Information Management software

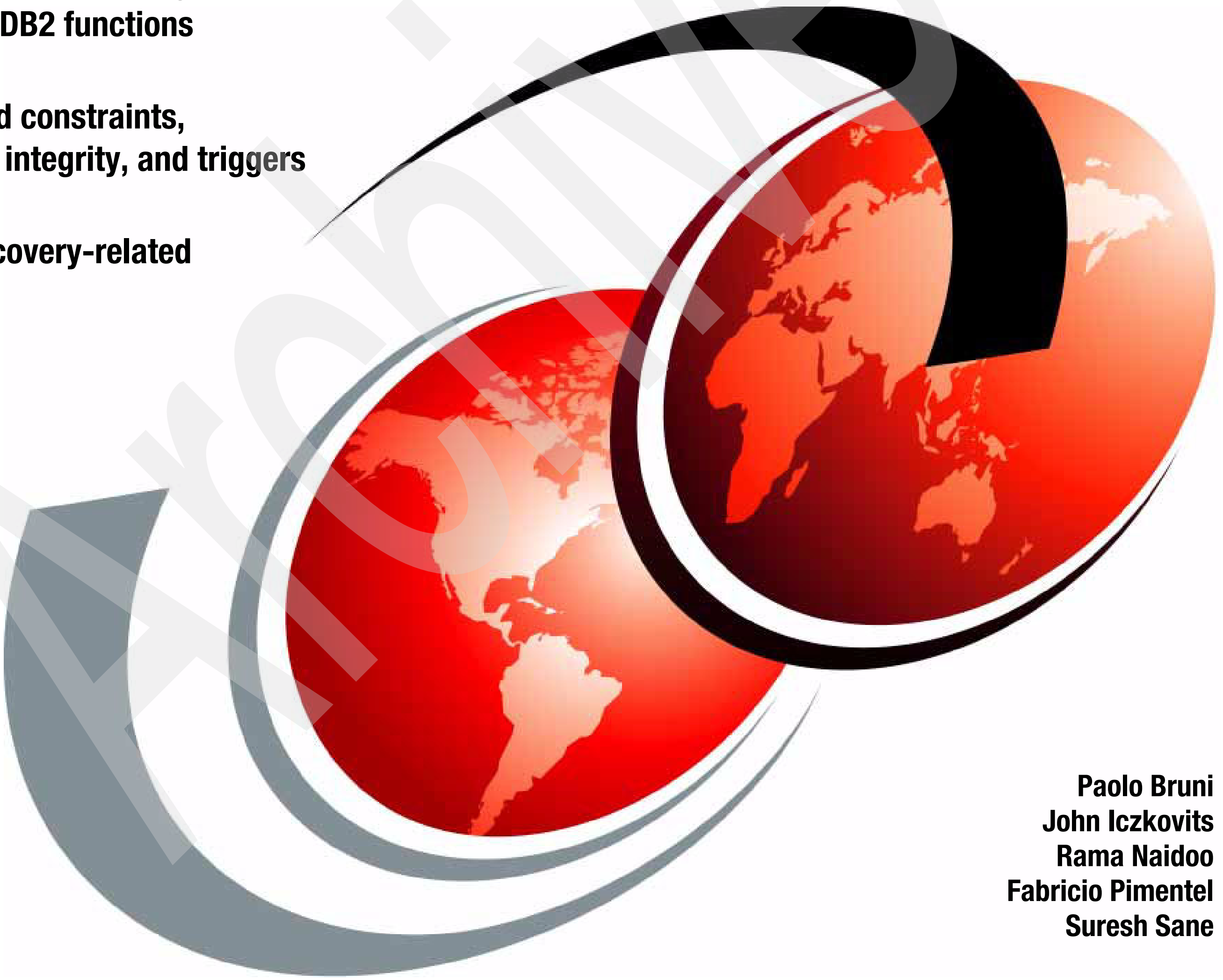
IBM

# Data Integrity with DB2 for z/OS

Assert information integrity by exploiting DB2 functions

Understand constraints, referential integrity, and triggers

Review recovery-related functions



Paolo Bruni
John Iczkovits
Rama Naidoo
Fabricio Pimentel
Suresh Sane

ibm.com/redbooks

Redbooks

International Technical Support Organization

# Data Integrity with DB2 for z/OS

July 2006

SG24-7111-00

**Note:** Before using this information and the product it supports, read the information in “Notices” on page xv.

**First Edition (July 2006)**

This edition applies to Version 8 of IBM Database 2 Universal Database for z/OS (DB2 UDB for z/OS), program number 5625-DB2.

# Contents

# Figures

# Examples

# Tables

# Notices

## Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

| | | |
|---|---|---|
| @server® | DFSMSdss™ | OS/390® |
| Redbooks (logo) ™ | DFSMShsm™ | Parallel Sysplex® |
| eServer™ | DFSORT™ | PR/SM™ |
| z/OS® | DRDA® | QMF™ |
| z/VM® | Enterprise Storage Server® | Redbooks™ |
| zSeries® | FlashCopy® | RACF® |
| z9™ | Geographically Dispersed Parallel Sysplex™ | S/390® |
| AIX® | GDPS® | SecureWay® |
| CICS® | IBM® | System z™ |
| Database 2™ | IMS™ | System z9™ |
| Domino® | MVS™ | System/390® |
| DB2® | | WebSphere® |

The following terms are trademarks of other companies:

EJB, Java, Java Naming and Directory Interface, JDBC, JVM, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

# Preface

DB2® provides functions to guarantee integrity at the system level and at the application level.

From the system point of view, DB2's integration with zSeries® and disk storage architecture is the cornerstone for data integrity. Logging functionality and COPY and RECOVER utilities are the building blocks for bringing the table space back to a current or consistent status in case of hardware or software failures or when application events need to be rerun.

From the application point of view, DB2 supports locking and commit at the transaction level, and general data integrity (at entity and semantic level) and a set of referential constraint rules for each parent/dependent table relationship. The tables linked by referential integrity are recognized during the execution of the QUIESCE utility. Other logical relations across tables, necessary to support business rules, are implemented via constraints, triggers, user defined functions, and user defined tables.

Informational constraints also exist, they are not enforced by the database manager, they are used to improve query performance.

In this IBM® Redbook, we briefly describe the integration of DB2 for z/OS® with System z™ architecture, we then explore the data integrity options and utilize the standard recovery functions for application-related issues.

The redbook is structured as follows:

- Chapter 1, “Introduction” on page 1
- Chapter 2, “Semantic integrity” on page 35
- Chapter 3, “Referential integrity” on page 49
- Chapter 4, “Triggers” on page 113
- Chapter 5, “Other integrity features” on page 153
- Chapter 6, “Recovery” on page 175

## The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

**Paolo Bruni** is a DB2 Information Management Project Leader at the ITSO, San Jose Center. He has authored several Redbooks about DB2 for z/OS and related tools, and has conducted workshops and seminars worldwide. During Paolo’s many years with IBM, previously as an employee and now as a contractor, his work has been mostly related to database systems.

**John Iczkovits** is a Consulting IT Specialist with IBM Advanced Technical Support (Dallas Systems Center) Americas. He provides DB2 for z/OS technical support and consulting services. His areas of expertise include DB2 data sharing, performance, and availability. His work in IBM includes experience supporting DB2 as a Systems Engineer, Database Administrator, IT Specialist, Consultant, and project lead. He has an operations background and 21 years of IT experience, ranging from database products such as DB2 and IMS™ to MVS™ systems support, including storage management. He has also co-authored IBM redbooks, white papers, and presented at SHARE, the DB2 Tech Conference, and local DB2 users groups.

**Rama Naidoo** is a Certified IT Specialist in Data/Content Management - Services from Australia. He has over 32 years of experience in data management and worked on several non-IBM database management systems before joining IBM in 1989. Rama co-authored the redbooks, *DB2 for OS/390 and Data Compression,* SG24-5261, and *DB2 for z/OS and OS/390 Version 7 Using the Utilities Suite, S*G24-6289. Rama holds a postgraduate degree in information technology from Monash University in Melbourne, Australia. His areas of expertise include data modeling, scientific numeric modeling, and project management.

**Fabricio Pimentel** is a Database Administrator for IBM in São Paulo, Brazil, supporting several very large installations in the US for the IBM DBA Service Center. He has worked with DB2 as a System Analyst and DBA as a customer before joining IBM. His areas of expertise include database administration and design. He holds a bachelor's degree in Computer Science from Pontificia Universidade Catolica de São Paulo.

**Suresh Sane** is a Database Architect with DST Systems in Kansas City, Missouri, USA. He co-authored the redbooks, *Squeezing the Most Out of Dynamic SQL,* SG24-6418, and *DB2 for z/OS Stored Procedures: Through the CALL and Beyond*, SG24-7083. He is actively involved with the International DB2 Users Group (IDUG) with numerous presentations and educational seminars in the US, Canada, and Australia as well as articles in the IDUG Solutions Journal. He has worked with DB2 since Version 1. He holds a bachelor's degree in Electrical Engineering from Indian Institute of Technology, Mumbai, India, and an MBA from Indian Institute of Management, Kolkutta, India.

A photo of the team is in Figure 1.

*Figure 1   Left to right: Suresh, John, Fabricio, Rama and Paolo (photo courtesy of Sangam Racherla)*

Thanks to the following people for their contributions to this project:

Rich Conway
Emma Jacobs
Bob Haimowitz
Leslie Parham
Sangam Racherla
International Technical Support Organization

Jeff Berger
John Campbell
Julie Chen
Steve Chen
James Guo
Claire McFeely
Roger Miller
Mary Petras
Akira Shibamiya
Jim Teng
Jay Yothers
IBM Silicon Valley Laboratory

Rick Butler
BMO Financial Group, Toronto, Canada

Randy Corum
DST Systems, Kansas City

Bart Steegmans
IBM Belgium

## Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

**ibm.com**/redbooks/residencies.html

## Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  **ibm.com**/redbooks

- Send your comments in an email to:

redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

# Introduction

The issues related to data integrity include a large variety of cases. Considering that your data is not usually in one place only, but exists as copies, and it is not always at rest, it could be useful to explore some of the following:

- Examples of controls in place in a production environment to protect the integrity of the data
  - Firewalls
  - Segregation of duties
  - Regular review of privileges held
  - A process to ensure a user's privileges are revoked when they change jobs or leave the company
  - High level recommendations on when to use encryption
  - A process to validate special requests for pulling data from production
  - An established Information Security policy
  - The use of ENABLE/DISABLE bind parameters to control the allowable system connection types
  - Use of GROUPS, such as RACF®, to distribute privileges (as opposed to granting directly to userids)
- The role of a data steward
- How to maintain data integrity when data is in transit between the production database and the business user's PC
- Controls to prevent corruption of data, either in a malicious manner or to benefit from fraud
- Controls which can be implemented to ensure that data is not shared within an organization
- Examples of controls in place in a development environment since production data is sometimes copied to development
  - Increased risk for this data in the development environment

- Compliance items
  - Recording for what purpose data was collected and ensuring that it is only used for that purpose.
  - The Hippocratic Database (HDB) Compliance Auditing application, called Eunomia (formerly known as PACT), enables companies to verify compliance with data disclosure laws, company policies, and customer preferences.
- Recovery flexibility and performance
- The role of an architect in establishing which source is the reference or book of record
- What to consider when using offshore resources to do development and production support
  - Possible increased risks
  - Differences in the legal system

In this chapter, we briefly review the functionalities offered by the host environment that can help in some of the issues above, and by DB2 in satisfying the more specific application data integrity requirements.

In the chapters that follow, we will go into more details of the functions that DB2 offers for application data integrity and recovery.

## 1.1 Host platform

The System z environment, the current evolution of S/390® and including zSeries, maintains its benchmark position as a flexible, efficient, and responsive platform for highly complex, integrated environments running a wide range of critical workloads.

System z and the new System z9™, z/OS, and DB2 have a unique partnership in the database industry, one which is key to IBM today and in the future. Nothing beats the availability of DB2 on System z hardware and software.

In this section, we discuss the System z hardware and some of the software features which DB2 uses to achieve synergy with the platform.

The foundation for the traditional strengths for data serving on the mainframe lies with the tight integration that System z and z/OS share with DB2 for z/OS. DB2 for z/OS is written to exploit the System z platform and as a result can offer advanced features and function. IBM DB2 for z/OS delivers rich function for highly scalable, industry-leading, high availability, IT infrastructure for your enterprise data and on demand business applications. The combined power and capacity of IBM System z with the high performance and availability of the z/OS operating system and the strength of the DB2 for z/OS data server can expand and extend your IT infrastructure and the business value of your data.

The combination of DB2 and System z provides a unique competitive advantage for on demand environments by providing a flexible, cost-effective and optimized foundation for Information on demand. This foundation allows you to better manage risk, supports your efforts to demonstrate compliance with policies and standards, and helps to simplify management of your information infrastructure. These capabilities are important to enable customers to use their core business data to drive insight and gain competitive advantage.

Synergy with the z/OS operating system and the System z hardware positions DB2 to *keep your data available*. This is different from other platforms and databases, which are focused on *recovering quickly*. The ability to recover quickly is an important attribute, however, your business goals are better achieved with a solution that focuses on data integrity and the elimination of outages, planned or unplanned. This design point difference characterizes your business applications and maintenance processes, and therefore your ability to meet business goals and service-level agreements.

Parallel Sysplex® and DB2 data sharing implementation offer DB2 applications and administrators the highest potential availability when implementing maintenance or applying a version upgrade allowing you to meet your customers' expectations.

DB2's online schema evolution allows structural changes to occur without drop and create, providing increased availability for your data and your business.

Key platform synergy items drive business resiliency and reduce cost of ownership to levels beyond any other offerings:

- Special engines for Linux®, Java™, and DB2
- Hardware assistance for compression, encryption, sorting, and Unicode conversion
- Exploitation of 64-bit memory addressability
- Exceptional workload management support, governing the priorities of processes running on z/OS, and balancing connections from distributed platforms
- Full utilization of the Parallel Sysplex architecture, enabling a scalable, clustered, and incremental growth architecture
- Support for new workloads, using specialized processors for Java and Linux workloads

- Integration with other core z/OS components, such as those for disk storage management, access control, UNIX® system services, and failover
- Tooling and instrumentation for diagnosis and manageability
- Integration and synergy with the disk storage architecture

**Note:** For more information, see *Disk storage access with DB2 for z/OS*, REDP-4187-00.

## 1.1.1 z/OS and OS/390 system integrity

The zSeries, S/390, z/OS, and OS/390® commitment to system integrity means that unauthorized users and programs cannot bypass the hardware isolation functions that protect other users or programs, cannot obtain control in an authorized execution status, and cannot bypass the system-level security functions provided by the SecureWay® Security Server for OS/390 and z/OS. IBM has had a formal commitment to system integrity since 1973. IBM continues this commitment with each new generation of zSeries, z/OS, S/390, and OS/390. The S/390 Division backs up this commitment through the proactive efforts of the System Integrity Competency Center. In addition, IBM will investigate, accept, process, and resolve, as a high-severity problem, any report (APAR) for a z/OS or OS/390 system integrity exposure found by our customers.

### zSeries and S/390 hardware isolation functions

The IBM zSeries and S/390 enterprise servers provide two kinds of hardware isolation functions that create a strong foundation for security. Basic isolation functions, such as storage protection keys, multiple address spaces, and program execution status provide mandatory separation of users and applications from each other and from the system. This guarantees that applications and users cannot affect other users or the system itself, except in appropriate, authorized ways. In addition, PR/SM™ provides an advanced isolation function in which you can run multiple copies of z/OS, z/VM®, OS/390, Linux, and other operating systems on the same processor, each fully isolated in its usage of the processor, storage, and I/O subsystem.

This allows you to implement concurrent production and test system images on the same hardware. It also allows implementation of multiple isolated production environments, such as you might want in a service bureau environment, or more importantly on the Internet, where you might want a strong guarantee that malicious users of the network cannot affect your main production system. For this case, you could have a production environment that communicates directly with the Internet, and one isolated from such direct contact, both running concurrently on the same hardware platform. IBM's PR/SM support has received an E4 security certification under the United Kingdom IT Security Evaluation and Certification Scheme, the first general purpose product to receive this high level of certification.

The basic isolation functions segregate users from each other and from the operating system running on the hardware. z/OS and OS/390 take full advantage of these functions to help ensure the integrity of the system and your data. The functions include:

#### *Storage protection keys*

The hardware provides 16 protection keys that the system can assign to running programs and to areas of storage with options that can require that a program's key match the storage area's key before the program can write into the storage, or optionally, before the program can even read from the storage. Using keys prevents user programs from tampering with storage owned by the system and can also lead to improved integrity by minimizing the errors that could occur if system code accidentally tried to write into the wrong area of user (or system) storage.

### *Authorized versus unauthorized execution states*

The hardware provides capabilities to allow a program to run in either an authorized or unauthorized execution state. Authorized programs run in either "supervisor" status or in a "system" storage protection key with a value between zero and seven. Unauthorized programs run in "problem program" status and with a storage protection key of 8 through 15. In addition, a "semi-privileged" status can allow problem-status programs, in controlled situations, to gain access to supervisor status functions or a system storage key. z/OS zSeries, S/390 Hardware Isolation Functions, and OS/390 also support a controlled mechanism (Authorized Program Facility (APF)) by which a problem-status program can switch to an authorized status or key.

An unauthorized program cannot issue privileged system instructions (such as instructions to change the status of the system or to initiate I/O), nor can it become authorized except through controlled system hardware or software interfaces. Thus, it cannot interfere with the operating system nor with other users' programs.

### *Multiple address spaces*

In addition to the isolation provided by storage protection keys and execution states, the hardware and software further isolate user (and system) programs into different "address spaces." Each address space has the ability to read common system storage but it can neither read nor write the non-shared storage belonging to another address space unless allowed to do so by an authorized program. Thus, unauthorized user programs cannot interfere with programs in another address space. Where programs in different address spaces must share data, the operating system provides mechanisms to allow such sharing in a controlled, safe way.

## 1.1.2 System-level security

System-level security provides the ability to identify and authenticate users of the system, control their access to system resources (databases, other data, transactions, programs, and so on), and audit their use of those resources.

The use of small computers and data processing have increased the need for data security. z/OS incorporates the SecureWay Security Server, which provides a platform that gives you solid security for your entire enterprise, including support for the latest technologies. A feature of z/OS, the SecureWay Security Server comes with these major components:

- Resource Access Control Facility (RACF) is the primary component of the SecureWay Security Server for z/OS. With RACF, the Security Server is able to incorporate additional components that aid in securing your system as you make your business data and applications accessible by your intranet, extranets, or the Internet.
- DCE Security Server provides user and server authentication for applications using the client server communications technology contained in the Distributed Computing Environment for OS/390. Beginning with OS/390 Security Server Version 2 Release 5, the DCE Security Server can interoperate with users and servers that make use of the Kerberos V5 technology.

  Integrated with RACF, OS/390 DCE support allows RACF-authenticated OS/390 users to access DCE-based resources and application servers without further authentications. In addition, DCE application servers can convert a DCE-authenticated user identity into an RACF identity and gain full RACF access control to z/OS resources.

- Communications Server for z/OS and OS/390 together with the Security Server provide basic firewall capabilities on the OS/390 platform to reduce or eliminate the need for non-OS/390 platform firewalls in many customer installations. The Communications Server provides the firewall functions of IP packet filtering, IP security (VPN or tunnels), and Network Address Translation (NAT).

  The Security Server provides the firewall functions of FTP proxy support, SOCKS daemon support, logging, configuration, and administration.
- LDAP Server provides secure access from applications and systems on the network to directory information held on OS/390 using the Lightweight Directory Access Protocol.

