Progress®



OpenEdge® Getting Started:
Database Essentials

OpenEdge®

Table of Contents

Copyright....7

Preface....9

Purpose....9
Audience....10
Organization....10
References to ABL data types....10
Typographical conventions....11
OpenEdge messages....12
Obtaining more information about OpenEdge messages....12

Chapter 1: Introduction to Databases....15

Advantages of a database....15
Elements of a relational database....16
Tables....17
Rows....17
Columns....17
Keys....18
Indexes....18
Applying the principles of the relational model....19
OpenEdge database and the relational model....21
Database schema and metaschema....21
Sports 2000 database....22
Key points to remember....23

Chapter 2: Database Design....25

Design basics....25
Data analysis....26
Logical database design....27
Table relationships....27
One-to-one relationship....28
One-to-many relationship....29
Many-to-many relationship....29
Normalization....30
First normal form....31
Second normal form....33
Third normal form....34
Denormalization....35
Defining indexes....35

Indexing basics....36
Choosing which tables and columns to index....40
Indexes and ROWIDs....41
Calculating index size....41
Eliminating redundant indexes....42
Deactivating indexes....43
Physical database design....43

Chapter 3: OpenEdge RDBMS....45

OpenEdge database file structure....46
Other database-related files....47
OpenEdge architecture....48
Storage areas....48
Guidelines for choosing storage area locations....50
Extents....50
Clusters....50
Blocks....51
Other block types....52
Storage design overview....54
Mapping objects to areas....55
Determining configuration options....56
System platform....56
Connection modes....56
Client type....57
Database location....57
Database connections....57
Relative- and absolute-path databases....59

Chapter 4: Administrative Planning....61

Data layout....61
Calculating database storage requirements....62
Sizing your database areas....65
Database areas....72
Data area optimization....72
Primary recovery (before-image) information....73
After-image information....74
System resources....76
Disk capacity....76
Disk storage....76
Projecting future storage requirements....77
Comparing expensive and inexpensive disks....78
Understanding cache usage....79
Increasing disk reliability with RAID....79

OpenEdge in a network storage environment....80
Disk summary....80
Memory usage....81
Estimating memory requirements....81
Optimizing memory usage....85
CPU activity....87
Tuning your system....88
Understanding idle time....89
Fast CPUs versus many CPUs....89
Tunable operating system resources....89

Chapter 5: Database Administration....91

Database administrator role....91
Security administrator role....92
Ensuring system availability....92
Database capacity....93
Application load....93
System memory....93
Additional factors to consider in monitoring performance....94
Testing to avoid problems....94
Safeguarding your data....94
Why backups are done....95
Creating a complete backup and recovery strategy....95
Using PROBKUP versus operating system utilities....97
After-imaging implementation and maintenance....98
Testing your recovery strategy....99
Maintaining your system....100
Daily monitoring tasks....100
Monitoring the database log file....100
Monitoring area fill....101
Monitoring buffer hit rate....101
Monitoring buffers flushed at checkpoint....101
Monitoring system resources (disks, memory, and CPU)....101
Periodic monitoring tasks....101
Database analysis....102
Rebuilding indexes....102
Compacting indexes....102
Fixing indexes....103
Moving tables....103
Moving indexes....103
Truncating and growing BI files....103
Dumping and loading....104
Periodic event administration....105
Annual backups....105

Archiving....105
Modifying applications....106
Migrating OpenEdge releases....106
Profiling your system performance....107
Establishing a performance baseline....107
Performance tuning methodology....108
Summary....108

Index....109

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- References to ABL data types
- Typographical conventions
- OpenEdge messages

Purpose

OpenEdge Getting Started: Database Essentials introduces the principles of a relational database, database design, and the architecture of the OpenEdge® database. The book also introduces planning concepts for a successful database deployment, and the database administration tasks required for database maintenance and tuning. You should use this book if you are unfamiliar with either relational database concepts or database administration tasks.

For the latest documentation updates see the OpenEdge Product Documentation on PSDN: https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

Audience

This book is for users who are new database designers or database administrators and who require conceptual information to introduce them to the tasks and responsibilities of their role.

Organization

Introduction to Databases on page 15

Presents an introduction to relational database terms and concepts.

Database Design on page 25

Provides an overview of database design techniques.

OpenEdge RDBMS on page 45

Explains the architecture and configuration supported by an OpenEdge database. This chapter also provides information on storage design and client/server configurations.

Administrative Planning on page 61

Offers administrative planning advice for block sizes, disk space, and other system resource requirements.

Database Administration on page 91

Introduces the database administrator role and discusses the associated responsibilities and tasks.

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.
- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.

