6. Attempt to fetch records from the same data files.
	7. Receive status code 84 (Record or Page Locked) if both tasks are using cursor stability and Task 2 attempts to fetch the same records that Task 1 has already locked, or receive 85 (File Locked) if one of the tasks is using an exclusive transaction.
	8. Retry the fetch if needed.
9. Update the records.	
10. End the transaction.	

	11. The fetch is successful.
	12. Update the records.
	13. End the transaction.

Since a transaction temporarily locks records, pages, or tables against updates by other applications, an application should not pause for operator input during a transaction. This is because no other application can update the records, pages, or tables accessed in the transaction until the operator responds and the transaction is terminated.

Note: Reading records within a cursor stability transaction does not guarantee that a subsequent update succeeds without conflict. This is because another application may have already locked the index page that Zen needs to complete the update.

Avoiding Deadlock Conditions

A *deadlock condition* occurs when two applications are retrying operations on tables, data pages, index pages, or records that the other one has already locked. To minimize the occurrence of deadlock situations, have your application commit its transactions frequently. Do not attempt to retry the operation from your application; Zen attempts a reasonable number of retries before returning an error.

Deadlock Conditions under Exclusive Isolation Level

When you use the exclusive isolation level, Zen locks the entire data file against updates by other applications; thus, it is possible for a deadlock to occur if your applications do not access data files in the same order, as shown in the following table.

Task 1	Task 2
1. Begin a transaction.	
	2. Begin a transaction.
3. Fetch from File 1.	
	4. Fetch from File 2.
5. Fetch from File 2.	
6. Receive lock status.	
7. Retry Step 5.	
	8. Fetch from File 1.

9. Receive lock status.
10. Retry Step 8.

Deadlock Conditions under Cursor Stability Isolation Level

When you use the cursor stability isolation level, other applications can read and update records or pages in the file you are accessing (records or pages that your application has not locked).

Passive Control

If your application performs single record fetch and update sequences that are not logically connected, you can use the Zen passive method of concurrency. Using this method, you can fetch and update (or delete) records without performing transactions or record locks. These operations are referred to as *optimistic* updates and deletes.

By default, if your task does not use transactions or explicit record locks to complete update and delete operations, your task cannot overwrite another task's changes. The feature that ensures this data integrity is passive control, sometimes referred to as optimistic concurrency control. With passive control, your task does not perform any type of locking. If another task modifies a record after you originally fetched it, you must fetch the record again before you can perform an update or delete operation.

Under passive control, if another application updates or deletes a record between the time you fetch it and the time you issue an update or remove operation, your application receives a conflict status. This indicates that another application has modified the data since you originally fetched it. When you receive a conflict status, you must fetch the record again before you can perform the update or remove operation.

Passive control allows an application that was designed for a single-user system to run on a network without implementing lock calls. However, passive control is effective only when an application is operating in a lightly used network environment or on files in which the data is fairly static. In a heavily used environment or on files that contain volatile data, passive control may be ineffective.

Atomicity in Zen Databases

The principle of atomicity states that if a given statement does not execute to completion, then it should not leave partial or ambiguous effects in the database. For example, if a statement fails after it has inserted three out of five records but does not undo that insert, then the database is not in a consistent state when you retry the operation. If the statement is atomic and it fails to complete execution, then all changes are rolled back, so that the database is in a consistent state. In this example, if all five records are not successfully inserted, then none of them are inserted.

The atomicity rule is especially significant for statements that modify multiple records or tables. It also makes retrying failed operations simpler, because any previous attempt is guaranteed not to have left any partial effects.

Zen enforces atomicity in two ways:

1. Any UPDATE, INSERT, or DELETE statement is defined to be atomic. Zen guarantees that, if a multirecord or multitable modification operation fails, none of the effects of that modification remain in the database.

   This is true for update, insert, or delete operations whether or not they are performed inside or outside of procedures.

2. You may specify stored procedures as ATOMIC when you create them. Such procedures apply the rule of atomicity to their entire execution. Therefore, not only do UPDATE, INSERT, or DELETE statements within an ATOMIC procedure execute atomically, but if any other statements within that procedure fail, all effects of the procedure's execution thus far are rolled back.