OS/390 includes the SecureWay Security Server for OS/390 to provide comprehensive, centralized, and integrated system-security functions for the system and the applications running on it. Unlike some platforms where the applications must perform a large part of the security processing, with z/OS or OS/390 and the SecureWay Security Server, applications often do not need to perform any separate security processing at all. Where they do need to provide some security processing, they can make use of the system's centralized security interfaces to minimize the application specific programming and to guarantee compatible security decisions and auditing with the rest of the system.

z/OS and OS/390 provide security functions for traditional batch and online applications (based in TSO/ E, CICS®, IMS, and so on) as well as applications running in the UNIX System Services environment. UNIX applications running in either single-user or multi-user (client server) mode have full access to the centralized security capabilities provided by the system.

z/OS and OS/390 can identify and authenticate users before they access the system, using a variety of mechanisms including:

- User ID and password, or user ID and PassTicket with applications such as IBM's Global Sign-On product
- OSF Distributed Computing Environment (DCE) credentials or Kerberos Version 5 credentials
- Industry-standard X.509 digital certificates with applications such as DominoGo Webserver

At your option, you can also allow access to selected system resources by unauthenticated (anonymous, or public) users. Note, however, that OS/390 eliminates some "security holes" contained in other UNIX systems, such as allowing assignment of an "authenticated" user ID for an *rlogin* session based on the host name or address of the connecting machine, without requiring a password or other means of truly authenticating the connecting user.

You may also assign surrogate or public IDs to authenticated users to change their access privileges when using programs such as Domino® Go Webserver, or you may grant access to resources based on the application program or server chosen by the user. These capabilities allow a high degree of control and System-Level Security for z/OS and OS/390 customization of your security environment, all with as much or as little auditing of system and resource access as you desire.

Then, with strong security at the system level, you have a good security foundation both for local work and for processing of network (Internet, intranet, extranet) work.

S/390 and OS/390 and their predecessors have provided inherent, robust security for decades, with security providing a key design point for hardware, operating system, subsystems, and applications. Security requirements have changed over time from the earlier days when system-level security would suffice, to today's environment, which requires comprehensive network- and transaction-level security. S/390 and OS/390 have evolved to support these newer requirements, and that evolution will continue to provide both enhanced functionality and the enhanced security you need to manage it.

The OS/390 platform is at the cutting edge of emerging technology. Products now available for OS/390 allow an enterprise to use the system in many ways. In addition to the traditional batch and transaction-based workloads that OS/390 supports with DB2, CICS, and IMS, recent additions, in the form of z/OS and OS/390 Firewall Technologies, WebSphere® Application Server, and WebSphere Commerce Suite, have made it easy for you to expand access to your business data to include the wider audience of the Internet, with appropriate security controls to ensure safety both for you and for your customers.

IBM will expand the capabilities of that LDAP server in the future to provide a generic, standard access point for all security and directory information on OS/390. This "secure directory" facility will allow you to have a central repository for security and directory information within your enterprise.

### 1.1.3 Transaction-level security

Transaction-level security allows two entities on the Internet to conduct a transaction privately and with authentication. It would allow a consumer to transact business with a merchant, for example, or a merchant to transact business with a supplier or bank. This level of security provides a basis to enable the payment for goods and services to occur with privacy and with assurance that each party knows the identity of the other parties that participate in the transaction.

Two leading technologies provide transaction-level security today:

- SSL (Secure Socket Layer), a de facto standard developed by Netscape Communications Corporation and generally used by Web browsers and servers, provides a private channel between the client and server that ensures privacy of the data, authentication of the session partners, and data integrity for the messages. SSL involves the use of both public- and private-key-based encryption.
- SET (Secure Electronic Transaction) is an open standard, multi-party protocol used for conducting secure bank card payments over the Internet. SET also provides authentication, data integrity, and data privacy, again through the use of encryption technology.

The hardware and software features of the zSeries and S/390 enterprise servers, together with the z/OS and OS/390 operating systems, provide an ideal platform for your applications that need transaction-level security, allowing you to conduct business safely over the Internet:

- Integrated cryptography provides the fundamental encryption and decryption operations required for SSL and SET requests and allows implementation of the Virtual Private Network (VPN) connections provided by z/OS and OS/390 Firewall Technologies for network-level security.
- The HTTP Server component of the IBM WebSphere Application Server for z/OS and OS/390 provides a Web server integrated with the operating system and supporting SSL version 3.0. Additionally, the HTTP Server makes full use of the functions of the SecureWay Security Server when running transactions on the system, allowing the transactions to run with a local authenticated identity based on an X.509 digital certificate provided by the client user and allowing access to local data based either on the client identity or a surrogate identity assigned by your administrators.
- IBM CommercePOINT Payment provides the first suite of end-to-end solutions to enable quick, easy, and highly secure credit card commerce on the Internet. This suite integrates the SET protocol 1.0 into appropriate stages of the commerce life cycle using digital

certificate technology. CommercePOINT Payment supplies a Wallet for client payment information, an eTill allowing merchants to accept various payment schemes, a Gateway providing a link to financial institutions, and a Registry for SET to issue and manage the digital certifications that underlie all SET transactions.

## 1.1.4 zSeries cryptography

The best way to secure information over the Internet is to encrypt it. IBM System z provides exceptional performance and function via cryptography coprocessors and accelerators that are individually specialized to address various encryption needs. The z/OS operating system provides the infrastructure to exploit the strengths of each cryptographic feature. The performance advantages of hardware-assisted cryptography are readily available to applications, such as banking and finance, via the cryptography interfaces of z/OS.

### *Cryptographic features*

New features are available on System z (z9 EC, z9 BC, z990, z890).

A third generation cryptographic feature, the Crypto Express2, combines the functions of the PCICA and the PCIXCC in a single feature that is expected to provide improved secure key and system throughput. The Crypto Express2 feature supports a mixture of both secure and clear key applications. Crypto Express2 also offers CVV generation and verification services for 19-digit PANs providing advanced anti-fraud security. In addition, it supports applications that require clear key RSA operations using fewer than 512-bits. This capability is designed to enable easier migration of some additional cryptographic applications to System z servers without requiring you to rewrite the applications.

The CP Assist for Cryptographic Function (CPACF) is incorporated into every central processor that ships with the IBM System server families. The CPACF feature delivers cryptographic support on every Central Processor (CP) with Data Encryption Standard (DES) and Triple DES (TDES) data encryption/decryption along with SHA-1 hashing. The CPACF integrated in every central processor of System z9 EC and z9 BC enhances cryptography by providing support for the Advanced Encryption Standard (AES) and SHA-256 hashing algorithm. Because these cryptographic functions are implemented in each central processor (CP), the potential throughput scales with the number of processor units (PUs) ordered with each system.

The PCIX Cryptographic Coprocessor (PCIXCC) is a replacement for the PCICC and the CMOS Cryptographic Coprocessor Facility that were originally available for zSeries processors. PCIXCC provides support for all of the security-related cryptographic functions available with its predecessor cryptograpic coprocessor features. In addition, PCIXCC also supports use of encrypted key values and user-defined extensions (UDX).

Optional cryptographic hardware features for zSeries servers include the zSeries PCI Cryptographic Coprocessor (PCICC) feature which has a tamper-proof design and supports symmetrical encryption, as well as Public Key encryption. The PCI Cryptographic Coprocessor is scalable and programmable. PCICC is used throughout the financial sector.

The zSeries PCI Cryptographic Accelerator (PCICA) feature was designed to perform the computationally intensive public key cryptographic operations in hardware. Its aim was to provide cryptographic support for e-business. The PCI Cryptographic Accelerator Feature is available on z990 and supported on z900 servers. It may be carried forward on upgrades from z900 to z990 servers.

Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols are public key cryptography-based extensions to TCP/IP networking. SSL /TLS helps to ensure private communications between parties on the Internet with the intent of allowing information such as the credit card number to be passed from customer to marketing application without the threat of interception.

System z servers provide the performance and scale you need to handle security-rich Web transactions. System z has focused on improving SSL/TLS encryption performance, and it shows. For example, z990 servers offer speed, with capabilities of greater than 11,000 SSL handshakes/second with z/OS 1.4 measured on a z990 with 16 CPs and 6 PCICA features. (To put that into some perspective, as recently as 1998, zSeries SSL performance was approximately 13 SSLs/second.) This ultra-fast and security-rich SSL comes courtesy of special hardware in the optional Crypto Express2 feature (when one or both of the two PCI-X adapters are configured as an accelerator) and the PCI Cryptographic Accelerator (PCICA) features.

### 1.1.5 System z integrity features

Most z/OS customers have business requirements for continuous system availability. System downtime or unplanned outages, even of short duration, can cost millions of dollars in lost revenue or other significant negative business impact. Thus, z/OS customers do not think about availability in terms of minimizing the time of an outage; they think in terms of minimizing outages. From the beginning, for z/OS as well as its predecessors, a basic assumption has been that hundreds or even thousands of users would depend on it. In fact, the anticipated mean time between failures of IBM z900 systems approaches 30 years.

z/OS has a long history, so do its users' suites of applications. z/OS customers have millions of dollars invested in business-critical applications that have been operating and evolving on z/OS and its predecessors for 20 years or more. Ensuring that this customer investment is preserved requires each z/OS release to maintain compatibility with prior releases, regardless of the complexities involved. It presents challenges when architectural changes are implemented, but application compatibility is expected for every z/OS release.

An operating system must maintain data integrity. A system that is up but quietly corrupting data is worse than one that is down. z/OS address spaces separate applications from each other to minimize the risk of one program corrupting another program's private storage or data area. Storage-protect keys prevent user programs from altering system storage. Extensive system locking and serialization techniques coordinate system events and actions. Data integrity is a core attribute of z/OS. And the *z* in zSeries and z9 System stands for *zero down time*.

The IBM System z9-109 servers, see Figure 1-1, have availability designs that have been carried forward from zSeries machines and several new ones. They are described in *IBM System z9 109 Configuration Setup*, SG24-7203, and *IBM System z9 109 Technical Guide*, SG24-7124.

*Figure 1-1 IBM System z9-109*

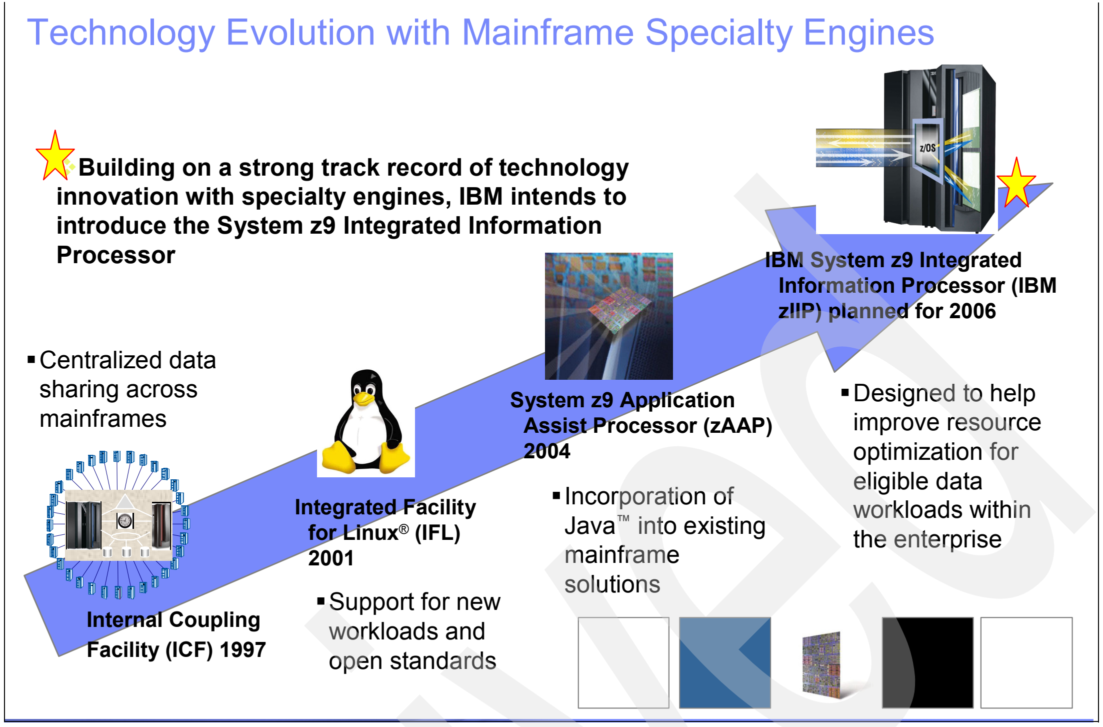
The z/OS mainframe operating system and its predecessors have been part of commercial computing for decades. z/OS includes characteristics on which FORTUNE 500 companies rely, such as highly usable capacity, strong concurrency, consistent application compatibility, pervasive real-time recovery, and robust data integrity.

### 1.1.6 The zIIP

The zSeries platform maintains its place as the main choice for mission critical systems. Recent announcements by IBM ensure that the System z9 platform maintains this key role, but it can also accept new workloads at the right cost. The new System z9 Integrated Information Processor (zIIP) is designed to run specific database workloads. Following on the success of the Internal Coupling Facility engine for Sysplex environments, the Integrated Facility for Linux (IFL), and the System z9 Application Assist Processor (zAAP) for Java workloads, the zIIP is designed to help improve resource optimization and lower the cost of eligible workloads, enhancing the role of the mainframe as the data hub of the enterprise.

Customers using zIIP processors can free up capacity in their System z9 general computing resources by moving some database-oriented workloads onto the dedicated zIIP engines instead of running on the main System z9 processors. See Figure 1-2.

DB2 for z/OS Version 8 is the first IBM software fully capable of exploiting zIIP for workloads such as those of distributed environments, star-schema executions, and functions of DB2 utilities related to index management. In the future, IBM plans to offer additional zIIP exploitation by other IBM subsystems. In addition, key new DB2 9 for z/OS functions have been announced that will support data serving on System z, including enabling high volume transaction processing for the next wave of Web applications, strengthening the security features within the product, and improving performance in the data warehousing arena, and new DB2 functions that will support DB2 scan acceleration via the DS8000.



Technology Evolution with Mainframe Specialty Engines

Building on a strong track record of technology innovation with specialty engines, IBM intends to introduce the System z9 Integrated Information Processor

- Centralized data sharing across mainframes

Internal Coupling Facility (ICF) 1997

Integrated Facility for Linux® (IFL) 2001

- Support for new workloads and open standards

System z9 Application Assist Processor (zAAP) 2004

- Incorporation of Java™ into existing mainframe solutions

IBM System z9 Integrated Information Processor (IBM zIIP) planned for 2006

- Designed to help improve resource optimization for eligible data workloads within the enterprise

*Figure 1-2 Mainframe specialty engines*

## 1.1.7 VSAM share options

When allocating a VSAM file, you have the ability to use any of four options for cross region integrity and two options for cross system integrity. IDCAMS allocations allow you to define these options by using the SHAREOPTIONS keyword.

DB2 defined data sets are created automatically with SHAREOPTIONS of (3,3). What these options mean is:

- The data set can be fully shared by any number of users.
- The user is responsible for maintaining both read and write integrity for the data the program accesses.

User defined data sets are generally created with the same attributes.

Although this may sound like users are on their own and have no protection, we do have some fail-safe mechanisms:

- IRLM controls locking issues inside of DB2. In data sharing, further protection is provided through the locking portion of the coupling facility. DB2 assures that no double updates are occurring.
- RACF should be set up to protect DB2 data sets, since it is possible to load data into DB2 objects outside of DB2, thereby bypassing the integrity provided by IRLM. You can accomplish this in a variety of ways, including using the offline utility DSN1COPY and user programs outside of DB2. You must take special care when loading or modifying data outside of the control of DB2. RACF should be employed to protect DB2 data sets from being updated by unauthorized programs and individuals.

## 1.1.8 Data sharing integrity

Data sharing delivers availability, workload balancing, and flexible growth benefits, and, through use of the coupling facility (CF), can also avoid the high overheads of frequent disk I/O and intersystem message passing. Both interquery and intraquery parallelism also use this architecture.

DB2 has been extended from its single-system initial structure to implement data sharing using the CF for global locking and intersystem buffer coherency. Use of the CF is the key factor, allowing multisystem data sharing with good performance characteristics. In addition, several optimizations have further reduced the overhead for data sharing, global locking, and buffer coherency. Most recent are the CF request batching and Locking protocol level 2 introduced by DB2 V8 (see Chapter 8 of the *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465).

DB2's implementation of retained locks, recovery logging, and CF failure recovery ensures that data integrity is maintained across the failure of any hardware or software element in the sysplex. DB2's robust design for data sharing builds on the strengths of the S/390 Parallel Sysplex to provide DB2 users with unprecedented levels of capacity, availability, and parallelism.

## 1.1.9 Global resource serialization

In a multitasking, multiprocessing environment, resource serialization is the technique used to coordinate access to resources that are used by more than one program. When multiple users share data, a way to control access to that data is necessary. Users, who update data, for example, need exclusive access to that data. If several users try to update the same data at the same time, the result can be data that is incorrect or corrupted. In contrast, users, who only read data, can safely access the same data at the same time.

Global resource serialization (GRS) offers the control needed to ensure the integrity of resources in a multisystem environment. Combining the systems that access shared resources into a global resource serialization complex enables you to serialize resources across multiple systems. In a global resource serialization complex, programs can serialize access to data sets on shared disk volumes at the data set level rather than at the disk volume level. A program on one system can access one data set on a shared volume while other programs on any system can access other data sets on the volume. Because GRS enables jobs to serialize resources at the data set level, it can reduce contention for these resources and minimize the chance of an interlock occurring between systems.

You can request access to a resource as exclusive or shared. When GRS grants shared access to a resource, no exclusive users are granted access to the resource simultaneously. Likewise, when GRS grants exclusive access to a resource, all other requestors for the resource wait until the exclusive requestor frees the resource.

## 1.1.10 DB2 controls

DB2 has strong and granular access control. It controls access to its objects by a set of privileges. Default access is none. Until access is granted, nothing can be accessed. This is called *discretionary access control* (DAC).

DB2 has extensive auditing features. For example, you can answer questions such as “Who is privileged to access what objects?” and “Who has actually accessed the data?”.

The catalog tables describe the DB2 objects, such as tables, views, table spaces, packages, and plans. Other catalog tables hold records of every granted privilege or authority. Every catalog record of a grant contains information such as name of the object, type of privilege, IDs that receive the privilege, ID that grants the privilege, and time of the grant.

The audit trace records changes in authorization IDs, changes to the structure of data, changes to values (updates, deletes, and inserts), access attempts by unauthorized IDs, results of GRANT and REVOKE statements, and other activities which are of interest to auditors.

You can use the z/OS Security Server, previously named Resource Access Control Facility (RACF), to:

- Control access to the DB2 environment
- Facilitate granting and revoking to groups of users
- Ease the implementation of multilevel security in DB2
- Fully control all access to data objects in DB2

DB2 defines sets of related privileges, called *administrative authorities*. You can effectively grant many privileges by granting one administrative authority.

Security-related events and auditing records from RACF and DB2 can be loaded into DB2 databases for analysis. The DB2 Instrumentation Facility Component can also provide accounting and performance-related data. This kind of data can be readily loaded into a standard set of DB2 tables (definitions are provided). Security and auditing specialists can easily query this data to review all security events.

### DB2 and multilevel security

A multilevel security (MLS) system is a security environment. It allows the protection of data based on both traditional discretionary access controls, and controls that check the sensitivity of the data itself through mandatory access controls.

These mandatory access controls are at the heart of an MLS environment. They prevent unauthorized users from accessing information at a classification to which they are not authorized. They also prevent users from changing the classification of information to which they do have access. These mandatory access controls provide a way to segregate users and their data from other users and their data regardless of the discretionary access they are given though access lists.

To create an MLS environment, you must have a combination of software and hardware components that enforce the security requirements needed for such a system. The security relevant portion of software and hardware components that make up this system is also known as the *Trusted Computing Base*.