Convention	Description
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is `200`:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

```
OpenEdge-install-dir/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

Introduction to Databases

Before you can administer an OpenEdge® database, it is important to understand the basic concepts of relational databases. This chapter introduces those concepts.

For details, see the following topics:

- Advantages of a database
- Elements of a relational database
- Applying the principles of the relational model
- OpenEdge database and the relational model
- Key points to remember

Advantages of a database

A *database* is a collection of data that can be searched in a systematic way to maintain and retrieve information. A database offers you many advantages, including:

- **Centralized and shared data** — You enter and store all your data in the computer. This minimizes the use of paper, files, folders, as well as the likelihood of losing or misplacing them. Once the data is in the computer, many users can access it through a computer network, regardless od the users' physical or geographical locations.
- **Current data** — Since users can quickly update data, the data available is current and ready to use.
- **Speed and productivity** — You can search, sort, retrieve, make changes, and print your data, as well as tally up the totals more quickly than performing these tasks by hand.

- **Accuracy and consistency** — You can design your database to validate data entry, thus ensuring that it is consistent and valid. For example, if a user enters "OD" instead of "OH" for Ohio, your database can display an error message. It can also ensure that the user is unable to delete a customer record that has an outstanding order.
- **Analysis** — Databases can store, track, and process large volumes of data from diverse sources. You can use the data collected from varied sources to track the performance of an area of business for analysis, or to reveal business trends. For example, a clothes retailer can track faulty suppliers, customers' credit ratings, and returns of defective clothing, and an auto manufacturer can track assembly line operation costs, product reliability, and worker productivity.
- **Security** — You can protect your database by establishing a list of authorized user identifications and passwords. The security ensures that the user can perform only permitted operations. For example, you might allow users to read data in your database but they are not allowed to update or delete the data.
- **Crash recovery** — System failures are inevitable. With a database, data integrity is assured in the event of a failure. The database management system uses a transaction log to ensure that your data will be properly recovered when you restart after a crash.
- **Transactions** — The transaction concept provides a generalized error recovery mechanism that protects against the consequences of unexpected errors. Transactions ensure that a group of related database changes always occur as a unit; either all the changes are made or none of the changes are made. This allows you to restore the previous state of the database should an error occur after you began making changes, or if you simply decided not to complete the change.

  To satisfy the definition of a transaction, a database management system must adhere to the following four properties:

  - **Atomicity** — The transaction is either completed or entirely undone. There can be no partial transaction.
  - **Consistency** — The transaction must transform the database from one consistent state to another.
  - **Isolation** — Each transaction must execute independent of any other transaction.
  - **Durability** — Completed transactions are permanent.

  Using the first letter of each of the four properties, satisfying these properties defines your database transactions as ACID compliant.

Now that the advantages of a database system have been discussed, the elements of relational databases follows.

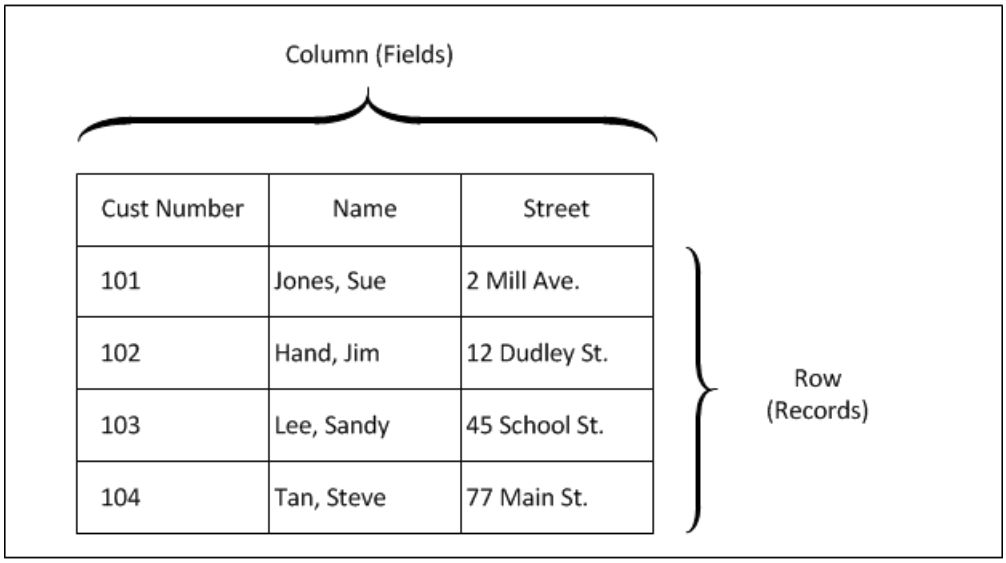
Elements of a relational database

Relational databases are based on the relational model. The *relational model* is a group of rules set forth by E. F. Codd based on mathematical principles (relational algebra), and it defines how database management systems should function. The basic structures of a relational database (as defined by the relational model) are tables, columns (or fields), rows (or records), and keys. This section describes these elements.

Tables

A *table* is a collection of logically related information treated as a unit. Tables are organized by rows and columns. The following figure shows the contents of a sample Customer table.

Figure 1: Columns and rows in the Customer table



Other common tables include an Order table in a retail database that tracks the orders each customer places, an Assignment table in a departmental database that tracks all the projects each employee works on, and a Student Schedule in a college database table that tracks all the courses each student takes.

Tables are generally grouped into three types:

- **Kernel tables** — Tables that are independent entities. Kernel tables often represent or model things that exist in the real world. Some example kernel tables are customers, vendors, employees, parts, goods, and equipment.
- **Association tables** — Tables that represent a relationship among entities. For example, an order represents an association between a customer and goods.
- **Characteristic tables** — Tables whose purpose is to qualify or describe some other entity. Characteristic only have meaning in relation to the entity they describe. For example, order-lines might describe orders; without an order, an order-line is useless.

Rows

A table is made up of rows (or records). A *row* is a single occurrence of the data contained in a table; each row is treated as a single unit. In the Customer table shown in Figure 1: Columns and rows in the Customer table on page 17, there are four rows, and each row contains information about an individual customer.

Columns

Rows are organized as a set of *columns* (or fields). All rows in a table comprise the same set of columns. In the Customer table, shown in Figure 1: Columns and rows in the Customer table on page 17, the columns are Cust Number, Name, and Street.

Keys

There are two types of keys: 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 good primary key has the following characteristics:

- It is *mandatory*; that is, it must store non-null values. If the column is left blank, duplicate rows can occur.
- It is *unique*. For example, the social security column in an Employee or Student table is an example of an unique key because it uniquely identifies each individual. The Cust Number column in the Customer table uniquely identifies each customer. It is not practical to use a person's name as an unique key because more than one customer might have the same name. Also, databases do not detect variations in names as duplicates (for example, Cathy for Catherine, Joe for Joseph). Furthermore, people do sometimes change their names (for example, through a marriage or divorce).
- It is *stable*; that is, it is unlikely to change. A social security number is an example of a stable key because but it is unlikely to change, while a person's or customer's name might change.
- It is *short*; that is, it has few characters. Smaller columns occupy less storage space, database searches are faster, and entries are less prone to mistakes. For example, a social security column of nine digits is easier to access than a name column of 30 characters.

A *foreign key* is a column value in one table that is required to match the column value of the primary key in another table. In other words, it is the reference by one table to another. If the foreign key value is not null, then the primary key value in the referenced table must exist. It is this relationship of a column in one table to a column in another table that provides the relational database with its ability to join tables. Database Design on page 25 describes this concept in more detail.

When either a primary key or foreign key is comprised of multiple columns, it is considered a *composite key*.

Indexes

An *index* in a database operates like the index tab on a file folder. It points out one identifying column, such as a customer's name, that makes it easier and quicker to find the information you want.

When you use index tabs in a file folder, you use those pieces of information to organize your files. If you index by customer name, you organize your files alphabetically; and if you index by customer number, you organize them numerically. Indexes in the database serve the same purpose.

You can use a single column to define a simple index, or a combination of columns to define a composite or compound index. To decide which columns to use, you first need to determine how the data in the table is accessed. If users frequently look up customers by last name, then the last name is a good choice for an index. It is typical to base indexes on primary keys (columns that contain unique information).

An index has the following advantages:

- Faster row search and retrieval. It is more efficient to locate a row by searching a sorted index table than by searching an unsorted table.
- In an application written with OpenEdge ABL (Advanced Business Language), records are ordered automatically to support your particular data access patterns. Regardless of how you change the table, when you browse or print it, the rows appear in indexed order instead of their stored physical order on disk.
- When you define an index as unique, each row is unique. This ensures that duplicate rows do not occur. A unique index can contain nulls, however, a primary key, although unique, cannot contain nulls.
- A combination of columns can be indexed together to allow you to sort a table in several different ways simultaneously (for example, sort the Projects table by a combined employee and date column).

- Efficient access to data in multiple related tables.
- When you design an index as unique, each key value must be unique. The database engine prevents you from entering records with duplicate key values.

Applying the principles of the relational model

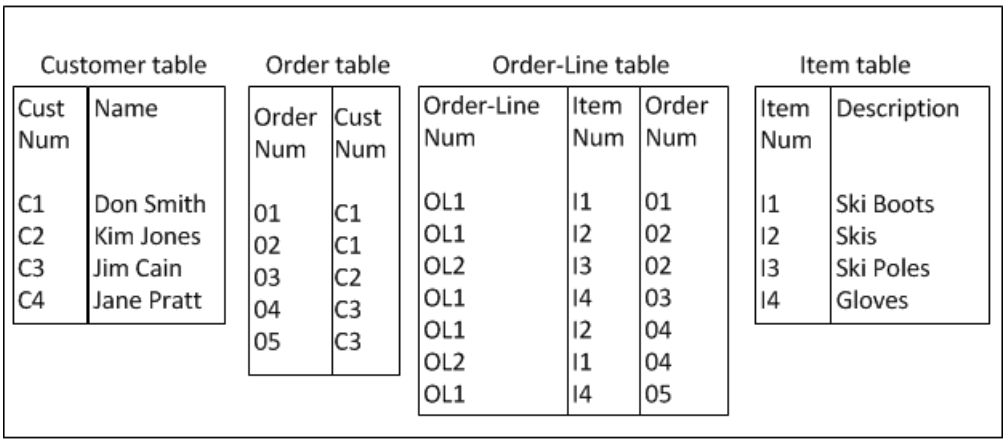
The relational model organizes data into tables and allows you to create relationships among tables by referencing columns that are common to both—the primary and foreign keys. It is easiest to understand this concept of relationships between tables with a common business example.

A hypothetical business needs to track information about customers and their orders. The business' database, as shown in Figure 2: Example of a relational database on page 20, includes the following tables:

- **The Customer table** — The Customer table shows four rows, one for each individual customer. Each row has two columns: Cust Num and Name. To uniquely identify each customer, every customer has a unique customer number. Each column contains exactly one data value, such as C3 and Jim Cain. The primary key is Cust Num.
- **The Order table** — The Order table shows five rows for orders placed by the customers in the Customer table. Each Order row contains two columns: Cust Num, from the Customer table, and Order Num.The primary key is Order Num. The Cust Num column is the foreign key that relates the two tables. This relationship lets you find all the orders placed by a particular customer, as well as information about a customer for a particular order.
- **The Order-Line table** — The Order-Line table shows seven rows for the order-lines of each order. Each order-line row contains three columns: Order-Line Num; Item Num, from the Item table; and Order Num, from the Order table. The primary key is the combination of Order Num and Order-Line Num. The two foreign keys, Order Num and Item Num, relate the Customer, Order, and Item tables so that you can find the following information:
  - All the order-lines for an order
  - Information about the order for a particular order-line
  - The item in each order-line
  - Information about an item

- **The Item table** — The Item table shows four rows for each separate item. Each Item row contains two columns: Item Num and Description. Every item in the Item table has a unique item number. Item Num is the primary key.

Figure 2: Example of a relational database



Customer table

Cust Num	Name
C1	Don Smith
C2	Kim Jones
C3	Jim Cain
C4	Jane Pratt

Order table

Order Num	Cust Num
01	C1
02	C1
03	C2
04	C3
05	C3

Order-Line table

Order-Line Num	Item Num	Order Num
OL1	I1	01
OL1	I2	02
OL2	I3	02
OL1	I4	03
OL1	I2	04
OL2	I1	04
OL1	I4	05

Item table

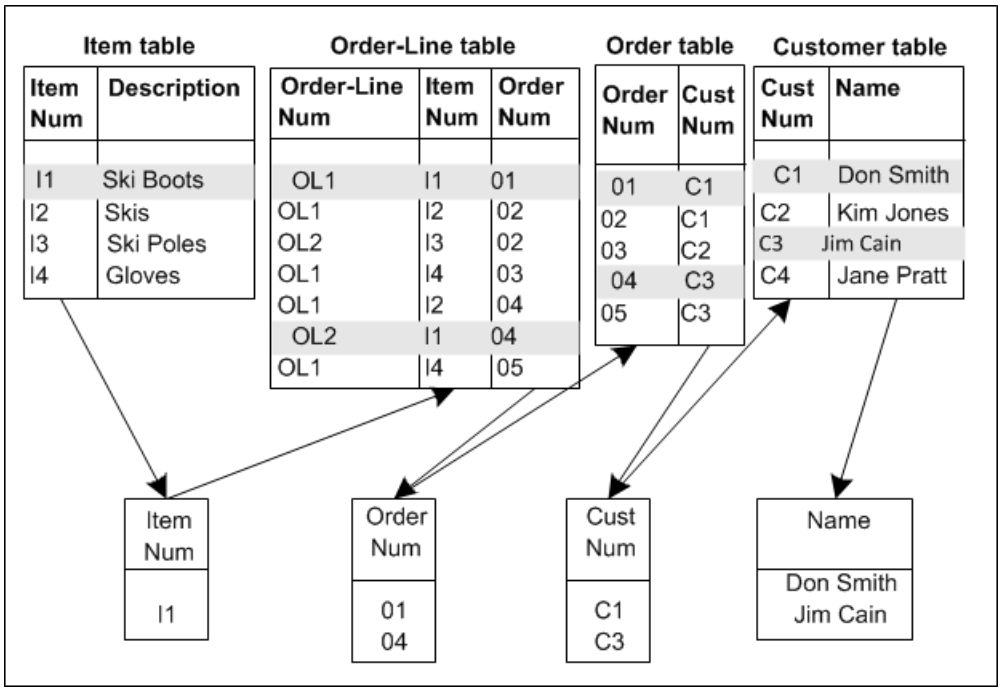
Item Num	Description
I1	Ski Boots
I2	Skis
I3	Ski Poles
I4	Gloves

Suppose you want to find out which customers ordered ski boots. To gather this data from your database, you must know what item number identifies ski boots and who ordered them. There is no direct relationship between the Item table and the Customer table, so to gather the data you need, you join four tables using their primary/foreign key relationships, following these steps:

1. Select the Item table row whose Description value equals ski boots. The Item Number value is I1.
2. Next, locate the Orders that contain Item I1. Because the Order table does not contain Items, you first select the Order-Lines that contain I1, and determine the Orders related to these Order-Lines. Orders 01 and 04 contain Item Number I1.
3. Now that you know the Order Numbers, you can find out the customers who placed the orders. Select the 01 and 04 orders, and determine the associated customer numbers. They are C1 and C3.
4. Finally, to determine the names of Customers C1 and C3, select the Customer table rows that contain customer numbers C1 and C3. Don Smith and Jim Cain ordered ski boots.

The following figure illustrates the steps outlined in the previous procedure.

Figure 3: Selecting records from related tables



By organizing your data into tables and relating the tables with common columns, you can perform powerful queries. The structures of tables and columns are relatively simple to implement and modify, and the data is consistent regardless of the queries or applications used to access the data. Figure 3: Selecting records from related tables on page 21 shows the primary key values as character data for clarity, but a numeric key is better and more efficient.

OpenEdge database and the relational model

The OpenEdge database is a relational database management system (RDBMS).You can add, change, manipulate, or delete the data and data structures in your database as your requirements change.

Database schema and metaschema

The logical structure of the OpenEdge database consists of the elements of a relational database: tables, columns, and indexes. The description of the database's structure, the tables it contains, the columns within the tables, views, etc. is called the database *schema* or the *data definitions*.

The underlying structure of a database that makes it possible to store and retrieve data is called the *metaschema*. That is, the metaschema defines that there can be database tables and columns and the structural characteristics of those database parts. All metaschema table names begin with an underscore (_).

Note: The metaschema is a set of tables that includes itself. Therefore, you can do ordinary queries on the metaschema to examine all table and index definitions, including the definitions of the metaschema itself.

The physical structure of the database and its relationship to the logical structure is discussed in OpenEdge RDBMS chapter.

Sports 2000 database

The Sports 2000 database is one of several sample databases provided with the product, and it is frequently used in the documentation to illustrate database concepts and programming techniques. This database holds the information necessary to track customers, take and process orders, bill customers, and track inventory. The following table describes the tables of the Sports 2000 database. For details about the fields and indexes of the Sports 2000 database, you can use either the Data Dictionary or the Data Admin Tool to create table and index reports. For details on how to create reports, see the online Help.

Table 1: The Sports 2000 database

Table	Description
Benefits	Contains employee benefits
BillTo	Contains bill to address information for an order
Bin	Represents the bins in each warehouse that contain items
Customer	Contains customer information including balance and address
Department	Contains a master listing of departments
Employee	Stores employee information including name and address
Family	Tracks an employee's family information
Feedback	Contains customer feedback regarding likes and dislikes
InventoryTrans	Contains information about the movement of inventory
Invoice	Contains financial information by invoice for the receivables subsystem
Item	Provides quick reference for stocking, pricing, and descriptive information about items in inventory
Local-Default	Contains format and label information for various countries
Order	Contains sales and shipping header information for orders
Order-Line	Provides identification of and pricing information for a specific item ordered on a specific order
POLine	Contains the PO detail information including the item and quantity on the PO
PurchaseOrder	Contains information pertaining to the purchase order including PO number and status
Ref-Call	Contains all history for a customer
Salesrep	Contains names, regions, and quotas for the sales people

Table	Description
ShipTo	Contains the ship to address information for an order
State	Provides U.S. state names, their abbreviations, and sales region
Supplier	Contains a supplier's name, address, and additional information pertaining to the supplier
SupplierItemXr	Lists all of the items that are supplied by a particular supplier
TimeSheet	Records time in and out, hours worked, and overtime
Vacation	Tracks employee vacation time
Warehouse	Contains warehouse information including warehouse name and address

Key points to remember

The following are some key points to remember:

- A database is an electronic filing system for organizing and storing data that relates to a broad subject area, such as sales and inventory.
- A database is made up of tables. A table is a collection of rows about a specific subject, such as customers.
- A row is a collection of pieces of information about one thing, such as a specific customer.
- A column is a specific item of information, such as a customer name.
- An index is a set of pointers to rows that you use as the basis for searching, sorting, or otherwise processing rows, such as a customer number.
- 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. It cannot contain null data.
- An index in a database operates like the index tab on a file folder, making it easier to find information.
- A foreign key is a column (or group of columns) in one table whose values are required to match the value of a primary key in another table.

2

Database Design

It is important to understand the concepts relating to database design. This chapter presents an overview of database design.

For details, see the following topics:

- Design basics
- Data analysis
- Logical database design
- Table relationships
- Normalization
- Defining indexes
- Physical database design

Design basics

Once you understand the basic structure of a relational database, you can begin the database design process. Designing a database is an iterative process that involves developing and refining a database structure based on the information and processing requirements of your business. This chapter describes each phase of the design process.

Data analysis

The first step in the database design cycle is to define the data requirements for your business. Answer the following questions to get started:

- 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 kind of reports do I want to generate?
- What will I do with this information?
- What kind of data control and security does this system require? For information on how a user is identified and authenticated and access is authorized, see *OpenEdge Getting Started: Identity Management*.
- Where is expansion most likely to occur?
- Will multiple clients or sites utilize one common database? Is any information shared between the clients? For a complete introduction to multi-tenancy, see *OpenEdge Getting Started: Multi-tenancy Overview*.
- Do you anticipate large tables that can be partitioned horizontally? Horizontal table partitioning allows you to design a physical database layout that aligns storage with specific data values or ranges. The physical separation of data into partitions can improve performance, maintenance, and data availability. For overview information on table partitioning, see *OpenEdge Getting Started: Table Partitioning*.

It is never too early to consider the security requirements of your design. For example:

- Will any data need to be encrypted?
- Will I need to audit changes to my data?

For complete discussions of OpenEdge support for auditing and transparent data encryption, see *OpenEdge Getting Started: Core Business Services - Security and Auditing.*

To answer some of these questions, list all the data you intend to input and modify in your database, along with all the expected outputs. For example, some of the requirements a retail store might include are the ability to:

- Input data for customers, orders, and inventory items
- Add, update, and delete rows
- Sort all customer addresses by zip code
- List alphabetically all customers with outstanding balances of over $1,000
- List the total year-to-date sales and unpaid balances of all customers in a specific region
- List all orders for a specific item (for example, ski boots)
- List all items in inventory that have fewer than 200 units, and automatically generate a reorder report
- List the amount of overhead for each item in inventory
- Track customer information to have a current listing of customer accounts and balances
- Track customer orders, and print customer orders and billing information for both customers and the accounting department
- Track inventory to know which materials are in stock, which materials need to be ordered, where they are kept, and how much of your assets are tied up with inventory

- Track customer returns on items to know which items to discontinue and which suppliers to notify

The process of identifying the goals of the business, interviewing, and gathering information from the different sources who will use the database is a time-consuming but essential process. Once you the information gathered, you are ready to define your tables and columns.

Logical database design

Logical database design helps you define and communicate your business' information requirements. When you create a logical database design, you describe each piece of information you need to track and the relationships among, or the business rules that govern, those pieces of information.

Once you create a logical database design, you can verify with users and management that the design is complete (that is, it contains all of the data that must be tracked) and accurate (that is, it reflects the correct table relationships and enforces the business rules).

Creating a logical data design is an information-gathering, iterative process. It includes the following steps:

- Define the tables you need based on the information your business requires.
- Determine the relationships between the tables.
- Determine the contents (or columns) of each table.
- Normalize the tables to at least the third normal form.
- Determine the primary keys and the column domain. A *domain* is the set of valid values for each column. For example, the domain for the customer number can include all positive numbers.

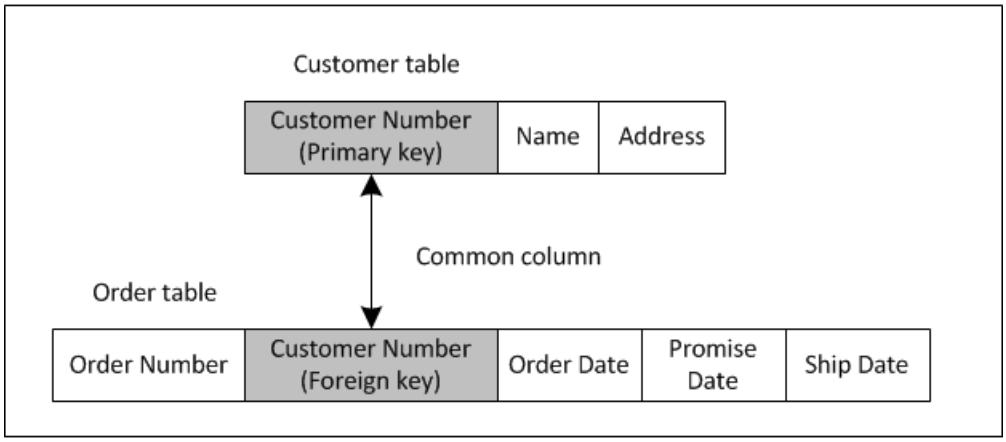
At this point, you do not consider processing requirements, performance, or hardware constraints.

Table relationships

In a relational database, tables relate to one another by sharing a common column or columns. This column, existing in two or more tables, allows the tables to be *joined*. When you design your database, you define the table relationships based on the rules of your business. The relationship is frequently between primary and foreign key columns; however, tables can also be related by other nonkey columns.

The following figure illustrates that the Customer and Order tables are related by a foreign key—the Customer Number.

Figure 4: Relating the Customer table and the Order Table



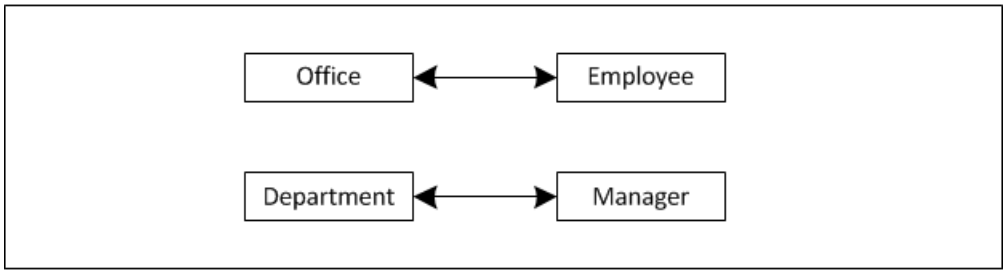
If the Customer Number is an index in both tables, you can quickly do the following:

- Find all the orders for a given customer and query information for each order (such as order date, promised delivery date, the actual shipping date)
- Find customer information for each order using an order's customer number (such as name and address)

One-to-one relationship

A *one-to-one* relationship exists when each row in one table has only one related row in a second table. For example, a business might decide to assign one office to exactly one employee. Thus, one employee can have only one office. The same business might also decide that a department can have only one manager. Thus, one manager can manage only one department. The following figure shows these one-to-one relationships.

Figure 5: Examples of one-to-one relationships

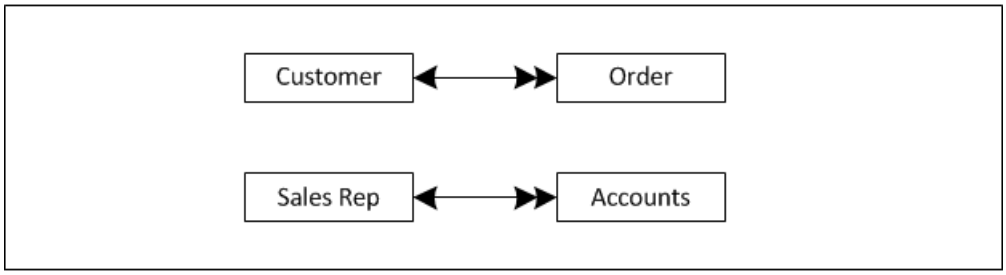


The business might also decide that for one office there can be zero or one employee, or for one department there can be no manager or one manager. These relationships are described as zero-or-one relationships.

One-to-many relationship

A *one-to-many* relationship exists when each row in one table has one or many related rows in a second table. The following figure shows examples: one customer can place many orders, or a sales representative can have many customer accounts.

Figure 6: Examples of one-to-many relationships



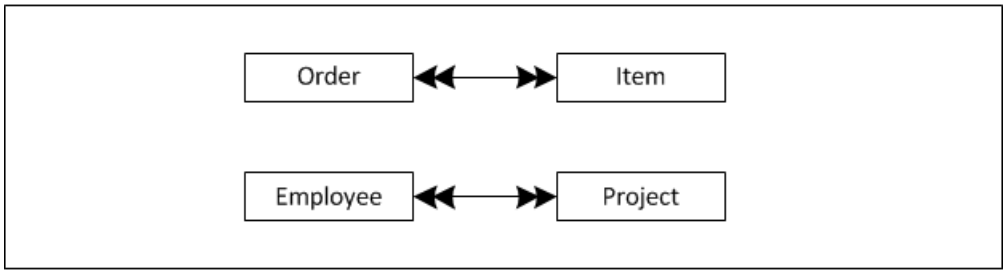
However, the business rule might be that for one customer there can be zero-or-many orders, one student can take zero-or-many courses, and a sales representative can have zero-or-many customer accounts. This relationship is described as a zero-or-many relationship.

Many-to-many relationship

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. The following figure shows examples of:

- An order can contain many items, and an item can appear in many different orders
- An employee can work on many projects, and a project can have many employees working on it

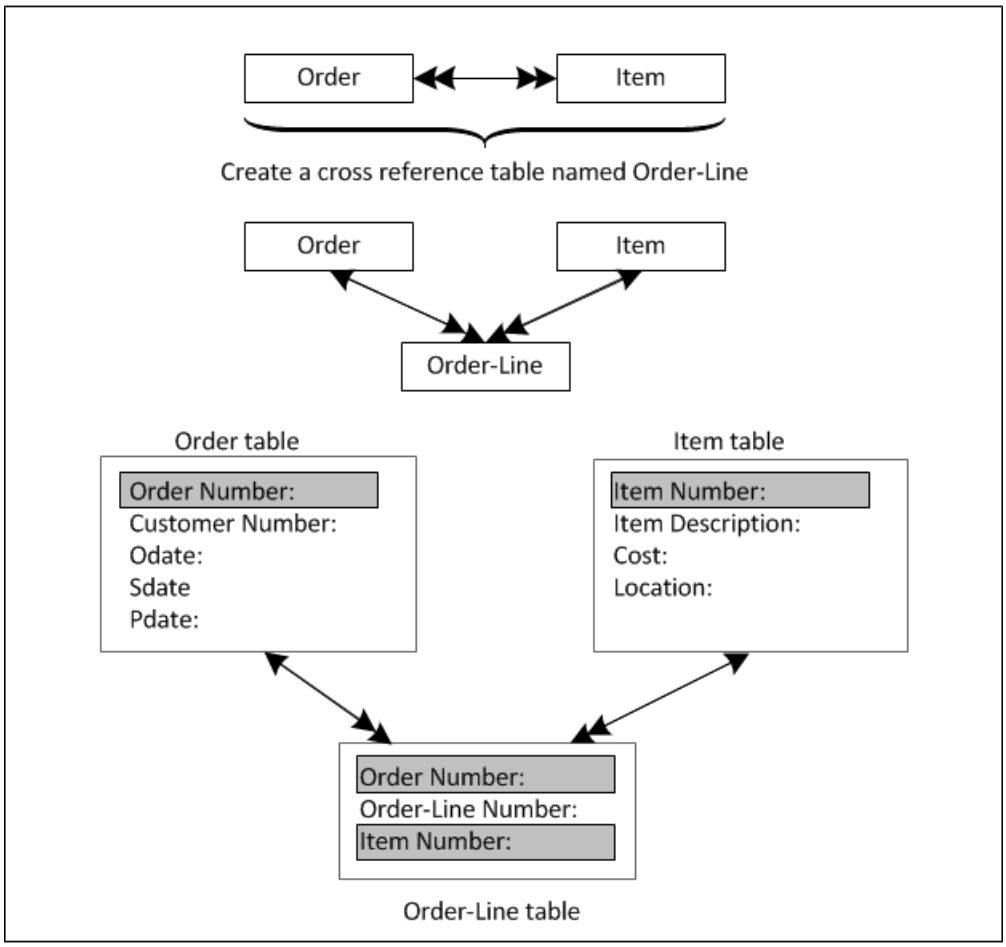
Figure 7: Examples of the many-to-many relationship



Accessing information in tables with a many-to-many relationship is difficult and time consuming. For efficient processing, you can convert the many-to-many relationship tables into two one-to-many relationships by connecting these two tables with a cross-reference table that contains the related columns.

For example, to establish a one-to-many relationship between Order and Item tables, create a cross-reference table Order-Line, as shown in the following figure. The Order-Line table contains both the Order Number and the Item Number. Without this table, you would have to store repetitive information or create multiple columns in both the Order and Item tables.

Figure 8: Using a cross-reference table to relate Order and Item tables



Normalization

Normalization is an iterative process during which you streamline your database to reduce redundancy and increase stability. During the normalization process, you determine in which table a particular piece of data belongs based on the data itself, its meaning to your business, and its relationship to other data. Normalizing your database results in a data-driven design that is more stable over time.

Normalization requires that you know your business and know the different ways you want to relate the data in your business. When you normalize your database, you eliminate columns that:

- Contain more than one value
- Are duplicates or repeat
- Do not describe the table in which they currently reside
- Contain redundant data
- Can be derived from other columns

The result of each iteration of the normalization process is a table that is in a *normal form*. After one complete iteration, your table is said to be in first normal form; after two, second normal form; and so on. The sections that follow describe the rules for the first, second, and third normal forms.

A perfectly normalized database represents the most stable data-driven design, but it might not yield the best performance. Increasing the number of tables and keys, generally leads to higher overhead per query. If performance degrades due to normalization, you should consider denormalizing your data. See the Denormalization on page 35 for more information.

First normal form

The first rule of normalization is that you must remove duplicate columns or columns that contain more than one value to a new table. The columns of a table in the first normal form have these characteristics:

- They contain only one value
- They occur once and do not repeat

First, examine an un-normalized Customer table, as shown in the following figure.

Table 2: Un-normalized Customer table with several values in a column

Cust Num	Name	Street	Order Number
101	Jones, Sue	2 Mill Ave.	M31, M98, M129
102	Hand, Jim	12 Dudley St.	M56
103	Lee, Sandy	45 School St.	M37, M40
104	Tan, Steve	67 Main St.	M41

Here, the Order Number column has more than one entry. This makes it very difficult to perform even the simplest tasks, such as deleting an order, finding the total number of orders for a customer, or printing orders in sorted order. To perform any of those tasks, you need a complex algorithm to examine each value in the Order Number column for each row. You can eliminate the complexity by updating the table so that each column in a table consists of exactly one value.

The following figure shows the same Customer table in a different un-normalized format which contains only one value per column.

Table 3: Un-normalized Customer table with multiple duplicate columns

Cust Num	Name	Street	Order Number1	Order Number2	Order Number3
101	Jones, Sue	2 Mill Ave.	M31	M98	M129
102	Hand, Jim	12 Dudley St.	M56	Null	Null
103	Lee, Sandy	45 School St.	M37	M140	Null
104	Tan, Steve	67 Main St.	M41	Null	Null

Here, instead of a single Order Number column, there are three separate but duplicate columns for multiple orders. This format is also not efficient. What happens if a customer has more than three orders? You must either add a new column or clear an existing column value to make a new entry. It is difficult to estimate a reasonable maximum number of orders for a customer. If your business is brisk, you might have to create 200 Order Number columns for a row. But if a customer has only 10 orders, the database will contain 190 null values for this customer.

Furthermore, it is difficult and time consuming to retrieve data with repeating columns. For example, to determine which customer has Order Number M98, you must look at each Order Number column individually (all 200 of them) in every row to find a match.

To reduce the Customer table to the first normal form, split it into two smaller tables, one table to store only Customer information and another to store only Order information. Table 4: Customer table reduced to first normal form on page 32 shows the normalized Customer table, and Table 5: Order table created when normalizing the Customer table on page 32 shows the new Order table.

Table 4: Customer table reduced to first normal form

Cust Num (Primary key)	Name	Street
101	Jones, Sue	2 Mill Ave.
102	Hand, Jim	12 Dudley St.
103	Lee, Sandy	45 School St.
104	Tan, Steve	67 Main St.

Table 5: Order table created when normalizing the Customer table

Order Number (Primary key)	Cust Num (Foreign key)
M31	101
M98	101
M129	101
M56	102
M37	103
M140	103
M41	104

There is now only one instance of a column in the Customer and Order tables, and each column contains exactly one value. The Cust Num column in the Order table relates to the Cust Num column in the Customer table.

A table that is normalized to the first normal form has these advantages:

- It allows you to create any number of orders for each customer without having to add new columns.
- It allows you to query and sort data for orders very quickly because you search only one column—Order Number.

- It uses disk space more efficiently because no empty columns are stored.

Second normal form

The second rule of normalization is that you must move those columns that do not depend on the primary key of the current table to a new table. A table is in the second normal form when it is in the first normal form and only contains columns that give you information about the key of the table.

The following table shows a Customer table that is in the first normal form because there are no duplicate columns, and every column has exactly one value.

Table 6: Customer table with repeated data

Cust Num	Name	Street	Order Number	Order Date	Order Amount
101	Jones, Sue	2 Mill Ave.	M31	3/19/05	$400.87
101	Jones, Sue	2 Mill Ave.	M98	8/13/05	$3,000.90
101	Jones, Sue	2 Mill Ave.	M129	2/9/05	$919.45
102	Hand, Jim	12 Dudley St.	M56	5/14/04	$1,000.50
103	Lee, Sandy	45 School St.	M37	12/25/04	$299.89
103	Lee, Sandy	45 School St.	M140	3/15/05	$299.89
104	Tan, Steve	67 Main St.	M41	4/2/04	$2,300.56

However, the table is not normalized to the second rule because it has these problems:

- The first three rows in this table repeat the same data for the columns Cust Num, Name, and Street. This is *redundant data*.
- If the customer Sue Jones changes her address, you must then update all existing rows to reflect the new address. In this case, you would update three rows. Any row with the old address left unchanged leads to *inconsistent data*, and your database will lack *integrity*.
- You might want to trim your database by eliminating all orders before November 1, 2004, but in the process, you also lose all the customer information for Jim Hand and Steve Tan. The unintentional loss of rows during an update operation is called an *anomaly*.

To resolve these problems, you must move data. Note that Table 6: Customer table with repeated data on page 33 contains information about an individual customer, such as Cust Num, Name, and Street, that remains the same when you add an order. Columns like Order Num, Order Date, and Order Amount do not pertain to the customer and do not depend on the primary key Cust Num. They should be in a different table. To reduce the Customer table to the second normal form, move the Order Date and Order Amount columns to the Order tables, as shown in Table 7: Customer table on page 33 and Table 8: Order table on page 34.

Table 7: Customer table

Cust Num (Primary key)	Name	Street
101	Jones, Sue	2 Mill Ave.

Cust Num (Primary key)	Name	Street
102	Hand, Jim	12 Dudley St.
103	Lee, Sandy	45 School St.
104	Tan, Steve	67 Main St.

Table 8: Order table

Order Number (Primary key)	Order Date	Order Amount	Cust Num (Foreign key)
M31	3/19/05	$400.87	101
M98	8/13/05	$3,000.90	101
M129	2/9/05	$919.45	101
M56	5/14/04	$1,000.50	102
M37	12/25/04	$299.89	103
M140	3/15/05	$299.89	103
M41	4/2/04	$2,300.56	104

The Customer table now contains only one row for each individual customer, while the Order table contains one row for every order, and the Order Number is its primary key. The Order table contains a common column, Cust Num, that relates the Order rows with the Customer rows.

A table that is normalized to the second normal form has these advantages:

- It allows you to make updates to customer information in just one row.
- It allows you to delete customer orders without eliminating necessary customer information.
- It uses disk space more efficiently because no repeating or redundant data is stored.

Third normal form

The third rule of normalization is that you must remove columns that can be derived from existing columns. A table is in the third normal form when it contains only independent columns, that is, columns not derived from other columns.

The following table shows an Order table with a Total After Tax column that is calculated from adding a 10% tax to the Order Amount column.

Table 9: Order table with derived column

Order Number (Primary key)	Order Date	Order Amount	Total After Tax	Cust Num (Foreign key)
M31	3/19/05	$400.87	$441.74	101

Order Number (Primary key)	Order Date	Order Amount	Total After Tax	Cust Num (Foreign key)
M98	8/13/05	$3,000.90	$3,300.99	101
M129	2/9/05	$919.45	$1011.39	101
M56	5/14/04	$1,000.50	$1,100.55	102
M37	12/25/04	$299.89	$329.87	103
M140	3/15/04	$299.89	$329.87	103
M41	4/2/04	$2,300.56	$2,530.61	104

To reduce this table to the third normal form, eliminate the Total After Tax column because it is a dependent column that changes when the Order Amount or tax changes. For your report, you can create an algorithm to obtain the amount for Total After Tax. You need only keep the source value because you can always derive dependent values. Similarly, if you have an Employee table, you do not need to include an Age column if you already have a Date of Birth column, because you can always calculate the age from the date of birth.

A table that is in the third normal form gives you these advantages:

- It uses disk space more efficiently because no unnecessary data is stored
- It contains only the necessary columns because superfluous columns are removed

Although a database normalized to the third normal form is desirable because it provides a high level of consistency, it might impact performance when you implement the database. If this occurs, consider denormalizing these tables.

Denormalization

Denormalizing a database means that you reintroduce redundancy into your database to meet processing requirements.

To reduce Table 9: Order table with derived column on page 34 to the third normal form, the Total After Tax column was eliminated because it contained data that can be derived. However, when data access requirements are considered, you discover that this data is constantly used. Although you can construct the Total After Tax value, your customer service representatives need this information immediately, and you do not want to have to calculate it every time it is needed. If it is kept in the database, it is always available on request. In this instance, the performance outweighs other considerations, so you denormalize the data by including the derived field in the table.

Defining indexes

An index on a database table speeds up the process of searching and sorting rows. Although it is possible to search and sort data without using indexes, indexes generally speed up data access. Use them to avoid or limit row scanning operations and to avoid sorting operations. If you frequently search and sort row data by particular columns, you might want to create indexes on those columns. Or, if you regularly join tables to retrieve data, consider creating indexes on the common columns.

On the other hand, indexes consume disk space and add to the processing overhead of many data operations including data entry, backup, and other common administration tasks. Each time you update an indexed column, OpenEdge updates the index, and related indexes as well. When you create or delete a row, OpenEdge updates each index on the affected tables.

As you move into the details of index design, remember that index design is not a once-only operation. It is a process, and it is intricately related to your coding practices. Faulty code can undermine an index scheme, and masterfully coded queries can perform poorly if not properly supported by indexes. Therefore, as your applications develop and evolve, your indexing scheme might need to evolve as well. The following sections discuss indexes in detail:

- Indexing basics on page 36
- Choosing which tables and columns to index on page 40
- Indexes and ROWIDs on page 41
- Calculating index size on page 41
- Eliminating redundant indexes on page 42
- Deactivating indexes on page 43

Indexing basics

This section explains the basics of indexing, including:

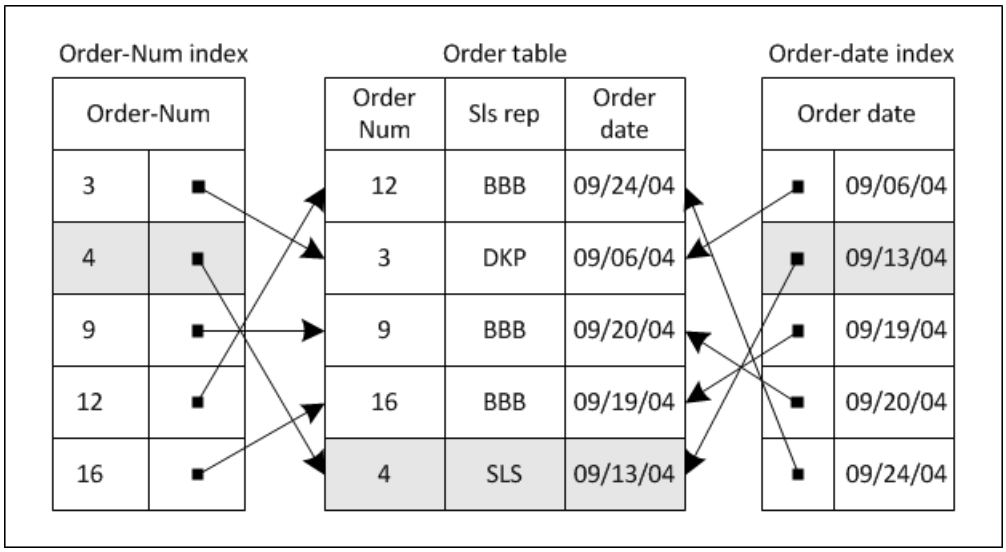
- How indexes work on page 36
- Reasons for defining an index on page 37
- Sample indexes on page 38
- Disadvantages of defining an index on page 40

How indexes work

A database index works like a book index. To look up a topic, you scan the book index, locate the topic, and turn to the pages where the information resides. The index itself does not contain the information; it only contains page numbers that direct you to the pages where the information resides. Without an index, you would have to search the entire book, scanning each page sequentially.

Similarly, if you ask for specific data from a database, the database engine uses an index to find the data. An index contains two pieces of information—the index key and a row pointer that points to the corresponding row in the main table. The following figure illustrates this using the Order table from the Sports 2000 database.

Figure 9: Indexing the Order table



Index table entries are always sorted in numerical, alphabetical, or chronological order. Using the pointers, the system can then access data rows directly, and in the sort order specified by the index.

Every table should have at least one index, the *primary index*. When you create the first index on any table, OpenEdge assumes it is the primary index and sets the Primary flag accordingly. In the above figure, the Order-Num index is the primary index.

Reasons for defining an index

There are four benefits to defining an index for a table:

- **Direct access and rapid retrieval of rows**.

  The rows of the tables are physically stored in the sequence the users enter them into the database. If you want to find a particular row, the database engine must scan every individual row in the entire table until it locates one or more rows that meet your selection criteria. Scanning is inefficient and time consuming, particularly as the size of your table increases. When you create an index, the index entries are stored in an ordered manner to allow for fast lookup.

  For example, when you query for order number 4, OpenEdge does not go to the main table. Instead, it goes directly to the Order-Num index to search for this value. OpenEdge uses the pointer to read the corresponding row in the Order table. Because the index is stored in numerical order, the search and retrieval of rows is very fast.

  Similarly, having an index on the date column allows the system to go directly to the date value that you query (for example, 9/13/04). The system then uses the pointer to read the row with that date in the Order table. Again, because the date index is stored in chronological order, the search and retrieval of rows is very fast.

- **Automatic ordering of rows**.

  An index imposes an order on rows. Since an index automatically sorts rows sequentially (instead of the order in which the rows are created and stored on the disk), you can get very fast responses for *range* queries. For example, when you query, "Find all orders with dates from 09/6/04 to 09/20/04," all the order rows for that range appear in chronological order.

Note: Although an index imposes order on rows, the data stored on disk is in the order in which it was created. You can have multiple indexes on a table, each providing a different sort ordering, and the physical storage order is not controlled by either of the indexes.

- **Enforced uniqueness**.

  When you define a unique index for a table, the system ensures that no two rows can have the same value for that index. For example, if order-num 4 already exists and you attempt to create an order with order-num 4, you get an error message stating that 4 already exists. The message appears because order-num is a unique index for the order table.

- **Rapid processing of inter-table relationships**.

  Two tables are related if you define a column (or columns) in one table that you use to access a row in another table. If the table you access has an index based on the corresponding column, then the row access is much more efficient. The column you use to relate two tables does not need to have the same name in both tables.

Sample indexes

The following figure lists some indexes defined in the Sports 2000 database, showing why the index is defined.

Table 10: Reasons for defining some Sports 2000 database indexes

Table	Index name	Index column(s)	Primary	Unique
Customer	cust-num	cust-num	YES	YES
–	**Why the index is defined:** • Rapid access to a customer given a customer's number. • Reporting customers in order by number. • Ensuring that there is only one customer row for each customer number (uniqueness). • Rapid access to a customer from an order, using the customer number in the order row.			
	name	name	NO	NO
	Why the index is defined: • Rapid access to a customer given a customer's name. • Reporting customers in order by name.			
	zip	zip	NO	NO
	Why the index is defined: • Rapid access to all customers with a given zip code or in a zip code range. • Reporting customers in order by zip code, perhaps for generating mailing lists.			
Item	item-num	item-num	YES	YES

Table	Index name	Index column(s)	Primary	Unique
–	**Why the index is defined:** • Rapid access to an item given an item number. • Reporting items in order by number. • Ensuring that there is only one item row for each item number (uniqueness). • Rapid access to an item from an order-line, using the item-num column in the order-line row.			
Order-line	order-line	order-num line-num	YES	YES
–	**Why the index is defined:** • Ensuring that there is only one order-line row with a given order number and line number. The index is based on both columns together since neither column alone needs to be unique. • Rapid access to all of the order-lines for an order, ordered by line number.			
	item-num	item-num	NO	NO
	Why the index is defined: • Rapid access to all the order-lines for a given item.			
Order	order-num	order-num	YES	YES

<table>
<tr><th>Table</th><th>Index name</th><th>Index column(s)</th><th>Primary</th><th>Unique</th></tr>
<tr><td>–</td><td colspan="4">Why the index is defined:
• Rapid access to an order given an order number.
• Reporting orders in order by number.
• Ensuring that there is only one order row for each order number (uniqueness).
• Rapid access to an order from an order-line, using the order-num column in the order-line row.</td></tr>
<tr><td></td><td>cust-order</td><td>cust-num
order-num</td><td>NO</td><td>YES</td></tr>
<tr><td></td><td colspan="4">Why the index is defined:
• Rapid access to all the orders placed by a customer. Without this index, all of the records in the order file would be examined to find those having a particular value in the cust-num column.
• Ensuring that there is only one row for each customer/order combination (uniqueness).
• Rapid access to the order numbers of a customer's orders.</td></tr>
<tr><td></td><td>order-date</td><td>order-date</td><td>NO</td><td>NO</td></tr>
<tr><td></td><td colspan="4">Why the index is defined:
• Rapid access to all the orders placed on a given date or in a range of dates.</td></tr>
</table>

Disadvantages of defining an index

Even though indexes are beneficial, there are two things to remember when defining indexes for your database:

- Indexes take up disk space.(See the Calculating index size on page 41)
- Indexes can slow down other processes. When the user updates an indexed column, OpenEdge updates all related indexes as well, and when the user creates or deletes a row, OpenEdge changes all the indexes for that table.

Define the indexes that your application requires, but avoid indexes that provide little benefit or are infrequently used. For example, unless you display data in a particular order frequently (such as by zip code), then sorting the data when you display it is more efficient than defining an index to do automatic sorting.

Choosing which tables and columns to index

If you perform frequent adds, deletes, and updates against a small table, you might not want to index it because of slower performance caused by the index overhead. However, if you mostly perform retrievals, then it is useful to create an index for the table. You can index the columns that are retrieved most often, and in the order they are most often retrieved.

You do not have to create an index if you are retrieving a large percentage of the rows in your database (for example, 19,000 out of 20,000 rows) because it is more efficient to scan the table. However, it is worth the effort to create an index to retrieve a very small number of rows (for example, 100 out of 20,000) because then OpenEdge only scans the index table instead of the entire table.

Indexes and ROWIDs

An index is a list of index values and row IDs (ROWIDs). *ROWIDs* are physical pointers to the database tables that give you the fastest access to rows. ROWIDs do not change during the life of a row—they only change when you dump and reload a database. If you delete a row and create a new identical row, it might have a different ROWID. The database blocks of an index are organized into a tree structure for fast access. OpenEdge locates rows within an index by traversing the index tree. Once a row is located, the ROWID accesses the data. OpenEdge does not lock the whole index tree while it looks for the row, it only locks the block that contains the row. Therefore, other users can simultaneously access rows in the same database.

Calculating index size

You can estimate the approximate maximum amount of disk space occupied by an index by using this formula:

```
Number of rows * (7 + number of columns in index + index column storage) * 2
```

For example, if you have an index on a character column with an average of 21 characters for column index storage and there are 500 rows in the table, the index size is:

```
500 * (7 + 1 + 21) * 2 = 29,000 bytes
```

The size of an index is dependent on four things:

- The number of entries or rows.
- The number of columns in the key.
- The size of the column values, that is, the character value "abcdefghi" takes more space than "xyz." In addition, special characters and mulit-byte Unicode characters take even more space.
- The number of similar key values.

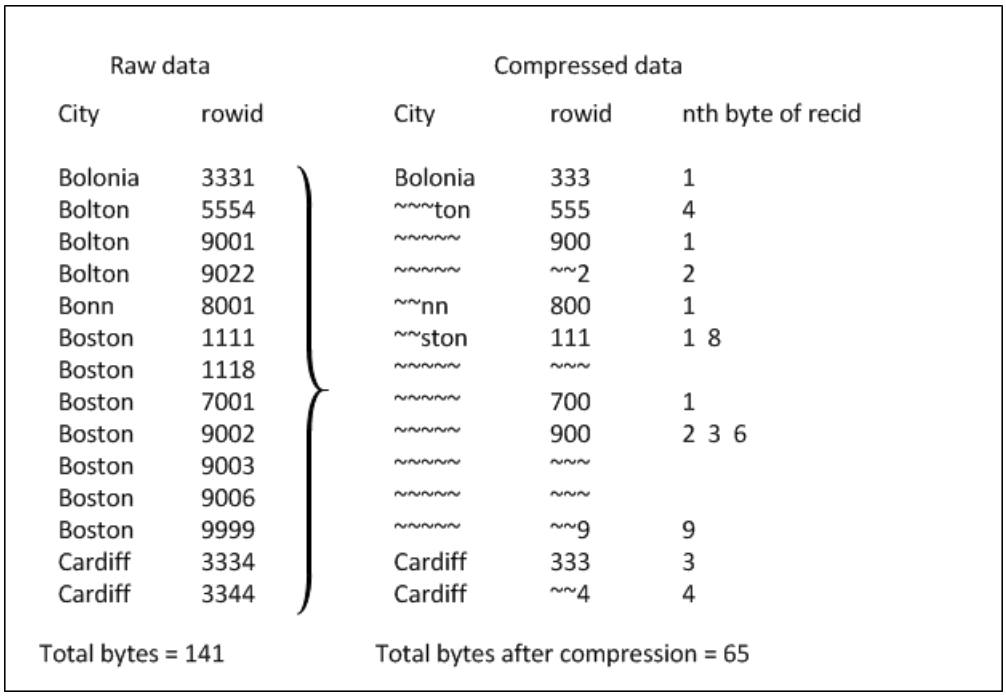
You will never reach the maximum because OpenEdge uses a data compression algorithm to reduce the amount of disk space an index uses. In fact, an index uses on average about 20% to 60% less disk space than the maximum amount you calculated using the previously described formula.

The amount of data compressed depends on the data itself. OpenEdge compresses identical leading data and collapses trailing entries into one entry. Typically non-unique indexes get better compression than unique indexes.

Note: All key values are compressed in the index, eliminating as many redundant bytes as possible.

The following figure shows how OpenEdge compresses data.

Figure 10: Data compression



The City index is stored by city and by ROWID in ascending order. There is no compression for the very first entry "Bolonia." For subsequent entries, OpenEdge eliminates any characters that are identical to the leading characters of Bolonia. Therefore, for the second entry, "Bolton," it is not necessary to save the first three characters "Bol" since they are identical to leading characters of Bolonia. Instead, Bolton compresses to "ton." Subsequently, OpenEdge does not save redundant occurrences of Bolton. Similarly, the first two characters of "Bonn" and "Boston" ("Bo") are not saved.

For ROWIDs, OpenEdge eliminates identical leading digits. It saves the last digit of the ROWID separately and combines ROWIDs that differ only by the last digit into one entry. For example, OpenEdge saves the leading three digits of the first ROWID 333 under ROWID, and saves the last digit under nth byte. Go down the list and notice that the first occurrence of Boston has a ROWID of 1111, the second has a ROWID of 1118. Since the leading three digits (111) of the second ROWID are identical to the first one, they are not saved; only the last digit (8) appears in the index.

Because of the compression feature, OpenEdge can substantially decrease the amount of space indexes normally use. In the above figure, only 65 bytes are used to store the index that previously took up 141 bytes. That is a saving of approximately 54%. As you can see, the amount of disk space saved depends on the data itself, and you can save the most space on the non-unique indexes.

Eliminating redundant indexes

If two indexes share the same leading, contiguous components for the same table, they are redundant. Redundant indexes occupy space and slow down performance.

Deactivating indexes

Indexes that you seldom use can impair performance by causing unnecessary overhead. If you do not want to delete an index that is seldom used, then you should deactivate it. Deactivating an index eliminates the processing overhead associated with the index, but it does not free up the index disk space. For information on how to deactivate indexes, see *OpenEdge Data Management: Database Administration*. To learn deactivate indexes using SQL, see *OpenEdge Data Management: SQL Reference.*

Physical database design

The *physical database design* is a refinement of the logical design. In this phase, you examine how the user will access the database. During this phase, ask yourself:

- What data will I commonly use?
- Which columns in the table should I index based on data access?
- Where should I build in flexibility and allow for growth?
- Should I denormalize the database to improve performance?

At this stage you might denormalize the database to meet performance requirements.

Once you determine the physical design of your database, you must determine how to map the database to your hardware. Maintaining the physical database is the primary responsibility of a database administrator. The OpenEdge RDBMS on page 45 discusses the physical storage of the OpenEdge database.

3

OpenEdge RDBMS

When administering an OpenEdge database, it is important to understand its architecture and the configuration options it supports. This chapter presents an overview of the OpenEdge Release 10 database.

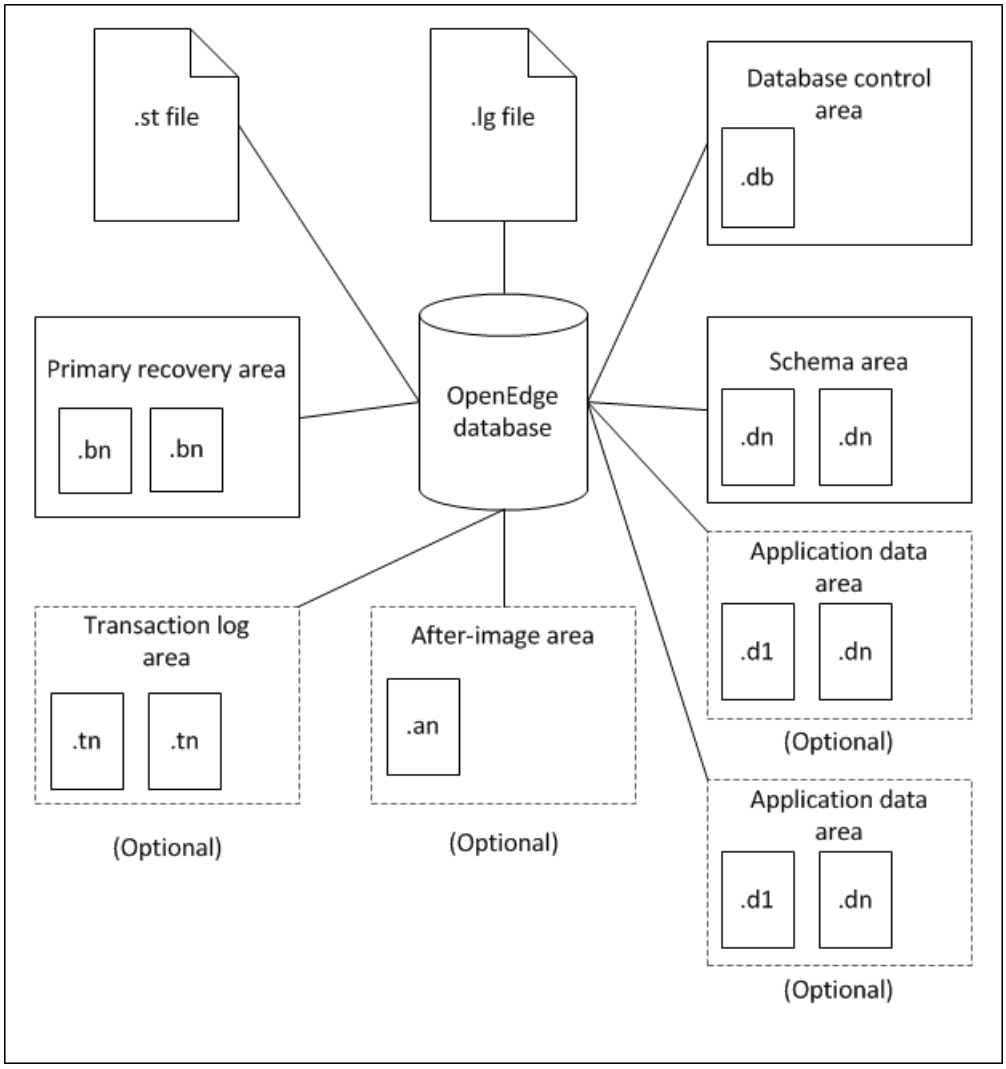
For details, see the following topics:

- OpenEdge database file structure
- OpenEdge architecture
- Storage design overview
- Determining configuration options
- Relative- and absolute-path databases

OpenEdge database file structure

The OpenEdge database contains more than data. The following figure illustrates the components of an OpenEdge database. Descriptions of the files follow the table.

Figure 11: OpenEdge RDBMS



As shown in the above figure, a typical OpenEdge database consists of:

- A structure description (`.st`) file, which defines the structure of the database. The `.st` file is a text file with a `.st` filename extension. The administration utility `PROSTRCT CREATE` uses the information in the `.st` file to create the areas and extents of the database. It is the database administrator's responsibility to create the `.st` file. For detailed information about structure description files, see *OpenEdge Data Management: Database Management*.
- A log (`.lg`) file, which is a text file. The `.lg` file contains a history of significant database events, including server startup and shutdown, client login and logout, and maintenance activity.
- One database (`.db`) control area, which is a binary file containing a database structure extent. The control area and its `.db` file act as a table of contents for the database engine, listing the name and location of every area and extent in the database.

- One primary recovery (before-image) area, which contains one or more extents with a .b*n* filename extension. The .b*n* files store notes about data changes. In the event of hardware failure, the database engine uses these notes to undo any incomplete transactions and maintain data integrity.
- One schema area, which contains at least one variable-length extent with a .d*n* filename extension. The schema area contains the master and sequence blocks, as well as schema tables and indexes. Progress Software Corporation recommends that you place all your application data in additional data areas, but if you do not create application data areas, the schema area contains your user data.
- Optionally, application data areas, which contain at least one variable-length extent with a .d*n* filename extension. Application data areas contain user data, indexes, CLOBs and BLOBs.
- Optionally, one after-image area when after-imaging is enabled. The after-image area can contain many fixed-length and variable-length extents with the .a*n* filename extension. In the event of hardware failure, the database engine uses the .a*n* file and the most recent backup to reconstruct the database.
- Optionally, one transaction log area when two-phase commit is in use. The transaction log area contains one or more fixed-length extents with the .t*n* filename extension; variable-length extents are not allowed. The transaction log lists committed two-phase commit transactions.

An OpenEdge database is collectively all the files described above: the control area, schema area, data areas, recovery files, and log files. You should treat these files as an indivisible unit. For example, the phrase "back up the database" means "back up the database data: .db, .d*n*, .lg, .d*n*, .t*n*, .a*n*, and .b*n* files together."

Other database-related files

While maintaining your database, you might encounter files with the extensions listed in the following table:

Table 11: Other database-related files

File extension	Description
.abd	Archived binary dump of audit data
.bd	Binary dump file (table-based)
.blb	Object data file for a BLOB or CLOB
.cf	Schema cache file
.cp	Binary compiled code page information file
.cst	Client database-request statement cache file
.dfsql	SQL data definition file
.dsql	SQL table dump file
.d	ABL table dump file
.df	ABL data definition file
.fd	ABL bulk loader description file

File extension	Description
.ks	OpenEdge Key store for encryption-enabled databases
`.lic`	License file
`.lk`	Lock file
`.repl.properties`	OpenEdge® Replication properties file
`.repl.recovery`	OpenEdge Replication file for maintaining replication state information
`.rpt`	License usage report file

OpenEdge architecture

The architecture for the OpenEdge database is known as Type II. Prior to Release 10, the supported architecture was known as Type I. OpenEdge Release 10 continues to support the Type I architecture, but since Type II offers significant advantages in both storage efficiency and data access performance, you should consider migrating your legacy databases to a Type II storage architecture.

The Type II architecture contains these elements, as described in the following sections:

- Storage areas on page 48
- Extents on page 50
- Clusters on page 50
- Blocks on page 51

The elements are defined in your database structure definition file. For details on the structure definition file, see *OpenEdge Data Management: Database Administration*.

Storage areas

A storage area is a set of physical disk files, and it is the largest physical unit of a database. With storage areas, you have physical control over the location of database objects: you can place each database object in its own storage area, you can place many database objects in a single storage area, or you can place objects of different types in the same storage area. Even though you can extend a table or index across multiple extents, you cannot split them across storage areas.

Certain storage areas have restrictions on the types of extents they support. See the Extents on page 50 for a definition of extents. The transaction log storage area, used for two-phase commit, uses only fixed-length extents but it can use more than one. The other storage areas can use many extents but they can have only one variable-length extent, which must be the last extent.

Storage areas are identified by their names. The number and types of storage areas used varies from database to database. However, all OpenEdge databases must contain a control area, a schema area, and a primary recovery area.

The database storage areas are:

- Control area on page 49

- Schema area on page 49
- Primary recovery area on page 49
- Application data area on page 49
- After-image area on page 49
- Encryption Policy area on page 50
- Audit data and index areas (optional) on page 50
- Transaction log area on page 50

Control area

The control area contains only one variable-length extent: the database structure extent, which is a binary file with a .db extension. The .db file contains the _area table and the _area-extent tables, which list the name of every area in the database, as well as the location and size of each extent.

Schema area

The schema area can contain as many fixed-length extents as needed; however, every schema area should have a variable-length extent as its last extent. The schema area stores all database system and user information, and any objects not assigned to another area. If you choose not to create any optional application data areas, the schema area contains all of the objects and sequences of the database.

Primary recovery area

The primary recovery area can contain as many fixed-length extents as needed, as long as the last extent is a variable length extent. The primary recovery area is also called the before-image area. The files, named .b*n*, record data changes. In the event of a database crash, the server uses the contents of the .b*n* files to perform crash recovery during the next startup of the database. *Crash recovery* is the process of backing out incomplete transactions.

Application data area

The application data storage area contains all application-related database objects. Defining more than one application data area allows you to improve database performance by storing different objects on different disks. Each application data area contains one or more extents with a .d*n* extension.

After-image area

The optional after-image area contains as many fixed-length or variable-length extents as needed. After-image extents are used to apply changes made to a database since the last backup. Enable after-imaging for your database when the risk of data loss due to a system failure is unacceptable.

Encryption Policy area

For databases enabled for transparent data encryption, a dedicated area called the "Encryption Policy Area" is required to hold your encryption policies. The Encryption Policy Area is a specialized Type II application data area. You cannot perform any record operation on the data in the Encryption Policy Area with either an SQL or an ABL client. The Encryption Policy Area contains one or more extents with a `.dn` extension, but it is defined in your structure definition file with an "`e`" token.

Audit data and index areas (optional)

For databases enabled for auditing, specifying an application data area exclusively for audit data is recommended. If you anticipate generating large volumes of audit data, you can achieve better performance by also creating a dedicated area for audit indexes and separating the data and indexes. Both the audit data and audit index areas are application data areas with no special restrictions.

Transaction log area

The transaction log area is required if two-phase commit is used. This area contains one or more fixed-length extents with the `.tn` filename extension; variable-length extents are not allowed.

Guidelines for choosing storage area locations

When choosing the locations for your database storage areas, consider the following:

- Protect against disk failures by creating the after-image storage area on a separate physical disk from the disks that store the database control and primary recovery areas.
- Improve performance by creating the primary recovery area on a separate disk from the disk that stores the database control area and its extents.
- If using two-phase commit, create the transaction log area in the same directory as the database control area to simplify management.

Extents

Extents are disk files that store physical blocks of database objects. Extents make it possible for an OpenEdge database to extend across more than one file system or physical volume.

There are two types of extents: fixed-length and variable-length. With *fixed-length extents* you control how much disk space each extent uses by defining the size of the extent in the `.st` file. *Variable-length extents* do not have a predefined length and can continue to grow until they use all available space on a disk or until they reach the file system's limit on file size.

Clusters

A *cluster* is a contiguous allocation of space for one type of database object. Data clusters reduce fragmentation and enable your database to yield better performance from the underlying file system.

Data clusters are specified on a per-area basis. There is one cluster size for all extents in an area. The minimum size of a data cluster is 8 blocks, but you can also specify larger clusters of 64 or 512 blocks. All blocks within a data cluster contain the same type of object. The high-water mark of an extent is increased by the cluster size.

In the Type I architecture, blocks are laid out one at a time. In the Type II architecture, blocks are laid out in a cluster at one time. With the Type II architecture, data is maintained at the cluster level, and blocks only include data associated with one particular object. Additionally, all blocks of an individual cluster are associated with a particular object. In Release 10 of OpenEdge, existing storage areas use the Type I architecture, as do schema. New storage areas can use the Type II architecture or the Type I architecture. To use the Type II architecture, you must allocate clusters of 8, 64, or 512 blocks to an area. If you do not, you will get the Type I architecture. Cluster sizes are defined in your structure definition file. For details on the structure definition file, see *OpenEdge Data Management: Database Administration*.

Blocks

A block is the smallest unit of physical storage in a database. Many types of database blocks are stored inside the database, and most of the work to store these database blocks happens behind the scenes. However, it is helpful to know how blocks are stored so that you can create the best database layout.

The most common database blocks are divided into three groups:

- Data blocks on page 51
- Index blocks on page 52
- Other block types on page 52

Data blocks

Data blocks are the most common blocks in the database. There are two types of data blocks: RM blocks and RM chain blocks. The only difference between the two is that RM blocks are considered full and RM chain blocks are not full. The internal structure of the blocks is the same. Both types of RM blocks are social. *Social blocks* can contain records from different tables. In other words, RM blocks allow table information (records) from multiple tables to be stored in a single block. In contrast, index blocks only contain index data from one index in a single table.

The number of records that can be stored per block is tunable per storage area. See the Data layout on page 61 for a discussion of calculating optimal records per block settings.

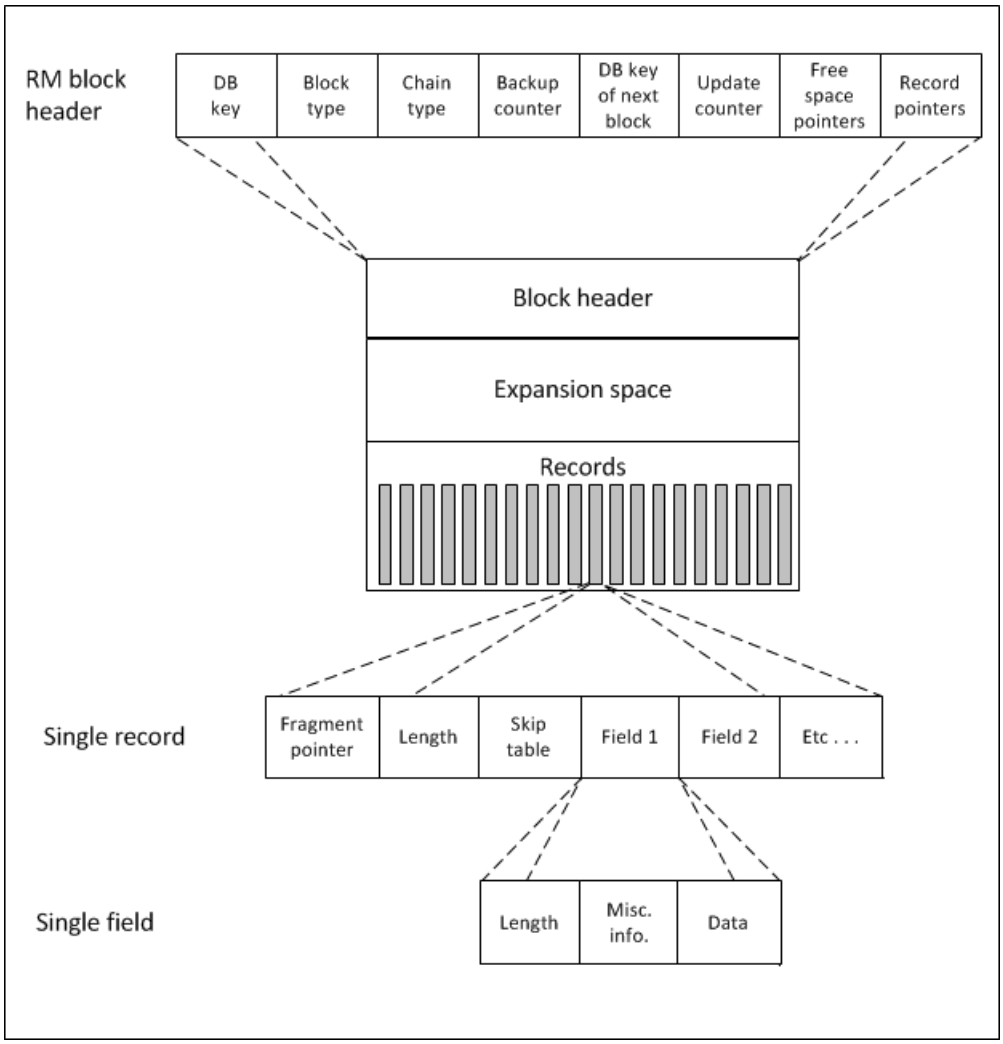
Each RM block contains four types of information:

- Block header
- Records
- Fields
- Free space

The block header contains the address of the block (dbkey), the block type, the chain type, a backup counter, the address of the next block, an update counter (used for schema changes), free space pointers, and record pointers. For a Type I storage area, the block header is 16 bytes in length. For a Type II storage area, the block header is variable; the header of the first and last block in a cluster is 80 bytes, while the header for the remaining blocks in a cluster is 64 bytes. Each record contains a fragment pointer (used by record pointers in individual fields), the Length of the Record field, and the Skip Table field (used to increase field search performance). Each field needs a minimum of 15 bytes for overhead storage and contains a Length field, a Miscellaneous Information field, and data.

The following figure shows the layout of an RM block.

Figure 12: RM Block layout



Index blocks

Index blocks have the same header information as data blocks, and have the same size requirements of 16 bytes for Type I storage areas, and 64 or 80 bytes for Type II storage areas. Index blocks can store the amount of information that can fit within the block, and that information is compressed for efficiency. As stated earlier, index blocks can only contain information referring to a single index.

Indexes are used to find records in the database quickly. Each index in an OpenEdge RDBMS is a structured B-tree and is always in a compressed format. This improves performance by reducing key comparisons. A database can have up to 32,767 indexes. Each B-tree starts at the root. The root is stored in an `_storageobject` record. For the sake of efficiency, indexes are multi-threaded, allowing concurrent access. Rather than locking the whole B-tree, only those nodes that are required by a process are locked.

Other block types

There are other types of blocks that are valuable to understand. These include:

- Master blocks on page 53

- Storage object blocks on page 53
- Free blocks on page 53
- Empty blocks on page 53

Master blocks

The master block contains the same 16-byte header as other blocks, but this block stores status information about the entire database. It is always the first block in the database and it is found in Area 6 (a Type I storage area). This block contains the database version number, the total allocated blocks, time stamps, and status flags. You can retrieve additional information from this block using the Virtual System Table (VST) `_mstrblk`. For more information on VSTs, see *OpenEdge Data Management: Database Administration*.

Storage object blocks

Storage object blocks contain the addresses of the first and last records in every table by each index. If a user runs a program that requests the first or last record in a table, it is not necessary to traverse the index. The database engine obtains the information from the storage object block and goes directly to the record. Because storage object blocks are frequently used, they are pinned in memory. This availability further increases the efficiency of the request.