Transaction Control in Procedures

Because triggers are always initiated by an external data change statement (INSERT, DELETE, or UPDATE), and all data change statements are defined to be atomic, the following statement are not allowed in triggers or in any procedures invoked by triggers:

- START TRANSACTION
- COMMIT WORK
- ROLLBACK WORK (including RELEASE SAVEPOINT and ROLLBACK TO SAVEPOINT)

In other words, triggers follow the same rules as ATOMIC compound statements.

No user-initiated COMMIT WORK, ROLLBACK WORK, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statement can cause a system-begun transaction (for purposes of atomicity) to end.

A. Sample Collations Using International Sorting Rules

This appendix provides sample collations of language-specific strings, using the ISR tables provided in Btrieve in the following languages:

- German Sample Collations
- Spanish Sample Collations
- French Sample Collations

German Sample Collations

This is a sample of unsorted and sorted strings that use the German character set:

Unsorted Data

Datei	abzüglich	Abriß	Äffin
Ähre	Rubin	aufwärmen	Jacke
ächten	Bafög	Behörde	berüchtigt
bescheißen	zugereiste	Beschluß	Blitzgerät
Bürger	Abgänger	Dämlich	darüber
daß	Aufwasch	absägen	Defekt
dösen	drängeln	drüber	dürr
Efeu	Effekt	einfädeln	einschlägig
dunkel	englisch	Ente	einsetzen
Engländer	entführen	Bergführer	Haselnuß
Füllen	für	Zöllner	fußen
hätte	gefährden	gefangen	Gegenüber
gesinnt	Härte	Haß	Fußgänger
häßlich	hatte	Gewäschshaus	Kahl
Höhe	Jaguar	jäh	Jähzorn
Jux	Käfer	Kaff	Käfig
Kreisförmig	Kreißsaal	Lüftchen	Jahr
luxuriös	Pflügen	pfütze	einhüllen
Reißbrett	Reißer	Prügel	Zögern
Abgang	Raub	Regreß	Zobel
Säge	Führer	Führung	regulär

schnüffler	Rübe	Zoll	Rübli
säen	Rätsel	Salz	Schnörkel
Abschluß	strategisch	Gespann	dünkel
Gewähr	Zone	entblößen	Zugegen
Däne	Straßenkreuzung	Zügel	

Sorted Data

Abgang	drängeln	Gewähr	regulär
Abgänger	drüber	Härte	Reißbrett
Abriß	dunkel	Haselnuß	Reißer
absägen	Dünkel	Haß	Rübe
Abschluß	dürr	häßlich	Rubin
abzüglich	Efeu	hatte	Rübli
ächten	Effekt	hätte	säen
Äffin	einfädeln	Höhe	Säge
Ähre	einhüllen	Jacke	Salz
aufwärmen	einschlägig	Jaguar	Schnörkel
Aufwasch	einsetzen	jäh	schnüffler
Bafög	Engländer	Jahr	Straßenkreuzung
Behörde	englisch	Jähzorn	strategisch
Bergführer	entblößen	Jux	Zobel
berüchtigt	Ente	Käfer	Zögern
bescheißen	entführen	Kaff	Zoll
Beschluß	Führer	Käfig	Zöllner
Blitzgerät	Führung	Kahl	Zone
Bürger	Füllen	Kreisförming	zugegen
Dämlich	für	Kreißsaal	Zügel
Däne	fußen	Lüftchen	Zugereiste
darüber	Fußgänger	luxuriös	
daß	gefährden	pflügen	
Datei	gefangen	Pfütze	
Defekt	Gegenüber	Prügel	
dösen	gesinnt	Rätsel	

Spanish Sample Collations

This is a sample of sorted and unsorted strings that use the Spanish character set:

- Unsorted Data
- Sorted Data

Unsorted Data

acción	añal	añoso	baja	xilografia
abdomen	bético	betún	Borgoña	zoo
búsqueda	acá	zarigüeya	cañada	té
abdicación	cáñamo	caos	cartón	segunda
cigüeña	clarión	cónsul	cúpola	señoría
chaqué	chófer	descortés	desparej	tiña
desparapajo	desteñir	educación	elaboración	ópera
émbolo	epítome	hórreo	época	sordina
estúpido	Eucaristía	flúido	horrendo	tísico
barbárico	garañón	garguero	gruñido	títere
hélice	heróina	gárgara	garanon	tío
herionómano	fréir	helio	horrible	tipo
iglú	ígneo	intentar	interés	repleto
ínterin	accompañanta	interior	jícara	manoseado
jinete	judicial	lactar’	lácteo	bebé
lúpulo	lustar	llana	llegada	típico
llorar	judío	máquina	maraña	tirón
living	maravilla	lívído	marqués	según
llama	manómetro	marquesina	fábula	titán
mí	miasma	obstáculo	obstante	periódico
opiata	ordeñar	ordinal	pabellón	sórdido

pábilo	penumbera	peña	peor	peón
perímetro	período	rábano	réplica	tea
república	señorita	rabia	xilófono	

Sorted Data

abdicación	descortés	hórreo	máquina	repleto
abdomen	desparapajo	horrible	maraña	réplica
acá	desparej	iglú	maravilla	república
acción	desteñir	ígneo	marqués	según
accompañanta	educación	intentar	marquesina	segunda
añal	elaboración	interés	mí	señoría
añoso	émbolo	ínterin	miasma	señorita
baja	epítome	interior	obstáculo	sórdido
barbárico	época	jícara	obstante	sordina
bebé	estúpido	jinete	ópera	té
bético	Eucaristía	judicial	opiata	tea
betún	fábula	judío	ordeñar	tiña
Borgoña	flúido	lactar’	ordinal	tío
búsqueda	fréir	lácteo	pabellón	típico
cañada	garanon	lívído	pábilo	tipo
cáñamo	garañón	living	peña	tirón
caos	gárgara	llegada	penumbera	tísico
cartón	garguero	llorar	peón	titán
chaqué	gruñido	lúpulo	peor	títere
chófer	hélice	lustar	perímetro	xilófono
cigüeña	helio	llama	periódico	xilografia
clarión	heróina	llana	período	zarigüeya

cónsul	herionómano	manómetro	rábano	zoo
cúpola	horrendo	manoseado	rabia	

French Sample Collations

This is a sample of sorted and unsorted strings that use the French character set:

Unsorted Data

ou	lésé	péché	999
OÙ	haïe	coop	caennais
lèse	dû	côlon	bohème
gêné	lamé	pêche	LÈS
cæsium	resumé	Bohémien	pêcher
les	CÔTÉ	résumé	Ålborg
cañon	du	Haie	pécher
cote	colon	l’âme	resume
élève	Canon	lame	Bohême
0000	relève	gène	casanier
élevé	COTÉ	relevé	Grossist
Copenhagen	côte	McArthur	Aalborg
Größe	cølibat	PÉCHÉ	COOP
gêne	révélé	révèle	Noël
île	aïeul	nôtre	notre
août	@@@@@	CÔTE	COTE
côté	coté	aide	air
modelé	MODÈLE	maçon	MÂCON
pèche	pêché	pechère	péchère

Sorted Data

@@@@@	coop	Haie	OÙ
0000	COOP	haïe	pèche
999	Copenhagen	île	pêche
Aalborg	cote	lame	péché
aide	COTE	l’âme	PÉCHÉ
aïeul	côte	lamé	pêché
air	CÔTE	les	pécher
Ålborg	coté	LÈS	pêcher
août	COTÉ	lèse	pechère
bohème	côté	lésé	péchère
Bohême	CÔTÉ	MÂCON	relève
Bohémien	du	maçon	relevé
caennais	dû	McArthur	resume
cæsium	élève	MODÈLE	resumé
Canon	élevé	modelé	résumé
cañon	gène	Noël	révèle
Casanier	gêne	NOËL	révélé
cølibat	gêné	notre	
colon	Größe	nôtre	
côlon	Grossist	ou	

B. Sample Database Tables and Referential Integrity

This appendix covers the following topics:

Overview of the Demodata Sample Database

The sample database Demodata is provided as part of the Zen product and is frequently used in the documentation to illustrate database concepts and techniques. Even if you are already familiar with Zen, you may want to review the information in this appendix in order to become acquainted with the new sample database.

Even though you may not be working in an academic environment, you can use the sample database examples both as a template and a reference to help you design and develop your own customized information systems. You can use the sample queries and other aspects included in our example, since it reflects a real-life scenario.