### Why multilevel security

The primary arena where MLS is valuable is governmental agencies that need a security environment that keeps information classified and compartmentalized between users. In addition to the fundamental identification and authentication of users, auditing and accountability of the actions by authenticated users on these systems is provided by the security environment.

In such highly secure environments, to manage the compartmentalization of information between users, each compartment is on its own system. This makes it difficult for classified information to spill from one system to another, since the connections between systems can be highly controlled. With MLS, these systems can be consolidated onto a single system, with each compartment independent of the other, so that no transfer of data can occur between compartments within that system. This takes advantage of the cost savings of not having to manage multiple systems, but only a few, or one system.

Commercial clients may also find some features of MLS useful, such as to separate sensitive customer information from the general populace or from other users. New governmental regulations, such as HIPAA (see the following list), or corporate mergers are examples where security of information based on the information itself is important in the commercial world.

MLS is implemented at the operating system level. DB2 Version 8 participates in this scheme and provides MLS security to the row level.

This additional mandatory access control feature helps your business address the most common security issues. It does this along with the rock solid z/OS partition separation, certification for common criteria, improved DB2 encryption features, and the new DB2 and IMS encryption tool. Together these features also help your business comply with existing and new regulations such as:

- Health Insurance Portability and Accountability Act of 1996 (HIPAA) in the U.S.A. for health care
- Gramm-Leach-Bliley Act of 1999 (GLBA) in the U.S.A. for financial services
- Sarbanes-Oxley, an act which aims to protect investors by improving the accuracy and reliability of corporate disclosures
- Personal Information and Electronic Documents (PIPEDA) Act in Canada
- United Kingdom Data Protection Act (Oct 1998)
- European Union Data Protection Directive 95/46/EC

## Multilevel security concepts

*Mandatory access control* (MAC) imposes additional restrictions upon users. Now users access data based on a comparison of the classification of the user and the classification of the data as well as the standard *discretionary access control* (DAC) checking. This additional security check verifies that users can access only data and resources that their classification allows them to. This happens regardless of whether they have discretionary access to such data or resources. Mandatory means that subjects cannot control or bypass the access.

An MLS system is a security environment that allows the protection of data. It is based on both traditional discretionary access controls and controls that check the sensitivity of data itself through mandatory access controls. Using MLS allows you to classify objects and users with security labels that are based on hierarchical security levels and non-hierarchical security categories.

You can implement MAC via RACF at the operating system level. The key advantage is that MAC security can be integrated across the platform with the same security for files, print, and DB2.

RACF has several options that you can turn on and off to manipulate different aspects of an MLS environment. It is possible to have some features of multilevel security on at one time (creating a partial MLS environment). Because of this, commercial customers may find this type of environment useful to meet these needs as opposed to running a complete multilevel security environment.

Using MLS, you can define security for DB2 objects by assigning security labels to your DB2 objects. You can define a hierarchy between those objects. MLS then restricts access to the object based on the security level of that object.

## Row level security as a subset of multilevel security

In today’s complex world, organizations may have considerable needs to restrict access to data in their database applications. Privacy and data protection legislation, antitrust legislation, and considerations of national security are a few of the reasons why organizations need to ensure that people in one part of the business do not know and cannot determine what is happening in another part.

Companies may lose business if they cannot demonstrate to security-conscious potential customers that data relating to them is strictly protected from unauthorized access. Regulatory authorities may require that computer systems be separately maintained if it cannot be shown that the different divisions of a company are prevented from accessing details of the other divisions' operations.

In an increasingly interconnected era, organizations may want to offer limited access to their operational systems to the clients, suppliers, and trading partners. They may not want to give those people freedom to roam through data pertaining to their competitors.

You can summarize the requirement as follows, using a customer order database as an example. It must include:

- A means of marking a customer as being in a set of protected customers
- A means of propagating such markings to related data after the customer marking is made
- A means of marking new data for such customers and transient data relating to such customers with the security markings of the customer
- The restriction of access to data based on such markings
- A means of ensuring that the security markings of the data are not changed other than by authorized users and processes

You can give a group of people access to a table and limit each user to only a subset of the data based on the particular individual's MLS definition. And you can do this without creating views or placing extra predicates in the SQL. Instead, you place a security label on the data row and then associate the security label with each of the users. The Data Manager layer knows how to compare these security labels and can see whether a particular user is allowed to access a row. It is flexible in terms of having one set of tables and many different ways of accessing the data or providing a subset of the data.

z/OS is designed to meet the stringent security requirements of multi-agency access to data. This solution leverages zSeries leadership in scale, high availability, and self managing capabilities for highly secure single-database hosting.

### 1.1.11 Auditing your DB2 applications

DB2 provides many controls that you can apply to data entry and update. Some of the controls are automatic; some are optional. All of the controls prohibit certain operations and provide error or warning messages if those operations are attempted.

Here we describe typical concerns for ensuring data accuracy and consistency. This section is not exhaustive, other combinations of techniques are possible. For example, you can use table check constraints or a view with the check option to ensure that data values are members of a certain set. Or you can set up a master table and define referential constraints. You can also enforce the controls through application programs and restrict the INSERT and UPDATE privileges only to those programs.

To ensure that the required data is present, define columns with the NOT NULL clause. You can also control the type of data by assigning column data types and lengths. For example, alphabetic data cannot be entered into a column with one of the numeric data types. Data that is inserted into a DATE column or a TIME column must have an acceptable format, and so on.

For suggestions about assigning column data types and the NOT NULL attribute, see *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03.

In some cases, you must ensure that the data in a column or a set of columns is unique. The preferred method for ensuring that data values are unique is to create a unique index on a column or set of columns. See *The Official Introduction to DB2 UDB for z/OS* by Susan Graziano Sloan, IBM Press, Prentice Hall, for suggestions about indexes.

Triggers and table check constraints enhance the ability to control data integrity. Triggers are very powerful for defining and enforcing rules that involve different states of DB2 data. For example, a rule can prevent a salary column from more than a ten percent increase. A trigger can enforce this rule and provide the value of the salary before and after the increase for comparison. See Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03, for information about using the CREATE TRIGGER statement to create a trigger. A check constraint designates the values that specific columns of a base table can contain. A check constraint can express simple constraints, such as a required pattern or a specific range, and rules that refer to other columns of the same table.

You might need to check that the table definition expresses required constraints on column values as table check constraints. For a full description of the rules for those constraints, see the CREATE TABLE information in Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03. An alternative technique is to create a view with the check option, and then insert or update values only through that view. For example, suppose that, in table T, data in column C1 must be a number between 10 and 20. Suppose also that data in column C2 is an alphanumeric code that must begin with A or B. Create view V1 with the following statement:

```
CREATE VIEW V1 AS SELECT * FROM T WHERE C1 BETWEEN 10 AND 20 AND (C2 LIKE'A%' OR C2 LIKE
'B%') WITH CHECK OPTION;
```

Because of the CHECK OPTION, view V1 allows only data that satisfies the WHERE clause. You cannot use the LOAD utility with a view, but that restriction does not apply to user-written exit routines.

Several types of user-written routines are pertinent here:

- Validation routines

  You can use validation routines to validate data values. Validation routines access an entire row of data, check the current plan name, and return a nonzero code to DB2 to indicate an invalid row.

- Edit routines

  Edit routines have the same access as validation routines and can also change the row that is to be inserted. Auditors typically use edit routines to encrypt data and to substitute codes for lengthy fields. However, edit routines can also validate data and return nonzero codes.

- Field procedures

  Field procedures access data that is intended for a single column; they apply only to short-string columns. However, they accept input parameters, so generalized procedures are possible. A column that is defined with a field procedure can be compared only to another column that uses the same procedure.

## Referential integrity ensures that data is consistent across tables

When you define primary and foreign keys, DB2 automatically enforces referential integrity. Therefore, every value of a foreign key in a dependent table must be a value of a primary key in the appropriate parent table. Use referential integrity to ensure that a column allows only specific values. Set up a master table of allowable values and define its primary key. Define foreign keys in other tables that must have matching values in their columns. In most cases, you should use the SET NULL delete rule. DB2 does not enforce informational referential constraints across subsystems. For information about the means, implications, and limitations of enforcing referential integrity, see *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-03.

Triggers offer an efficient means of maintaining an audit trail. You can define a trigger to activate in response to certain DELETE, INSERT, or UPDATE statements that change data. You can qualify a trigger by providing a list of column names when you define the trigger. The qualified trigger is activated only when one of the named columns is changed. A trigger that performs validation for changes that are made in an UPDATE operation must access column values both before and after the update. Transition variables (available only to row triggers) contain the column values of the row change that activated the trigger. The old column values and the column values from after the triggering operation are both available. See *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03, for information about when to use triggers.

Locks can ensure that data remains consistent, even when multiple users try to access the same data at the same time. From an auditing standpoint, you can use locks to ensure that only one user is privileged to change data at a given time. You can also ensure that no users are privileged to access uncommitted data. If you use repeatable read (RR), read stability (RS), or cursor stability (CS) as your isolation level, DB2 automatically controls access to data by using locks. However, if you use uncommitted read (UR) as your isolation level, users can access uncommitted data and introduce inconsistent data. Auditors must know which applications use UR isolation, and they must know whether these applications can introduce inconsistent data or create security risks. For static SQL, you can determine which plans and packages use UR isolation by querying the catalog.

For static SQL statements, use the following query to determine which plans use UR isolation:

```
SELECT DISTINCT Y.PLNAME FROM SYSIBM.SYSPLAN X, SYSIBM.SYSSTMT Y
WHERE (X.NAME = Y.PLNAME AND X.ISOLATION = ’U’) OR Y.ISOLATION = ’U’ ORDER BY Y.PLNAME;
```

For static SQL statements, use the following query to determine which packages use UR isolation:

```
SELECT DISTINCT Y.COLLID, Y.NAME, Y.VERSION FROM SYSIBM.SYSPACKAGE X, SYSIBM.SYSPACKSTMT Y
WHERE (X.LOCATION = Y.LOCATION AND X.LOCATION = ’ ’ AND X.COLLID = Y.COLLID AND X.NAME =
Y.NAME AND X.VERSION = Y.VERSION AND X.ISOLATION = ’U’) OR Y.ISOLATION = ’U’
ORDER BY Y.COLLID, Y.NAME, Y.VERSION;
```

For dynamic SQL statements, turn on performance trace class 3 to determine which plans and packages use UR isolation.

### Consistency between systems

When an application program writes data to both DB2 and IMS, or to both DB2 and CICS, the subsystems prevent concurrent use of data until the program declares a point of consistency.

Database balancing is a technique that can alert you to lost and incomplete transactions. Database balancing determines, for each set of data, whether the opening balance, the control totals, and the processed transactions equal the closing balance and control totals. DB2 has no automatic mechanism to calculate control totals and column balances and compare them with transaction counts and field totals. Therefore, to use database balancing, you must design these mechanisms into the application program.

Use your application program to maintain a control table. The control table contains information to balance the control totals and field balances for update transactions against a user view. The control table might contain these columns:

- View name
- Authorization ID
- Number of logical rows in the view (not the same as the number of physical rows in the table)
- Number of insert transactions and update transactions
- Opening balances
- Totals of insert transaction amounts and update transaction amounts
- Relevant audit trail information, such as date, time, workstation ID, and job name

The program updates the transaction counts and amounts in the control table each time it completes an insert or update to the view. To maintain coordination during recovery, the program commits the work only after it updates the control table. After the application processes all transactions, the application writes a report that verifies the control total and balancing information.

When you control data entry, you perform only part of a complete security and auditing policy.

You must also verify the results when data is accessed and changed. The following are methods for determining whether your data is consistent:

- Automatically checking the consistency of data

  Whenever an operation changes the contents of a data page or an index page, DB2 verifies that the modifications do not produce inconsistent data. Additionally, you can run the DSN1CHKR utility to verify the integrity of the DB2 catalog and the directory table spaces. You can also run this utility to scan the specified table space for broken links, damaged hash chains, or orphan entries. For more information, see Part 3 of *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-03.

- Submitting SQL queries to check data consistency

  If you suspect that a table contains inconsistent data, you can submit an SQL query to search for a specific type of error. For example, a view allows an insert or update to table T1 only if the value in column C1 is between 10 and 20 and if the value in C2 begins with A or B. To check that the control has not been bypassed, issue the following statement:

  ```
  SELECT * FROM T1 WHERE NOT (C1 BETWEEN 10 AND 20 AND (C2 LIKE 'A%' OR C2 LIKE
  'B%'));
  ```

  If the control has not been bypassed, DB2 returns no rows and thereby confirms that the contents of the view are valid. You can also use SQL statements to get information from the DB2 catalog about referential constraints that exist. For several examples, see *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03.

- Checking data consistency with the CHECK utility

  The CHECK utility helps to ensure data consistency in the following ways:

  – CHECK INDEX

    The CHECK INDEX utility checks the consistency of indexes with the data to which the indexes point. It determines whether each index pointer points to a data row with the same value as the index key. If an index key points to a LOB, the CHECK INDEX utility determines whether the index key points to the correct LOB.

  - CHECK DATA

    The CHECK DATA utility checks referential constraints (but not informational referential constraints). It determines whether each foreign key value in each row is a value of the primary key in the appropriate parent table. The CHECK DATA utility also checks table check constraints and checks the consistency between a base table space and any associated LOB table spaces. It determines whether each value in a row is within the range that was specified for that column when the table was created.

  - CHECK LOB

    The CHECK LOB utility checks the consistency of a LOB table space. It determines whether any LOBs in the LOB table space are invalid.

  See *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-03, for more information about the CHECK utility.

- Checking data consistency with the DISPLAY DATABASE command

  You can check data consistency with the DISPLAY DATABASE command. If you allow a table to be loaded without enforcing referential constraints on its foreign key columns, the table might contain data that violates the constraints. DB2 places the table space that contains the table in the CHECK-pending status. You can determine which table spaces are in CHECK-pending status by using the command:

  ```
  DISPLAY DATABASE ..... RESTRICT
  ```

  You can also use the DISPLAY DATABASE command to display table spaces with invalid LOBs. See Chapter 2 of *DB2 UDB for z/OS Version 8 Command Reference,* SC18-7416-03, for information about using this command.

- Checking data consistency with the REPORT utility

  You can use the REPORT utility to determine:

  - Which table spaces contain a set of tables that are interconnected by referential constraints
  - Which LOB table spaces are associated with which base tables

  See *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-03, for information about using the REPORT utility.

- Checking data consistency with the operation log

  You can use the operation log to verify that DB2 is operated reliably and to reveal unauthorized operations and overrides. The operation log consists of an automated log of DB2 operator commands (such as those that start or stop the subsystem or its databases) and any abend of DB2. The operation log records the following information:

  - Command or condition type
  - Date and time when the command was issued
  - Authorization ID that issued the command

  You can obtain database connection code information from the system log (SYSLOG), the SMF data set, or the automated job scheduling system. To obtain the information, use SMF reporting, job-scheduler reporting, or a user-developed program. You should review the log report daily and keep a history file for comparison. Because abnormal DB2 termination can indicate integrity problems, you should implement an immediate notification procedure to alert the appropriate personnel (DBA, systems supervisor, and so on) of abnormal DB2 terminations.

- Using internal integrity reports to check data consistency

  You can generate internal integrity reports for application programs and for utilities. For application programs, you should record any DB2 return codes that indicate possible data integrity problems, such as inconsistency between index and table information, physical errors on database disk, and so on. All programs must check the SQLCODE or the SQLSTATE for the return code that is issued after an SQL statement is run. DB2 records, on SMF, the occurrence (but not the cause) of physical disk errors and application program abends. The program can retrieve and report this information; the system log (SYSLOG) and the DB2 job output also have this information. However, in some cases, only the program can provide enough detail to identify the exact nature of problem.

You can incorporate these integrity reports into application programs, or you can use them separately as part of an interface. The integrity report records the incident in a history file and writes a message to the operator's console, a database administrator's TSO terminal, or a dedicated printer for certain codes. The recorded information includes:

- Date
- Time
- Authorization ID
- Terminal ID or job name
- Application
- Affected view or affected table
- Error code
- Error description

When a DB2 utility reorganizes or reconstructs data in the database, it produces statistics to verify record counts and to report errors. The LOAD and REORG utilities produce data record counts and index counts to verify that no records were lost. In addition to that, keep a history log of any DB2 utility that updates data, particularly REPAIR. Regularly produce and review these reports, which you can obtain through SMF customized reporting or a user-developed program.

### 1.1.12 Other security enhancements

While DB2 for z/OS V8 provides many enhancements for security, there is more to come. Roles will be used in the recently announced DB2 9 for z/OS to provide a more flexible technique than groups or users in assigning and controlling authorization, while improving consistency with the industry.

- A network trusted context provides a technique to work with other environments more easily, improving flexibility.
- The *instead of trigger* is an SQL technique that allows a trigger to be used in place of a view, with the possibility of updates, consistent with DB2 for LUW.
- Improved audit selectivity is necessary to verify that security is functioning.
- Secure Socket Layer (SSL) implementation provides encryption of data on the wire.
- DB2 allows encryption for the key disk resources used by DB2 (tables, LOBs, indexes, image copies, logs and archive logs).

Keep in mind that DB2 and z/OS strive to provide Regulatory Compliance and to get Common Criteria security certification. z/OS V1.7 has been Security Certified at EAL4+ for CAPP and LSPP. One year after receiving Common Criteria Security Certification of EAL3+, z/OS V1.7 with the RACF optional feature has achieved EAL4+ for Controlled Access Protection Profile (CAPP) and Labeled Security Protection Profile (LSPP) in March 2006. This certification assures customers that z/OS V1.7 has gone through a long and rigorous testing process and conforms to standards sanctioned by the International Standards Organization. Achieving EAL4+ certification will further enable z/OS to be adopted by governments and governmental agencies for mission-critical and command-and-control operations.

## Roles

The database ROLE is a “virtual authid” assigned via TRUSTED CONTEXT. It provides additional privileges only when in a trusted environment using existing primary AUTHID. It can optionally be the OWNER of DB2 objects. The definition is:

```
CREATE ROLE PROD_DBA;
GRANT DBADM … TO PROD_DBA;
CREATE TRUSTED CONTEXT DBA1 …
DEFAULT ROLE PROD_DBA OWNER(ROLE);
```

DB2 privileges are assigned to the defined role. The role exists as an object independent of its creator, so creation of the role does not produce a dependency on its creator.

This capability can allow a DBA to have privileges to create objects and manage them for a time, even though ownership belongs to another id.

The role can be assigned and removed from individuals via the trusted authorization context as necessary. This allows a DBA to perform object maintenance during a change control window on a Saturday night, for example. But when Monday arrives, the same DBA does not have the authority to do this same work.

Audit trails of the work completed during the maintenance window are available for verification by a security administrator or auditor.

Examples of database roles are:

- Dynamic SQL access to DB2 tables using JDBC™ or CLI, but only when running on a specific server.
- DBA can be temporarily assigned a DBA ROLE for weekend production table admin work with no table access at other times.
- DBA uses a ROLE for CREATE statements, so that the ROLE owns the objects that the DBA creates.
- Project librarian is assigned a BIND ROLE only when running on the production code library server and cannot BIND from any other server.

## Trusted security context

Today, you have the option to set a system parameter, which indicates to DB2 that all connections are to be trusted. It is unlikely that all connection types, such as DRDA®, RRS, TSO, and batch, from all sources fit into this category. It is likely that only a subset of connection requests for any type and source may be trusted or that you want to restrict trusted connections to a specific server. More granular flexibility allows for the definition of trusted connection objects. For instance, a user connecting from home may have fewer privileges than when connected via a trusted security context, such as a campus.

Once defined, connections from specific users via defined attachments and source servers allow trusted connections to DB2. The users defined in this context can also be defined to obtain a database role.