Free blocks

Free blocks have a header, but no data is stored in the blocks. These blocks can become any other valid block type. These blocks are below the high-water mark. The *high-water mark* is a pointer to the last formatted block within the database storage area. Free blocks can be created by extending the high-water mark of the database, extending the database, or reformatting blocks during an index rebuild. If the user deletes many records, the RM blocks are put on the RM Chain. However, index blocks can only be reclaimed through an index rebuild or an index compress.

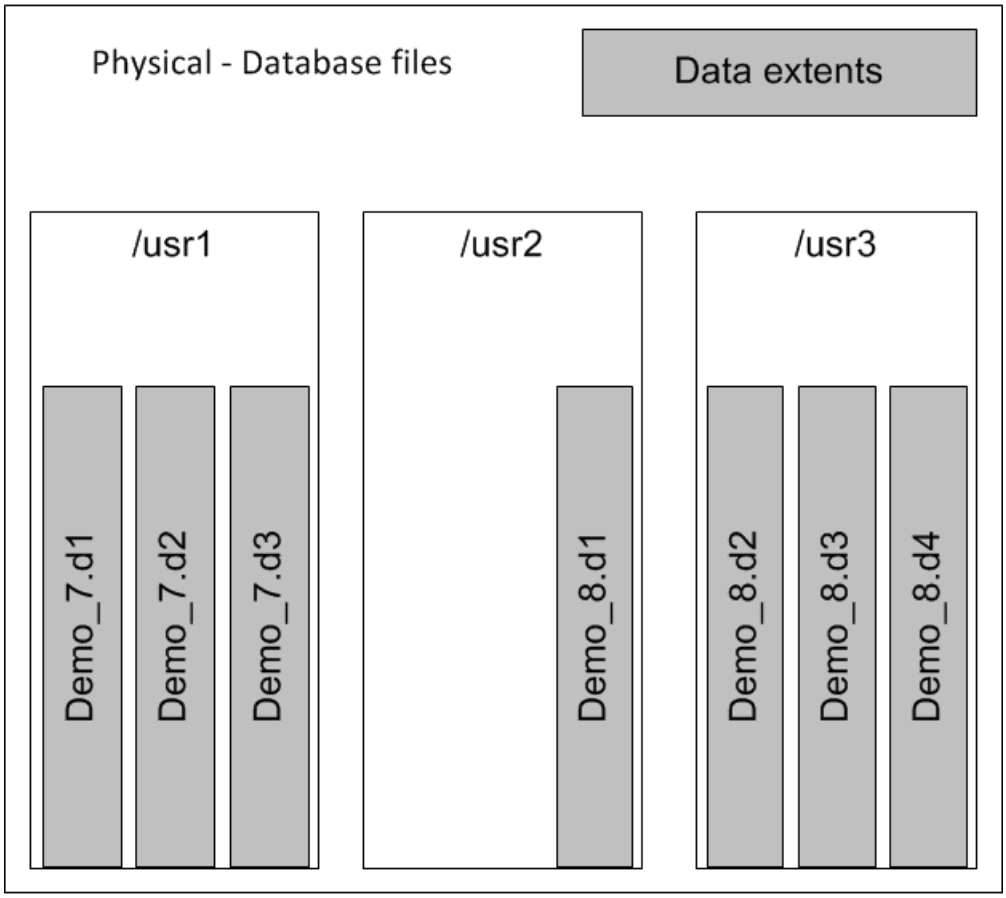
Empty blocks

Empty blocks do not contain header information. These blocks must be formatted prior to use. These blocks are above the high-water mark but below the total number of blocks in the area. The *total blocks* are the total number of allocated blocks for the storage area.

Storage design overview

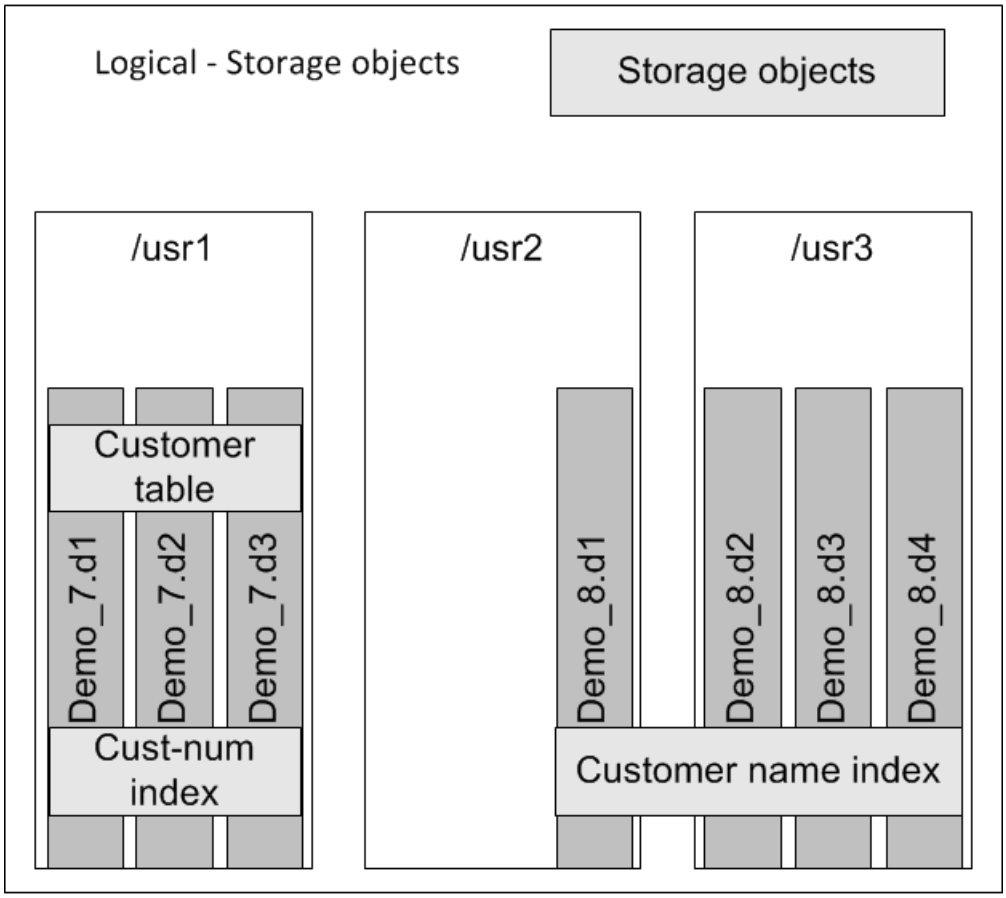
The storage design of the OpenEdge RDBMS is divided into a physical model and a logical model. You manipulate the physical storage model through ABL, OpenEdge SQL, and database administration utility interfaces. The following figure shows how areas can be stored on and span different file slices. Observe that although an area's extents are comprised of many disk files, an extent can only be associated with one storage area.

Figure 13: Physical storage model



The logical storage model overlays the physical model. Logical database objects are described in the database schema and include tables, indexes, and sequences that your application manipulates. The following figure illustrates how logical objects can span physical extents.

Figure 14: Logical storage model



Mapping objects to areas

Creating optimal storage areas and proper relationships between areas and objects enhances your database performance. You can define one object per area, or you can have multiple objects in an area. Use the information in the following table to guide your design decision.

Table 12: Guidelines for storage areas

If your database contains . . .	Then you should . . .
Many large or frequently updated tables	Create a storage area for each table. This will give you easier administration and better performance.
Many small tables	Create storage areas to represent a distinct subject (for example Sales, Inventory, etc.) and assign related tables to that storage area. This will give you easier administration.
A mixture of large and small tables	Combine the above two strategies.

If your database contains . . .	Then you should . . .
Many indexes for large or frequently updated tables	Create a storage area for each index. This will give you flexibility in administration, and the potential for better performance.
Many of small indexes	Create storage areas representing a distinct subject for these indexes. This will give you easier administration.

Determining configuration options

The OpenEdge RDBMS architecture and its wide range of supported system platforms allow flexible configuration options. A very simple configuration might support a single user on a PC system. A complex configuration might include heterogeneous, networked systems supporting hundreds of users accessing multiple databases, including non-OpenEdge databases.

Your business needs determine your choices for the following variables:

- System platform on page 56
- Connection modes on page 56
- Client type on page 57
- Database location on page 57
- Database connections on page 57

The sections that follow explain how these variables affect your configuration.

System platform

The OpenEdge RDBMS provides a multi-threaded database server that can service multiple network clients. Each server can handle many simultaneous requests from clients. The server processes simple requests as a single operation to provide rapid responses, and it divides complex requests into smaller tasks to minimize the impact on other users. OpenEdge supports a variety of hardware and software platforms to suit varying configuration needs.

Connection modes

OpenEdge databases run in one of two *connection modes*: single-user or multi-user. Connection modes control how many users can access a database.

Single-user mode

A database running in *single-user mode* allows only one user to access a specified database at a time. If another user is already accessing a database, you cannot connect to that database from a different session.

Running a database in single-user mode is required when you perform system administration tasks that require exclusive access to the database.

Multi-user mode

A database running in *multi-user mode* allows more than one user to access it simultaneously. A broker coordinates all the database connection requests, and servers retrieve and store data on behalf of the clients. The broker process locks the database to prevent any other broker or single-user process from opening it.

Batch mode

When a client runs in *batch mode*, processing occurs without user interaction. Batch mode is convenient for large-scale database updates or procedures that can run unattended. Both single-user and multi-user processes can run in batch mode. Intensive multi-user batch jobs can degrade the response time for interactive users and should be scheduled to run at a time that will not negatively impact interactive users, such as overnight.

Interactive mode

When a client runs in *interactive mode*, the user interacts directly with an application connected to the database. Both single-user and multi-user processes can run in interactive mode.

Client type

OpenEdge supports both self-service clients and network clients.

Self-service clients

A *self-service client* is a multi-user session that runs on the same machine as the broker. Self-service clients access the database directly through shared memory, rather than through a server. Self-service clients perform server and client functions in one process, because the server code is part of the self-service client process. The database engine provides self-service clients with nearly simultaneous access to the database.

Network clients

A *network client* can either be local or remote, but it cannot connect to a database directly, so it must use a server. Network clients access the database through a server process that the broker starts over a network connection. The network client does not have access to shared memory, and it must communicate with a server process.

Database location

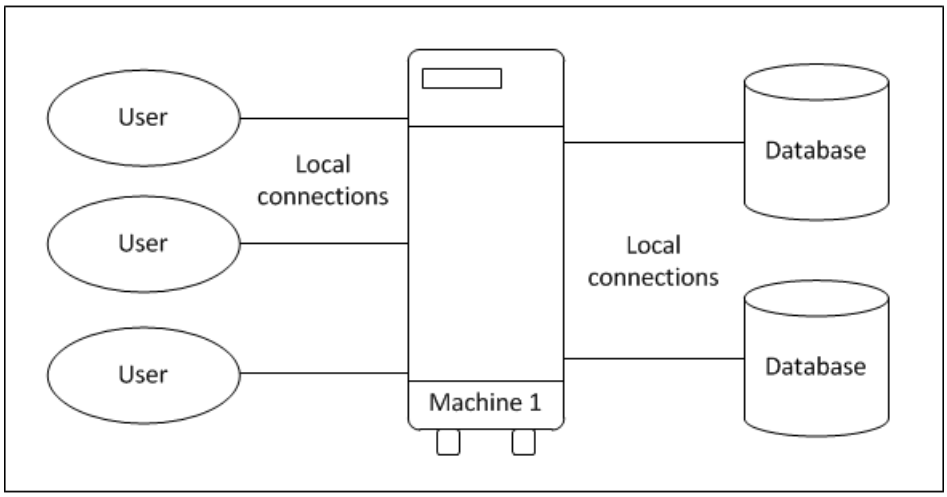
An OpenEdge database can be either local or remote. A database located on the same machine as the application session is local. A database located on a machine that is networked to the machine containing the application session is remote.

Database connections

When the user connects to more than one database, there are two basic configurations:

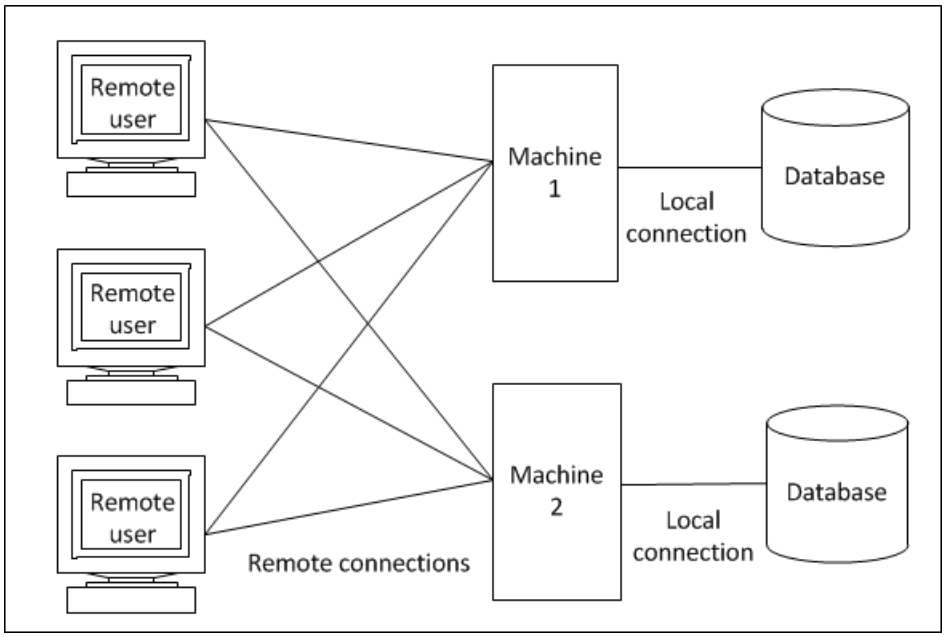
- **Federated —** All databases are local. The following figure illustrates federated database connections.

Figure 15: Federated database configuration



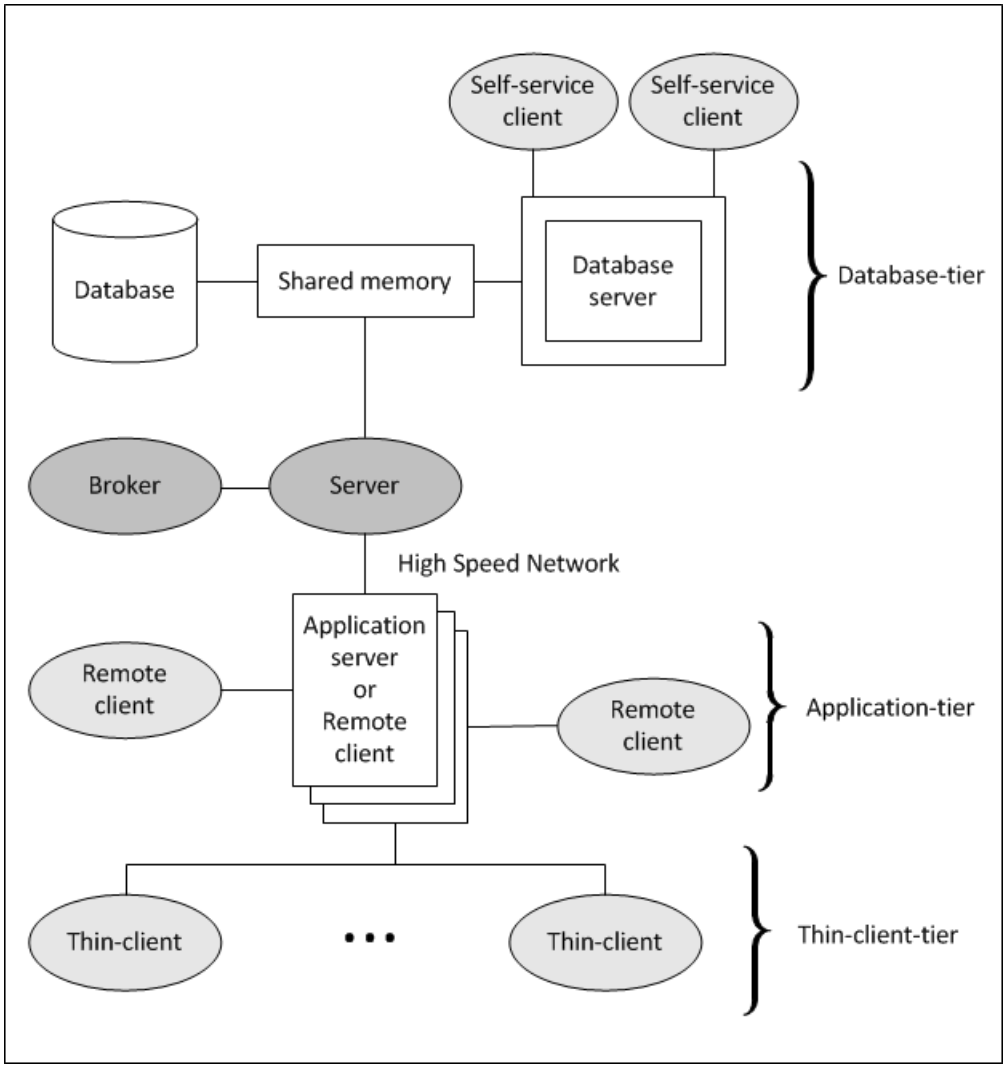
- Distributed — One or more of the databases reside on one or more remote machines in the network, and OpenEdge sessions connect to the database using a single networking protocol. The following figure illustrates distributed database connections.

Figure 16: Distributed database configuration



A *multi-tier configuration* is more complex than the basic federated and distributed models. A multi-tier configuration consists of a database tier that supports self-service clients, an application tier that supports remote clients, and a thin-client tier. Multi-tier configurations might improve system performance for a large installation. The following figure illustrates a three-tier configuration.

Figure 17: Sample multi-tier configuration



The OpenEdge RDBMS architecture provides multiple simultaneous paths to a database. Each self-service client can access the database and service its own requests. Each network server queues and runs requests for one or more network clients. The database broker initializes shared memory and starts a new server for each additional client or set of clients that access the database. By removing the server as a bottleneck, the OpenEdge architecture increases overall performance.

Relative- and absolute-path databases

OpenEdge databases can be either relative-path or absolute-path:

- **Relative-path databases** — Relative-path databases are the simplest form of OpenEdge databases. Relative-path databases are made up of a minimum number of extents that are all stored in the same directory as the database control area. The control area contains relative paths to all the extents. A relative-path database is used in the following situations:

- When the database must be opened across a network for demonstration purposes
- When the database must be copied with OS tools, for example, when installing a demonstration database
- When the database is the master database to be associated with a new application

Use the PRODB utility to create a relative-path database from an empty database. If you use PROSTRCT with the LIST qualifier after creating the relative-path database, you will see a dot at the start of each extent name. The dot indicates a relative path.

Schema must be loaded into an empty relative-path database to make it useful. Any standard technique for loading schema, such as dump and load or PROCOPY, can be used. The database maintains its relative path as long as its structure is not changed. As soon as areas or extents are added, the database becomes an absolute-path database.

- **Absolute-path databases** — Absolute-path databases are used in most production situations. With an absolute-path database, extents associated with the database can be stored anywhere on your system. The control area contains absolute paths to each extent. Absolute-path databases should not be copied using your operating system tools; use PROBKUP and PROCOPY so that all underlying files are properly backed up or copied.

4

Administrative Planning

As a Database Administrator, you maintain and administer an OpenEdge database. A well-planned database simplifies your job by reducing the time spent maintaining the database structure.

For details, see the following topics:

- Data layout
- Database areas
- System resources
- Disk capacity
- Memory usage
- CPU activity
- Tunable operating system resources

Data layout

Proper database design is essential because all data originates from a storage area. Consider the following important facts:

- A database is made up of storage areas.
- Each storage area can contain one or more objects.
- A *database object* is a table or an index or LOB.

- There are other objects, such as sequences and schema. Currently, OpenEdge controls the locations of these objects.
- Each storage area can contain one or more extents or volumes on disk.
- The *extents* are the physical files stored at the operating system level.
- Each extent is made up of blocks, and you can determine the block size for your database. The block sizes that you can choose from are: 1KB, 2KB, 4KB, and 8KB.
- You can have only one block size per database, but each area can have differing numbers of records per block.

There are several things to consider when determining the layout of a database. The first is your mean record size. This is easy to learn if you have an existing OpenEdge database, because this information is included in the output of running the database analysis utility. Other information you must consider is specific to your application and answers the following questions:

- Is the table mostly accessed sequentially or randomly?
- Is the table frequently used, or is it historical and thus used infrequently?
- Do the users access the table throughout the day, or only for reporting?
- Do individual records grow once they are inserted, or are they mostly static in size?

The answers to these questions help determine the size and layout of a database.

Calculating database storage requirements

The OpenEdge database engine stores all database fields and indexes in variable-length format and does not store trailing blanks in character fields or leading zeros in numeric fields. The benefits of this variable-length storage technique are:

- Disk storage is reduced.
- An application is easier to change since you can increase the maximum storage space for a field without copying or restructuring the entire database. You can put more characters into newly created records without affecting existing records.

Formulas for calculating field storage

The following table lists the formulas for calculating the field storage values (in bytes) for different data types.

Table 13: Formulas for calculating field storage

ABL Data type (SQL equivalent)	Value	Field storage in bytes
CHARACTER (VARCHAR)	Character string	1 + *number of characters*, excluding trailing blanks. If the number of characters is greater than 240, add 3 to the number of characters instead of 1.
DECIMAL (DECIMALor NUMERIC)	Zero	1
	Nonzero	2+(# significant digits + 1)/2

ABL Data type (SQL equivalent)	Value	Field storage in bytes
INTEGER (INTEGER)	1 to 127	1
	128 to 32,511	2
	32,512 to 8,323,071	3
	8,323,072 to 2,147,483,647	4
INT64 (BIGINT)	1 to 127	1
	128 to 32,511	2
	32,512 to 8,323,071	3
	8,323,072 to 2,147,483,647	4
	2,147,483,648 to 545,460,846,591	5
	545,460,846,592 to ~139,636,000,000,000	6
	~139,636,000,000,000 to ~35,750,000,000,000,000	7
	Greater than ~35,750,000,000,000,000	8
DATE (DATE)	Date	Same as INTEGER. Dates are stored as an INTEGER representing the number of days since a base date, defined as the origin of the ABL DATE data type.
DATE-TIME	Date and time	DATE + 4
DATE-TIME-TZ	Date, time, and time zone	DATE + 8
LOGICAL (BIT)	False	1
	True	2

Sample to estimate storage requirements

The following example demonstrates how to estimate storage requirements. Consider a database with a single customer table with three fields:

- **Cust-num** — An integer field that is always three digits
- **Name** — A character field containing 12–30 characters, with an average of 21 characters
- **Start-date** — A date field

If the table is indexed on just one field (Name) and you expect to have about 500,000 records in the customer table, the following table lists formulas (and examples) for estimating storage requirements.

Table 14: Calculating database size

Database component	Size
Field storage	= *Cust-num* + *Name* + *Start-date* = 3 + 21 + 3 = 27
Customer table	= *number of records* x *field storage* x 1.5 = 500,000 x 27 x 1.5 = 20,250,000
Name index	= *number of records* x (7 + *number of fields in index* + *index field storage*) x 2 = 500,000 x (7 + 1 + 21) x 2 = 29,000,000

These formulas are conservative and often result in a large estimate of your database size.

Database-related size criteria

When planning the size of your database, use the formulas described in the following table to calculate the approximate amount of disk space (in bytes) required for a database. For limits on any of these elements, see *OpenEdge Data Management: Database Administration*.

Table 15: Formulas for calculating database size

Size	Formula
Database size	*schema size* + *data table size* + *index size*
Schema size	Typically, between 1 and 100MB[1]
Data table size	Sum of individual table sizes
Individual table size	*number of records* x *field storage* x 1.5
Index size	Sum of individual index sizes
Individual index size	*number of records in the table being indexed* x (7 + *number of fields index +field storage*) x 2

Additional disk space is required for the following purposes:

[1] To determine the schema size, load the schema into an empty database and check the size of your database—this is the size of your schema.

- Sorting (allow twice the space required to store the table)
- Temporary storage of the primary recovery (BI) area
- Before-image storage in the primary recovery area
- After-image storage in the after-image storage area

Sizing your database areas

When trying to determine the size of an area, you must look at the makeup of the information being stored in that area; remember, an area can contain one or more tables or indexes. The default area should generally be reserved for schema and sequence definitions, as this will make conversions easier in the future. The first step in this process, if you already have an OpenEdge database, is to do a table analysis using the `PROUTIL` utility. See *OpenEdge Data Management: Database Administration for details*.

The following sample output shows a portion of a table analysis:

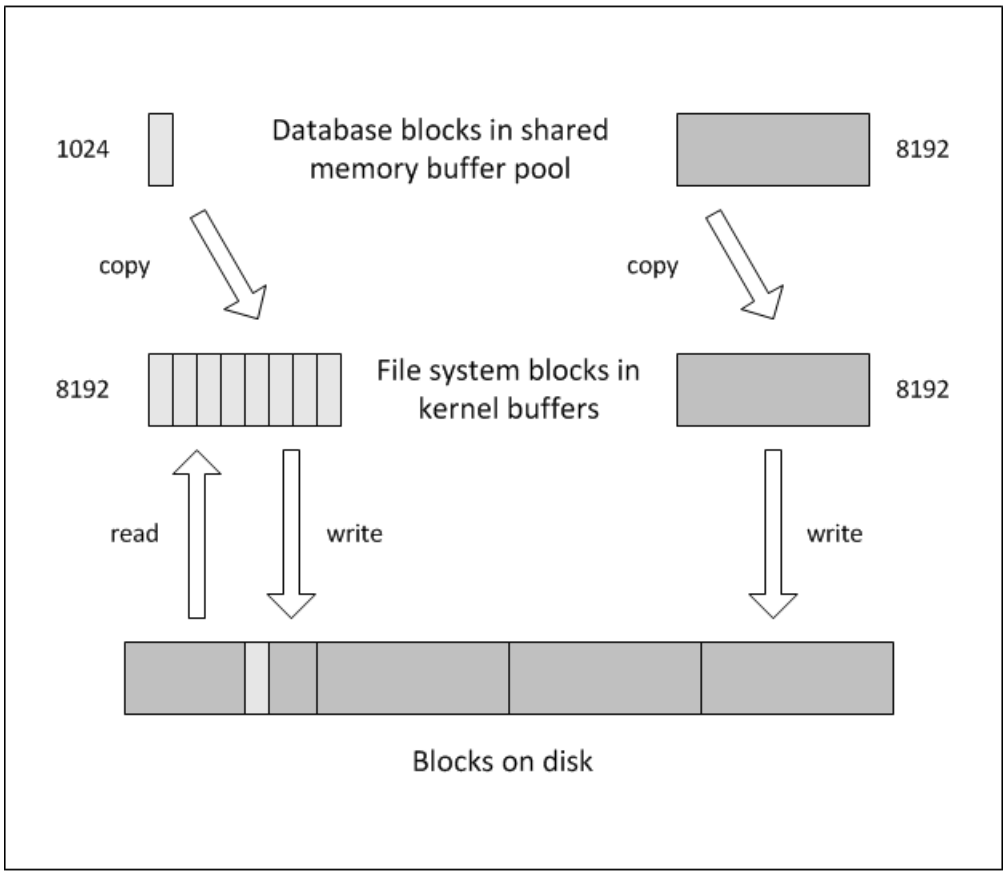
```
              ---Record---              ---Fragment---          ---Scatter---
Table        Records  Bytes      Min  Max     Mean   Count    Factor  Factor
Work Orders  12,383   6,109,746  60   10,518  493    21,131   1.6     4.0
RECORD BLOCK SUMMARY FOR AREA "Inventory" : 8
------------------------------------------------------
                                  -Record Size (B)- ---Fragments--- Scatter
Table               Records    Size  Min   Max  Mean   Count Factor    Factor
PUB.Bin                 770   26.2K    33    35    34     770    1.0       1.4
PUB.InventTrans          75    3.6K    43    51    48      75    1.0       0.9
PUB.Item                 55    7.6K   103   233   141      55    1.0       1.0
PUB.POLine             5337  217.5K    40    44    41    5337    1.0       1.0
PUB.PurchaseOrder      2129   68.6K    33    36    33    2129    1.0       1.0
PUB.Supplier             10    1.1K    92   164   117      10    1.0       1.0
PUB.SupplierItmXref      56    1.1K    18    20    19      56    1.0       1.0
PUB.Warehouse            14    1.3K    82   104    92      14    1.0       1.7
                ------------------------------------------------------------
Subtotals:             8446  327.0K    18   233    39    8446    1.0       1.1
```

After doing a table analysis, focus on record count (Records) and mean record size (Mean). Look at every table and split them according to mean record size. Each record contains approximately 20 bytes of record overhead that is included in the calculation of the mean.

Block sizes

In the vast majority of cases, choose an 8KB-block size on UNIX, and a 4KB-block size in Windows. Selecting these block sizes conforms to the operating system block size, which usually yield the best performance. It is good practice to match or be a multiple of the operating system block size, if possible. Matching the database block size to the file system block size helps prevent unnecessary I/O, as shown in the following figure:

Figure 18: Matching database and file block sizes



In Windows, the operating system assumes that files and memory are handled in 4KB chunks. This means that all transfers from disk to memory are 4KB in size. The Windows operating system has been highly optimized for 4KB and often the performance at an 8KB setting is not as good as 4KB.

On UNIX operating systems, the block size is generally 8KB or a multiple of 8K. The block size is tunable. Generally, an 8K database block size is best on UNIX systems.

There are exceptions to every rule. The intention is to make a best estimate that will enhance performance and assist OpenEdge in meshing with your operating system.

Specifying records per block

OpenEdge allows you specify a maximum number of records per block per area value within the range of 1 to 256. The number of records per block must be a power of 2 (1, 2, 4, 8..., 256). Depending on the actual size of your record, you might not be able fit the specified maximum number of records in a block.

To determine the number of records per block:

1. Take the mean record size from your table analysis.
2. Add 2 to the record size for the directory entry overhead. The mean record size includes the record overhead, but excludes the directory entry overhead.
3. For an 8K block size, divide 8192 by the number in Step 2 on page 67. For a 4K block size, divide 4096 by the number in Step 2 on page 67.
4. Round the number from Step 3 on page 67 up or down to the closest power of 2.

Most of the time, the record length will not divide into this number evenly so you must make a best estimate. If your estimate includes too many records per block, you run the risk of fragmentation (records spanning multiple blocks). If your estimate includes too few records per block, you waste space in the blocks. The goal is to be as accurate as possible without making your database structure too complex.

Example of calculating records per block

The following example demonstrates how to determine the best records per block setting:

1. Assume you retrieved the following information from your table analysis:
   - The database has 1 million records.
   - The mean record size is 59 bytes.
2. Add the directory entry overhead (2 bytes) to determine the number of the actual size of the stored record, as shown:

```
Mean record size + overhead = actual storage size
     59 + 2 = 61
```

3. Divide that number into your database block size to determine the optimal records per block, as shown:

```
Database block size / actual storage size = optimal records per block
     8192 / 61 = 134
```

4. Choose a power of 2 from 1 to 256 for the records per block. You have two choices: 128 and 256. If you choose 128, you will run out of record slots before you run out of space in the block. If you choose 256, you run the risk of record fragmentation. Make your choice according to the nature of the records. If the records grow dynamically, then you should choose the lower number (128) to avoid fragmentation. If the records are added, but not typically updated, and are static in size, you should choose the higher number (256) of records per block. Generally OpenEdge will not fragment a record on insert; most fragmentation happens on update. Records that are updated frequently are likely to increase in size.

If you choose the lower value, you can determine this cost in terms of disk space. To do this, take the number of records in the table and divide by the number of records per block to determine the number of blocks that will be allocated for record storage, as shown:

```
Number of records / records per block = allocated blocks
     1,000,000 / 128 = 7813
```

Next, calculate the number of unused bytes per block by multiplying the actual storage size of the record by the number of records per block and subtracting this number from the database block size, as shown:

```
Database block size - (Actual storage size * records per block) = Unused space per block
     8192 - (61 * 128) = 384
```

Take the number of allocated blocks and multiply them by the unused space per block to determine the total unused space, as shown:

```
Allocated blocks * unused space per block = total unused space
     7813 * 384 = 3000192
```

In this case, the total unused space that would result in choosing the lower records per blocks is less than 3MB. In terms of disk space, the cost is fairly low to virtually eliminate fragmentation. However, you should still choose the higher number for static records, as you will be able to fully populate your blocks and get more records per read into the buffer pool.

Unused slots

Another item to consider when choosing to use 256 records per block is that you will not be using all the slots in the block. Since the number of records determines the number of blocks for an area, it might be important to have all of the entries used to obtain the maximum number of records for the table.

Determining space to allocate per area

You must determine the quantity of space to allocate per area. OpenEdge keeps data and index storage at reasonable compaction levels. Most data areas are kept between 90 and 95 percent full, and indexes are maintained at 95 percent efficiency in the best case. However, when calculating the amount of space, it is advisable to use 85 percent as the expected efficiency. Using the 1-million-record example previously discussed, you can see that the records plus overhead would take 61 million bytes of storage, as shown:

```
(Mean record size + overhead) * Number of records = record storage size
(59 + 2) * 1,000,000 = 61 million bytes
```

This is only actual record storage. Now, divide the record storage value by the expected fill ratio. The lower the ratio, the more conservative the estimate. For example:

```
Record storage size / Fill ratio = Total storage needed
61,000,000 / .85 = 71,764,706 bytes
```

To determine the size in blocks, divide this number by 1KB (1024 bytes). The resulting value is in the proper units for expressing the amount of space needed when you create your structure description file (*dbname*.st). The structure description file uses kilobytes regardless of the block size of the database. For example:

```
71,764,706 / 1024 = 70083 (1KB blocks)
```

If there are other objects to be stored with this table in a storage area, you should do the same calculations for each object and add the individual calculations to determine the total amount of storage necessary. Beyond the current storage requirements for existing records, you should factor additional storage for future growth requirements.

Distributing tables across storage areas

Now that you know how many records can fit in each block optimally, you can review how to distribute the tables across storage areas. Some of the more common reasons to split information across areas include:

- Controlled distribution of I/O across areas
- Application segmentation
- Improve performance of offline utilities

Knowing how a table is populated and accessed, helps decide whether or not to break a table out to its own area. In tables where records are added to a table in primary index order, most of the accesses to these records are done in sequential order via the primary index, and there might be a performance benefit in isolating the table. If this is a large table, the performance benefit gained through isolation can be significant. There are two reasons for the performance improvement:

- One database read will extract multiple records from the database, and the other records that are retrieved are likely to be used. This improves your buffer hit percentage.
- Many disk drive systems have a feature that reads ahead and places items in memory that it believes are likely to be read. Sequential reads take advantage of this feature.

Finally, databases can contain different types of data in terms of performance requirements. Some data, such as inventory records, is accessed frequently, while other data, such as comments, is stored and read infrequently. By using storage areas you can place frequently accessed data on a "fast disk." However, this approach does require knowledge of the application.

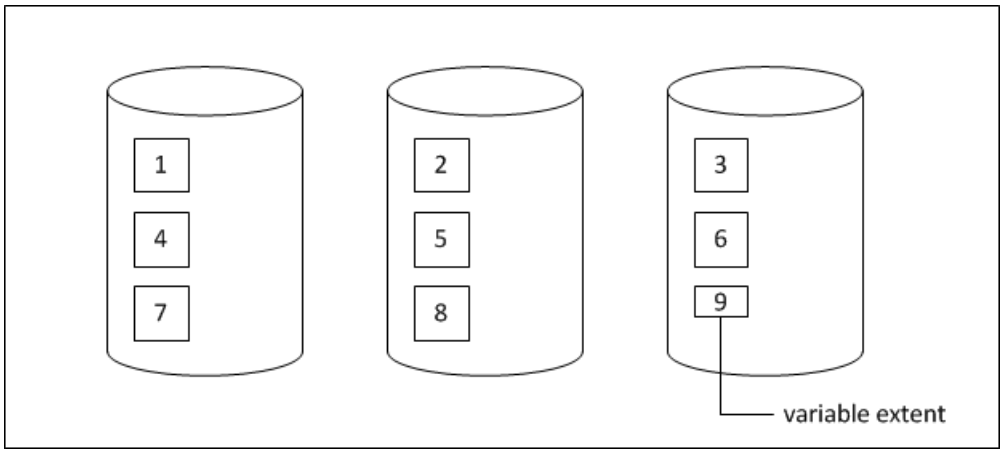
Using extents

Most DBAs choose powers of 2 for their extent sizes because these numbers are easier to monitor from the operating system level. Each area should have a variable extent as the last area to allow for growth. Monitoring and trending should prevent you from needing this last extent, but it is preferable to have it available should it be needed.

For example, the Determining space to allocate per area on page 68, we calculated that the size of an area needed to be 70,083 1KB blocks. You can choose one extent with 80,000 1KB blocks to store the data, with room for expansion, and one variable extent; or you could choose eight fixed extents with 10,000 1KB blocks each, and one variable extent.

Many extents allow you to distribute your data over multiple physical volumes if you do not use RAID on your system. For example, if you chose eight fixed extents and one variable extent for your area, you can "stripe" your extents across three drives, as shown in Figure 19: Manually striped extents on page 70. You put the first, fourth, and seventh extents on the first drive, the second, fifth, and eighth extents on the second drive, and the third, sixth, and variable extents on the third drive. OpenEdge fills these extents in order. By striping the extents, you will have a mix of old and new data. While striping your extents is not as efficient as hardware striping, it does help eliminate variance on your drives.

Figure 19: Manually striped extents



Even if you do have hardware striping, you might want to have multiple extents. The default file limit is 2GB per extent. If you want to store more than this amount, you need to have multiple extents per area, or you can enable large file support, which allows you to allocate extents up to 1TB in size.

While it is possible to have one very large extent, this will not give you the best performance. The best size varies across operating systems, but 1GB seems to be a safe number across all operating systems with modern file systems. In Windows, you should use NTFS file systems for best performance.

Index storage

Record storage is fairly easy to calculate, but index storage is not because index compression makes calculation difficult. The ever-evolving compression algorithms make the calculation even harder. You can run a database analysis and use the index-specific output to make your decisions. Remember to allow room for growth and general overhead, the same as with data storage.

If you have an existing database, you can take statistics to determine index storage size. Without a database, you have to estimate the size. The number and nature of indexes can vary greatly between applications. Word indexes and indexes on character fields tend to use more space, while numeric indexes are significantly more efficient in terms of storage. There are databases where indexes use more storage than data, but these are the exception and not the rule.

In general, indexes account for approximately 30 percent of total storage. Therefore, you can take 50 percent of your data storage as an estimate of index storage. Remember that this percent might vary greatly, depending on your schema definition. Consider this estimate as a starting point and adjust and monitor it accordingly.

The following example highlights a portion of a database analysis report that shows the proportion of data storage to index storage within an existing database. Use this information to determine the allocation of disk resources to the areas that are going to contain the data, as shown:

```
SUMMARY FOR AREA "Student Area": 8
----------------------------------
                   Records                Indexes                Combined
Name           Size   Tot percent    Size   Tot percent   Size   Tot percent
PUB.stuact       18.9M  12.6            9.7M   6.4            28.6M   19.0
PUB.student      30.3M  20.1           20.1M  13.4            50.5M   33.5
                ------------------------------------------------------------
Total           115.3M  76.4           35.6M  23.6           150.8M  100.0
```

Primary recovery area

Proper sizing of the primary recovery area, also known as the before-image file, is important to your overall database system. This area is responsible for the recoverability of your database on an ongoing basis. Because the primary recovery area is written to frequently, if it is on a slow disk update performance will be affected. The size of this area varies depending on both the length of transactions and the activity on your system.

The primary recovery area is made up of clusters, which are tunable in size. When records are modified, notes are written to this area. If a problem occurs, or if the user decides to "undo" the changes, this area is used to ensure that no partial updates occur.

For example, assume you want to modify all of the records in a table to increase a value by 10 percent. You want this to happen in an all-or-nothing fashion because you can not determine which records are modified if the process terminates abnormally. In this case, you have one large transaction that modifies all of the records. If a problem occurs during the transaction, all of the modifications are rolled back to the original values. Why is this important? If you have several of these processes running simultaneously, the primary recovery area can grow substantially.

The structure of the area is a linked list of clusters. The smaller the cluster size the more frequent the checkpoints occur. A *checkpoint* is a synchronization point between memory and disk. While there is a potential performance benefit from infrequent checkpoints, this must be tempered with the amount of time it takes to recover the database. Large cluster sizes can also increase database startup and shutdown time when the database needs to back out incomplete transactions or perform crash recovery.

The best way to determine the before-image cluster size is to monitor the database at the time of the day when the most updates to the database are being made, and review the duration of your checkpoints throughout the week.

Ideally, checkpoints should not happen more than once every two minutes. If you are checkpointing more often, you should increase your before-image cluster size. This does not mean you should decrease the cluster size if it is happening less frequently. The default of 512KB is fine for smaller systems with low update volume, but a value of 1024KB to 4096KB is best for most other systems. The cluster size can be modified from small (8KB) to large (greater than 256 MB).

The cluster size influences the frequency of the checkpoints for the database. As users fill up a cluster with notes, they are also modifying shared memory. The page writers (APWs) are constantly scanning memory, looking for modified buffers to write to disk. At the first checkpoint, all of the modified buffers are put in a queue to be written prior to the next checkpoint. The buffers on the modified buffer queue are written by the page writers at a higher priority than other buffers. If all of the buffers on the queue are written prior to the next checkpoint, it is time to schedule the current modified buffers. If all of the buffers are not written, then you must write all of the previously scheduled buffers first, and then schedule the currently modified buffers. If you are checkpointing at the proper frequency and you are still flushing buffers at checkpoint, you should add an additional APW and monitor further. If adding the APW helps but does not eliminate the problem, add one more. If adding the APW does not help, look for a bottleneck on the disks.

The format of the primary recovery area has been discussed, but its size has not. There is no formula for determining the proper size because the size of the area is dependent on the application. Progress Software Corporation recommends that you isolate this area from other portions of the database for performance reasons. If you only have one database, you can isolate this area to a single disk (mirrored pair), as the writes to this area are sequential and would benefit from being placed on a single nonstriped disk. If you have several databases, you might want to store your primary recovery areas on a stripe set (RAID 10) to increase throughput.

Database areas

This section details database area optimization. Although some of the information presented in this section might be found in other sections of this book or in other manuals, it is repeated here to present the most common solutions to area optimization in one place for easy reference. The goal of area optimization is to take advantage of the OpenEdge architecture and the operating system.

Data area optimization

Following a few simple rules can make your data areas easy to maintain and can provide optimal performance for users.

Splitting off the schema

By splitting off the schema and sequences from other portions of your database, you can make future upgrades to OpenEdge and your application more transparent to the user. One of the operations that takes place during most major version changes of OpenEdge is a metaschema change. The *metaschema* is a set of definitions for the internal structure of data. Your schema definitions create the rules by which data can be added or modified within your application. Metaschema serves the same purpose for the definition of your schema. By eliminating information other than your schema definitions and sequences from the metaschema area, it reduces the task of updating this information.

Choosing an appropriate block size

Matching the database block size to the operating system allows for a more efficient transfer of data. If you have a block size that is too small, the operating system retrieves more blocks than your request. While that transfer of additional information might be useful, if the additional information is not used by the application, then the transfer of additional information is wasted. Larger blocks are also generally better because of the way indexes behave. If each block contains more information, then you will require fewer index blocks and fewer index levels to contain the data. Index levels in a B-tree are important. If you can eliminate a level from an index, you can save one additional I/O operation per record request.

Keeping extents sized to eliminate I/O indirection

In an unjournaled file system, I/O indirection happens when a single inode table cannot address all of the addresses within a file. The *inode* is a mapping of logical to physical addresses in a file. Think of the inode table as a table of contents for the file. In theory, a file can be large (more than 4GB) before indirection occurs, but the conditions need to be perfect. Inode indirection is not a concern for journaled file systems.

In the real world, it is possible to see indirection at a file size of 500MB, but on most systems you do not see indirection until 1GB or higher. The penalty for smaller database area extents is fairly low. Generally, you only need to verify that the number of file descriptors (open files) available from the operating system is high enough to support the extents.

Keeping areas small for offline utilities

While many utilities for OpenEdge are available online, there are still some utilities that require you to shut down the database prior to running them. For these utilities, limiting the amount of information per area reduces the amount of downtime needed for the utility to run.

The best example of this is an index rebuild. If you only need to rebuild one index, you still must scan the entire area where the records for that index are stored to ensure you have a pointer to every record. If all of your tables are in one area, this can take a significant amount of time. It is much faster to scan an area where all the records are from one table.

Always have an overflow extent for each area

The last extent of every area, including the primary recovery area, but not the after-image areas, should be variable length. Monitoring storage capacity and growth should eliminate the need to use the variable extent. It is preferable to have it defined in case you fill all of your fixed extents. The variable extent allows the database to grow as needed until it is possible to extend the database.

Enabling large files

You should always have large files enabled for your database. Large file support allows you to support extent sizes in excess of 1TeraByte (TB), provided that the operating system supports large files. On UNIX, you need to enable operating system support for large files on each file system where the database resides. In Windows, a file can fill the entire volume. By default, large file support is enabled for all databases created in OpenEdge 12.0 under Enterprise license. All databases created in OpenEdge 12.0 under sub-Enterprise license will have large file support disabled by default.

Partitioning data

Partitioning data by functional areas is a reasonable way to split your information into small pieces to reduce the size of a given area. This activity allows you to track the expansion of each portion of the application independent of each other.

Primary recovery (before-image) information

On systems where updates are done frequently, it is important to make the read and write access to this database area as efficient as possible. The write access is more important.

The following sections provide simple tips to create an efficient environment for the primary recovery area.

- Extent size rules on page 73
- Sequential access on page 74
- BI grow option on page 74

Extent size rules

These rules apply to both the primary recovery area and the data areas:

- Do not make them too large
- Always make the last extent variable length

- Enable large files as a safety valve

Enabling large files is particularly important on file system that holds the primary recovery area because this is the place most likely to experience issues. A large update program with poor transaction scoping or a transaction held open by the application for a long period of time can cause abnormal growth of this area. If the fixed portion of an area is 2GB in size, you start extending your variable portion in that same transaction. Only then do you notice that you might need more than 2GB of recovery area to undo the transaction. If you are large-file enabled and have enough disk space, there is no problem. If you are not large-file enabled, the database might crash and not be recoverable because there is no way to extend the amount of space for the recovery area without going through a proper shutdown of the database.

Sequential access

The primary recovery area is sequentially accessed. Items are written to and read from this area in a generally linear fashion. If you are able to isolate a database's primary recovery area from other database files and other databases, then it is a good idea to store the extents for this area on a single disk (mirror).

While striping increases the throughput potential of a file system, it is particularly effective for random I/O. If the area is not isolated from the database, or you are storing the primary recovery areas of several databases on the same disk, use striping because the I/O will be fairly randomized across the databases.

BI grow option

The BI grow option to PROUTIL allows you to preformat BI clusters before the user enters the database. Preformatting allows you to write more recovery notes without needing to format new clusters while the database is online. Formatting clusters online can have a negative impact on performance. Your database must be shut down for you to grow the `BI` file (primary recovery area). For more information, see *OpenEdge Data Management: Database Administration*.

After-image information

The after-image file is used to enable recovery to the last transaction or to a point in time in the case of media loss. After-imaging is critical for a comprehensive recovery strategy.

The after-image file is like the before-image file in the sequential nature of its access. It does not have automatic reuse like the before-image file because it requires intervention from the administrator to reuse space. After-imaging is the only way to recover a database to the present time in the case of a media failure (disk crash). It also provides protection from logical corruption by its "point-in-time" recovery ability.

For example, assume a program accidentally runs and incorrectly updates every customer name to "Frank Smith." If you have mirroring, you now have two copies of bad data. With after-imaging, you can restore last night's backup and roll forward today's after-image files to a point in time just prior to running the program. After-image should be a part of every high-availability environment. For more details on implementing and managing after-image files, see *OpenEdge Data Management: Database Administration*.

Always use multi-volume extents

OpenEdge supports one or more after-image extents per database when after-imaging is enabled. Each extent of the after-image file is its own area with a unique area number, but it is more common to refer to them as extents. You need more than one extent for each after-image file to support a high-availability environment.

Each extent has five possible states. They are:

- **Empty** — An extent that is empty and ready for use.

- **Busy** — The extent that is currently active. There can only be one busy extent per database.
- **Full** — An extent that is closed and contains notes. A full extent cannot be written to until the extent is marked as empty and readied for reuse.
- **Locked** — An extent that is full and has not been replicated by OpenEdge Replication. You will only see this state when you have OpenEdge Replication enabled.
- **Archived** — An extent that is full and has been archived by AI File Management, but has not been replicated by OpenEdge Replication. You will only see this state when AI File Management and OpenEdge Replication are enabled.

Multiple extents allow you to support an online backup of your database. When an online backup is executed the following occurs:

1. A latch is established in shared memory to ensure that no update activities take place.
2. The modified buffers in memory are written (pseudo-checkpoint).
3. An after-image extent switch occurs (if applicable).
4. The busy after-image extent is marked as full, and the next empty extent becomes the busy extent.
5. The primary recovery area is backed up.
6. The latch that was established at the start of the process is released.
7. The database blocks are backed up until complete.

Isolate for disaster recovery

The after-image files must be isolated to provide maximum protection from media loss. If you lose a database or a before-image drive, you can replace the drive, restore your backup, and use the after-image file to restore your database. If you lose an after-image drive, you can disable after-imaging and restart the database. You will only lose active transactions if the after-image extents are isolated from the rest of the database. Sometimes this is difficult to do because you might have several file systems accessing the same physical drive, but the isolation must be at the device and file system levels.

Sizing after-image extents

The after-image area differs from all other areas because each extent can be either fixed or variable length. Each extent is treated as its own area. It is fairly common for people to define several (more than 10) variable-length extents for the after-imaging.

To choose a size, you must know how much activity occurs per day and how often you intend to switch after-image extents. You can define all of your extents as variable length and see how large they grow while running your application between switches. To accommodate above normal activity, you need extra extents. If you are concerned about performance, you should have the after-image extents fixed length so you are always writing to preformatted space. Preformatting allows you to gain:

- Performance by eliminating the formatting of blocks during the session
- Use of a contiguous portion of the disk

Most operating systems are fairly good at eliminating disk fragmentation. However, if you have several files actively extending on the same file system, there is a high risk of fragmentation.

System resources

The remainder of this chapter describes the various resources used by the OpenEdge database, as well as other applications on your system, and it gives you a greater understanding of each resource's importance to meet your needs. The resources in reverse performance order, from slowest to fastest, are:

- Disk capacity on page 76
- Memory usage on page 81
- CPU activity on page 87

Disk capacity

The disk system is the most important resource for a database. Since it is the only moving part in a computer, it is also the most prone to failure. Reliability aside, a disk is the slowest resource on a host-based system, and it is the point where all data resides.

There are three overall goals for a database administrator in terms of disk resources. They are:

- **Quantity** — Having enough disk space to store what you need
- **Reliability** — Having reliable disks so your data will remain available to users
- **Performance** — Having the correct number and maximum speed for your disks to meet your throughput needs

These goals sound simple, but it is not always easy to plan for growth, or to know what hardware is both reliable and appropriate to meet your needs. With these goals in mind, this section examines the problems that might present roadblocks to fulfilling each of these goals.

Disk storage

The following sections describe how to determine if you have adequate disk storage.

Understanding data storage

The following is a list of critical data stored on your system. Used in this context, the term "data" is a more inclusive term than one that defines simple application data. Critical data can include:

- Databases
- Before-image files
- After-image files
- Key store (for database enabled for Transparent Data Encryption)
- Application files (ABL or SQL code, third-party applications)
- Temporary files
- Client files

Data also refers other possible data storage requirements, which can include:

- Backup copies of your databases
- Input or output files
- Development copies of your databases
- Test copies of your databases

Other critical elements stored on your disk include:

- Operating system
- Swap or paging files
- Your OpenEdge installation

If this information is already stored on your system, you know or you can determine the amount of data you are storing.

If you are putting together a new system, planning for these data storage elements can be a difficult task. You need to a complete understanding of the application and its potential hardware requirements. One of the first things you must know is how much data you intend to store in each table of your database, and which tables will grow and which are static. See the database storage calculations discussed in the Calculating database storage requirements on page 62.

Determining data storage requirements

In existing database environments, the first step to determine data storage requirements is to take an inventory of your storage needs. The types of storage are:

- **Performance oriented** — Database or portions of databases, before-image (`BI`) areas, after-image (`AI`) areas
- **Archival** — Historical data, backups
- **Sequential or random** — Data entry and report generation, or spot queries

Determining current storage using operating system commands

Use one of the following options, depending on your operating system, to determine the current storage used:

- Use `df` to determine the amount of free space available on most UNIX systems. There are switches (`-k` to give the result in kilobytes and `-s` to give summary information) that will make the information easier to read.
- Use `bdf` on HPUX to report the same information as described in the previous bullet.
- Use the disk properties option in Windows to provide a graphical display of disk storage information.

Projecting future storage requirements

Perform a detailed analysis of your current needs and projected growth to estimate storage requirements. The following sections describe this analysis process:

- Examining your growth pattern on page 78
- Moving archival data off the production machine on page 78

Examining your growth pattern

Many companies experience exponential growth in their data storage needs when the business grows to meet demand or absorbs additional data due to acquisition. The database administrator must be informed when these business decisions occur to appropriately plan for growth of the database. Some growth is warranted, but in most cases much of data stored can be archived out of the database and put on backup media. For information that is not directly related to current production requirements, one option is to use a secondary machine with available disk space. The secondary machine can serve as both secondary storage for archival data and as a development or replication server. When you move data that is not mission-critical off the production machine to another location, you can employ less expensive disks on the archival side and more expensive disks for production use.

Moving archival data off the production machine

It is important to understand how archived data is to be used, if at all, before making any plans to move it off of the production system. Some users might want to purge the data, but first you should archive the data so if at a later date, it can be easily recovered. Your archive method can be as simple as a tape; however, it is important to remember that you might need to read this tape in the future. Archive data in a format that you can access at a later date. For your database, you should archive the data in a standard OpenEdge format, remembering to also archive all the files not included in an OpenEdge backup such as after-image and key store files. For your application, you should archive the information in a format that will not depend on third-party software.

Comparing expensive and inexpensive disks

When you buy a disk, you are really buying two things: storage capacity and throughput capacity. For example, if you need to buy 72 gigabytes (GB) of capacity, you can purchase a single 72GB unit or four 18GB units. The storage capacities of these disks are exactly the same, but the four-drive configuration potentially has four times greater throughput capacity. Each drive is capable of doing approximately 100 input/output (I/O) operations per second regardless of its size. Therefore the four-drive system has the potential to perform 400 I/O operations per second.

For several reasons, it is generally less expensive to purchase fewer larger disks to get to your desired capacity:

- The 72GB drive is only marginally more expensive than a single 18GB drive, so buying four 18GB drives to obtain the same capacity as a 72GB drive will substantially increase your cost.
- You must have the storage space to hold the additional drives; more storage space can lead to additional cost.
- You might need more controllers to efficiently run the additional disks, also adding to the cost.

However, if performance is more important than cost saving, you should purchase a greater number of physical disk drives to give you the greatest throughput potential. The additional cost of the multi-disk solution can be offset by increases in performance (user efficiency, programmer time, customer loyalty) over time.

If you are using a database only as archival storage, and performance is not critical, fewer large disks are recommended to keep costs lower.

Understanding cache usage

Disk vendors provide for caching on their disk arrays to increase performance. However, there is a limit to how effective this cache can be. If your system is doing hundreds of thousands of reads per hour, the cache becomes saturated after a short period of time. When your cache is saturated, performance reverts to conventional disk speeds. When estimating performance, consider as performing at conventional speeds.

Consider that an index rebuild can quickly saturate the disk cache because it performs many modifications in a short time frame. The potential performance benefit of a disk cache should not alter the layout of these disks—they should be treated as identical to any other disk.

Increasing disk reliability with RAID

It is important to understand the terms reliability and performance as they pertain to disks. Reliability is the ability of the disk system to accommodate a single- or multi-disk failure and still remain available to the users. Performance is the ability of the disks to efficiently provide information to the users.

Adding redundancy almost always increases the reliability of the disk system. The most common way to add redundancy is to implement a Redundant Array of Inexpensive Disks (RAID).

There are two types of RAID:

- **Hardware** — The most commonly used hardware RAID levels are: RAID 0, RAID 1, RAID 5, and RAID 10. The main differences between these RAID levels focus on reliability and performance as previously defined.
- **Software** — Software RAID can be less expensive. However, it is almost always much slower than hardware RAID, because it places a burden on the main system CPU to manage the extra disk I/O.

The different hardware RAID types are as follows:

- **RAID 0 (Striping)** — RAID 0 has the following characteristics:
  - **High performance** — Performance benefit for randomized reads and writes
  - **Low reliability** — No failure protection
  - **Increased risk** — If one disk fails, the entire set fails

  The disks work together to send information to the user. While this arrangement does help performance, it can cause a potential problem. If one disk fails, the entire file system is corrupted.
- **RAID 1 (Mirroring)** — RAID 1 has the following characteristics:
  - **Medium performance** — Superior to conventional disks due to "optimistic read"
  - **Expensive** — Requires twice as many disks to achieve the same storage, and also requires twice as many controllers if you want redundancy at that level
  - **High reliability** — Loses a disk without an outage
  - **Good for sequential reads and writes** — The layout of the disk and the layout of the data are sequential, promoting a performance benefit, provided you can isolate a sequential file to a mirror pair

  In a two disk RAID 1 system, the first disk is the primary disk and the second disk acts as the parity, or mirror disk. The role of the parity disk is to keep an exact synchronous copy of all the information stored on the primary disk. If the primary disk fails, the information can be retrieved from the parity disk.

  Be sure that your disks are able to be hot swapped so repairs can be made without bringing down the system. Remember that there is a performance penalty during the resynchronization period of the disks.

On a read, the disk that has its read/write heads positioned closer to the data will retrieve information. This data retrieval technique is known as an optimistic read. An optimistic read can provide a maximum of 15 percent improvement in performance over a conventional disk. When setting up mirrors, it is important to consider which physical disks are being used for primary and parity information, and to balance the I/O across physical disks rather than logical disks.

- **RAID 10 or 1+0** — RAID 10 has the following characteristics:
  - **High reliability** — Provides mirroring and striping
  - **High performance** — Good for randomized reads and writes.
  - **Low cost** — No more expensive than RAID 1 mirroring

  RAID 10 resolves the reliability problem of striping by adding mirroring to the equation.

Note: If you are implementing a RAID solution, Progress Software Corporation recommends RAID 10.

- **RAID 5** — RAID 5 has the following characteristics:
  - **High reliability** — Provides good failure protection
  - **Low performance** — Performance is poor for writes due to the parity's construction
  - **Absorbed state** — Running in an absorbed state provides diminished performance throughout the application because the information must be reconstructed from parity

Caution: Progress Software Corporation recommends not using RAID 5 for database systems.

It is possible to have both high reliability and high performance. However, the cost of a system that delivers both of these characteristics is higher than a system that is only delivers one of the two.

OpenEdge in a network storage environment

OpenEdge supports reliable storage of data across the network, provided your operating system complies with the rules of NFS Protocol Version 3.0.

Storage Area Networks (SANs) are becoming more popular than NFS Protocol. You can purchase one storage device and attach it to several hosts through a switching device. The same rules apply in a SAN environment as they do in a conventional or RAID environment. The SAN maintains database integrity. Because multiple machines have access to the disks in a SAN, you must be very aware of the drive/controller usage and activity of other hosts using the SAN. Check your SAN vendor's documentation regarding monitoring to see if they provide an interface into the activity levels of the disks in the array.

Disk summary

Disks are your most important resource. Purchase reliable disk arrays and configure them properly to allow consistent fast access to data, and monitor them for performance and fill rate. Monitor your disks and track their fill rate so you do not run out of space. Track the consumption of storage space to accurately plan for data archiving, and to allow planning time for system expansion due to growth.

Memory usage

The primary function of system memory is to reduce disk I/O. Memory speed is orders of magnitude faster than disk speed. From a performance perspective, reading and writing to memory is much more efficient than reading and writing to disk. Because memory is not a durable storage medium, long-term storage on memory is not an option. There are RAM disks that do provide durable data storage, but they are cost prohibitive for most uses.

Maximizing memory management includes:

- Allocating memory for the right tasks
- Having enough memory to support your needs

Systems manage memory with paging. There are two types of paging:

- **Physical paging**— Identifies when information is needed in memory; information is retrieved from temporary storage on disk (paging space)
- **Virtual paging** — Occurs when information is moved from one place in memory to another

Both kinds of paging occur on all systems. Under normal circumstances, virtual paging does not degrade system performance to any significant degree. However, too much physical paging can quickly lead to poor performance. Paging varies by hardware platform, operating system, and system configuration. Because virtual paging is fairly inexpensive, a significant amount can be done with no adverse effect on performance. Physical paging will usually be high immediately after booting the system, and it should level off at a much lower rate than virtual paging. Most systems can sustain logical paging levels of thousands of page requests per second with no adverse effect on performance. Physical paging levels in the thousands of requests is too high in most cases. Physical paging should level into the hundreds of page requests per second on most systems.

If physical paging continues at a high rate, then you must either adjust memory allocation or install more memory. It is important to remember that these numbers are only guidelines because your system might be able to handle significantly more requests in both logical and physical page requests with no effect on performance.

Estimating memory requirements

To determine memory usage needs, take an inventory of all of the applications and resources on your system that use memory. For OpenEdge-based systems, these processes include:

- Operating system memory
- Operating system required processes
- Operating system buffers
- OpenEdge-specific memory
- OpenEdge executables
- Database broker
- Remote client servers
- Other OpenEdge servers
- Client processes (batch, self-service, and remote)

Operating system memory estimates

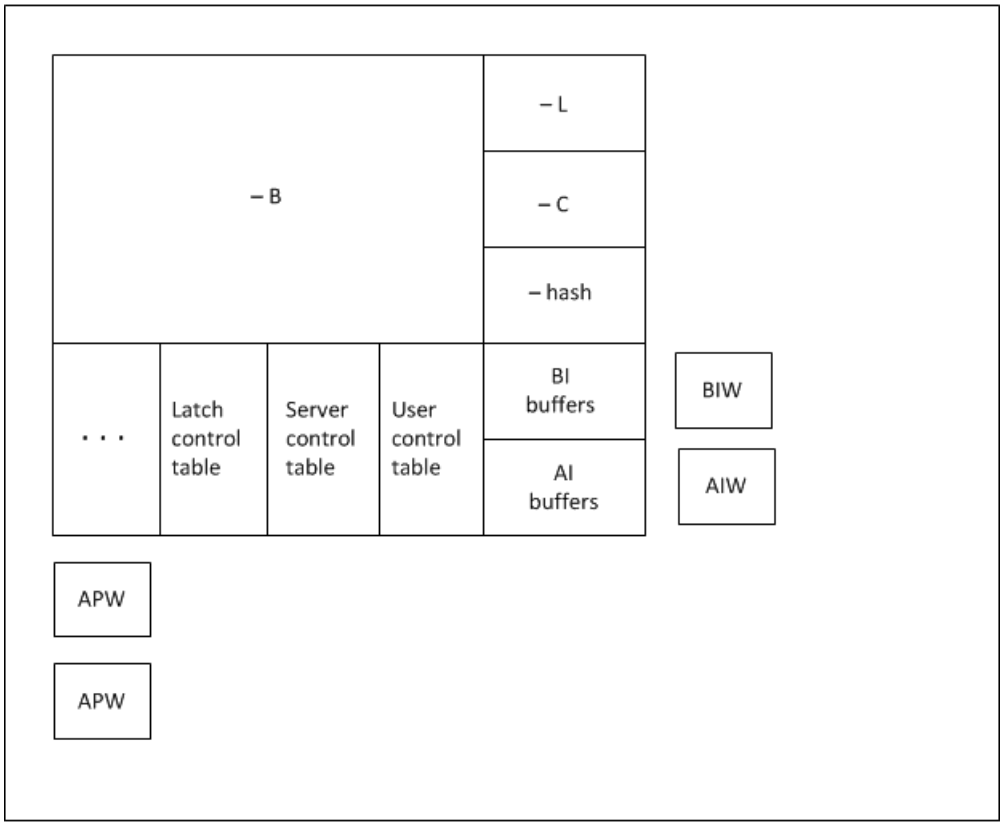
Operating system memory usage varies from machine to machine. Operating system buffers are generally a product of how much memory is in the machine. Most systems will reserve 10 to 15 percent of RAM for operating system buffers. Operating system buffers are tunable on most systems. See your operating system product documentation for details.

Understanding memory internals

The primary broker process allocates shared memory for users to access data within the database. The users also use structures within memory to allow for concurrent access to information without corrupting this information. For example, two users who want to update the same portion of memory with different updates could lead to shared memory corruption. Latches prevent this corruption, similar to a lock. When a process locks a record, it is allowed to update the record without interference from other processes trying to make simultaneous changes. A *latch* is a lock in shared memory that allows a user to make modification to a memory block without being affected by other users.

Figure 20: Shared memory resources example on page 82 shows an example of shared memory resources. This illustration should only be considered an example because it is incomplete and is not to scale. Database buffers account for more than 90 percent of shared memory, and the various latch control structures account for less than 1 percent.

Figure 20: Shared memory resources example



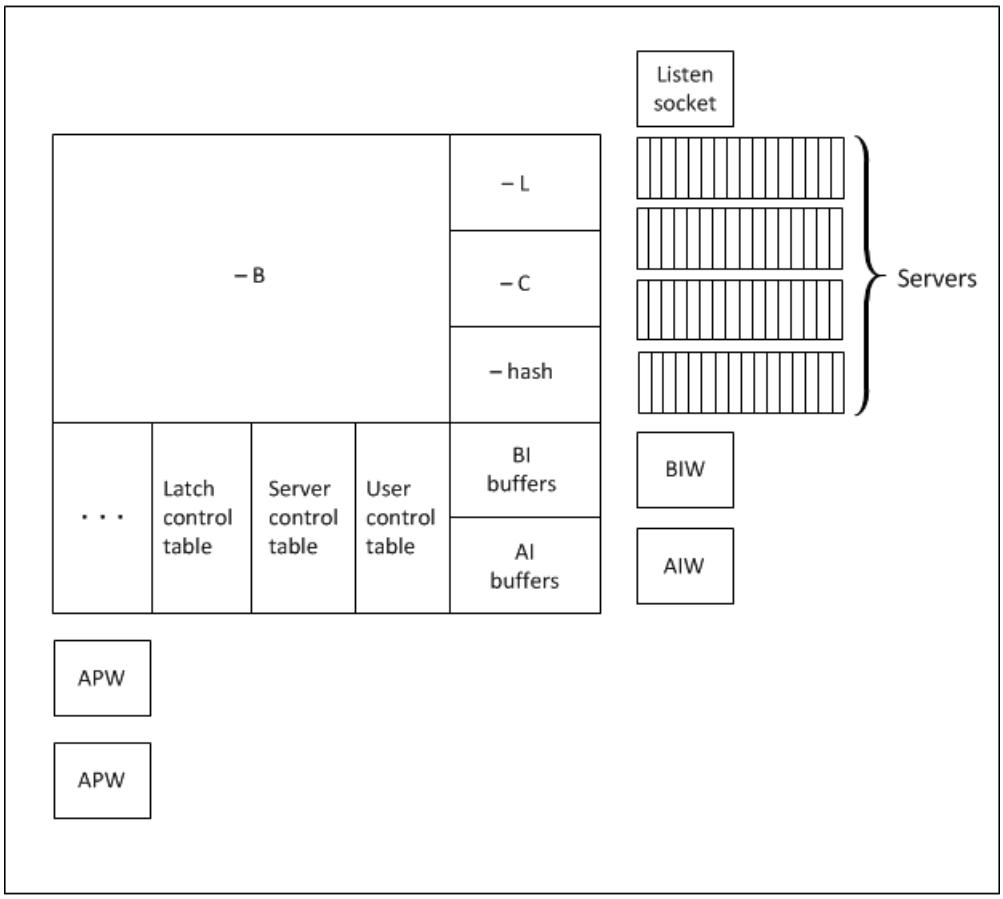
As Figure 20: Shared memory resources example on page 82 illustrates, there are many resources inside of shared memory. Local users (both end-user processes and batch processes) update these structures. If two users access this database simultaneously and both users want to make an update to the lock table (`-L`), the first user requests the resource by looking into the latch control table. If the resource is available, the user establishes a latch on the resource using an operating system call to ensure that no other process is doing the same operation. Once the latch is enabled, the user makes the modification to the resource and releases the latch. If other users request the same resource, they must retry the operation until the resource latch is available.

The database buffers are vitally important. They provide a caching area for frequently accessed portions of the database so that information can be accessed from disk once and from memory several times. Because memory is so much faster than disk, this provides an excellent performance improvement to the user when tuned properly.

The other processes shown in Figure 20: Shared memory resources example on page 82 are page writers. These Asynchronous Page Writer (APW) processes write modified database buffers to disk. You can have more than one APW per database. The other writers, After-image Writer (AIW) and Before-image Writer (BIW), write after-image and before-image buffers to disk. There can only be a single BIW and a single AIW per database.

Figure 21: Shared memory resource—adding remote clients on page 84 illustrates how the process of adding remote clients adds a TCP/IP listen socket and server processes. The remote clients send a message to the listen socket, which in turn alerts the broker process. The broker process references both the user control table and the server control table to determine if the user can log in, and to which server the user can attach. If a server is not available, one server is started, depending on the server parameters for this broker. Parameters such as `-Mn`, `-Mi`, and `-Ma`, control the number of servers a broker can start, the number of clients a server can accept, and when new servers are started. See *OpenEdge Data Management: Database Administration* for details. Once the proper server has been determined, a bi-directional link opens between that server and the remote client. This link remains open until the user disconnects or until the broker is shut down.

Figure 21: Shared memory resource—adding remote clients



OpenEdge-specific memory estimates

OpenEdge uses demand-paged executables. *Demand-paged executables,* also known as *shared executables*, reserve text or static portions of an executable that is placed in memory and shared by every user of that executable. For brokers and servers, the dynamic, or data portion, of the executable is stored in memory (or swap/paging files) for every user or instance of the executable. OpenEdge dynamic memory allocation is estimated based on the number of users and a some of the startup parameters for the brokers.

Estimate the amount of memory used by the database broker, at 110 percent to the database buffers parameter (`-B`). However, if you have a high value for lock table entries (`-L`) or index cursors (`-c`), you must increase the estimate. Record locks consume 64 bytes each, and index cursors consume 84 bytes each. Also, if you have a very low setting for database buffers (less than 2000), the overhead for the other parameters is greater than 10 percent of the `-B` value.

For example, if database buffers (-B) are set to 20,000 on an 8KB-block-size database, you allocate 160,000KB in database buffers. If you add 10 percent of this amount, your total allocation is approximately 176,000KB, or 176MB for the database broker.