Structure of the Demodata Sample Database

The physical structure of the database consists of the elements of a relational database: tables, columns, rows, keys, and indexes.

The database contains 10 tables with various relationships between them. It contains data on students, faculty, classes, registration, and so forth.

Assumptions

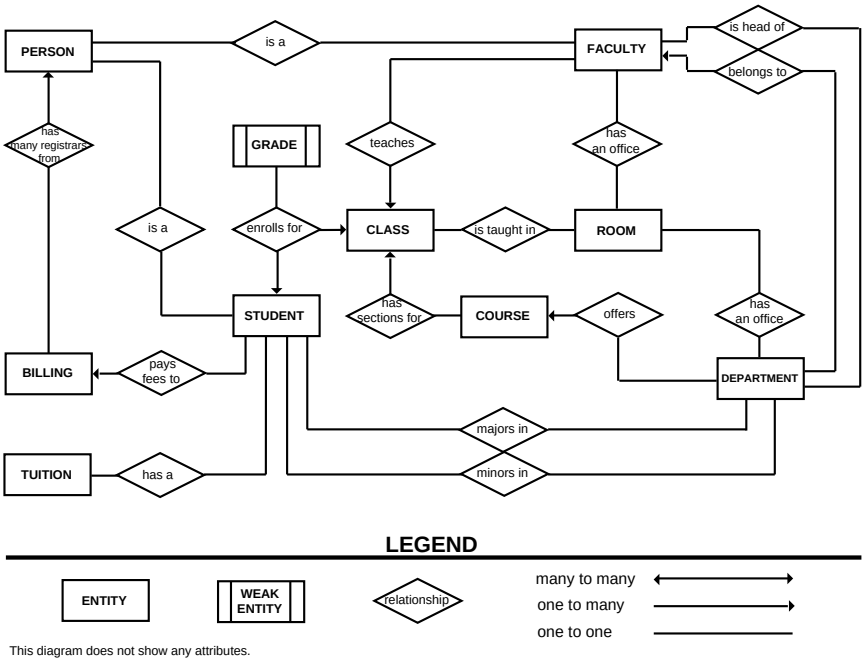
Following are some assumptions around which the database was built:

- The scope of the database is one semester.
- A student cannot take the same course more than once. For example, a student cannot enroll in Algebra I, Sections 1 and 2.
- A faculty member can be a student, but a faculty member cannot teach and enroll in the same class.
- Any course is offered by only one department.
- In order for a student to receive a grade, they must be enrolled in a class, and a faculty member must be assigned to teach the class.
- Faculty members belong to a single department, but they can teach for many departments.
- All students have a Student ID that is based on the US standard of a social security number.
- All faculty members have a Faculty ID that is based on the US standard of a social security number.
- All other persons have a Person ID that is based on the US standard of a social security number.
- Rooms are unique within the same building.
- Two classes cannot be taught in the same room at the same time.
- A faculty member can only be teaching one class at a given time.
- Prerequisites are not required for enrollment in a class.
- Departments imply majors.
- A course can only be taught by one faculty member throughout the semester.
- A telephone number or zip code does not correlate to a state.
- A registrar cannot be a faculty member or student.

- When a person is entered into the database, they can complete a survey of which they must answer all the questions or none of the questions.
- Credit hours for a course are not necessarily equal to the number of hours that a class convenes.
- An e-mail address does not have to be unique.

Entity Relationships

Entities are objects that describe primary components in the database. When designing a database, it is important to define the entities and their relationships to one another before proceeding further. In the Demodata database, CLASSES, STUDENTS, FACULTY, GRADES, and so forth are entities. The entities and their relationships to one another are outlined in the following diagram:

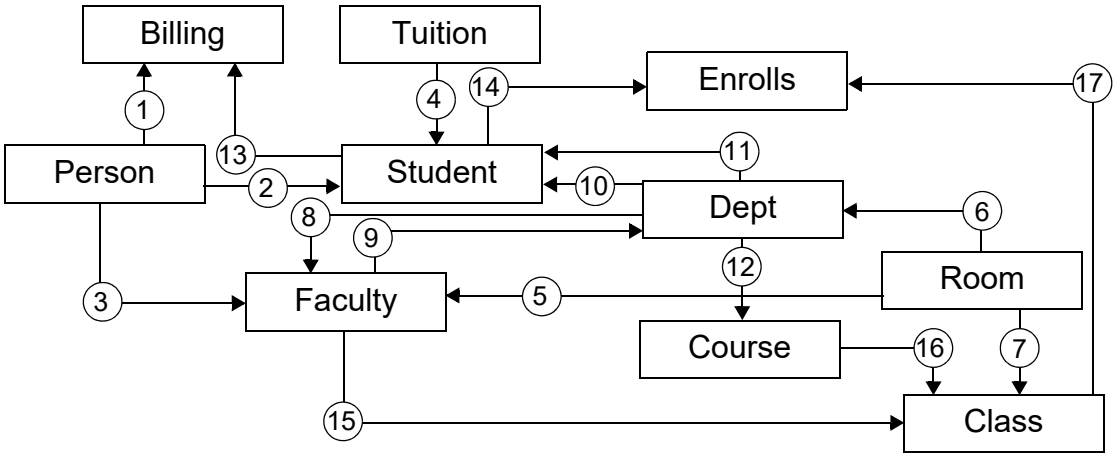


GRADES is a *weak entity*. It is dependent upon a student taking a class, so its existence is dependent upon the validity of other entities. The STUDENT and FACULTY tables create common information, since a student could be a faculty member and vice versa. The common information is in the PERSON table.

Example of Referential Integrity in the Demodata Database

This topic describes a referential integrity design applied to the Demodata sample database and includes a SQL script that can implement the design in the Demodata sample database.

The following diagram depicts a set of references among tables in Demodata. Boxes represent tables. Arrows indicate a referential constraint from a parent table to a referencing table. For example, in constraint 1, a foreign key in the Person table references a primary key in the Billing table.



Note: This diagram also serves as a dependency graph to indicate the order in which tables must be populated for their references to one another to be valid. This order is reflected in the SQL script.

The following table lists these constraints and the relationships among the tables and columns involved with referential integrity.

Constraint	Referencing Table	Foreign Key	Referenced Table	Primary Key
1	BILLING	Registrar_ID	PERSON	ID
2	STUDENT	ID	PERSON	ID
3	FACULTY	ID	PERSON	ID
4	STUDENT	Tuition_ID	TUITION	ID
5	FACULTY	Building_Name, Room_Number	ROOM	Building_Name, Number

Constraint	Referencing Table	Foreign Key	Referenced Table	Primary Key
6	DEPT	Building_Name, Room_Number	ROOM	Building_Name, Number
7	CLASS	Building_Name, Room_Number	ROOM	Building_Name, Number
8	FACULTY	Dept_Name	DEPT	Name
9	DEPT	Head_Of_Dept	FACULTY	ID
10	STUDENT	Major	DEPT	Name
11	STUDENT	Minor	DEPT	Name
12	COURSE	Dept_Name	DEPT	Name
13	BILLING	Student_ID	STUDENT	ID
14	ENROLLS	Student_ID	STUDENT	ID
15	CLASS	Faculty_ID	FACULTY	ID
16	CLASS	Course_Name	COURSE	Name
17	ENROLLS	Class_ID	CLASS	ID