Trusted security context:

- Identifies “trusted” DDF, RRS Attach, or DSN application servers
- Allows selected DB2 authids on connections without passwords
- Reduces complexity of password management
- Reduces the need for an all-inclusive “system authid” in application servers
- Provides more visibility and auditability regarding which user is currently running
- Enables mixed security capabilities from a single application server

A WebSphere example of Trusted Security Context/ROLE:

- WebSphere connection pool can be created with one DB2 AUTHID
- WebSphere can reuse pooled connections to DB2 with different AUTHIDs
- DB2 AUTHIDs can be given privileges that are only available when executing in WebSphere:
  - Dynamic SQL access for JDBC only when using WebSphere

## 1.1.13 DB2 column level encryption

DB2 Version 8 ships a number of built-in functions which allow you to encrypt data at the *cell* level. These functions are ENCRYPT_TDES (or ENCRYPT), DECRYPT_BIN, DECRYPT_CHAR, and GETHINT.

### Create and Insert

The SET ENCRYPTION PASSWORD statement allows you to specify a password as a key to encryption. In Example 1-1, the EMPNO in EMPL is encrypted with a password.

*Example 1-1 DB2 data encryption*

```
CREATE TABLE EMPL
    (EMPNO       VARCHAR(64) FOR BIT DATA,
     EMPNAME     CHAR(20),
     CITY        CHAR(20) NOT NULL DEFAULT 'KANSAS CITY',
     SALARY      DECIMAL(9,2))
   IN DSNDB04.RAMATEST ;
COMMIT ;

SET ENCRYPTION PASSWORD = 'PEEKAY' WITH HINT 'ROTTIE' ;

INSERT INTO EMPL(EMPNO,EMPNAME, SALARY)
VALUES (ENCRYPT('12346'),'PAOLO BRUNI',20000.00)  ;

INSERT INTO EMPL(EMPNO,EMPNAME, SALARY)
VALUES (ENCRYPT('12347'),'RAMA NAIDOO',20000.00)  ;
```

When creating a column for data encryption, you must define it as VARCHAR. The length of the VARCHAR depends on the password and the password hint. Assuming EMPNO is VARCHAR(6) before encryption, you can compute the final length of VARCHAR as follows:

```
Maximum length of non-encrypted data 6 bytes
Number of bytes to the next multiple of 8 2 bytes
24 bytes for encryption key 24 bytes
Encrypted data column length 32 bytes
```

Therefore, define the column for encrypted data as VARCHAR(32) FOR BIT DATA. If you use a password hint, DB2 requires an additional 32 bytes to store the hint. You must define the EMPNO column as VARCHAR(64) FOR BIT DATA as in Example 1-1.

You are responsible for managing all these keys. Make sure you have a mechanism in place to manage the passwords that are used to encrypt the data. Use the password hint to “remember” the password. The GETHINT function returns the password hint for every row in the table.

```
SELECT GETHINT(EMPNO) FROM EMPL ;
```

```
---------+---------+---------+---------+
HINT
---------+---------+---------+---------+
ROTTIE
ROTTIE
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
```

Without the password, there is no way to decrypt the data. These encryption functions use the Triple Data Encryption Standard (DES) to perform the encryption.

### Select

In order to retrieve the data, the DECRYPT_CHAR function must be applied to EMPNO as:

```
SET ENCRYPTION PASSWORD = 'PEEKAY'    ;
SELECT SUBSTR(DECRYPT_CHAR(EMPNO),1,6) AS EMPNO,
       EMPNAME,CITY,SALARY
       FROM EMPL ;
```

which decrypt the EMPNO based on the password:

```
---------+---------+---------+---------+---------+---------+----
EMPNO   EMPNAME              CITY                          SALARY
---------+---------+---------+---------+---------+---------+----
12346   PAOLO BRUNI          KANSAS CITY                 20000.00
12347   RAMA NAIDOO          KANSAS CITY                 20000.00
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
```

### Insert

Data is inserted into EMPL, and values for EMPNO are encrypted with the ENCRYPT function.

Because you can specify a different password for every row that you insert, you can encrypt data at the cell level in your tables.

```
SET ENCRYPTION PASSWORD = 'ITSOSJ' WITH HINT 'SANJOSE'  ;
INSERT INTO EMPL(EMPNO,EMPNAME, SALARY)
VALUES (ENCRYPT('12346'),'PAOLO BRUNI',20000.00)  ;

SET ENCRYPTION PASSWORD = 'NAIDOO' WITH HINT 'ROWVILLE' ;
INSERT INTO EMPL(EMPNO,EMPNAME, SALARY)
VALUES (ENCRYPT('12347'),'RAMA NAIDOO',20000.00)  ;
```

In this case, the GETHINT function returns:

```
SELECT GETHINT(EMPNO) FROM EMPL ;

---------+---------+---------+---------+
HINT
---------+---------+---------+---------+
SAN JOSE
ROWVILLE
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
```

### Requirements

The DB2 built-in encryption functions require:

- DB2 V8.
- Integrated Cryptographic Service Facility (ICSF).
- On z990, CPACF is required (PCIXCC card is not, unless DRDA encryption is necessary).
- Pre-z990, cryptographic coprocessor is required.

Each CP on the z990 and later models has an assist processor on the chip in support of cryptography. This feature provides for hardware encryption and decryption support. PCIXCC provides a cryptographic environment with added function. To learn more about PCIXCC, refer to *IBM @server zSeries 990 (z990) Cryptography Implementation*, SG24-7070.

Applications that need to implement DB2 encryption must apply the DB2 encrypt and decrypt built-in functions to each column to be encrypted or decrypted. All encrypted columns must be declared “for bit data”. Unchanged read-applications see data in encrypted form. Applications may apply a different key for each column, but may also supply the key in a special register. We strongly recommend, for performance, that you specify the key in the special register.

The LOAD and UNLOAD utilities do not support the DB2 built-in encryption functions, but do handle broader encryption. SQL-based programs such as DSNTIAUL do support encryption. Encryption of numeric fields is not supported. The length of encrypted columns must allow for an additional 24 bytes, rounded up to a double-word boundary, for storing the encryption key. Space usage may be a concern if you plan to use DB2 to encrypt small columns.

Indexes are also encrypted. Predicates that depend on the collating sequence of encrypted columns (for example, range predicates) may produce wrong results (unless modified to use built-in functions correctly).

For example, the following statement produces the wrong results.

```
SELECT COUNT(*) WHERE COL =:HV;
```

The following statement produces the correct results with almost no impact to performance.

```
SELECT COUNT(*) WHERE COL = ENCRYPT_TDES(:HV);
```

The following statement produces the wrong results.

```
SELECT COUNT(*) WHERE COL < ENCRYPT_TDES(:HV);
```

The next statement produces the correct results with a large impact on performance.

```
SELECT COUNT(*) WHERE DECRYPT_CHAR(COL) <:HV;
```

### 1.1.14 IBM Data Encryption for IMS and DB2 Databases

IBM Data Encryption for IMS and DB2 Databases is the tool that provides you with a data encryption tool for both IMS and DB2 for z/OS databases in a single product. It enables you to protect your sensitive and private data for IMS at the segment level and for DB2 at the table level.

This tool performs row level encryption using EDITPROCs. Unlike the DB2 encryption functions shipped with DB2, the Data Encryption Tool uses different keys to encrypt different tables. The encryption keys can be either *clear*, such as the DB2 encryption functions, or *secure*. Plus they are managed through ICSF. Clear keys generally perform better. The tool also supports single, double, or triple DES. Again, refer to *IBM @server zSeries 990 (z990) Cryptography Implementation*, SG24-7070, to learn more about the clear and secure keys.

Customization consists of:

- Building a user exit routine (DB2 EDITPROC exit)
- Putting the user-specified encryption key label in the exit routine

The tool provides sample jobs to help you build the user exit (DB2 EDITPROC exit) and specify the encryption key label. Alternatively, the ISPF screen can be used to build the exit.

You can find more information about the IBM Data Encryption for IMS and DB2 Databases by visiting the Web at:

`http://www.ibm.com/software/data/db2imstools/db2tools/ibmencrypt.html`

The IBM Data Encryption for IMS and DB2 Databases tool supports all versions of DB2, and it encrypts only the whole row. No application changes are required. However you can include the EDITPROC only at the CREATE time of a table. Drop, create, runstats, and bind are necessary for adding the EDITPROC to an existing table. The applications do not need to be aware of encryption keys.

## 1.2 Information integrity

Leading-edge organizations have already switched from the data age to the information age, where their data is a critical resource capable of generating information that enables them to stay ahead of the competition. In the era of on demand information, more data than ever is generated, updated, managed, and exchanged. Data sits at the bottom of the information hierarchy, and without investing in its quality, no matter how good the tools and techniques, effective information will remain elusive, and if the competition has mastered the data foundations, they will take the lead.

Information integrity is the result of people, processes, *and* technology. Often data corruption issues arise from sources other than technology. A good data foundation requires a systematic, integrated approach to build an environment of people, processes, and technology that will foster sustainable information integrity.

Information integrity is the effectiveness of data needed to support the organization's strategic objectives, such as the ability to manage its assets and conduct its core operations. The level of information integrity required to effectively support operations varies by product line or functional area, depending on their unique information requirements. For example, a finance department will require high-quality data, while a marketing database may be able to achieve its objectives with less stringent information integrity requirements.

Information integrity can be measured across multiple dimensions, each of which is used to gauge how well a data element meets a company's information integrity goals, and ultimately, its information needs. These dimensions are summarized in Table 1-1.

*Table 1-1 Information integration dimensions*

| Dimension | Description | An example of lack of information integrity |
|---|---|---|
| Completeness | The data element is populated when required. | Customer's first name is missing. |
| Validity | The data element contains a valid data format or domain value. | State code is "ZZ". |
| Consistency | The data element is consistent with other related data or business rules. | Client's birth date is 01/01/1990 and retirement date is 01/01/2005. |
| Uniqueness | The data element is unique — there are no duplicate values. | Customer number is used twice in information file. |
| Accuracy (electronic) | The data element agrees with validated data from another source. | Address does not match with post office database. |
| Accuracy (real) | The data reflects the real-world object, event, or transaction. | Address does not match client's application form. |
| Precision | The data element holds data at sufficient granularity. | Product costs are stored as dollars, not dollars and cents. |
| Accessibility | The data element is accessible when required. | Birth date is in a source system, but not in the centralized customer information file. |
| Timeliness | The data element is available when it is required. | Data uploaded to data warehouse monthly, not daily. |
| Clarity | The data element is clearly defined and understood. | Customer record has free-text "order status" field, so the field cannot be analyzed. |
| Sufficiency | The data element contents are sufficient without making assumptions. | Order status "delayed" is used to identify out-of-stock items (because it is assumed that all delayed items are out of stock). |

Information integrity issues increase with the organization's dependence on data. Imbalances in the interrelationship between people, processes, and technology are exposed by corporate initiatives such as:

- Customer relationship management (CRM) and Enterprise Resource Planning (ERP) implementations
- Outsourcing
- New compliance requirements, such as Sarbanes-Oxley, Basel II, and the U.S.A. PATRIOT Act
- Business intelligence and data warehousing
- Performance improvement
- New delivery channels (such as self-service)
- System upgrades
- Cost reduction
- Joint ventures, mergers, acquisitions, and divestitures

The tangible benefits of data quality are reflected in cost reductions and improved business metrics. There are less-tangible costs associated with poor information integrity that require more effort to identify, quantify, and monitor, but the deteriorating quality of the information will ultimately affect the bottom line. Harvesting information integrity benefits requires an investment in an enterprise information integrity framework. The return on investment is high compared to the direct and indirect costs related to poor data quality. An initial investment in an enterprise information integrity framework can be leveraged across multiple systems, processes, and organizational units, further compounding the initial return on investment.

IBM Business Consulting Services can help organizations achieve information integrity through established best practices, proven through numerous successful customer engagements. The IBM enterprise information integrity framework highlights areas that must be managed to achieve information integrity. The IBM enterprise information integrity methodology captures the process for designing and implementing the IBM enterprise information integrity framework.

The information integrity framework highlights the various areas that must be managed in order to achieve sustainable information integrity. The framework, shown in Figure 1-3, is used to describe what a business environment looks like, but does not identify how the framework is developed or how an organization would make the transition to a new environment described in a framework. The information integrity methodology describes how you can tailor and implement the framework to help achieve and sustain information integrity.

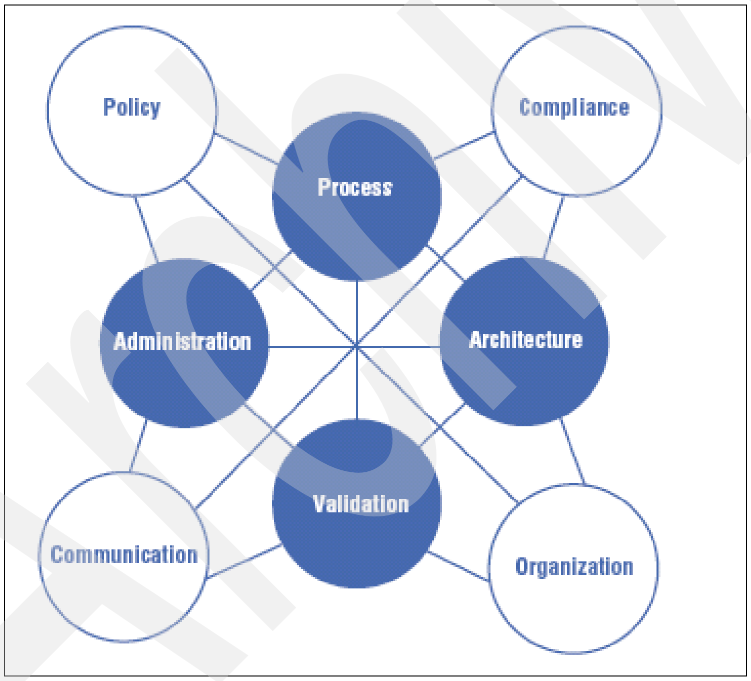


*Figure 1-3 Information integration framework*

An information integrity policy helps an enterprise to develop a strategic direction for the governance of its information assets, and directs, standardizes, and safeguards the corporation's information assets. An information integrity policy enables an organization to recognize that corporate data is a critical resource and to articulate corporate objectives with respect to information integrity.

Information organization focuses on empowering employees and defining:

- Roles and responsibilities — Data steward, information integrity governance council, chief information integrity officer, data architect, data administrator, database administrator, business expert, business user, and information integrity analyst
- Structure and work group design — Allocation of work activities, reporting relationships and the combination of resources needed to meet information integrity goals
- Job design — Assignment of groups of related tasks, activities, and procedures to specific roles
- Skills and behavior development — Development of skill sets and behaviors required to support new processes at the individual and group levels
- Performance management — Development and tracking of key success indicators for individuals, groups, and the overall business with respect to information integrity
- Communication — Education of everyone in the organization about the importance of information integrity, and creation of an awareness and understanding of the impact individuals have in the data value chain.

### *Information integrity data administration*

Information integrity needs a clear understanding within an organization of the meaning and usage of data elements, as well as their of standardization. Information integrity data administration involves envisioning a data administration framework that is applied consistently across the organization, verifying that data is standardized and that all stakeholders can readily access information about the current meaning and use of the data. The key to successful information integrity administration is the design of organizational structure and processes. Organizational structure and processes help ensure that business events, which cause changes to reference data, can be proactively captured, documented, and distributed. The key elements of information integrity administration are the metadata model, business rules, and the data model.

- Metadata — Documenting definitions for data elements helps build a solid foundation for the enterprise-wide metadata strategy. Valid domains for each key data element that identify the value ranges and types must also be defined.
- Business rules — Business rules are policies governing business actions. A business rule describes how to make a transition from one state to another or how to prohibit such a transition. Business rules govern updating, creating, or deleting objects and generally result in constraints or integrity parameters on data relationships or values. Since they tend to be volatile or subject to frequent change, there must be a single source where business rules are documented and distribution is controlled. This helps prevent different business areas or application development teams from creating inconsistent business rules that govern the same activity.
- Data model — Data design calls for understanding the business requirements and translating them into a data model. While a conceptual data model captures the business at a very high level, it provides an overview of the business entities in the scope of any system. A logical data model captures the entities and their relationships and cardinality, as well as rules governing the establishment of relationships. The business rules recorded in the logical data model and supporting documentation need to be incorporated into the physical designs (that is the physical data model and the physical database design). Lack of attention to the business rules results in an inability to meet business requirements and could potentially cause information integrity issues. The physical data model addresses the alignment of the data with the metadata model by implementing data type constraints, domain rules, and business rules within the data administration framework.

### *Information integrity architecture*

The information integrity architecture component of the framework is composed of data architecture, technology, and infrastructure, and it is closely linked to information integrity administration. It includes key information integrity best practices, such as verifying that:

- Data is electronically validated at all entry points.
- Systems sharing common data elements are linked so that they always share the most current view of those data elements.
- All data is held in a staging area where information integrity processes can be applied to it before it is populated into operating systems.
- Data redundancies are eliminated or streamlined through good architecture.
- Systems development methodologies incorporate data quality best practices.

Strong data architecture results from robust data design practices and enforcement of standards. This contributes significantly to the overall improvement of information integrity across the enterprise, meeting the core objective of the framework. As with information integrity administration, it is vital to link information integrity architecture directly to the change management process.

### *Information integrity framework compliance*

The interrelationships between organization, process, and technology are continually evolving and can cause the information integrity framework components to become outdated if a strong change management process is not present. For example, new errors can be caused by changes in data interfaces, staff, new business policies, and regulatory requirements. If existing processes and systems do not detect the errors, an enterprise will not be aware of the loss in data. To mitigate this risk, the framework compliance component of the information integrity framework assesses the information integrity program within an organization by evaluating the effectiveness of the integrated framework in sustaining data quality. The results of the framework compliance can then be used to certify systems, procedures, and business units to meet the standards set out in the information integrity policies and to identify areas where the framework elements need to be strengthened due to a change in circumstances. The process for executing information integrity framework compliance represents a condensed version of the methodology used to design and implement an information integrity assessment.

## 1.3 DB2 and data integrity

The SQL statements INSERT, UPDATE, and DELETE, as well as DB2 utilities, modify data in an existing database. Whenever you modify existing data, the logical integrity of the data can be affected. For example, an order for a nonexistent product could be entered into the orders table, a customer with outstanding orders could be deleted from the customer table, or the order number could be updated in the orders table and not in the items table. In each of these cases, the integrity of the stored data is lost.

Data integrity is actually a combination of:

- Entity integrity

  Each row of a table has a unique identifier.

- Semantic integrity

  The data in the columns properly reflects the types of information the column was designed to hold.

- Referential integrity

  The relationships between tables are enforced.

- Domain integrity

  Domain integrity applies to every column for which a domain is associated. A domain defines a pool of all values that are valid for a column associated with this domain.

  If different columns draw their values from the same domain, then it makes sense to do comparisons of the columns, because you would be comparing compatible columns. In particular, if domains were implemented, it would be possible to enforce a constraint that the domain of the foreign key must belong to the same domain (or subset) of the parent key it references.

Well designed databases incorporate these principles so that when you modify data, the database itself prevents you from doing anything that might harm the data integrity.

In this section, we briefly introduce each of the functions. For more information about DB2 data integrity, refer to Chapter 10, "Using constraints to maintain data integrity", of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

## 1.3.1 Entity integrity

An entity is any person, place, or thing to be recorded in a database. Each table represents an entity, and each row of a table represents an instance of that entity. For example, if an order is an entity, the ORDERS table represents it and each row in the table is an instance of a specific order.

To identify each row in a table, the table must have a *primary key*. The primary key is a unique value that identifies each row. This requirement is called the entity integrity constraint.

For example, the ORDERS table primary key is ORDER_NUM. The ORDER_NUM column holds a unique system-generated order number for each row in the table. To access a row of data in the ORDERS table, you can use the following SELECT statement:

```
SELECT * FROM ORDERS WHERE ORDER_NUM = 90090
```

Using the order number in the WHERE clause of this statement enables you to localize a row easily because the order number uniquely identifies that row. If the index allowed duplicate order numbers, it would be almost impossible to access one single row, because all other columns of this table allow duplicate values.