Remote client servers are estimated to use approximately 3MB to 5MB. The number of remote client servers is limited by the -Mn parameter. The default number is 5.

Client processes will vary, depending on the startup options chosen. However, with average settings for -mmax and -Bt, the new memory allocated per process is 5MB to 10MB. This range applies to application server processes, too. Remote users usually use more memory (10MB to 20MB per process) because they require larger settings for -mmax and -Bt to provide acceptable performance across the network. The memory requirements for a remote user (that is, -mmax and -Bt settings) do not impact the memory requirements on the host.

Example memory budget

Here is an example of a machine with 1GB of RAM, 50 local users, and one 8KB-block-size database using 10,000 database buffers:

- Operating system memory
  - 28MB OS
  - 100MB OS buffers
- OpenEdge memory
  - 16MB executable
  - 88MB database broker ((8KB * 10000) * 1.1)
  - 250MB to 500MB for users

Total memory requirement: 582MB to 832MB.

The system can run without significant paging, allowing you to use the additional memory for other applications or to further increase the memory utilization for OpenEdge by increasing database broker parameters, like -B. Once the broker is as efficient as possible, you can look into increasing local user parameters like -mmax.

In many cases there other applications running on the system also. You should consider the memory used by these additional applications to accurately determine memory estimates.

Optimizing memory usage

Optimizing memory usage is taking advantage of the memory that you have. The general idea is to share memory where you can, use memory efficiently where possible, increase memory on the broker side first, and then increase client memory usage. In most cases, systems benefit from additional memory, but sometimes it is not possible to purchase additional memory. This section focuses on the pros and cons of user and broker memory and how to best use the available memory.

Buffer hits

The greater the percentage of time that the buffer is used, the lower the percentage of time the disks are used. Since memory is faster than disks, you will get better performance with a better buffer hit percentage. For example, a 90 percent buffer hit percentage equates to 10 disk reads for every 100 requests to the database manager. If you increase your buffer hit percentage to 95 percent, you are only doing 5 disk reads for the same number of requests, which is a 50 percent reduction in requests to the disk. A small change in buffer hit percentage can equate to a large reduction in disk I/O. This is especially noticeable at the high end; changing from 95 percent buffer hit percentage to 96 percent represents a 20 percent reduction in disk I/O.

Increasing memory

Because memory is a limited resource, it is important to use it properly. In the case of a database application, it is important to increase broker parameters first because the payback extends across all users and the system as a whole. This fact is demonstrated with the previous database buffer hit percentage example that shows how a small change on the broker side can dramatically affect the entire system. The size of the change on the broker is usually smaller relative to any self-service client changes you might make.

For example, a 1000 buffer increase for the `-B` parameter on the broker of an 8KB-block-size database costs you 8MB of RAM, and an 80KB increase on the `-mmax` parameter for a 100-user system costs the same 8MB of RAM. In the majority of cases, a buffer increase has a greater overall impact than the client change. This simple example demonstrates that you should always tune your broker parameters first. Once that is complete and you still have RAM to allocate, you can then focus on the self-service client parameters.

Decreasing memory

Determining where to cut back on memory is difficult. When decreasing memory you should use the opposite approach used for increasing memory; look at client parameters before broker parameters.

Direct IO

OpenEdge bypasses operating system buffers with the use of `-directio`, limiting the need for operating system buffers. Operating system buffers are still used for temporary file I/O for any client processes that reside on that machine. Most operating system manufacturers allow you to modify the number of buffers that can be allocated to the operating system. If you are using the `-directio` startup option, then you can reduce the amount of operating system buffers to approximately 10 percent in most cases. One major exception is systems with very limited memory (less than 256MB), where leaving the parameter at its default value is the best practice.

Private buffers (-Bp)

Private buffers allow a read-intensive user to isolate a portion of the buffer pool. Up to 25 percent of buffers can be allocated as private buffers. Private buffers work as follows:

1. The user requests a number of buffers to be allocated as private.
2. As the user reads records, if the corresponding buffers are not already in the buffer pool, the records are read into these buffers.

3. Instead of following the rules for buffer eviction, the user only evicts buffers that are in their private buffers. By default, the least recently used buffer is evicted. Private buffers are maintained on their own chain and are evicted by the user who brought them into memory.
4. If a user wants a buffer that is currently in another user's private buffers, this buffer is "transferred" from the private buffers to the general buffer pool. The transfer is a process of removing the buffer from the user's private buffer list and adding it to the general buffer list. The buffer itself is not moved.

Alternate Buffer Pool

When increasing the size of the buffer pool (-B) is not possible or does not improve buffer hit ratios, creating an Alternate Buffer Pool and designating objects to use it, might be beneficial.

The Alternate Buffer pool gives the database administrator the ability to modify buffer pool behavior by designating objects or areas to consume buffers from the Alternate Buffer Pool, rather than from the primary buffer pool. The size of the Alternate Buffer Pool is specified with the startup parameter `-B2`. Both the Alternate Buffer Pool and the primary buffer pool consume shared memory, and the sum total of both buffer pools is limited by the established shared memory maximums.

Specifying the best objects for the Alternate Buffer Pool is application-specific. Tables considered "hot" (very active) are good candidates, as are their related indexes. Tables and indexes that are governed by an encryption policy are also considered good candidates because the cost of encrypting and decrypting blocks as they are written and read from disk can be high.

The Alternate Buffer Pool operates under the following rules:

- Allocation of an Alternate Buffer Pool requires an Enterprise database license. The `-B2` startup parameter is ignored for non-enterprise databases.
- If you change buffer pool assignments for objects at runtime, existing buffers remain in the buffer pool where they were originally allocated.
- Private read-only buffers (`-Bp`) are always obtained from the primary buffer pool regardless of the buffer pool designation of the object.
- Database control and recovery areas cannot be assigned to the Alternate Buffer Pool.
- If you do not specify a size for the Alternate Buffer Pool at startup, all objects and areas consume buffers from the primary buffer pool.
- You can increase the size of the Alternate Buffer Pool with the `PROUTIL INCREASETO` utility.

For more information on the Alternate Buffer Pool, see *OpenEdge Data Management: Database Administration*.

CPU activity

All resources affect CPU activity. As a database or system administrator, there are only a few things you can do to more efficiently use the CPU resources of your machine. The major consumer of CPU resources for your system should be the application code. Therefore, the greatest impact on CPU consumption can be made with application changes. Other resources are affected by application code as well, but there are things that you can do as an administrator to minimize problems associated with other resources. This is not the case with CPU resources. Slow disks can increase CPU activity by increasing the waits on I/O. If there is significant context switching, system delay time will increase. CPU activity is divided into four categories:

- **User time** — The amount of time spent performing user tasks, such as running applications and database servers.

- **System time** — The amount of time devoted to system overhead such as paging, context switches, scheduling, and various other system tasks.
- **Wait on I/O time** — The amount of time the CPU is waiting for another resource (such as disk I/O).
- **Idle time** — The amount of unallocated time for the CPU. If there are no jobs in the process queue and the CPU is not waiting for a response from some other resource, then the time is logged as idle. On some systems, such as Windows, wait on I/O is logged as idle. This is because the CPU is idle and waiting for a response. However, this time does not accurately reflect the state of performance on the system.

Tuning your system

If I/O wait is having a noticeable impact on performance, it is clearly an issue, however not all wait for I/O is cause for concern. Given that CPUs are much faster than other resources, the CPU must wait for these slower resources some percentage of the time.

When tuning a system, it is desirable to have some idle time available, but is not a requirement. It is possible to use 100 percent of the CPU time on a two-CPU system with two processes. This does not mean that the system was performing poorly; rather it means the system is running as fast as possible. In performance tuning, you try to push the bottleneck to the fastest resource, the CPU. Under ideal circumstances, CPU activity is divided into 70 percent user time, 20 percent system time, 0 percent wait for I/O time, and 10 percent idle time.

The ratio of user time to system time should be approximately 3 to 1. However, this ratio varies widely when user time is below 20 percent of total CPU usage, as system time tends to consume a much larger portion. In some cases, system time is greater than user time due to poor allocation of resources. However, as you increase user time through performance tuning, system time should level off at one-third or less of user time. This can be determined by looking at your CPU resources from monitoring tool.

Indications that you are out of CPU resources might be masking an issue with another resource. In most cases it is a disk issue. If you see a CPU bottleneck, first determine that you do not have a runaway process, and then make sure that your other resources are working efficiently.

CPU usage and the -spin parameter

Broker-allocated shared memory can be updated by multiple clients and servers. To prevent shared memory from being updated by two users simultaneously, OpenEdge uses spin locks. Each portion of memory contains one or more latches to ensure that two updates cannot happen simultaneously. When a user modifies a portion of shared memory, the user gets a latch for that resource and makes the change. Other users who need the resource respect the latch. By default, when a latch is established on a resource and a user needs that resource, that user tries for the resource once and then stops trying. On a single CPU system, you should only try the operation once because the resource cannot be freed until the resource that has the latch can use the CPU. This is the reason for this default action.

On a multiple CPU system, the default action is not very efficient because a significant amount of resource time is used to activate a different process on the CPU. This effort is wasted if the resource is not available. Typically, a resource is only busy and latched for a very short time, so it is more efficient to ask to obtain the resource many times rather than just once, and then go to the end of the CPU queue and then proceed through the queue back to the top before asking a second time to get the resource. The `-spin` parameter determines the number of times to ask before proceeding to the end of the CPU queue.

Generally, a setting between 2,000 and 10,000 works for the majority of systems, but this varies greatly. Monitor the number of naps-per-second per resource. If the naps-per-second value for any given resource exceeds 30, you might try increasing the value of `-spin`. This can be done while the system is running through the `PROMON R&D` option.

Understanding idle time

Idle time can be positive because it means that the CPUs have capacity to support growth. It is not necessary for idle time to always be greater than zero. If idle time is zero for prolonged periods and there is no significant amount of time logged to wait, you must look deeper into CPU activity to determine the correct course of action.

For example, look at the CPU queue depths. *CPU queue depth* is the number of processes waiting to use the CPU. If there are always several processes in the queue, you should do one of the following:

- Increase CPU capacity or increase CPU efficiency
- Reduce demand by either fixing code or moving processing to slow periods of the day

If the wait for I/O is high and there is no idle time, you need to either increase disk efficiency, reduce I/O throughput, or modify your processing schedule. If the wait for I/O is 10 percent or less and you still have idle time, you might want to look at those items outlined in the previous paragraph, but there no problem that requires urgent attention.

Fast CPUs versus many CPUs

The best scenario is having a large number of fast CPUs in your system. However, in many cases you must choose between having multiple, slower CPUs, or a single, faster CPU. The benefit of fast CPUs is that they run single-threaded operations very quickly, for example end-of-day processing such as applying all of your payments prior to running your general ledger trial balance. On the other hand, multiple CPUs allow you to do two different operations simultaneously, for example one user can be entering orders while another is shipping products. This has obvious benefits to the business in terms of efficiency.

The best way to decide is by looking at your options and your application to determine. For example, an application that does a significant amount of single-threaded operations will benefit from a design that has fast CPUs, even at the expense of having fewer total CPUs in the system. An application that is mostly data entry will benefit from a design that has more CPUs, even at the expense of each CPU being slower. This is another case where having knowledge of your application and workload allows you to make intelligent system decisions.

Tunable operating system resources

The OpenEdge RDBMS relies on operating system resources, to perform its operations. The following resources directly impact the performance of your database:

- **Semaphores** — Operating system constructs that act as resource counters. The database engine uses semaphores to synchronize the inter-process activities of server and self-service client processes that are connected to a database. By default, each database has an array of semaphores, one for each user or server. Each process uses its semaphore when it must wait for a shared resource. Semaphores are not used for single-user sessions. The Semaphore sets (`-semsets`) startup parameter allows you to change the number of semaphore sets available to the broker. You might also need to set some kernel or system parameters to increase the number of semaphores.
- **File descriptors** — An identifier assigned to a file when it is opened. There is a system limit on the number of file descriptors. Each database process (clients, remote client servers, and the broker) uses several file descriptors for both database files and application files. You might have to adjust the system file descriptor limit.
- **Processes** — Brokers, servers, clients, and background writers run as processes. Processes implement the client, the server, or the functions for both on a node. Processes also coordinate communication among

nodes and provide operating system and application services for the node. A kernel or system parameter in the operating system limits the number of active processes that can run on a system. You might have to raise the parameter to allow for more processes.

- **Shared Memory** — An area in system memory that multiple users can access concurrently. You might have to adjust both the maximum shared-memory segment size and the total number of available segments to improve system performance.

For information on setting kernel and system parameters, see your operating system documentation.

5

Database Administration

As a database administrator, you have many responsibilities.

For details, see the following topics:

- Database administrator role
- Ensuring system availability
- Safeguarding your data
- Maintaining your system
- Daily monitoring tasks
- Periodic monitoring tasks
- Periodic event administration
- Profiling your system performance
- Summary

Database administrator role

The specific tasks of the database administrator vary from company to company, but the role usually includes the following common responsibilities:

- **Providing predictable system availability** — To provide reliable system availability to users, the administrator must understand the state of the system at any given moment, and where it is going in terms

of resource utilization, This involves knowing how the system works and having a deeper understanding of the trends in system utilization over time. It also requires a complete understanding of the business, which can only be provided by the management of the company. The database administrator is charged with translating business plans into technology choices.

- **Providing a resilient system that can recover from disasters** — The database administrator must look at potential problems and determine if and how these problems are addressed by the disaster recovery plan. It is important to document items that will not be addressed and those items that will be addressed in the disaster recovery plan. The focus of the plan should be on recovery rather than on the backup process; a backup has no value if you cannot recover it in the event of a disaster.
- **Providing reliable system performance** — Users must know how long a task is going to take so they can plan schedule their tasks accordingly. The application plays a significant role in overall performance, however it is the administrator's job to ensure that tasks take the same amount of time every time they run, by taking advantage of system resources and eliminating bottlenecks.
- **Performing periodic maintenance** — This task might only be performed once a year, but must be considered in a complete administration plan. The small details of each task might change at the time of execution, but the overall process should be laid out in advance.

Security administrator role

For many installations, the role of the security administrator is assumed by the database administrator. The security administrator addresses issues surrounding the following topics:

- **Authentication** — Authentication assures that the identity asserted by one entity can be validated as authentic. For example, when a user logs on to an application, the authentication service assures that the user is permitted to access the application and its features.
- **Authorization** — Authorization grants or denies an entity access to capabilities based on the entity's validated identity.
- **Auditing** — Auditing is the action of generating a trail of secure and nonrepudiatable events during the execution of a business process. Auditing provides a history of application and data events that can be used to validate that all audited users and components, and their actions, were both anticipated and legal in the context of the application and its environment.
- **Encryption** — Transparent data encryption provides for data privacy of specified database objects while the data is "at rest" in your OpenEdge database, regardless of the location of the database or who has a copy of it. OpenEdge combines various cryptography technologies and processes to provide the security administrator with control over who can gain access to private, encrypted data.

For detailed information regarding security topics, see *OpenEdge Getting Started: Core Business Services - Security and Auditing.*

Ensuring system availability

The primary goal of the database administrator is to make sure that data is available to users. Most OpenEdge-based applications require multiple resources to function properly. This section discusses monitoring those resources.

Running out of resources is the second most likely cause of system failure. (Hardware failure is the most common.) It takes persistence to ensure maximum system availability, and maximizing system availability is the role of the database administrator.

When determining the source of a problem, the first question to ask is, "What has changed?" If you are monitoring your system, you can determine if there is a change in the amount of work (reads, writes, commits) being done, the number of users on the system, or if the system is consuming more resources than usual.

Monitoring resources is extremely important because f you know how fast you are using additional resources, you can determine when you will run out. Monitoring also allows you to plan for growth and avoid potential problems within the database and at the system level. On most systems, disk capacity and database storage areas are the most dynamic areas in terms of growth.

Database capacity

It is important to understand how much data is in your database today, and how much growth is expected. On existing databases, you should first consider the storage area high-water mark. The high-water mark is established by the number of formatted blocks in an area. In an area with many empty (unformatted) blocks, data is allocated to the empty blocks before the system extends the last extent of the area. The goal is to never use the variable extent, but to have it available if necessary.

Plan your excess capacity to be sufficient to cover your desired uptime. Each environment has a different amount of required uptime. Some systems can come down every evening, while 24x7 operations might only plan to be shut down once a year for maintenance. With careful planning you can leave your database up for long periods of time without the needing to shutdown. In most cases, the OpenEdge database does not need to be shut down for maintenance.

Your operating system might need to be shut down more often than your database for maintenance or an upgrade. Examples of operating system maintenance include: clearing memory, installing additional hardware, or modifying the operating system kernel parameters. In Windows, it is generally necessary to reboot the system every 30 to 90 days to avoid problems, while on most UNIX systems once a year is more common. You must plan for growth to cover this period of uptime for your system.

Application load

One statistic that most administrators do not keep track of is the amount of work that is completed per day on their systems. By monitoring database activity such as commits and database requests, you can determine when the greatest workload on the system occurs, and the growth pattern of this workload over time.

Workload information can be valuable information. If you encounter a problem, you can see whether there is an abnormal workload on the system or if there is some other issue. In cases where additional customer records are added to the database, you might notice that the workload on the system is increasing even though the number of users and the number of transactions are not increasing. This indicates that there might be an application efficiency issue that needs to be addressed before it becomes a problem. It will also help you to understand your need for additional resources prior to loading more records into the database. By knowing this information prior to the event, you can plan effectively.

System memory

Memory usage increases as additional users and additional functionality are added to the system. There is a dramatic change in performance when the memory resource is exhausted. The amount of paging and swapping are key indicators to monitor. An administrator should focus on physical paging as the primary indicator of memory usage.

Additional factors to consider in monitoring performance

The following list identifies other factors you should consider when monitoring performance:

- One of the greatest impacts on performance and availability of data over time is a fluctuation in the number of users.
- The volume of data involved is another factor to consider when evaluating performance.
- A poorly written query on a large amount of data causes a performance impact.

Remember that performance impact is not always linear. An example of this is a repeating query that scans the entire table for a particular record or pattern. When the table is small, all of the information can be stored in memory. But once the table grows beyond the size of the buffer pool, it can cause a significant amount of degradation to the system due to continual physical reads to the disk. This not only affects the query in question, but all other users accessing the database.

Testing to avoid problems

The best way to avoid problems on your system is to test prior to implementation. Most users only think of testing their application code. It is also necessary to test other aspects of the system including administration scripts and applications, such as backup software, hardware, middleware, and other infrastructure.

Testing is a long and meticulous process if done properly. There are three types of testing:

- **Limits testing** — Exceeds hardware and software system limits and ensures system reliability
- **End-to-end testing** — Examines an entire operation, checks the integrity of individual processes, and eliminates compatibility issues between processes, for example, looking at the order entry process from the customer's phone call to the delivery of goods
- **Unit testing** — Examines a process in isolation; unit testing should be done during early development and again, prior to implementation, as the initial step in the user acceptance process

You can also run tests on the individual system hardware components in isolation to ensure there are no faults with any item. Once this testing is complete, you can run a stress test to test the items together. A well designed test includes your application components, making an end-to-end test possible. For a complete check of the system, execute the stress test while running at full capacity, and then simulate a crash of the system to check system resiliency.

Safeguarding your data

Resiliency is the ability to recover in the case of a disaster. The goal in a disaster is to appear to the outside world as if nothing has happened. In the case of a Web site, if a customer or potential customer cannot access your site, that customer might be lost forever. OpenEdge supports many features to improve availability of your database in the event of system failures, such as failover cluster support, and OpenEdge Replication, but the most basic element of a recovery strategy is a good backup and a solid restoration process.

This section addresses basic questions regarding the creation of an effective recovery strategy and shows you where to invest your time and energy to create a complete solution. It is referred to as focusing on a recovery strategy. You want to focus on recovering from various situations, and not merely on the process of backing up your system. For example, if you have a good backup of your database, but you neglect to back up your application code, your ability to recover is in as much jeopardy as if you did not have a complete database backup.

Why backups are done

More and more companies rely on online systems to do very basic portions of their business. If a system is down for any period of time, it affects the ability of these companies to do business. Consequently, it is important to protect your data and applications. Problems can occur for several reasons, including:

- Hardware failure
- Software failure
- Natural disaster
- Human error
- Security breach

The goal of a complete backup strategy is to appear to the outside world as if nothing has happened, or at worst to minimize the amount of time that you are affected by the problem. A secondary, but equally important, goal is to reduce or eliminate data loss in case of a failure.

The best way to increase system resiliency is to prevent failure in the first place, and the best way to do this is to implement redundancy. Including disk mirrors in your system design minimizes the probability of hardware problems that cause system failure. You can also include OpenEdge Replication in your system to maintain an exact copy of your database.

Even with redundancy it is possible to encounter other issues that will cause a system outage. This is the reason to implement a complete backup strategy.

A complete backup strategy considers these factors:

- Who performs the backups
- Which data gets backed up
- Where the backups are stored
- When the backups are scheduled
- How the backups are performed
- How often the current backup strategy is reviewed and tested

A backup strategy must be well-designed, well implemented, and periodically reviewed and, if necessary, changed. Sometimes, the only time a problem is found is when the backup is needed, and by then it is too late. When systems change it is often necessary to modify the backup strategy to account for the change. You should periodically test your backup strategy to ensure that it works prior to a problem that precipitates its need.

Creating a complete backup and recovery strategy

A complete backup strategy should balance the probability of a problem occurring with the amount of data that would be lost in a given situation, with the amount of time and resources spent on backups. A disk failure is a more likely event than a fire or a flood. To reduce cost of data loss in the event of disk failure, you should have redundancy at the disk level. A fire or flood is less likely, and it is understood that some data would be lost should such an event occur.

It is important to include the users in the disaster-planning process. They are the real owners of the data, and can provide input to help decide the probability/data loss/cost trade-offs. It is sometimes helpful to put a price on lost data and downtime because it makes it easier to do the final cost/benefit analysis.

While it is possible to provide near-100-percent reliability, there is a large cost. For example, an emergency services organization needs as close to 100 percent reliability as possible, since a failure on its part could cost lives. It must have complete, duplicate systems at a second location, with available staff in case the primary location is knocked out by a disaster. Your situation might allow you to lose some data and availability of the system in exchange for lower cost. It is important to focus on the entire system and not just your databases and applications—you must also weigh the cost/benefit of each strategy.

The sections that follow identify the key questions to ask when developing a complete backup-and-recovery strategy.

Who does the backup?

In most cases the system administrator performs backups of the system on a regular basis. When the system administrator is unavailable, other personnel must be responsible for the backups. It is important to have a documented process so that backups are created consistently.

What does the backup contain?

An application consists of many diverse components, including databases, application code, user files, input files, and third-party applications. Remember that your application includes:

- Your OpenEdge installation and its associated service packs
- Your database and all related files
- Your application source and object code
- Middleware such as the AppServer
- Associated operating system files

The best way to determine what needs to be backed up in your organization is to walk through the vital processes and note activities and systems such as:

- The systems that are involved
- The software application files
- The data that is used throughout the process

Where does the backup go?

The media that you use for backups must be removable so they can be archived off site to protect data from natural disaster. Always consider the size of your backup in relation to your backup media. For example, tapes with a large storage capacity are a practical and reliable option to back up a 20GB database.

Tape compatibility is also a consideration because you might want to use the backup tapes on more than one system. This allows you to back up on one system and then restore to another system in the event of a system failure. A Digital Linear Tape (DLT) is supported on many platforms and can be used to either move data from one system to another, or to retrieve an archive.

Archiving off site is as important as the backup itself. If a fire, flood, or other natural disaster destroys your building, you can limit your data loss by having your backup at a separate location. A formalized service can be utilized, or you can simply store the backup tapes at a different facility. It is important to make sure you have access to your archives 24 hours a day, seven days a week.

How to label a backup

Proper labeling of your backup media is essential. Every label should contain the following:

- The name of specific items stored on the tape. A tape labeled "nightly backup" has no meaning if you cannot cross reference the items contained in the nightly backup.
- The date and time when the tape was created. In situations where multiple tapes are made for one day, you must know which tape is more current.
- The name or the initials of the person who made the tape. This ensures accountability for the quality of the backup.
- Instructions to restore the tape. There should be detailed restore instructions archived with each tape. The instructions should be easy to follow and should include specific command information required to restore the backup.
- The volume number and total number of volumes in the complete backup set. Labels should always read "Volume n of n."

When do you do a backup?

You should perform a backup as often as practical, balancing the amount of data loss in a failure situation against the interruption to production that a backup causes. To achieve this balance, consider these points:

- Static information, like application files that are not being modified, only need to be backed up once a week.
- Most database application data should be backed up at least once a day.

In cases where data is backed up once a day, it is possible to lose the work of an entire day if the disks fail or a natural disaster strikes at the end of the day. If you perform multiple backups throughout the day but only archive once a day, you are be better protected from a hardware, software, or user error, but your protection from most natural disasters is identical. By moving the intra-day tapes from the computer room to a different area of your facility, you decrease the probability of a fire in the computer room destroying your tapes.

Using PROBKUP versus operating system utilities

Among database and system administrators there are debates concerning the use of the OpenEdge `PROBKUP` utility versus operating system utilities. There is also a great deal of misinformation about when and how to back up a system while leaving it online. The following sections discuss these issues in detail.

Understanding the OpenEdge PROBKUP utility

The OpenEdge `PROBKUP` utility was created to back up databases. It has many nice features that make it unique, including:

- The `PROBKUP` utility is *database-aware*.

  Database aware means that the utility understands the structure and format of the database. `PROBKUP` can scan each block to ensure it is the proper format during the backup process. It takes longer to do this scan, but the added integrity check of potentially seldom-used blocks is worth the small performance degradation.

  Also, if the structure of the database changes, the syntax of the `PROBKUP` command does not need to change when the structure of the database changes, regardless of the number of disks/areas/extents.

- `PROBKUP` has an online option.

This online option allows you to back up a database while the database is available to users.

How PROBKUP works

The following steps briefly identify the PROBKUP process:

1. Establish a database latch (online only).
2. Do a pseudo checkpoint (online only).
3. Switch AI files (if applicable).
4. Back up the primary recovery area (online only).
5. Release the database latch (online only).
6. Back up the database.

The database is backed up from the high-water marks downward. Free blocks are compressed to save space. Online backups represent the database at the time the backup started. All transactions started after the backup begins are not in the backup, and will not be in the database when a restore and transaction rollback occurs.

The reason for the pseudo checkpoint in an online backup is to synchronize memory with disk prior to backing up the database. This synchronization is critical because then the PROBKUP utility can back up all of the data that is in the database and in memory at that moment. Other utilities can only back up the information on disk, thus missing all of the "in memory" information.

Adding operating system utilities to augment PROBKUP