The following script implements the references described in this topic. You can copy and paste the script in ZenCC to apply it to Demodata. We recommend you make a copy of Demodata and use the script on the copy.

```
ALTER TABLE Person (ADD PRIMARY KEY (ID)); -- uses existing PersonID index
ALTER TABLE Billing ADD CONSTRAINT Billing_Person FOREIGN KEY (Registrar_ID)
   REFERENCES Person ON DELETE RESTRICT; -- creates a new Billing_Person index
ALTER TABLE Student ADD CONSTRAINT Student_Person FOREIGN KEY (ID)
   REFERENCES Person ON DELETE RESTRICT; -- uses existing StudentID index
ALTER TABLE Faculty ADD CONSTRAINT Faculty_Person FOREIGN KEY (ID)
   REFERENCES Person ON DELETE RESTRICT; -- uses existing FacultyID index

ALTER TABLE Tuition (ADD PRIMARY KEY (ID)); -- uses existing UK_ID index
ALTER TABLE Student ADD CONSTRAINT Student_Tuition FOREIGN KEY (Tuition_ID)
   REFERENCES Tuition ON DELETE RESTRICT; -- uses existing TuitionID index

ALTER TABLE Room (MODIFY Building_Name CHAR(25) NOT NULL); -- Primary key must be NOT NULL
ALTER TABLE Room (MODIFY Number UINTEGER NOT NULL); -- Primary key must be NOT NULL
ALTER TABLE Room (ADD PRIMARY KEY (Building_Name, Number)); -- uses existing Building_Number index
ALTER TABLE Faculty ADD CONSTRAINT Faculty_Room FOREIGN KEY (Building_Name, Room_Number)
   REFERENCES Room ON DELETE RESTRICT; -- uses existing Building_Room index
ALTER TABLE Dept ADD CONSTRAINT Dept_Room FOREIGN KEY (Building_Name, Room_Number)
   REFERENCES Room ON DELETE RESTRICT; -- uses existing Building_Room index
ALTER TABLE Class ADD CONSTRAINT Class_Room FOREIGN KEY (Building_Name, Room_Number)
   REFERENCES Room ON DELETE RESTRICT; -- creates a new Class_Room index

ALTER TABLE Dept (ADD PRIMARY KEY (Name)); -- uses existing Dept_Name index
```

```
ALTER TABLE Faculty ADD CONSTRAINT Faculty_Dept FOREIGN KEY (Dept_Name)
   REFERENCES Dept ON DELETE RESTRICT; -- uses existing Dept index
ALTER TABLE Student ADD CONSTRAINT Student_Dept_Major FOREIGN KEY (Major)
   REFERENCES Dept ON DELETE RESTRICT; -- creates a new Student_Dept_Major index
ALTER TABLE Student ADD CONSTRAINT Student_Dept_Minor FOREIGN KEY (Minor)
   REFERENCES Dept ON DELETE RESTRICT; -- creates a new Student_Dept_Minor index
ALTER TABLE Course ADD CONSTRAINT Course_Dept FOREIGN KEY (Dept_Name)
   REFERENCES Dept ON DELETE RESTRICT; -- uses existing DeptName index

ALTER TABLE Faculty (ADD PRIMARY KEY (ID)); -- uses existing FacultyID index
ALTER TABLE Dept ADD CONSTRAINT Dept_Faculty FOREIGN KEY (Head_Of_Dept)
   REFERENCES Faculty ON DELETE RESTRICT; -- uses existing Dept index
ALTER TABLE Class ADD CONSTRAINT Class_Faculty FOREIGN KEY (Faculty_ID)
   REFERENCES Faculty ON DELETE RESTRICT; -- creates a new Class_Faculty index

ALTER TABLE Student (ADD PRIMARY KEY (ID)); -- uses existing StudentID index
ALTER TABLE Billing ADD CONSTRAINT Billing_Student FOREIGN KEY (Student_ID)
   REFERENCES Student ON DELETE RESTRICT; -- creates a new Billing_Student index
ALTER TABLE Enrolls ADD CONSTRAINT Enrolls_Student FOREIGN KEY (Student_ID)
   REFERENCES Student ON DELETE RESTRICT; -- uses existing StudentID index

ALTER TABLE Course (ADD PRIMARY KEY (Name)); -- uses existing Course_Name index
ALTER TABLE Class ADD CONSTRAINT Class_Course FOREIGN KEY (Name)
   REFERENCES Course ON DELETE RESTRICT; -- creates a new Class_Course index

ALTER TABLE Class (ADD PRIMARY KEY (ID)); -- uses existing UK_ID index
ALTER TABLE Enrolls ADD CONSTRAINT Enrolls_Class FOREIGN KEY (Class_ID)
   REFERENCES Class ON DELETE RESTRICT; -- creates a new ClassID index
```

Table Design of the Demodata Sample Database

The following guide to the tables in the Demodata sample database. This information is included with each table:

- Columns in the table
- Data types for each column
- Size, or length, of the column in bytes
- Keys (blank if the column is not a key)
- Indexes (blank if the column does not have an index)
- BILLING Table
- CLASS Table
- COURSE Table
- DEPT Table
- ENROLLS Table
- FACULTY Table
- PERSON Table
- ROOM Table
- STUDENT Table
- TUITION Table