## 1.3.2 Semantic integrity

Semantic integrity ensures that data entered into a row reflects an allowable value definition for that row. The value must be within the column-specific properties, or allowable set of values, for that column. For example, the QUANTITY column of the ITEMS table permits only numbers. If a value outside the column-specific properties can be entered into a column, the semantic integrity of the data is violated.

Semantic integrity is enforced using the following constraints:

- Data type

  The data type defines the types of values that you can store in a column. For example, the data type SMALLINT allows you to enter values from -32,767 to 32,767 into a column.

- Default value

  The default value is the value inserted into the column when an explicit value is not specified. For example, the USER_ID column of the CUST_CALLS table defaults to the login name of the user if no name is entered.

- Check constraint

  The check constraint specifies conditions on data inserted into a column. Each row inserted into a table must meet these conditions. For example, the quantity column of the items table might check for quantities greater than or equal to one.

## 1.3.3 Referential integrity

Referential integrity refers to the relationship between tables. Because each table in a database must have a primary key, this primary key can appear in other tables because of its relationship to data within those tables. When a primary key from one table appears in another table, it is called a *foreign key*.

Primary and foreign keys are the important and basic components on which the relational theory is based. Primary keys enforce entity integrity by uniquely identifying entity instances. Foreign keys enforce referential integrity by completing an association between two entities. The primary key of a relational table uniquely identifies each record in the table. The primary key of a relational table can be a normal attribute that is guaranteed to be unique (such as EMPNO in EMPLOYEE table) or it can be generated by the DBMS (such as the IDENTITY column in DB2).

Foreign keys logically join tables and establish dependencies between tables. Tables can form a hierarchy of dependencies in such a way that if you change or delete a row in one table, you destroy the meaning of rows in other tables. For example, the CUSTOMER_NUM column of the CUSTOMER table is a primary key for that table and a foreign key in the ORDERS and CUST_CALL tables. If a CUSTOMER_NUM is deleted from the CUSTOMER table, the logical link across the three tables and this particular customer is destroyed.

When you delete a row that contains a primary key or update it with a different primary key, you destroy the meaning of any rows that contain that value as a foreign key. Referential integrity is the logical dependency of a foreign key on a primary key. The integrity of a row that contains a foreign key depends on the integrity of the row that it references, that is the row that contains the matching primary key.

If the referential integrity is implemented in the database, by default, the database does not allow you to violate referential integrity and gives you an error message if you attempt to delete rows from the parent table before you delete rows from the child table. You can, however, use the ON DELETE CASCADE option to cause deletes from a parent table to trigger deletes on child tables.

To define primary and foreign keys, and the relationship between them, you use options of the CREATE TABLE and ALTER TABLE statements.

For instance, an EMPLOYEE table and a DEPARTMENT table can be defined as shown in Example 1-2.

*Example 1-2 Referential integrity*

```
CREATE TABLE EMPLOYEE (
EMPNO INTEGER NOT NULL PRIMARY KEY,
NAME CHAR(30) NOT NULL,
SALARY DECIMAL(7,2) NOT NULL,
WORKDEPT SMALLINT);
```

```
CREATE TABLE DEPARTMENT (
DEPTNO SMALLINT NOT NULL PRIMARY KEY,
DEPTNAME VARCHAR(30),
MGRNO INTEGER NOT NULL, CONSTRAINT REF_EMPNO FOREIGN KEY (MGRNO) REFERENCES EMPLOYEE
(EMPNO) ON DELETE RESTRICT);
```

The primary key of the EMPLOYEE table is the EMPNO column. The foreign key of the DEPARTMENT table, the MGRNO column, reflects values from EMPNO. We specify a constraint name for the foreign key, REF-EMPNO. This name is used in error messages, queries to the catalog, and DROP FOREIGN KEY statements.

The delete rule applies when a row of the EMPLOYEE table is deleted and that row has dependents in the DEPARTMENT table. Since we have chosen RESTRICT, upon such deletion, an error occurs and no rows are deleted.

### 1.3.4 Domain integrity

A domain is basically the set of valid values that a column or data type can assume. Check constraints can provide functions similar to the relational concept of a domain. Check constraints can impose restrictions on the data values of a column through the specification of Boolean expressions. The expression is explicitly defined in the table DDL and is formulated in a way similar to the SQL WHERE clause. Updates and inserts to the data in the column trigger the evaluation of the expression and the update is either allowed or rejected for constraint violation.

However, there is more to domain integrity than what check constraint can provide, such as user-defined data types with strong type checking. This is how the function avoids problems such as adding U.S. dollars and Euros. To provide this extra function, use DB2 distinct types.

## 1.4 Example of integrity needed across applications

Suppose application A is being enhanced to provide a sales rollup report for the entire company and that a company wide organization hierarchy table already exists in application B as B.ORG.

Because this organizational info is needed to produce the required content by the developers supporting application A, they decide to join data from application A with the B.ORG table in application B.

Application A code changes are delivered to production.

On Monday night, the first batch run (with the enhancement) for application A works successfully. A few days later, it is discovered that the results from the first batch run are incorrect, some input was missing. So the plan is to restore the tables in Application A and rerun the "Monday" batch run.

Now if application A can be rerun and can use the current contents of B.ORG, all is fine.

If application A needs to use the exact contents of B.ORG as it was when the first run took place, there is a problem.

One approach is to request that application B temporarily restores the contents of B.ORG to the appropriate point in time on Monday before the first batch run. This may not be possible for application B because it is a 24/7 application or there may not be a usable quiesce point to which to go back.

An alternative could be to create a clone of B.ORG, and, assuming that a backup was available, restore the data to the clone and then point the code in application A to the clone, possibly using an alias.

If many applications need this core organization data, then consider establishing a Master table and let all the applications which need such information copy the data from the Master version into their database. Another possibility is to maintain organization history in a central Master table using an effective business date to represent how the organization has evolved.

This illustrates some points to consider when deciding to reuse another application's data.

Another approach would be to create an A.ORG table, and populate from B.ORG before the batch run and also include this A.ORG table in application A's referential structure.

## 1.4.1 Customer names and addresses across applications

Companies typically run many applications and some of these may use DB2 for z/OS.

Some of these applications are new and some have been in place for a long time. Applications are also purchased and acquired through mergers. Within those applications dealing with customer information, it is not surprising to see different formats for names and addresses as well as different levels of quality in the data.

In a financial institution, for example, a customer may purchase different products and have the customer’s name and address stored in different systems, such as Mortgage, Loan, Credit Card, and Retail Banking. Maintaining the integrity of a customer's name and address in different systems can be a challenge.

One alternative is to designate a master reference and propagate changes to other systems which could be DB2, other DBMSs, or flat files.

In other cases, information can be consolidated in a warehouse where the customer's complete portfolio is captured and where the name and address information is matched and "conformed."

IBM provides a solution in this area, it is IBM Master Data Management (MDM).

IBM defines MDM as the set of disciplines, technologies, and solutions used to create and maintain consistent, complete, contextual, and accurate business data for all stakeholders. MDM is an approach to control the proliferation and inconstancy of vital data. IBM Master Data Management is SOA-based middleware designed to provide organizations the most flexible framework to support enterprise structured and unstructured data and business services, aligned with key business processes. IBM brings together all the key core components required for a successful enterprise MDM strategy: information integration, content management, business intelligence, and master data management for specific data objects – including product, customer, and supplier – and master data solutions for specific industries.

2

# Semantic integrity

In this chapter, we describe two sets of DB2 functions which help control the integrity of your DB2 data: constraints and distinct types.

Constraints are rules that limit, restrict, and regulate the values that you can use to define, add, or modify a table. They help toward domain support. The first step to ensure the integrity of your data is choosing the right data type for each attribute of your tables. This way, you can avoid character strings in a numeric field or non-date values in a date field.

Distinct types augment the basic column-level integrity DB2 provides. They are user-defined data types that share their internal representation with a built-in data type, but are considered a separate and incompatible data type for SQL.

## 2.1 Constraints

Constraints enable enhanced data integrity without requiring procedural logic (such as in stored procedures and triggers). Check constraints are written using SQL syntax. They are easy to implement and consist of two components: a constraint name and a check condition.

The constraint name is an SQL identifier and is used to reference the constraint. If a constraint name is not explicitly coded, the DBMS typically creates a unique name automatically for the constraint.

The check condition defines the actual constraint logic defined by specifying basic predicates (>, <, =, <>, <=, >=), BETWEEN, IN, LIKE, and NULL. Furthermore, AND and OR can be used to string conditions together.

In this section, we describe the following types of constraints:

- Data constraint
- NOT NULL constraints
- Unique constraint
- Check constraints

### 2.1.1 Data constraint

DB2 currently provides 19 different built-in data types classified according to the type of the data:

- Numeric
- Strings
- Date and time
- Large objec

#### Common built-in data types

In the following tables, we show the common built-in data types.

The numeric data types are listed in Table 2-1.

*Table 2-1 Numeric data types - Contain digits*

| Data type | Description |
|---|---|
| SMALLINT | For small integers. The range is -32,768 to 32,767. |
| INTEGER | For large integers. The range is -2,147,483,648 to 2,147,483,647 |
| DECIMAL(p,s) | For numbers that have both whole and fractional parts. p is the whole or precision and s is the fractional or scale. |
| REAL | The single precision floating-point number is a short (32 bit) floating-point number with the range of -7.2E+75 to 7.2E+75. |
| FLOAT (or DOUBLE) | The double precision floating-point number is a long (64 bit) floating-point number with the range of -7.2E+75 to 7.2E+75. |

The character data types are listed in Table 2-2.

*Table 2-2 String data types - Contain alphanumeric characters*

| Data type | Description |
|---|---|
| CHAR | For fixed-length character string. The length is between 1 to 254 characters. |
| VARCHAR | For varying-length character string. The length is up to 32,672 characters. |
| GRAPHIC | For double-byte character set (DBCS) strings up to 127 bytes in length. |

The date and time data types are listed in Table 2-3.

*Table 2-3 Date and time data types - Contain dates and times*

| Data type | Description |
|---|---|
| DATE | Dates with three-part value that represent year, month, and day. |
| TIME | Times with three-part value that represent hour, minute, and second. |
| TIMESTAMP | Timestamps with seven-part value that represents date and time by year, month, day, hour, minute, second, and microsecond. |

From a user perspective, the date and time values appear to be character strings; however, they are physically stored as binary packed strings.

As an example, the following three columns defined in the EMPLOYEE table have the three data types mentioned:

- EMPNO CHAR(6) NOT NULL

  Employee identification number. It can use alphanumeric characters.
- HIREDATE DATE

  Date of hire. Using this built-in data, you will avoid non-date values in this field.
- SALARY DECIMAL(9,2)

  Annual salary in dollars.

## 2.1.2 NOT NULL constraints

We use the NULL values to represent unknown or missing data. But sometimes the business rules declare a null value unacceptable. For instance, in the EMPLOYEE table, each EMPLOYEE must have an identification number. In this case, you can use the NOT NULL constraint to ensure that this specific column will always have data on it.

There are situations where you do not want NULL values, you have a default value defined by your business rules, in cases such as hire date. You can always put a system date if hire date was not provided.

**Note:** NULL value is not the same as blank strings or zeros. NULL values are considered, but they are not included in the results of aggregate functions, such as MIN, MAX, AVG, and SUM.

## 2.1.3 Unique constraint

By default, a unique constraint is an SQL rule that ensures that no two values in the same column or in a specific group of columns are the same (entity integrity). For instance, in an employee table, the ID column must be unique because it represents one and only one employee. Enforcing the unique constraint in the EMPLOYEE table helps accesses and avoids insertion of a new employee using the same ID.

A unique constraint can be established using the clauses PRIMARY KEY (see 3.5.1, “Primary key” on page 82) or UNIQUE in the CREATE TABLE or ALTER TABLE. The columns specified in a unique constraint must be defined as NOT NULL or NOT NULL WITH DEFAULT and the data type cannot be LOB or ROWID (including a distinct type that is based on a ROWID data type).

A unique index enforces the uniqueness of the key during changes to the columns of the unique constraint. For each unique constraint, an unique index must be created. If the unique index already exists for the same set of columns, it will assume this index as the unique constraint rule to enforce integrity.

**Note:** DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete only if the CREATE statement is processed by the schema processor. For information about the schema processor (DSNHSP batch job with sample JCL provided in member DSNTEJ1S of the SDSNSAMP library), refer to*DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413.

## CREATE TABLE with UNIQUE constraint

There are two places where you can define the unique clause in the CREATE statement:

- In the column definition, as shown in Example 2-1.

*Example 2-1 Unique clause besides column definition*

```
CREATE TABLE SAMPLE.EMP
(EMPNO    CHAR(6) NOT NULL UNIQUE,
NAME      VARCHAR(60) NOT NULL,
WORKDEPT  CHAR(3),
) IN DATAINT.EMPTS
--
CREATE TABLE SAMPLE.EMP
(EMPNO    CHAR(6) NOT NULL CONSTRAINT EMPNO UNIQUE,
NAME      VARCHAR(60) NOT NULL,
WORKDEPT  CHAR(3),
) IN DATAINT.EMPTS
```

- At the end of column definition, as shown in Example 2-2. With this option, you can refer to more than one column in the same unique constraint clause.

*Example 2-2 Unique clause at the end of column definition*

```
CREATE TABLE SAMPLE.EMP
(EMPNO    CHAR(6) NOT NULL UNIQUE,
NAME      VARCHAR(60) NOT NULL,
WORKDEPT  CHAR(3),
CONSTRAINT EMPNO_NAME UNIQUE (EMPNO, NAME)
) IN DATAINT.EMPTS
```

The table definition is incomplete at this point. As unique constraint is enforced by a unique index, you must create a unique index with the same columns of the unique constraint.

### ALTER TABLE to add a UNIQUE constraint

In Example 2-3, we show an example of using ALTER to add a unique constraint. The table must have a unique index with a key that is identical to the unique key. The keys are identical only if they have the same number of columns and the *n*th column name of one is the same as the *n*th column name of the other.

*Example 2-3 Adding a unique constraint*

```
ALTER TABLE SAMPLE.EMP
   ADD UNIQUE(EMPNO);
--
ALTER TABLE SAMPLE.EMP
   ADD CONSTRAINT EMPNO UNIQUE(EMPNO);
--
ALTER TABLE SAMPLE.EMP
   ADD CONSTRAINT EMPNO UNIQUE(EMPNO, NAME);
```

### Considerations on constraints

Use the following considerations on constraints:

- If you need to add a constraint in a table (ALTER TABLE), the unique index must be already created.
- You cannot drop an index if it is the index to ensure the unique constraint rule. First, you need to drop the constraint using the ALTER TABLE statement.
- There is no limit to how many unique constraints a table can have, but a table cannot have more than one unique constraint defined on the same set of columns.
- Because unique constraints are enforced by indexes, all the limitations that apply to indexes apply to unique constraints.
- You can always define a name for your unique constraint. If the CONSTRAINT clause is omitted, DB2 assigns an internally generated name based on the column name.
- DB2 SYSIBM.SYSTABCONST catalog table has information on the unique constraints (see “SYSIBM.SYSTABCONST” on page 104).

### Why use unique constraint

Although a unique, system-required index is used to enforce a unique constraint, there is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns. So, unique index without a unique constraint cannot be used in a referential constraint. For a referential constrain, the parent table must have a defined primary key or a defined unique constraint.

## 2.1.4 Check constraints

Check constraints designate the set of values that specific columns of a base table can contain. The designated values are explicitly defined, through a Boolean expression, in the table DDL and formulated as SQL WHERE clauses. The check constraint is enforced every time a column is inserted or updated. If the modification does not comply with the Boolean expression, the statement fails with a specific SQL code constraint violation.

For example, if you want to make sure that no salary can be below 15,000 dollars, you can create the following check constraint:

```
CREATE TABLE EMPSAL (ID INTEGER NOT NULL, SALARY INTEGER CHECK (SALARY >= 15000));
```

Check constraints provide a very powerful way to support business rules in the database. Because check constraints cannot be bypassed, they provide better data integrity than equivalent logic programmed into the application.We recommend to use check constraints to support data integrity, domains, and business rules in your applications.

Using check constraints makes your programming task easier, because you do not need to code them in the application programs or with a validation routine. It makes your system more robust and consistent because they are implemented once, in the table DDL, and each constraint is always enforced. Constraints written in application logic, on the other hand, must be executed by each program that modifies the data to which the constraint applies. This can cause code duplication and inconsistent maintenance resulting in inaccurate business rule support. Another important benefit of check constraints is that plans and packages do not need to be rebound after check constraints are defined on or removed from a table. However, the program should be coded to report the constraint's name on which inserts are failing.

The catalog tables used with check constraint definitions are shown in 3.13.1, "DB2 catalog extensions" on page 99.

## Creating tables with check constraint

The check condition defines the actual constraint logic. The check condition can be defined using any of the basic predicates (>, <, =, <>, <=, >=), as well as BETWEEN, IN, LIKE, and NULL. You can use AND and OR to string conditions together. The syntax is checked at definition time.

Example 2-4 shows how to create a table with the following restrictions:

- Annual salary column (SALARY) cannot be less than 15000.
- The sex of the employee column (SEX) is "M" or "F".
- The employee bonus (BONUS) cannot be higher than the commission (COMM).

*Example 2-4 Employee table with check constraint*

```
CREATE TABLE EMPLOYEE (
    EMPNO CHAR(6) NOT NULL
, SEX CHAR(1) CONSTRAINT SEX_GENDER CHECK (SEX IN ('M', 'F'))
  , BIRTHDATE DATE
  , SALARY DECIMAL(9,2) NOT NULL
  , BONUS DECIMAL(9,2) NOT NULL
  , COMM DECIMAL(9,2)
  , CONSTRAINT MIN_SAL CHECK (SALARY >= 15000)
  , CONSTRAINT BONUS_COMM CHECK (BONUS <= COMM)
)
IN FABRICIO.FABTS1
```

There are two ways to define the check constraint:

- Column level:

  ```
  SEX CHAR(1) CONSTRAINT SEX_GENDER CHECK (SEX IN ('M', 'F'))
  ```

- Table level:

  ```
  CONSTRAINT MIN_SAL CHECK (SALARY >= 15000)
  ```

If a constraint name is not specified, DB2 provides a unique constraint name based on the first column referenced in the check condition.

DB2 checks the syntax of the check constraint but not the semantic. You must make sure you do not define constraints that contradict another one.

Some examples of contradictory constraints:

- A self-contradictory check constraint:

  `CHECK (SALARY > 50000 AND SALARY < 49999)`
- Two check constraints that contradict each other:

  `CHECK (BONUS > 30000) CHECK (BONUS < 2000)`
- Two check constraints, one of which is redundant:

  `CHECK (COMM > 1000) CHECK (COMM >= 1000)`
- A check constraint that contradicts the column definition:

  `BONUS DECIMAL (9,2) NOT NULL CHECK (BONUS IS NULL)`
- A check constraint that repeats the column definition:

  `COMM DECIMAL (9,2) NOT NULL CHECK (COMM IS NOT NULL)`

There are check constraint restrictions that you cannot include in the definition:

- Subselects, column functions, host variables, parameter markers, special registers, and columns defined with field procedures
- Columns from other tables
- The NOT logical operator

A check constraint is not checked for consistency with other types of constraints. For example, a column in a dependent table can have a referential constraint with a delete rule of SET NULL. You can also define a check constraint that prohibits nulls in the column. As a result, an attempt to delete a parent row fails, because setting the dependent row to null violates the check constraint.

## Adding check constraint

The options of the CURRENT RULES statement decide if an additional check constraint can be successfully added (ALTER TABLE) to a table already populated:

- STD

  An error occurs if a row does not satisfy the check constraint rule (see Example 2-5.)