The PROBKUP utility backs up the database and primary recovery area. This utility does not back up the after-image files, the key store, or the database log file.

All complete backup strategies use operating system utilities to back up additional important files on the system, such as programs, application support files, and user home directories. Some administrators choose to back up their database to disk with PROBKUP, and use an operating system utility to augment the database backup with other files to get a complete backup on one archive set. After-image files should be backed up to separate media from the database and before-image files to increase protection. For more information, see the After-imaging implementation and maintenance on page 98.

After-imaging implementation and maintenance

After-imaging provides an extra layer of protection around the database. Every high-availability system should implement after-imaging. It is essential that you have a good backup and recovery plan prior to implementing after-imaging.

Once started, the OpenEdge after-image feature keeps a log of all transactions that can be rolled forward into a backup copy of the database to bring it up-to-date.

The primary reason to enable after-imaging is to protect you from media loss. Media loss can be the loss of a database, a primary recovery area disk, or a backup tape. In the case of a lost backup, you can go to the previous backup and roll-forward to bring the system up-to-date. This assumes that your after-image backup was not stored on the same piece of media as the database backup that you were unable to recover. This is the main reason for doing a backup of your data on one tape or tape set and your after-image files to a second tape.

After-imaging can also be used to keep a warm standby copy of your database. This standby database can be stored on the same system as the primary copy. However, for maximum protection, you should store it on a different system. When you use after-imaging to implement a custom warm standby replication solution, you periodically update your standby database by transferring after-image files from the primary database and applying or rolling-forward those files to the standby database. In the case of a system failure, you apply the last after-image file to the standby database, and start using the standby database. If it is not possible to apply the last after-image file to the database, you only lose the data entered since the last application of the after-image file. In the event of a system failure, having implemented a warm standby database typically results in significantly less downtime for the users than if the database needs to be restored from a backup.

OpenEdge Replication

OpenEdge Replication provides hot standby capabilities. OpenEdge Replication has two major real-time functions:

- To distribute copies of information to one or more sites
- To provide failure recovery to keep data constantly available to customers

OpenEdge Replication automatically replicates a local OpenEdge database to remote OpenEdge databases running on one or more machines. Once OpenEdge Replication is installed, configured, and started, replication happens automatically. OpenEdge Replication offers users the ability to keep OpenEdge databases identical while also providing a hot standby in the event a database fails. When a database fails, another becomes active. Therefore, mission-critical data is always available to your users.

OpenEdge Replication provides the following benefits:

- Availability of mission-critical data 24 hours a day, seven days a week
- Minimal or no disruption in the event of unplanned downtime or disaster

OpenEdge Replication provides the following key features:

- Automated, real-time replication of databases for failover or disaster recovery
- Failback functionality
- A single source database and one or two target database configurations
- Data integrity between source and target databases
- Continued source database activity while administration tasks are being performed
- Replication activity reporting
- Online backup of source and target databases

For details on OpenEdge Replication, see OpenEdge Replication: User Guide.

Testing your recovery strategy

The only valid backup is your last tested backup. This underscores the need to test your backup strategy on a regular basis. This does not mean that you should delete your production database and restore from backup to see if it works. It is best if you can restore your database to a different location on disk, or better yet to a different system.

How often should you test your backup? If you are using an operating system utility to back up your database, it is a good idea to test your backup any time you modify the structure of your database. Since PROBKUP is database-aware you might not need to test for structure changes. However, you should still test at least twice a year.

You must test the backup and the process when there are changes in staff or in the event that the administrator is out of town. Any member of the IT staff should be able to perform a well-documented recovery process. If you do not have an IT staff, you must ensure that each person who might be required to restore the system can follow the documented recovery procedures.

Your recovery procedures should be scenario-based. Common recovery scenarios, such as a loss of a disk (for example, database, after-image, or application files), fire, flood, and so on, must be clearly documented with step-by-step instructions to describe how to determine the root cause of the problem and how to recover from the event.

You hope you will never need to use your recovery plan, however, a well documented and thoroughly tested plan is essential to your business.

Maintaining your system

Do not let the ease of OpenEdge maintenance allow you to become complacent about maintaining your system. The health of the system should be monitored every day. This can be done using a tool such as OpenEdge Management or by manually using scripts.

The areas that are most important to monitor are the areas that will cause a performance problem or that will cause the database to stop running. While the issues that cause the database to stop running are the most important to identify **before** they fail, performance problems are often equally important to system users. A slow system erodes the users' confidence in the system, and they will look elsewhere for their information. For example, a slow Web site can drive users to go elsewhere for information, and they might never return.

Daily monitoring tasks

It is easy to become complacent your system runs smoothly. Unless you set up a routine of daily tasks, you might put system monitoring at the bottom of your list of priorities. Your database or system might slip into a problematic state when you least expect it. Establishing and executing a simple set of daily tasks can prevent problems.

The sections that follow describe the resources you should monitor on a daily basis.

Monitoring the database log file

The database log file contains a wealth of information about database activities. OpenEdge places many pieces of information in the file, beyond simple logon and logoff information. For example, when the database is started, all of your startup parameter settings are placed in the log file. This information can be used to verify the settings in your parameter file (`.pf`) or your `conmgr.properties` file.

Regularly pay attention to error log entries in the log file because OpenEdge places serious errors in this file when they occur. Most users do not report an error to the administrator unless it happens more than once. Error details are automatically recorded in the log file, providing an administrator accurate data so that the appropriate corrective action can be taken.

The log file for a given database should be truncated on a regular basis to keep the length manageable. The `PROLOG` utility is used to keep the log file manageable. This utility should be run after you back up the log file.

Key Events

Enabling the database to save key events stores a streamlined account of the information written to the database log file within the database in the `_KeyEvt` table. Saving key events can provide the database administrator or a Progress technical support engineer, an efficient record of critical events in the history of the database. When the database is enabled to save key events, an additional background database process is created to periodically scan the database log file, creating records of the key events in the `_KeyEvt` table. For more information on key events, see *OpenEdge Data Management: Database Administration*.

Monitoring area fill

Check the fill rate of area extents every day. You need to know if your database is growing beyond the planned capacity so you can plan an to address increasing the size of the area when it is convenient for the users, particularly if downtime is required.

Monitoring buffer hit rate

The buffer hit rate measures the percentage of time the system is retrieving records from memory versus from disk. You should view the buffer hit rate throughout the day to ensure that the users are consistently getting acceptable performance. A goal for the buffer hit rate is 95 percent.

Monitoring buffers flushed at checkpoint

Buffers flushed at checkpoint are a good indicator of APW and checkpoint efficiency. The APW's job is to keep writable buffers at a low count. You do not want frequent checkpoints where the database and memory are synchronized. If you are seeing an increase in buffers flushed during your prime operating times and they cannot be attributed to an online backup, then you need to make adjustments.

Monitoring system resources (disks, memory, and CPU)

The system resources that require daily monitoring can vary from system to system, but disk fill rate, memory consumption, and CPU utilization are generally resources that must be monitored daily. Memory and CPU are less important because if over used, these resources will generally cause a performance issue, but overusing disk resources can cause an outage.

Periodic monitoring tasks

There are tasks that you must perform on a periodic basis to promote monitoring of your system. Some of these tasks, like database analysis, should be done monthly. Other tasks, such as a compacting an index, should only be done as needed.

Periodic OpenEdge maintenance tasks include:

- Database analysis on page 102

- Rebuilding indexes on page 102
- Compacting indexes on page 102
- Fixing indexes on page 103
- Moving tables on page 103
- Moving indexes on page 103
- Truncating and growing BI files on page 103
- Dumping and loading on page 104

Database analysis

The database analysis utility, `PROUTIL DBANALYS`, generates table and index storage information. You should run this utility at least once every three months. You can run this utility while users are accessing the database with low-to-moderate impact on performance. Use the utility's information to determine if you must rebuild your indexes or if you need to make modifications to your database structure.

The database analysis report details table storage information and helps you determine if tables must be moved to other areas or reorganized to reduce scatter. However, the area of index utilization is generally more dynamic and must be analyzed on a regular basis.

Index efficiency is important. If your data is 100 percent static, then you want your index utilization to be 100 percent to provide you with the maximum number of index entries per block. Unfortunately, this is not the case with most applications. Most applications perform substantial numbers of inserts and modifications, which impact the indexes. You should have sufficient space left in the index blocks to add additional key values without introducing the need for an index block split.

Rebuilding indexes

The purpose of an index rebuild, `PROUTIL IDXBUILD`, is to increase index efficiency and to correct errors. Use this utility when index corruption forces a rebuild of an index or when you can take your database offline for index maintenance. Your database must be offline when the index rebuild utility is run. You can use a combination of online utilities, including index fix and index compact, to approximate the effect of an index rebuild.

You will get significantly better organization within the indexes by sorting them prior to merging them back into the database. You can do this by opting to sort when running the index rebuild. Be aware that sorting requires substantial disk space (50 to 75 percent of the entire database size when choosing all indexes).

You cannot choose a level of compression with index rebuild. The utility tries to make indexes as tight as possible. While high compression is good for static tables, dynamic tables tend to experience a significant number of index splits right after an index rebuild runs. This affects the performance of updates to the table. Even with the decrease in update performance, the overall benefit of this utility is desirable. The performance hit is limited in duration, and the rebuild reduces I/O operations and decreases scatter of the indexes.

Compacting indexes

The index compact utility, `PROUTIL IDXCOMPACT`, is the online substitute for index rebuild. Run this utility when you determine that the indexes are not as efficient as desired and the database cannot be taken offline to perform an index rebuild. You can determine inefficiencies by reading the output of the database or index analysis utilities.

The benefit of the index compact utility over index rebuild is that it can be run with minimal performance impact while users are accessing the database. The resulting index efficiency is usually not as good as a sorted index rebuild, but the cost saving from eliminating downtime, and the minimal performance impact generally make this the preferred option.

The index compact utility allows you to choose the level of compression that you want for your index blocks. The default compression level for this utility is 80 percent—ideal for nonstatic tables. If your tables are static, you might want to increase the compression level. You can not choose a level of compression that is less than your present level of compression.

Fixing indexes

The index fix utility, `PROUTIL IDXFIX`, corrects leaf-level index corruption while the database is up and running, with minimal performance impact. Run this utility if you get an error indicating an index problem. If the index error is at the branch level rather than at the leaf level, you must use the index rebuild utility to correct the problem.

Moving tables

The purpose of the table move utility, `PROUTIL TABLEMOVE`, is to move a table from one storage area to another. This allows you to balance your I/O or group similar tables. You can run this utility while users are accessing the database, but users will be locked out of the table being moved for the duration of the move. The table move utility requires a significant amount of logging space which affects the primary recovery and after-image areas. In the primary recovery area, logging of the move process uses three to four times the amount of space occupied by the table itself. If you did not plan accordingly, you run the risk of crashing the database because of a lack of space in the primary recovery area. You should test your table move on a copy of your database **before** using it against production data.

Moving indexes

The index move utility, `PROUTIL IDXMOVE`, moves an index from one storage area to another. It works in the same manner as the table move utility and has the same limitations and cautions. See the Moving tables on page 103 for details. Note that, just as with the table move utility, you should test the index move utility on a copy of your database **before** using it against production data.

Truncating and growing BI files

The before-image file or primary recovery area varies in size, depending on the transaction scoping within your application. Sometimes you experience abnormal growth of this area, such as when you make schema changes or wholesale changes to a table in the database. These circumstances warrant truncating the `BI` file to recover the space. Use `PROUTIL TRUNCATE BI` to truncate your `BI` file.

You do not want to truncate the `BI` file unnecessarily. When the `BI` file is truncated for any reason, OpenEdge must reformat the recovery space to make it usable. If this reformatting is done while the system is running, it can cause noticeable performance degradation. Consequently, you should size your `BI` file to accommodate normal growth. You can extend your `BI` file with the `PROUTIL BIGROW` utility by specifying the number of clusters by which you want to grow the `BI` file. By pre-allocating these `BI` clusters and eliminating the formatting of clusters during normal processing, you can eliminate a substantial performance drag on the system.

Dumping and loading

Determining when to perform a dump and load is a constant struggle. If you set up your database correctly, your records should not get fragmented, and only your indexes should need reorganizing. Index reorganization is the primary benefit of a dump and load. Usually, about 80 to 90 percent of the benefit of a dump and load can be achieved with index maintenance using the index rebuild and compact utilities. However, you might need to dump and load to reorganize your database into different areas or extents due to changes in application requirements.

A fast dump and load process is important, since the system is not available to users during the process. Before performing a dump and load, it is important to have a good backup of your existing database. This backup provides you with a fallback position should something go wrong.

The basic dump and load options are as follows:

- Data Dictionary dump and load on page 104
- Bulk loader on page 104
- Binary dump and load on page 104

Data Dictionary dump and load

The option of using the Data Dictionary dump and load is viable, provided you follow these rules:

- Multi-thread both the dump and the load. Generally, you should add sessions on both the dump and load until you cause a bottleneck on the system.
- Use all of the disks on the system evenly to achieve maximum throughput. For example, you might want to make use of your `BI` disks because they will be idle during the dump portion of the process.
- Leave the indexes enabled during reload. This does not make for efficient indexes due to index splits during the load process, but since the indexes are built at the same time the data is loaded, you can take advantage of the multi-threaded nature of OpenEdge. The indexes can be reorganized later through an index compress.

Bulk loader

This option is simple. The bulk loader files are loaded sequentially with the indexes turned off, necessitating an index rebuild after the load completes. The bulk load process itself is fairly quick, but it is not possible to run multiple instances of this utility simultaneously. You must run an index rebuild after all your bulk loads are processed.

Binary dump and load

The binary dump and load is much faster than the previous methods described. It allows for multi-threading of both the dump and the load. It also supplies the option to build indexes during the load, eliminating the need for a separate second step to build indexes. This utility, when used with the index build option, provides the best overall dump and load performance in the majority of cases.

Periodic event administration

There are some tasks that are only performed occasionally on the system. These tasks require thought and advance planning to decrease their impact and maximize the benefit to the organization. Many of the items listed in this area can be considered luxury items, but if you are well positioned in the other aspects of your system, you can invest some time in these tasks to round out the management and maintenance of your system.

Periodic events include:

- Annual backups on page 105
- Archiving on page 105
- Modifying applications on page 106
- Migrating OpenEdge releases on page 106

Annual backups

The annual backup is generally viewed as a full backup of the system that can be restored in the event of an emergency. The most common use of the annual backup is for auditing purposes. These audits can occur several years after the backup is taken, so it is very important to be able to restore the system to its condition at the time of that backup. In the United States, it is possible for the Internal Revenue Service to audit your company as far back as seven years. How likely is it that you will be on the same hardware seven years from now? You might be on compatible hardware, but most likely you will be on different hardware with a different operating system. Consequently, it is important to plan thoroughly for such an eventuality.

One way to guarantee platform independence is to dump your important data to ASCII and back it up on a reliable, common, and durable backup medium. Some people prefer optical storage over tapes for these reasons. Also, do not overlook the application code and the supporting software such as the version of OpenEdge being used at the time of backup. If you are not going to dump to ASCII, you must obtain a complete image of the system. If you take a complete image and are audited, you must find compatible hardware to do the restoration. It is also important to use relative pathnames on the backup to give you greater flexibility during the restoration. Finally, you must document the backup as thoroughly as possible, and include that information with the media when sending the backup to your archive site.

Archiving

A good IT department always has a complete archiving strategy. It is generally not necessary to keep transactional data available online for long periods of time. In most cases, a 13-month rolling history is all that is necessary. This can and will change from application to application and from company to company. You need a thorough understanding of the application and business rules before making a decision concerning when to archive and how much data to archive. In most cases, you should keep old data available offline in case it is needed. In these scenarios, you should develop a dump-and-purge procedure to export the data to ASCII. This format is always the most transportable in case you change environments or you want to load some of the data into another application such as Microsoft Excel. Always make sure you have a restorable version of the data before you purge it from the database. An archive and purge can dramatically improve performance, since the system has far fewer records to scan when it is searching the tables.

Modifying applications

Changes to applications require careful planning to reduce interruptions to users. Although there might be a process to test application changes at your site, database administrators should consider it their responsibility to verify expected application changes. The most effective way to do this testing is to have a test copy of your database that is an exact image of what you have in production, and a thorough test plan that involves user participation.

Making schema changes

Schema changes can take hours to apply if they are not done properly. If the developers tested the application of schema changes against a small database, they might not notice a potential problem. A small database can apply an inefficient schema update in a short period of time and will not raise any red flags. If you have a full-size test environment, you can apply the schema change and know approximately how long it will take to complete. It is important to understand how long this process takes, since the users of the application are locked out of parts of the system during the schema update.

Making application code changes

The amount of time it takes to apply application code changes can be greatly reduced by an advance compilation of your code against a CRC-compatible copy of your production database. To maintain CRC compatibility, start by creating a basic database, which is one that contains no data—only schema definitions. Use the basic database to seed a production database and a development database. The basic database is also saved, so you will have three copies of your database. If you already have a production database in place, the basic database is obtained by dumping the schema from that database.

As development occurs on a test copy of the database, the production and basic databases remain unmodified. When you are ready to promote your schema changes from development to production, first make an incremental data definition dump from the OpenEdge Data Dictionary by comparing the development schema with the basic database. The incremental data definitions can be applied to the basic database, and you can compile your application against that database. Second, the incremental data definitions can be applied at a convenient time on the production database (after appropriate testing). While the incremental data definitions are being applied, you can move the r-code you created against the basic database into place, avoiding additional downtime to compile the application code.

Migrating OpenEdge releases

In most cases, migrating to a new release of OpenEdge, particularly a minor release upgrade, can be made without running any conversion utility. It is important to test even minor release upgrades and service packs or patches in your test environment prior to promoting the code to production.

When making a major release upgrade, you need to do additional analysis prior to making any changes. Major releases require that you test the conversion process for performance and reliability prior to applying the new release to your production environment.

Generally, the actual conversion of the database will only take a few minutes, so it is not a major undertaking to convert the test environment and verify that conversion. After the verification has been done on the test database, you can decide how to proceed with the production application and databases.

You might want to do a more complex conversion if for example, you have a significant amount of record fragmentation in the database. A complex conversion might start with a simple conversion to take advantage of new features in the new release, but then you add a dump and reload of your tables to establish a new database structure. By using a multi-phase approach, you can minimize your risk and provide a fallback position if there are problems during the migration. It is imperative to have good, tested backups before applying a new release of OpenEdge or application code to the system. This is also true when applying minor releases and patches to the system.

Profiling your system performance

Performance is a matter of perception. Users may say performance is slow or fast based on many factors. It is best to take perception out of the equation by establishing some baselines on important aspects of your system.

Establishing a performance baseline

The primary reason to establish a performance baseline is to enable you to quantify between changes in performance and changes in your load or application. The most difficult part of establishing a baseline is determining which operations are critical to the effective use of the system. You want the list of operations to be complete and concise. Too many items increases the amount of work needed to establish reliable baselines.

The basic rules outlined below allow you to narrow down the number of operations that need baselines to a manageable number. You might want to have two sets of baselines—one for daytime processing and one for end-of-day processing.

When generating your list of operations to baseline, include the following:

- Tasks that are done many times throughout the day (such as creating orders, process control, and data entry tasks)
- Tasks that are important to the user (such as Web site queries, customer support screens, and order entry tasks)

When generating your list of operations to baseline, do not include the following:

- Periodic tasks (such as monthly and weekly reports)
- Little-used portions of your application
- Reporting, as it generally falls into the above two categories, and you can schedule most reporting outside of your primary operating hours

Collecting your baseline statistics

Once you have determined what operations you want to benchmark, you can plan your strategy.

You can modify the application code to collect benchmark data, which is the most accurate method, but it is also time consuming and costly. An easier way to perform data collection is to time the operations on a stopwatch. This is fairly accurate and easy to implement. To determine the best timing baseline for each task, perform timing in isolation while nothing else is running on the system. When timing baselines have been established, repeat the task during hours of operation to establish your under-load baselines.

Understanding your results

Once your operation times have been calculated, you must analyze the results.

Remember, it is best to establish the baselines while there are no reports of any problems on the system to establish what is normal on your system. If users are reporting problems, you can compare the current timings against your baselines to see if the problem is real. If there is a material difference in the current timing, you must start analyzing performance on the system with monitoring tools such as `PROMON`, VSTs, OpenEdge Management, and operating system utilities.

Performance tuning methodology

Always analyze problems starting with the slowest resource and moving to the fastest. Thus, the first place to start is disks, then memory, and finally CPU efficiency. Before you begin to look at the system, you must make sure that the application is performing correctly. Correct application performance has the greatest effect on overall performance. This is easy to determine by looking at the number of database requests per user. If most users have tens of thousands of requests but a few users have millions of requests, you should ask those few users what they are doing with the system and look at those portions of the application for inefficiencies.

If these users are doing the same jobs as everyone else, the cause of the problem might be how they are using the application. This is more common with off-the-shelf applications than with custom applications. The business rules for the off-the-shelf application might not match your business process exactly. One example is an application that is written for companies with multiple divisions, but deployed at company with one division. The application expects the user to enter a division code as a unique identifier, but the employees at the single-division company, may not trained to enter a division code. The application might not have an index defined to find records efficiently without a supplied division code. While this can be rectified on the application side, you first must discover the problem before you can fix it. The solution might be to train users to enter a division code until an application modification can be made.

Once you have ruled out the application as a source of performance problems, you can start looking for common problems.

Summary

The ideal system plan accounts for all aspects of the system administration and maintenance, from those tasks performed daily to those tasks done on an irregular or occasional basis. If you plan and test properly, you can avoid potential problems and provide a predictable environment to users. In general, people do not like surprises. This is especially true with business systems, It is important to establish accurate schedules for all tasks because this builds user confidence in your system.

Index

A

Absolute-path database *59*
After image area *49*
After-image writers *82*
After-imaging *74–75*, *98*
Alternate Buffer Pool *87*
Annual backups *105*
Application data area *49*
Applications
  applying modifications *106*, *108*
  making code changes *106*
APWs *71*, *101*
Archived data *78*
Areas
  after image area *49*
  application data *49*
  control *49*
  primary recovery *49*
  schema *49*
  storage *48*
  transaction log *50*
Asynchronous Page Writers (APW) *82*
Auditing *92*

B

Backups
  annual *105*
  archiving *96*, *105*
  complete strategy *95*, *97*
  contents of *96*
  documenting strategy *99*
  how to label media *97*
  platform independent *105*
  reasons for *95*
  tape compatibility *96*
  testing *99*
  using operating system utilities *97*
  using PROBKUP *97–98*
  when to perform *97*
  who performs *96*
Batch mode *57*
Before-image writers *82*
BI (before-image) File *103*
Buffers
  hit rate *101*
  monitoring flushed at checkpoint *101*
Bulk loader *104*

C

Cache usage *79*
Columns
  defined *17*
Compound indexes *18*
Configurations
  database location *57*
  distributed database *57*
  federated database *57*
  system platforms *56*
conmgr.properties file *100*
Connection modes
  batch *57*
  interactive *57*
Connection Modes
  multi-user *57*
  single-user *56*
Control area *49*
CPU
  comparing fast versus many *89*
  idle time *87*, *89*
  managing activity *87*, *89*
  optimizing usage *87–88*
  queue depth *89*
  system time *87*
  user time *87*
  wait on I/O time *87*, *89*
  what to buy *89*
Cross-reference tables- *29*

D

Data blocks
  RM blocks *51*
  RM chain blocks *51*
Database
  activity *93*
  administrator roles *91*
  buffer hit rate *101*
  buffers *82*
  buffers flushed at checkpoint *101*
  Database Areas *65*
  dump and load *104*
  empty blocks *53*
  free blocks *53*
  high-water mark *53*
  index blocks *52*
  log file *100*
  master blocks *53*
  monitoring buffer hit rate *101*

Database *(continued)*
multi-volume extents *74*
optimizing data layout *61*
primary recovery area *71*
spin locks *88*
storage object block *53*
Database areas
before-image cluster size *71*
block sizes *72*
determining space to allocate *68*
distributing tables *69*
enabling large files *73*
extents *72–74*
index storage *70*
monitoring fill rate *101*
optimizing *72–73*
partitioning data *73*
primary recovery area *71*, *73–74*
sizing *65*
splitting off the schema *72*
using extents *69*
Database broker
memory requirements *84*
Database Configuration
distributed *57*
federated *57*
multi-tier *57*
Database Location *57*
Databases
absolute-path *59*
defined *15*, *29*
disk requirements *62*
distributed *57*
federated *57*
indexing *36*
metaschema *21*
normalization *30*
relative-path *59*
schema *21*
storage area location *50*
Demand page executables *84*
Disk capacity
managing *76*, *80*
Disk space
estimating *62*
Disks
cache usage *79*
comparing expensive and inexpensive *78*
data storage requirements *77*
determining current storage *77*
determining what to buy *78*
increasing reliability with RAID *79*
mirroring *79*
monitoring *101*
properties option in Windows NT *77*
striping *79*
Disks *(continued)*
swappable *79*
understanding data storage *76*
Dump and load
binary *104*
bulk loader *104*
data dictionary dump and load *104*

E

Empty blocks *53*
Examining growth patterns *78*
Extents *50*

F

Fields
defined *17*
foreign keys *27*
File extensions *47*
File structure *46*
Fixed-length extents *50*
Foreign keys *27*
Free blocks *53*

G

Growth patterns *78*

I

Idle time *89*
Index blocks *52*
Index Compact utility *102*
Index Fix utility *103*
Index Move utility *103*
Index Rebuild utility *102*
Indexes
advantages *37*
choosing tables and fields to index *40*
compound *18*
deactivating *43*
defined *18*
demo database *38*
disk space *41*
how indexes work *36*
reasons not to define *40*
record ID *41*
redundant *42*
size *41*
Interactive mode *57*

K

Key Events *101*

L

Latches
understanding 82
Locked record 82

M

Managing
disk capacity 76, 80
memory usage 81, 86
Many-to-many relationships 29
Master blocks 53
Memory
adding remote clients 82
AIWs 82
APWs 82
BIWs 82
decreasing 86
estimating requirements
for Progress 84
operating system 82
increasing usage 86
managing usage 81, 86
maximizing 81
optimizing usage 85
physical paging, See paging
private buffers 86
sample requirements 85
understanding internals 82
understanding shared memory resources 82, 84
virtual paging, See paging
Monitoring
daily tasks 100–101
database log file 100
periodic tasks 101, 104
your system 100, 104

N

Normalization
first normal form 31
second normal form 33
third normal form 34

O

One-to-many table relationships 29
One-to-one table relationships 28
OpenEdge Replication 99
Operating system
backup utilities 98
memory requirements 82
Operating System
resources
file descriptors 89
processes 89
shared memory 89

P

Paging
physical 81
virtual 81
Performance
establishing a baseline 107–108
profiling 107–108
tuning 108
Primary keys 18
Primary recovery area 49
PROBKUP utility 97–98
Progress
client process memory requirements 84
database broker memory requirements 84
migrating versions 106
PROLOG utility 100

R

RAID
hardware 79
software 79
Record ID 41
Records
defined 17
Recovery strategy 94, 97
Redundant array of inexpensive disks, See RAID
Relative-path database 59
RM blocks
layout 51
Rows
defined 17

S

SAN environment 80
Schema area 49
Schema changes 106
Security administrator role 92
Semaphores
defined 89
Shared executables 84
Shared memory
file descriptors 89
processes 89
semaphores 89
shared memory segments 89

Storage
logical 54
physical 54
temporary 64
variable-length technique 62
Storage area networks, See SAN
Storage areas
locating 50
transaction log area 50
Storage object block 53

T

Table Move utility 103
Table relationships
many-to-many 29
one-to-many 29
one-to-one 28
Tables
cross-reference 29
defined 17
Tables *(continued)*
normalization
first normal form 31
second normal form 33
third normal form 34
relationships 27
Temporary storage 64
Testing your system 94
Transparent Data Encryption 92

U

UNIX
file descriptors 89
processes 89
semaphores 89
shared memory 89

V

Variable-length extents 50