BILLING Table

Column	Data Type	Size	Keys
Student_ID	UBIGINT	8	PRIMARY, FOREIGN
Transaction_Number	USMALLINT	2	PRIMARY
Log	TIMESTAMP	8	
Amount_Owed	DECIMAL	7.2	
Amount_Paid	DECIMAL	7.2	
Registrar_ID	UBIGINT	8	FOREIGN

Column	Data Type	Size	Keys
Comments	LONGVARCHAR	65500	

CLASS Table

Column	Data Type	Size	Keys
ID	IDENTITY	4	PRIMARY
Name	CHARACTER	7	FOREIGN
Section	CHARACTER	3	
Max_Size	USMALLINT	2	
Start_Date	DATE	4	
Start_Time	TIME	4	
Finish_Time	TIME	4	
Building_Name	CHARACTER	25	FOREIGN
Room_Number	UINTEGER	4	FOREIGN
Faculty_ID	UBIGINT	8	FOREIGN

COURSE Table

Column	Data Type	Size	Keys
Name	CHARACTER	7	PRIMARY
Description	CHARACTER	50	
Credit_Hours	USMALLINT	2	
Dept_Name	CHARACTER	20	FOREIGN

DEPT Table

Column	Data Type	Size	Keys
Name	CHARACTER	20	PRIMARY
Phone_Number	DECIMAL	10.0	
Building_Name	CHARACTER	25	FOREIGN
Room_Number	UINTEGER	4	FOREIGN
Head_of_Dept	UBIGINT	8	FOREIGN

ENROLLS Table

Column	Data Type	Size	Keys
Student_ID	UBIGINT	8	PRIMARY, FOREIGN
Class_ID	INTEGER	4	PRIMARY, FOREIGN
Grade	REAL	4	

FACULTY Table

Column	Data Type	Size	Keys
ID	UBIGINT	8	PRIMARY, FOREIGN
Dept_Name	CHARACTER	20	FOREIGN
Designation	CHARACTER	10	
Salary	CURRENCY	8	
Building_Name	CHARACTER	25	FOREIGN
Room_Number	UINTEGER	4	FOREIGN
Rsch_Grant_Money	FLOAT	8	

PERSON Table

Column	Data Type	Size	Keys
ID	UBIGINT	8	PRIMARY
First_Name	VARCHAR	15	
Last_Name	VARCHAR	25	
Perm_Street	VARCHAR	30	
Perm_City	VARCHAR	30	
Perm_State	VARCHAR	2	
Perm_Zip	VARCHAR	10	
Perm_Country	VARCHAR	20	
Street	VARCHAR	30	
City	VARCHAR	30	
State	VARCHAR	2	
Zip	VARCHAR	10	
Phone	DECIMAL	10.0	
Emergency_Phone	CHARACTER	20	
Unlisted	BIT	1	
Date_Of_Birth	DATE	4	
Email_Address	VARCHAR	30	
Sex	BIT	1	
Citizenship	VARCHAR	20	
Survey	BIT	1	
Smoker	BIT	1	
Married	BIT	1	
Children	BIT	1	
Disability	BIT	1	
Scholarship	BIT	1	

Column	Data Type	Size	Keys
Comments	LONGVARCHAR	65500	

ROOM Table

Column	Data Type	Size	Keys
Building_Name	CHARACTER	25	PRIMARY
Number	UINTEGER	4	PRIMARY
Capacity	USMALLINT	2	
Type	CHARACTER	20	

STUDENT Table

Column	Data Type	Size	Keys
ID	UBIGINT	8	PRIMARY, FOREIGN
Cumulative_GPA	DECIMAL	5.3	
Tuition_ID	INTEGER	4	FOREIGN
Transfer_Credits	DECIMAL	4.0	
Major	CHARACTER	20	FOREIGN
Minor	CHARACTER	20	FOREIGN
Scholarship_Money	DECIMAL	19.2	
Cumulative_Hours	SMALLINT	2	

TUITION Table

Column	Data Type	Size	Keys
ID	INTEGER	4	PRIMARY

Column	Data Type	Size	Keys
Degree	VARCHAR	4	
Residency	BIT	1	
Cost_Per_Credit	REAL	4	
Comments	LONGVARCHAR	65500	