*Example 2-5 CURRENT RULES - STD*

```
SET CURRENT RULES = ‘STD’
ALTER TABLE EMPLOYEE2
  ADD  CONSTRAINT SEX CHECK(SEX IN ('M','F'))
---------+---------+---------+---------+---------+---------+---------+---------+
DSNT408I SQLCODE = -544, ERROR:  THE CHECK CONSTRAINT SPECIFIED IN THE ALTER
         TABLE STATEMENT CANNOT BE ADDED BECAUSE AN EXISTING ROW VIOLATES THE
         CHECK CONSTRAINT
```

- DB2

  The check constraint is added, but the table space that holds the altered table is placed in CHECK-pending status (see Example 2-6.)

*Example 2-6 CURRENT RULES - DB2*

```
SET CURRENT RULES = 'DB2';
ALTER TABLE EMPLOYEE2
  ADD  CONSTRAINT SEX CHECK(SEX IN ('M','F'))
---------+---------+---------+---------+---------+---------+---------+---------+
DSNT404I SQLCODE = 162, WARNING:  TABLE SPACE FABRICIO.FABTS2 HAS BEEN PLACED
         IN CHECK PENDING
```

### CHECK-pending

When a table space is placed in CHECK-pending, you are not be able to SELECT, INSERT and UPDATE, or DELETE from any table in the table space. The only statement that you will be able to execute is a drop table. The CHECK-pending status indicates that the object might be inconsistent.

Beside the common case when you add a check constraint, there are other cases that the table space can be placed in CHECK-pending status:

- The LOAD utility is run with CONSTRAINTS NO, and check constraints are defined on the table.
- CHECK DATA is run on a table that contains violations of check constraints.
- A point in time RECOVER introduces violations of check constraints.

### Resetting the CHECK-pending status

You can read a discussion of how to reset the CHECK-pending status at 3.9.1, “CHECK DATA” on page 89.

## 2.2 Distinct types

This section describes how distinct types augment the basic column-level integrity DB2 provides.

In this chapter, we discuss the following topics:

- Why distinct types
- Creating a distinct type
- Generated cast functions
- Comparing distinct types
- Assigning a distinct type
- Invoking routines with distinct types
- Errors with comparisons across distinct types
- Summary and usage recommendations

### 2.2.1 Why distinct types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in data type (its *source type*) but is considered to be a separate and incompatible data type for most operations.

**Note:** Distinct types are referred to as “user-defined types” (UDTs) in some literature. In DB2 UDB for z/OS, the term “distinct type” is used instead.

Consider the following example. You have created three tables using the statements listed in Example 2-7.

*Example 2-7 Tables with different currency*

```
CREATE TABLE EMP1
   (EMP_NO CHAR(6) NOT NULL,
   SALARY_USD DECIMAL(9,2));

CREATE TABLE EMP2
   (EMP_NO CHAR(6) NOT NULL,
   SALARY_EURO DECIMAL(9,2));

CREATE TABLE EMP3
   (EMP_NO CHAR(6) NOT NULL,
   SALARY_YEN DECIMAL(9,2));
```

Since the three columns SALARY_USD, SALARY_EURO, and SALARY_YEN are defined with compatible data types, DB2 allows a comparison among these columns. Such comparisons, while syntactically valid, are semantically incorrect. Comparisons make business sense in this case only after the proper exchange rate has been applied.

Another example is one where two columns, both defined as BLOBs, contain *audio* and *video* data. Comparisons between such columns, once again, are syntactically correct but semantically meaningless.

Distinct types provide a means to impose *strong typing* which enforces data integrity by ensuring that only valid comparisons are allowed. In addition, it ensures that functions that require an input of a certain type (such as *USD* or *audio*) are able to validate that the input is of the valid type. In this chapter, we explore this issue in some detail.

## 2.2.2 Creating a distinct type

The CREATE DISTINCT TYPE statement defines a distinct type, which is a data type that a user defines. A distinct type must be based on one of the built-in data types. For a detailed discussion of the authorization required and the syntax of the statement, see “CREATE DISTINCT TYPE” in *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

Example 1: Create a distinct type named SALARY_USD that is based on a decimal data type with a precision and scale of 9 and 2 respectively.

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
```

Example 2: Create distinct types named MPH and KPH that are based on an integer data type.

```
CREATE DISTINCT TYPE MPH AS INTEGER;

CREATE DISTINCT TYPE KPH AS INTEGER;
```

## 2.2.3 Generated cast functions

Successful execution of the CREATE DISTINCT TYPE statement also generates the following functions:

- A function to cast between the distinct type and its source type
- A function to cast between the source type and its distinct type

The statement:

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
```

generates the following two functions:

```
SALARY_USD (accepts DECIMAL as input) and
DECIMAL (accepts SALARY_USD as input)
```

In addition, the WITH COMPARISONS clause can be specified during the CREATE.

```
CREATE DISTINCT TYPE MPH AS INTEGER WITH COMPARISONS;
```

Note that the WITH COMPARISONS clause is required for compatibility with other members of the DB2 family. It cannot be used when the source data type is BLOB, CLOB, or DBCLOB. In DB2 z/OS, the clause has no effect when specified.

### 2.2.4 Comparing distinct types

The basic rule for comparisons is that data types of the operands must be compatible. For example, all numeric data types (SMALLINT, INTEGER, FLOAT, and DECIMAL) are compatible. This allows you to compare an INTEGER value with one defined as FLOAT.

However, you cannot compare an object of a distinct type to an object of a different type. You can compare an object with a distinct type only to an object with exactly the same distinct type. Similarly, you also cannot compare a distinct type directly to its source type. You can do so only by using a cast function. We illustrate this with an example.

You have created a distinct type with a statement like this:

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
```

You have also created a table using a statement like this:

```
CREATE TABLE EMP1
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_USD);
```

To retrieve all employees with a salary greater than 10,000 USD, you cannot specify a predicate like this:

```
WHERE SALARY > 10000
```

Instead you must use one of the two cast functions automatically created (as discussed in 2.2.3, “Generated cast functions” on page 43):

```
WHERE SALARY > SALARY_USD(10000)
```

or

```
WHERE DECIMAL(SALARY) > 10000
```

This casting satisfies the requirement that the compared data types are identical.

When using dynamic SQL, parameter markers (identified as question mark “?”) replace host variables. In this case, you must use one of these two alternatives:

```
WHERE SALARY > CAST (? AS SALARY_USD)
```

or

```
WHERE CAST(SALARY AS DECIMAL) >?
```

### 2.2.5 Assigning a distinct type

For assignments from columns to columns or from constants to columns of distinct types, the type of that value to be assigned must match the type of the object to which the value is assigned, or you must be able to cast one type to another.

We consider the three different types of assignments in the following three sections.

**Important:** The rules for assigning distinct types to host variables to columns of distinct types differ from the rules for constants and columns.

## Column to host variable

You can assign a column value of a distinct type to a host variable if you can assign a column value of the distinct type's source type to that host variable. For example, with the following definitions of the distinct type and the table using it:

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
```

and

```
CREATE TABLE EMP1
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_USD);
```

the values of the column SALARY can be retrieved into any host variable that is compatible with the DECIMAL data type.

## Host variable or constant to column

When you assign a value in a host variable to a column with a distinct type, the type of the host variable or constant must be able to cast to the distinct type. For a table of base data types and the base data types to which they can be cast, see "Promotion of Data Types" in the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

In the example we described in "Column to host variable" on page 45, any data type that is compatible with the DECIMAL data type is an acceptable host variable for assignment to the SALARY column.

## Column to column

DB2 does not let you assign the values of a column to another when they are of different distinct types. In this case, a function must exist that converts the value from one type to another. Since DB2 provides the cast functions only between a distinct type and its source type, you must write the function to convert from one distinct type to another. We illustrate this with an example.

You have created distinct types with a statements below:

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
CREATE DISTINCT TYPE SALARY_EURO AS DECIMAL(9,2);
```

You have also created tables using statements below:

```
CREATE TABLE EMP1
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_USD);

CREATE TABLE EMP2
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_EURO);
```

To insert all rows from EMP2 to EMP1, the following statement causes a type mismatch and will **not** work:

```
INSERT INTO EMP1
SELECT EMP_NO, SALARY
      FROM EMP2
```

You must first write a user defined function that accepts SALARY_EURO as input and returns SALARY_USD. This looks something like this:

```
CREATE FUNCTION EURO_TO_DOLLAR(SALARY_EURO)...
```

This function is an external user defined function (UDF) written in a host language such as C or COBOL. See “Create Function” in the *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426, for further information about this topic.

Once the user defined function is available, you can then issue:

```
INSERT INTO EMP1
SELECT EMP_NO, EURO_TO_DOLLAR(SALARY)
      FROM EMP2
```

### 2.2.6 Invoking routines with distinct types

DB2 enforces strong typing when you pass arguments to a routine (stored procedure or a user defined function). This means that you can pass arguments that have distinct types to a routine if either one of the following conditions is true:

- A version of the routine that accepts those distinct types is defined (Note: this applies to infix operators - you must create a function for handling CONCAT, +, -, *, and /).
- You can cast your distinct types to the argument types of the routine.

Assume you have created a distinct type as shown below:

```
CREATE DISTINCT TYPE FLIGHT_TIME AS TIME;
```

You have also created a table using the distinct type as shown below:

```
CREATE TABLE FLIGHTS
   (FLIGHT_NO SMALLINT NOT NULL,
    HOW_LONG FLIGHT_TIME);
```

If you want to invoke the built-in HOUR function that accepts only the TIME or TIMESTAMP data type as an argument, you must do one of two things, you cast it as:

```
SELECT FLIGHT_NO, HOUR(TIME(HOW_LONG)
FROM FLIGHTS;
```

Alternatively, you create a UDF as shown:

```
CREATE FUNCTION HOUR(FLIGHT_TIME)
RETURNS INTEGER
SOURCE SYSIBM.HOUR(TIME)
```

and then issue:

```
SELECT FLIGHT_NO, HOUR(HOW_LONG)
FROM FLIGHTS;
```

By creating a sourced function HOUR on this distinct type, you are telling DB2 to process it just like the built-in HOUR function. This must be done explicitly, that is, built-in functions are not automatically available to distinct types.

### 2.2.7 Errors with comparisons across distinct types

Suppose you have created distinct types with the statements:

```
CREATE DISTINCT TYPE SALARY_USD AS DECIMAL(9,2);
CREATE DISTINCT TYPE SALARY_EURO AS DECIMAL(9,2);
```

You have also created tables using statements below:

```
CREATE TABLE EMP1
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_USD);

CREATE TABLE EMP2
   (EMP_NO CHAR(6) NOT NULL,
   SALARY SALARY_EURO);
```

You now try to compare the two SALARY columns with different distinct types associated with them as shown in Example 2-8.

*Example 2-8   Comparing two different data types*

```
   -- Show all European employees who earn more than employee 123456.
   SELECT E2.EMP_NO,
          E2.SALARY,
   FROM EMP1 E1,
      EMP2 E2
   WHERE E1.EMP_NO = ‘123456’
   AND E2.SALARY > E1.SALARY;
```

This comparison results in the error reported in Example 2-9.

*Example 2-9   Incompatible data types*

```
   DSNT408I SQLCODE = -401, ERROR:  THE OPERANDS OF AN ARITHMETIC OR COMPARISON
            OPERATION ARE NOT COMPARABLE
   DSNT418I SQLSTATE   = 42818 SQLSTATE RETURN CODE
   DSNT415I SQLERRP    = DSNXOBFB SQL PROCEDURE DETECTING ERROR
   DSNT416I SQLERRD    = 920 0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
   DSNT416I SQLERRD    = X'00000398'  X'00000000'  X'00000000'  X'FFFFFFFF'
            X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

This message is the same one you receive when you attempt to compare incompatible base data types; for example, a numeric column to a character column. The presence of the user defined type is not explicit in the message.

## 2.2.8 Summary and usage recommendations

We have seen that distinct types provide a means to impose *strong typing* which enforces data integrity by ensuring that only valid comparisons are allowed and that functions requiring input of a certain type are properly invoked.

This integrity, however, comes at a price. Besides the need to create additional conversion functions when necessary, they make access to the underlying data more complex and add overhead when an external user defined function must be called for conversion. Distinct types prevent an ad hoc user from adding, for example, USD and Euro amounts together which would otherwise produce a meaningless result. In general, their use is justified when integrity is of great importance and ad hoc data access by users, who are only partially cognizant of the data types, is common. In such cases, distinct types can help prevent accidental loss of integrity.

You should consider the trade-offs carefully before deciding to use distinct types. For new application systems with ad hoc access, they provide a definite value but retrofitting into an existing application may make application development more complex.

3

# Referential integrity

Referential integrity (RI) is the status of a logical database in which all values of all foreign keys are valid. DB2-enforced RI is implemented by referential constraints. The definition of referential constraint is the requirement that non-null values of a designated foreign key are valid only if they equal values of the parent key of a designated table. For details, see *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

Maintaining RI requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

In this chapter, we discuss the following topics:

- Referential constraints
- RI in the relational model
- RI in DB2
- Functional implications
- Summary of design recommendations
- DB2 versus application RI
- REPORT utility
- CHECK utility
- LOAD utility
- Performance
- Migrating applications to RI
- DB2 catalog information and queries

## 3.1 Referential constraints

RI requires the enforcement of referential constraints on all LOAD, INSERT, UPDATE, and DELETE operations on the data in a table on which the referential constraints are defined.

The DB2 referential constraints are:

- Primary or unique key
- Foreign key with the specific delete rule

The DB2 RI DELETE rules are:

- CASCADE
- RESTRICT
- SET NULL
- NO ACTION

The referential constraints are enforced for all the following cases:

- INSERT: Check the validity of the primary key and in the dependent tables check if there is a primary key in the parent table.
- UPDATE: The update rule is always RESTRICT.
- DELETE: The effects of a delete depend on the delete rule specified.
- LOAD utility without ENFORCE NO.
- CHECK DATA utility.

A referential constraint is a rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's ID changes. You can define a referential constraint stating that the ID of the supplier in the table must match a supplier ID in the supplier information. This constraint prevents insert, update, or delete operations that would otherwise result in missing or incorrect supplier information.

A referential constraint is the rule that the non-null values of a foreign key are valid only if they also appear as values of its parent key. The table that contains the parent key is called the parent table of the referential constraint, and the table that contains the foreign key is a dependent of that table.

Referential constraints are optional and can be defined using CREATE TABLE or ALTER TABLE statements.

Referential constraints between base tables are also an important factor in determining whether a materialized query table can be used for a query. For instance, in a data warehouse environment, data is usually extracted from other sources, transformed, and loaded into data warehouse tables. In such an environment, the RI constraints can be maintained and enforced by other means than the database manager to avoid the overhead of enforcing them by DB2. However, referential constraints between base tables in materialized query table definitions are important in a query rewrite to determine whether or not a materialized query table can be used in answering a query. In such cases, you can use informational referential constraints to declare a referential constraint to be true to allow DB2 to take advantage of the referential constraints in the query rewrite. DB2 allows the user application to enforce informational referential constraints, while it ignores the informational referential constraints for inserting, updating, deleting, and using the LOAD and CHECK DATA utilities. Thus, the overhead of DB2 enforcement of RI is avoided and more queries can qualify for automatic query rewrite using materialized query tables.

DB2 enforces referential constraints when:

- An INSERT statement is applied to a dependent table.
- An UPDATE statement is applied to a foreign key of a dependent table.
- An UPDATE statement is applied to the parent key of a parent table.
- A DELETE statement is applied to a parent table. All affected referential constraints and all delete rules of all affected relationships must be satisfied in order for the delete operation to succeed.
- The LOAD utility with the ENFORCE CONSTRAINTS option is run on a dependent table.
- The CHECK DATA utility is run.

The order in which referential constraints are enforced is undefined. To ensure that the order does not affect the result of the operation, there are restrictions on the definition of delete rules and on the use of certain statements. The restrictions are specified in the descriptions of the SQL statements CREATE TABLE, ALTER TABLE, INSERT, UPDATE, and DELETE.

## 3.2 RI in the relational model

RI has been defined rigorously in numerous documents, articles, and papers. In this redbook, we define RI using examples to explain the concept and terminology involved.

RI is a method, a database concept that ensures consistency between table relationships. These relationships are based on the definition of a primary key and a foreign key.

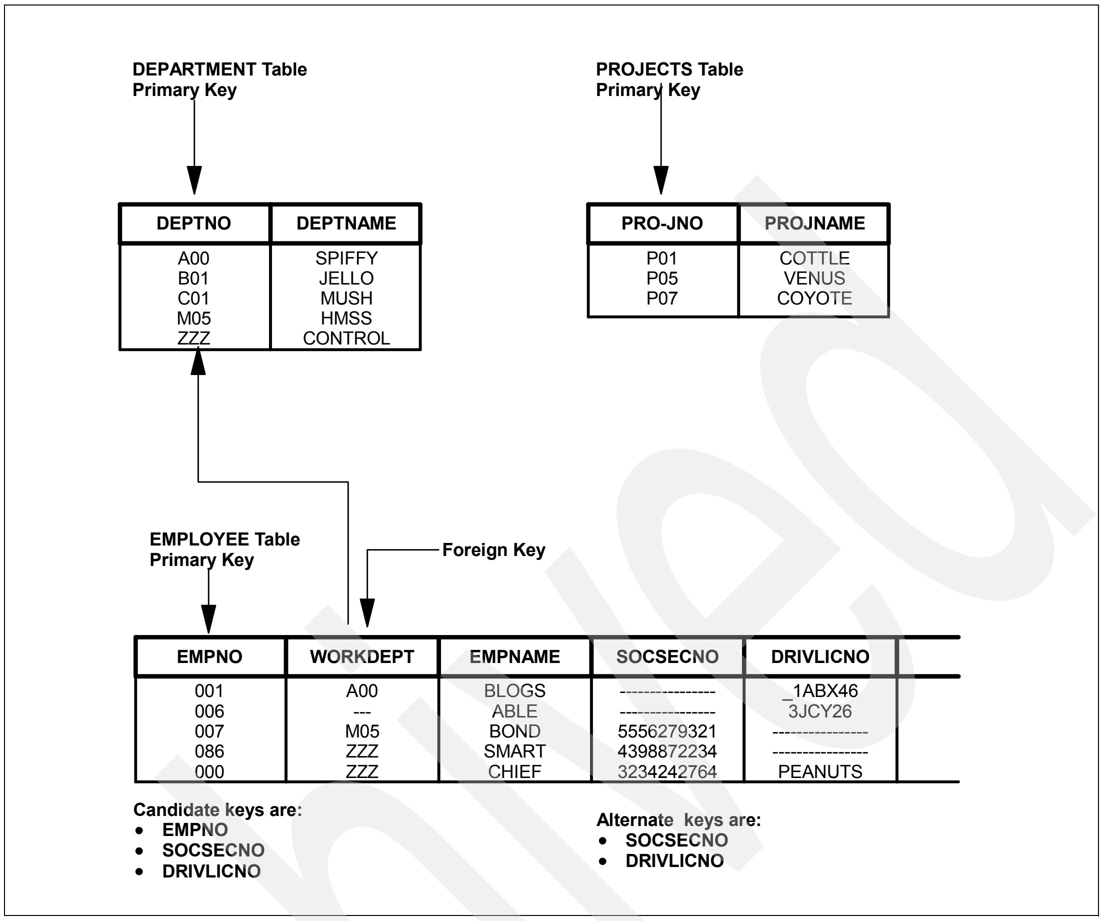
Several terms are used to explain the concept of RI, the most important are:

- Entity integrity
- Candidate keys
- Unique key
- Primary key
- Alternate key
- Foreign key
- Parent key
- Parent table
- Dependent table
- Independent table
- Parent row
- Dependent row
- Referentia constraint

We define these terms in this section.

### 3.2.1 RI concepts

Figure 3-1 on page 52 illustrates concepts behind RI and the use of RI terminology.



| DEPTNO | DEPTNAME |
|---|---|
| A00 | SPIFFY |
| B01 | JELLO |
| C01 | MUSH |
| M05 | HMSS |
| ZZZ | CONTROL |

| PRO-JNO | PROJNAME |
|---|---|
| P01 | COTTLE |
| P05 | VENUS |
| P07 | COYOTE |

| EMPNO | WORKDEPT | EMPNAME | SOCSECNO | DRIVLICNO |
|---|---|---|---|---|
| 001 | A00 | BLOGS | ---------------- | _1ABX46 |
| 006 | --- | ABLE | ---------------- | 3JCY26 |
| 007 | M05 | BOND | 5556279321 | ---------------- |
| 086 | ZZZ | SMART | 4398872234 | ---------------- |
| 000 | ZZZ | CHIEF | 3234242764 | PEANUTS |

**Candidate keys are:**
- **EMPNO**
- **SOCSECNO**
- **DRIVLICNO**

**Alternate keys are:**
- **SOCSECNO**
- **DRIVLICNO**

*Figure 3-1 RI terminology*

## Candidate key

A candidate key is an attribute, or a combination of attributes, of a table that uniquely identifies a row in a table. The primary key is selected from the pool of candidate keys.

In Figure 3-1 on page 52, EMPNO, SOCSECNO, and DRIVLICNO are the candidate keys in the EMPLOYEE table, because we assume that each is unique (at least within a country and state).

However, neither SOCSECNO nor DRIVLICNO is persistent, because it is possible that an employee does not have a social security number or a driver's license when joining an organization.

## Unique keys

A unique key is a key that is constrained (by a unique index) so that no two of its values are equal. The columns of a unique key cannot contain null values. A unique key can be defined using the UNIQUE clause of the CREATE TABLE or ALTER TABLE statement. When a unique key is defined in a CREATE TABLE DB2 concepts statement, the table is marked unavailable until the unique index is created by the user. However, if the CREATE TABLE statement is processed by the schema processor, DB2 automatically creates the unique index. When a unique key is defined in an ALTER TABLE statement, a unique index must already exist on the columns of that unique key. DB2 enforces the constraint during the execution of the LOAD utility and the SQL INSERT and UPDATE statements.

## Primary key

The criteria for selecting a primary key from a pool of candidate keys should be:

- Persistence of the key. In other words, the key must always be present for a row.
- Ability to update the key. In other words, you should not be able to update the primary key to another value.

  Some people argue that you should never be able to update the primary key to another value. If the primary key value needs to be updated, they recommend that you delete the row with the old value of the primary key and reinsert a row with the new value of the primary key.

A primary key has the following characteristics:

- Table can have exactly one primary key.
- Can be made up of one or more columns of a table.
- Each column must have the NOT NULL attribute.

In the EMPLOYEE table, EMPNO is the primary key because neither SOCSECNO nor DRIVLICNO is persistent, because it is possible that an employee does not have a social security number or a driver's license when joining an organization.

EMPNO is designated as the primary key because:

- EMPNO is an employee number that is automatically assigned when a person joins an organization and cannot be NULL.
- Once an employee number is assigned, generally, it is not changed for as long as an employee remains with the organization. Therefore, it is not updated.

The primary keys for each table in Figure 3-1 on page 52 are:

- DEPTNO for the DEPARTMENT table
- EMPNO for the EMPLOYEE table

## Alternate keys

Alternate keys are the remaining list or candidate keys after excluding the primary key.

In Figure 3-1 on page 52, SOCSECNO and DRIVLICNO are alternate keys of the EMPLOYEE table.

## Foreign key

A foreign key is a key that is specified in the definition of a referential constraint using the CREATE or ALTER statement. A foreign key refers to or is related to a specific parent key. There cannot be unmatched foreign key values. A table can have zero or more foreign keys. The value of a composite foreign key is null if any component of the value is null. A foreign key references a primary key or a unique key in the same table or another table and has the following characteristics:

- Table can have any number of foreign keys; from zero to '*n*'
- Can be made up of one or more columns of a table, but must match one for one (data type and column sequence) with the primary key it references.
- Columns may or may not be NULL. Any or all of these columns may be defined with the NULL attribute.

In Figure 3-1 on page 52, WORKDEPT is the foreign key in the EMPLOYEE table that references the primary key DEPTNO in the DEPARTMENT table.

## Parent key

A unique constraint that is referenced by the foreign key of a referential constraint is called the parent key. A parent key is either a primary key or a unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the constraint.

## Parent table

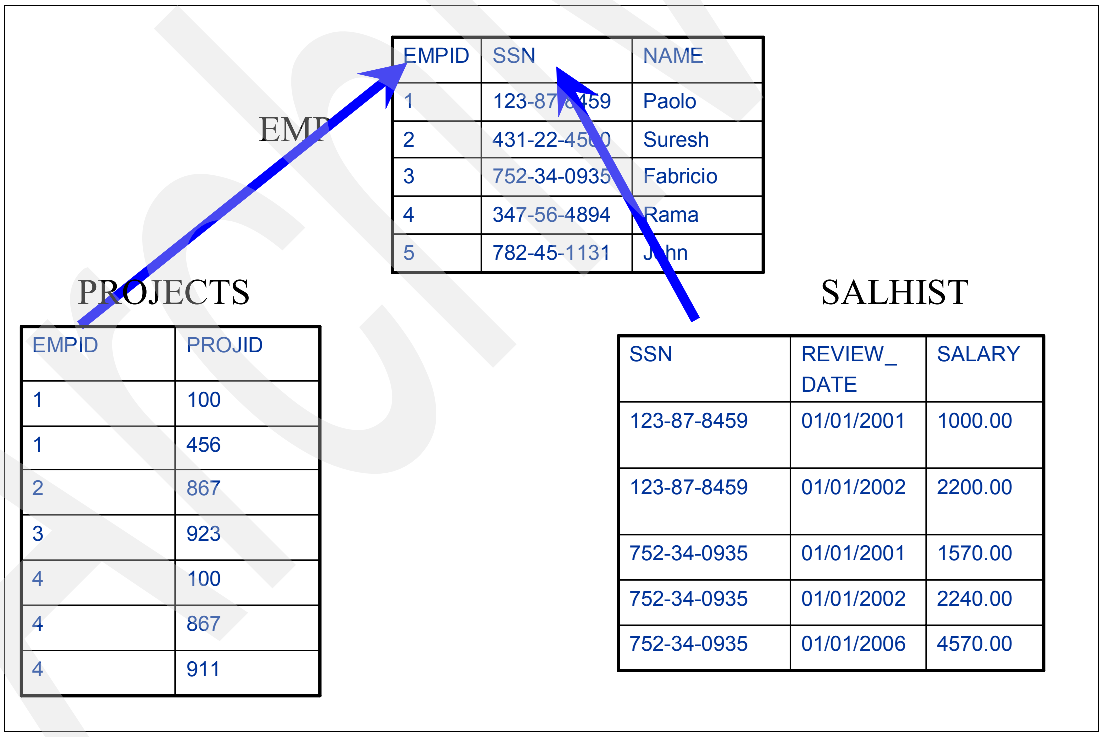
A parent table is a table that has a parent key that is referenced by one or more foreign keys in the same table or another table.

In Figure 3-1 on page 52:

- DEPARTMENT table is a parent table, because it has a primary key, DEPTNO, which is referenced by the foreign key WORKDEPT in the EMPLOYEE table.
- PROJECTS table has a primary key PROJNO, but because no foreign keys reference this table, the PROJECTS table is *not* a parent table.

A parent table sometimes is referred to as the referenced table.

Note that we are mostly considering foreign keys referencing primary key, however foreign key can also reference unique keys. Both primary and unique keys can be parent keys. In Figure 3-2, we show a simple example of foreign keys referencing both a primary and a unique key.



| EMPID | SSN | NAME |
|---|---|---|
| 1 | 123-87-8459 | Paolo |
| 2 | 431-22-4560 | Suresh |
| 3 | 752-34-0935 | Fabricio |
| 4 | 347-56-4894 | Rama |
| 5 | 782-45-1131 | John |

| EMPID | PROJID |
|---|---|
| 1 | 100 |
| 1 | 456 |
| 2 | 867 |
| 3 | 923 |
| 4 | 100 |
| 4 | 867 |
| 4 | 911 |

| SSN | REVIEW_ DATE | SALARY |
|---|---|---|
| 123-87-8459 | 01/01/2001 | 1000.00 |
| 123-87-8459 | 01/01/2002 | 2200.00 |
| 752-34-0935 | 01/01/2001 | 1570.00 |
| 752-34-0935 | 01/01/2002 | 2240.00 |
| 752-34-0935 | 01/01/2006 | 4570.00 |

*Figure 3-2 Primary and unique keys being parent keys*

## Dependent table

A dependent table is a table that has one or more foreign keys defined.

Generally, you may not have a primary key, but a dependent table can also be a parent table.

In Figure 3-1 on page 52, the EMPLOYEE table is a dependent table because it contains a foreign key WORKDEPT that references the DEPARTMENT table.

A dependent table sometimes is referred to as the referencing table.

## Independent table

An independent table is a table that is neither a parent table nor a dependent table, such as the PROJECTS table in Figure 3-1.

## Parent row

A parent row is a row of a parent table whose parent key value matches a foreign key value in the same table or in a dependent table.

A row in a parent table is not necessarily a parent row.

In Figure 3-1 on page 52:

- Rows with department names SPIFFY, HMSS, and CONTROL in the DEPARTMENT table are parent rows because they are referenced by rows BLOGS, BOND, SMART, and CHIEF in the EMPLOYEE table.
- Rows with department names JELLO and MUSH in the DEPARTMENT table are not parent rows, because they are not referenced by any row in the EMPLOYEE table.

## Dependent row

A dependent row is a row in a dependent table with at least one foreign key value that is not NULL.

The row is a dependent row of the parent row.

A row in a dependent table is not necessarily a dependent row.

Because a table can be a dependent of more than one parent table, a row in a dependent table can be a dependent row of more than one parent row in one or more parent tables.

In Figure 3-1 on page 52, rows with employee names BLOGS, BOND, SMART, and CHIEF in the EMPLOYEE table are dependent rows, because the foreign key WORKDEPT value is not NULL and references a parent row in the DEPARTMENT table. The row with employee name ABLE in the EMPLOYEE table is NOT a dependent row, because its foreign key WORKDEPT value is NULL.

## Entity integrity

The requirement that a primary key value must be UNIQUE and not NULL is known as the *Entity Integrity rule*.

In Figure 3-1 on page 52, the values of the primary keys DEPTNO in the DEPARTMENT table, PROJNO in the PROJECTS table, and EMPNO in the EMPLOYEE table obey the Entity Integrity rule.

## Referential constraint

All non-NULL values of a foreign key in a dependent table must match the values of the primary key in the parent table that the foreign key references. The time of consistency is not defined.

This requirement was originally known as Integrity of Reference or RI.

The term relationship is also used synonymously with referential constraint.

In Figure 3-1 on page 52, the constraint on the values of the foreign key WORKDEPT in the EMPLOYEE table is known as a referential constraint, because the non-NULL values of the foreign key WORKDEPT all reference a primary key DEPTNO value in the DEPARTMENT table.

#### General integrity

This extends referential integrity rules and is often confused with referential integrity. It applies to cases such as a parent in at least one of a number of tables or only unique across some tables or match on some function or combination (such as substring of a column and a hash of another).

## 3.2.2 RI rules and options

The relational model supports a set of rules that govern insert, update, and delete operations on tables. These rules are known as:

- INSERT rule
- DELETE rule
- UPDATE rule

#### INSERT rule

The INSERT rule is an implicit rule, which means that it is automatic and not explicitly specifiable on the foreign key declaration, such as the UPDATE and DELETE rules are.

This rule states that any row, inserted into a dependent table, must have its foreign key value as either:

- NULL, if NULLs are permitted for the columns in the foreign key, or
- Equal to the value of a parent key in the parent table that it references

Because there can be many foreign keys in a dependent table that can reference many parent tables, each inserted row must not violate the INSERT rule for any of the foreign keys.

In Figure 3-1 on page 52, all non-NULL values of the foreign key WORKDEPT in the EMPLOYEE table are equal to a primary key DEPTNO value in the DEPARTMENT table that it references.

#### DELETE rule

The DELETE rule is explicitly specified for each foreign key defined in a table. This rule states the requirements to be met when a row in a parent table is deleted.

The DELETE rule has three options:

- RESTRICT
  - A row of a parent table cannot be deleted if rows exist in the dependent tables with foreign key values equal to the parent key value of this row.
- CASCADE
  - All rows in the dependent tables with a foreign key value equal to the parent key value of this row also are deleted.
  - The delete is propagated to the dependents of the dependent tables.
  - If any of the deletes fail, the whole delete operation fails.

- SET NULL
  - All rows in the dependent tables with a foreign key value equal to the parent key value of this row have their foreign key values changed to NULL.

Each foreign key is associated with its own DELETE rule. All applicable DELETE rules are used to determine whether or not a delete is done. Therefore, a row in a parent table cannot be deleted if:

- One or more dependent rows have a DELETE rule of RESTRICT, or
- The deletion CASCADEs to any of its dependent rows, which are also parent rows that have dependent rows with a DELETE rule of RESTRICT.

### UPDATE rule

Like the DELETE rule, the UPDATE rule is explicitly specified for each foreign key defined in a table. This rule states the requirements to be met when the:

- Foreign key value of a row in a dependent table is updated.

  This rule is a corollary of the INSERT rule in "INSERT rule" on page 56 and is also not explicitly specifiable. This rule states that the foreign key of a row in a dependent table can be updated to a value that is NULL, if NULLs are permitted for the columns in the foreign key, or equal to the value of a parent key in the parent table that it references.

- Parent key value of a row in a parent table is updated.

Like the DELETE rule, the UPDATE rule has three options:

- RESTRICT rule

  The parent key value of a row of a parent table cannot be updated if rows in the dependent tables exist with foreign key values equal to the parent key value of this row.

- CASCADE

  If the parent key value of a row of a parent table is updated, then all rows in the dependent tables with a foreign key value equal to the parent key value of this row are also updated to the new value of the parent key.

- SET NULL

  If the parent key value of a row of a parent table is updated, then all rows in the dependent tables with a foreign key value equal to the parent key value of this row have their foreign key value updated to NULL.

Each foreign key is associated with its own UPDATE rule. All applicable UPDATE rules are used to determine whether or not the update will be done. Therefore, the parent key value of a row in a parent table cannot be updated if it has one or more dependent rows that have an UPDATE rule of RESTRICT.

## 3.3 RI in DB2

In this section, we introduce the DB2 implementation of RI by describing:

- Additional DB2 terminology
- Data definitions for RI
- Plan, package, and trigger considerations
- Maintaining RI when using data encryption
- Informational RI

## 3.3.1 Additional DB2 terminology

The DB2 implementation of RI introduces a few terms in addition to the ones described in Figure 3-1 on page 52. These terms are:

- Parent table space
- Dependent table space
- Independent table
- Independent row
- Self-referencing tables
- Self-referencing constraints
- Self-referencing row
- Delete-connected tables
- Descendent table
- Cycles
- Referential structure
- Table space set

Figure 3-3 shows the application extracted from the sample tables provided with DB2 that we used to define the additional terms. Details of the sample application, including the DDL for the tables, can be found in Appendix A. “DB2 sample tables” of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415. A more complete listing, including the index definition, is in the installation sample job DSNTEJ1.

Note that in all the figures shown in this redbook, the connecting arrows between the tables point to the dependent table from the parent table.

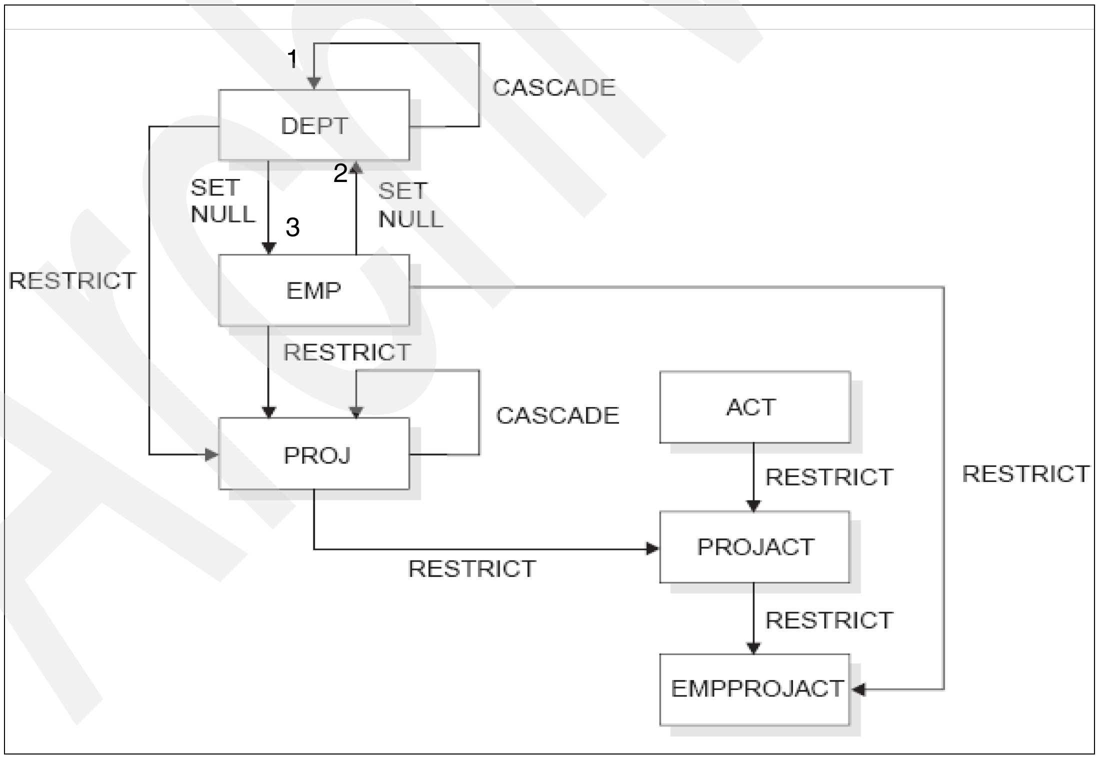


*Figure 3-3 Sample application with only DELETE rules shown*

Table 3-1 shows for each table its primary key, foreign keys if any, referenced table (parent), and referential constraint name. The numbers in Table 3-1 and Figure 3-3 help to map the defined constraint.

Table 3-1 Keys and constraints

| Table name | Primary key | Foreign key | Referenced table | Constraint name |
|---|---|---|---|---|
| DEPT | DEPTNO | ADMRDEPT | DEPT | ADMRDEPT (1) |
| | | MGRNO | EMP | MGRNO (2) |
| EMP | EMPNO | WORKDEPT | DEPT | WORKDEPT (3) |
| PROJ | PROJNO | DEPTNO | DEPT | DEPTNO |
| | | RESPEMP | EMP | RESPEMP |
| | | MAJPROJ | PROJ | MAJPROJ |
| ACT | ACTNO | - | - | - |
| PROJACT | PROJNO | PROJNO | PROJ | PROJNO |
| | ACTNO | ACTNO | ACT | ACTNO |
| | ACSTDATE | | | |
| EMPPROJACT | EMPNO | EMPNO | EMP | EMPNO |
| | PROJNO | PROJNO | PROJACT | PROJNO |
| | ACTNO | ACTNO | PROJACT | PROJNO |
| | EMSTDATE | EMSTDATE | PROJACT | PROJNO |

### Parent table space
A table space containing a parent table.

### Dependent table space
A table space containing a dependent table.

### Independent table
A table that is neither a parent table nor a dependent table.

### Independent row
All rows in an Independent table. In addition, it can be a row in a parent table or dependent table that is neither a parent row nor a dependent row.

### Self-referencing tables
A table that is a parent and a dependent in the same relationship.

In Figure 3-3, tables DEPT and PROJ are self-referencing tables.

### Self-referencing constraints
The referential constraint of a self-referencing table.

In Figure 3-3, the CASCADE delete rules on tables DEPT and PROJ are the self-referencing constraints.

## Self-referencing row

A row of a self-referencing table in which one of the foreign key values in the row is the same as the primary key value of itself.

## Delete-connected tables

A table T2 is said to be "delete-connected" to another table T1 if a delete of rows in table T1 can involve table T2. The following conditions determine whether tables are considered delete-connected:

- A self-referencing table is delete-connected to itself.

  In Figure 3-3, tables DEPT and PROJ are delete-connected to themselves.

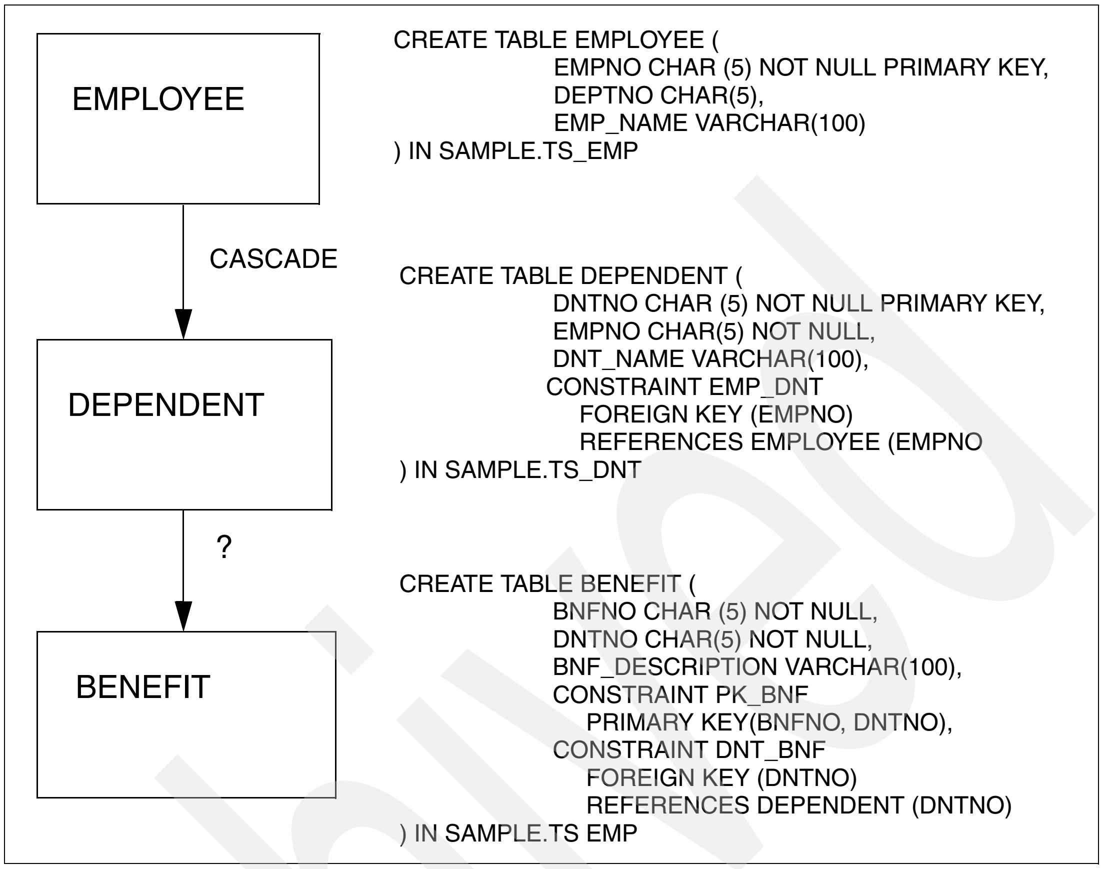
- Dependent tables are always delete-connected to their parents irrespective of the delete rule. In Figure 3-3:

  – Table EMP is delete-connected to table DEPT.
  – Table DEPT is delete-connected to table EMP and to itself.
  – Table PROJ is delete-connected to tables EMP and DEPT and to itself.
  – Table PROJACT is delete-connected to tables PROJ and ACT.
  – Table EMPPROJACT is delete-connected to tables PROJACT and EMP.

- A table is delete-connected to its grandparents and great grandparents when the delete rules between the parent, grandparents, and great grandparents use the CASCADE option.

  Figure 3-3 does not have an example or this. An example of a table that is delete-connected to its grandparent is shown in Figure 3-4.

  Table BENEFIT is delete-connected to its grandparent, table EMPLOYEE, because the delete rule on the table EMPLOYEE and DEPENDENT relationship is CASCADE. The delete rule between tables DEPENDENT and BENEFIT is irrelevant. The definitions of the required unique indexes for tables EMPLOYEE and DEPENDENT are not shown.



```
CREATE TABLE EMPLOYEE (
                    EMPNO CHAR (5) NOT NULL PRIMARY KEY,
                    DEPTNO CHAR(5),
                    EMP_NAME VARCHAR(100)
) IN SAMPLE.TS_EMP

CREATE TABLE DEPENDENT (
                    DNTNO CHAR (5) NOT NULL PRIMARY KEY,
                    EMPNO CHAR(5) NOT NULL,
                    DNT_NAME VARCHAR(100),
                   CONSTRAINT EMP_DNT
                       FOREIGN KEY (EMPNO)
                       REFERENCES EMPLOYEE (EMPNO
) IN SAMPLE.TS_DNT

CREATE TABLE BENEFIT (
                    BNFNO CHAR (5) NOT NULL,
                    DNTNO CHAR(5) NOT NULL,
                    BNF_DESCRIPTION VARCHAR(100),
                    CONSTRAINT PK_BNF
                        PRIMARY KEY(BNFNO, DNTNO),
                    CONSTRAINT DNT_BNF
                        FOREIGN KEY (DNTNO)
                        REFERENCES DEPENDENT (DNTNO)
) IN SAMPLE.TS EMP
```

*Figure 3-4 Delete-connected table*

A table is not delete-connected to its grandparents when the delete rule between the parent and grandparents is RESTRICT or SET NULL.

In Figure 3-3 on page 58:

- Table PROJACT is not delete-connected to tables EMP or DEPT because of the RESTRICT delete rules on the EMP-PROJ relationship and the DEPT-PROJ relationship.
- Table EMPPROJACT is not delete-connected to tables PROJ, ACT, or DEPT because EMP is a parent of PROJ just as DEPT is a parent of PROJ.

## Descendent table

A table T*n* is a descendent of another table T1, if it is a dependent of a dependent of table T1. The intervening DELETE, UPDATE, and implicit rules are not relevant to the definition of a descendent table. A table can be a dependent of itself and also a descendent.

In Figure 3-3 on page 58:

- Table PROJ is a descendent and dependent of table DEPT.
- Table PROJACT is a descendent of tables EMP and DEPT.
- Table EMPPROJACT is a descendent of tables PROJ, EMP. DEPT, and ACT.
- Table PROJ is a dependent and descendent of table PROJ.
- Table DEPT is a dependent and descendent of table DEPT.

### Cycle

A cycle is a path that connects a table to itself. The arrows in the path should all flow in the same direction. The intervening DELETE, UPDATE, and implicit rules are irrelevant.

Three cycles are shown in the sample application in Figure 3-3 on page 58:

- Two of these cycles, DEPT and PROJ, are the results of self-referencing constraints.
- The other one is the cycle between DEPT and EMP.

if the arrow for the relationship between DEPT and PROJ were in the opposite direction, then there would be a third cycle for DEPT, a cycle for EMP, and a second cycle for PROJ.

### Referential structure

A referential structure is a set of tables and relationships in which each table in the set is a parent or dependent of itself or some other table in the set. A referential structure contains descendents when there are more than two tables in the structure. Every table that is a parent or dependent is part of exactly one referential structure. Figure 3-3 on page 58 represents a single referential structure. Referential structures do not have names and are not referenced in any statement. The concept is introduced to help explain terms that are used in the rules of RI.

### Table space set

A table space set is the set of table spaces that contains the tables of a single referential structure. If two tables that belong to different referential structures share the same table space, the table space set is the union of the table spaces that are part of the two referential structures.

From the perspective of DB2 utilities, a table space set is a set of table spaces and partitions that should be recovered together for one or both of these reasons:

- Each of them contains a table that is a parent or descendent of a table in one of the others.
- The set contains a base table and associated auxiliary tables.

Table space sets do not have names, however, because a table space cannot be a member of more than one table space set. A table space set is uniquely identified by identifying any member of the set. The only way that DB2 identifies the table spaces and tables within a table space set is to specify it in the control statement of the REPORT utility, as described in "REPORT utility" on page 87. Table space sets also have these characteristics:

- Like referential structures, table space sets may include cycles. A table space cycle exists when a table space contains tables that are descendents of itself. This can affect the use of the LOAD utility as explained in "Loading tables involved in cycles" on page 94.
- A table space set is a unit of consistency with respect to RI as described in 6.7, "DB2 subsystem restart after abend" on page 211.

## 3.3.2 Data definitions for RI

This section describes:

- DDL extensions
- Restrictions of defining referential structures and recommendations about circumventing these restrictions

## DDL extensions

The DDL extensions to DB2 for RI support define:

- Primary key
- Foreign keys and the associated DELETE rules
- Foreign keys and the associated UPDATE rules

The rules that govern these definitions are documented along with some considerations and examples of their use. The sample application, shown in Figure 3-3 on page 58, is used in some of the examples.

DB2 can dynamically add and drop a primary key and referential constraints on tables without dropping and recreating the tables.

The ALTER TABLE SQL statement has been enhanced to provide this support. The syntax for both the CREATE and ALTER statements is not provided here but can be found in *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

## Primary key

The primary key in DB2 has the following characteristics:

- It is optional. A dependent or independent table may or may not have a primary key defined. A table can have only one primary key.
- It obeys the same rule as index keys (except for the NOT NULL attribute).

  Therefore:

  – The key can include no more than 64 columns.
  – No column can be named twice.
  – The sum of the column length attributes cannot be greater than 2000.
- The columns that form part of a primary key must be defined as NOT NULL or NOT NULL WITH DEFAULT.
- A primary key can have more columns than are necessary to achieve uniqueness, but this is not generally recommended because these additional columns would have to be reflected in the foreign key of the dependent tables.
- A unique index must be defined on ONLY the columns of the primary key. Also, the sequence of columns in this index must be the same as that of the primary key definition. This unique index will be the primary index.

A primary key may be defined in a CREATE TABLE statement or an ALTER TABLE statement. Primary key characteristics are:

- If the primary key is defined in the CREATE TABLE statement, the table is said to have an incomplete definition until the unique index has been defined on the columns of the primary key. DB2 catalog table SYSIBM.SYSTABLES shows in the STATUS column “I” for incomplete definitions and the column TABLESTATUS shows the reason with the value “P” (the table lacks a primary index). Also the UNIQUERULE column value of SYSIBM.SYSINDEXES has the value “P”, which represents the primary index.

  Use of a table with an incomplete definition is severely restricted: you can drop the table, create the primary index, and drop or create other indexes; you cannot load the table, insert data, retrieve data, update data, delete data, or create foreign keys that reference the primary key.

  The primary index should generally be created as soon as the table is created.

- If a primary key is defined using ALTER TABLE:
  - The unique index must already exist.

    If more than one unique index is on those columns, DB2 chooses one arbitrarily to be the primary index.
  - In some cases, plans, packages, and triggers referencing that table are invalidated. Check the “Plan, package, and trigger considerations” on page 69 for more details.
- A table can have only one primary index.
- If a table has more than one unique index on the columns of the primary key, then only one of those indexes is a primary index.

  If a primary key was defined in a CREATE TABLE statement, the primary index is the first unique index that was created on the columns of that primary key and in the same sequence as the primary key columns.
- The primary index may be ascending or descending.
- A table is defined as a parent table if it contains a parent key, which can be a primary index or a unique index.

  ALTER or REFERENCES authority on the parent table must be granted to all users who need to define that table as a parent in a foreign key.

  Note: The REFERENCES privilege does not replace the ALTER privilege. It was added to conform to the SQL standard. To define a foreign key that references a parent table, you must have either the REFERENCES or the ALTER privilege, or both.

The impact of dropping a primary key or primary index is:

- When the definition of the primary key of a table is dropped using the ALTER statement, all referential constraints in which that table is a parent also are dropped. However, the primary index is not dropped and it remains as a unique index. The UNIQUERULE column value of SYSIBM.SYSINDEXES is changed to ‘U’ to indicate that it is now only a unique index.

  In Figure 3-3 on page 58, dropping the primary key of the DEPT table will:
  - Drop the referential constraints ADMRDEPT, WORKDEPT, and DEPTNO
  - Designate the index on column DEPTNO of the DEPT table as a unique index in SYSIBM.SYSINDEXES
- The ALTER privilege is required on all the dependent tables in order to drop a primary key from a parent table. The ALTER authority cannot be REVOKEd from a user who has already created a foreign key referencing this table based on the authority granted to him earlier.
- If a primary index is dropped, the definition of its table is changed to incomplete, irrespective of whether another unique index exists on the columns of the primary key.

  The only way to complete the definition of the table again is to create another unique index on the columns of the primary key. Note that dropping the primary index does *not* cause the referential constraints to be dropped.

  In Figure 3-3 on page 58, dropping the primary index on column DEPTNO of the DEPT table will:
  - *Not* drop the referential constraints ADMRDEPT, WORKDEPT, and DEPTNO
  - Put the DEPT table into an incomplete status.

Some considerations in defining primary keys are:

- Each entity should have a primary key according to relational theory. While this rule is not enforced in DB2, the recommendation is to create a primary key for every table that data analysis has identified as having an entity key.

  However, because a primary key requires a unique index to be defined, the index maintenance overhead associated with defining a primary key may not be acceptable.

  In the case of private data, it seems unnecessary to define a primary key for the tables if no referential constraints are defined.
- Choose a primary key that cannot be updated. Besides avoiding the UPDATE rule restrictions, it enforces the good practice of having unique identifiers that remain the same for the lifetime of the entity occurrence.
- NOT NULL WITH DEFAULT generally is not recommended for primary key columns unless they have the TIMESTAMP attribute.
- The IDENTITY clause is allowed in the columns of a primary key. However, the CYCLE and MAXVALUE options can restrict the total rows of the tables. Once the MAXVALUE is reached, DB2 recycles the identity column and it is a duplicate column. Choose the minimum number of columns to ensure uniqueness of the primary key.

  More than the minimum number may be preferred for performance reasons. Using the “data in index” technique, an access to the index can retrieve all the desired information without accessing the data. This also incurs the overhead of more columns in the foreign keys of the dependent tables.
- A view that can be updated that is defined on a table with a primary key should include all columns of the key. Although this is necessary only if the view is used for inserts, the unique identification of rows can be useful if the view is used for update, delete, or select.

Example 3-1 shows examples of creating and dropping primary keys for some of the tables in the sample application in Figure 3-3 on page 58.

*Example 3-1 CREATE and ALTER statements with primary keys*

```
Create primary' key on table EMP

   CREATE TABLE EMP
   (PRIMARY KEY (EMPNO),
   EMPNO CHAR(6) NOT NULL,
   ...
The definition of the table is incomplete until the unique index is defined.

The primary key requires a unique index to be defined on it

   CREATE UNIQUE INDEX XEMPI
   ON EMP(EMPNO)
   ...
This makes the definition of the EMP table complete.

A primary key also can be defined for table DEPT by the ALTER TABLE statement, after the
table has been created.

   ALTER TABLE DEPT
     ADD PRIMARY KEY (DEPTNO)

A unique index on DEPTNO must already exist for this statement to execute successfully.

ALTER TABLE also can be used to drop the definition of the primary key of table DEPT

   ALTER TABLE DSN8210.DEPT
   DROP PRIMARY KEY
```

For more considerations about primary keys, see 1.1.4 “Key design” and 2.6.5 “Column sequence in a multi-column index” of *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134.

## Foreign keys

Foreign keys in DB2 have the following characteristics:

- A table can have any number of foreign keys defined; from zero to ‘*n*’.

  The DELETE rule is specified as part of the foreign key definition. The UPDATE rule cannot be specified, because the UPDATE RESTRICT rule is only supported implicitly.
- Can be NULL. A multi-column foreign key is considered NULL if at least one of the columns is NULL.
- The columns of a foreign key can be part of the columns of the primary key of this table.
- A column in a table can be a part of more than one foreign key.
- Each foreign key relationship defined can be assigned a user-defined constraint name. This constraint name must be unique for all referential, check, primary key, or unique key constraints previously specified on a particular table. Figure 3-3 on page 58 shows the constraint names of the referential constraints defined in the sample application.

  Every referential constraint is uniquely identified by the combination of a table name and a constraint name.

  Referential constraints are identified by name in:

  – Error messages, and
  – DROP FOREIGN KEY clause of the ALTER TABLE statement

A foreign key may be defined in a CREATE TABLE statement or an ALTER TABLE statement. To create a foreign key on a table referencing a parent table, the following conditions must be satisfied:

- The parent table must not be a DB2 catalog table.
- The parent table must exist.

  Therefore, some referential constraints can only be defined by the ALTER TABLE statement because the parent table identified in a FOREIGN KEY clause must already be described in the DB2 catalog. For example, a self-referencing constraint must be defined by a CREATE TABLE statement followed by an ALTER TABLE statement to add the referential constraint.
- The definition of the parent table must be complete. It must have a unique index, which is defined on the columns, that forms the primary key. Thus, the existing restrictions on index keys also apply by implication to foreign keys.
- The creator of the foreign key must have ALTER authority on the parent tables.
- The foreign key must have the:

  – Same number of columns as the primary key of the parent table, and

  – Description of the *n*th column of the foreign key must be identical to the description of the *n*th column of the primary key.

  However, the column names, default values, and null attributes may be different. If an index is defined on the foreign key columns, the index columns may be ascending or descending, which may be different from the ascending or descending attribute of the corresponding primary index.
- A foreign key cannot reference a view.

- The DELETE rules can be specified for each foreign key definition as RESTRICT, CASCADE, SET NULL, or NO ACTION. The SET NULL option can be specified only if at least one of the columns of the foreign key is NULLable.

  If the value of the register is “DB2”, the delete rule defaults to RESTRICT; if the value is “SQL”, the delete rule defaults to NO ACTION.

  The DELETE rules apply only to the DML, *not* to the DDL. Therefore, a DROP TABLE statement is not influenced by the DELETE rules of the foreign key.
- When ALTER TABLE is used to add a foreign key to a table:
  - If the table is populated, its table space is put into CHECK-pending.

    See “Resetting the CHECK-pending status” on page 90 for more details.
  - In some cases, plans, packages, and triggers referencing that table are invalidated. Check the “Plan, package, and trigger considerations” on page 69 for more details.

The impact of dropping foreign keys is:

- When all the foreign keys in a dependent table are dropped, any CHECK-pending status on the table space in which the table resides is reset. This assumes that the table space only contains one table. If any other table resides in the table space, the CHECK-pending status on the table space is not reset.
- To DROP FOREIGN KEY, the ALTER authority is required on both the table containing the foreign key and the parent table that it is referencing.

Some considerations involving foreign keys are:

- When a row is inserted into a dependent table, each non-null foreign key insert value must match some value of the primary key of the parent table of the relationship.

  For multi-column foreign keys, the INSERT rule is verified only when all the columns are not NULL. If one of the columns is NULL, DB2 does not verify the values in the other columns.

  If any referential constraint is violated by an INSERT operation, an error occurs and no rows are inserted.
- When any column of a foreign key is updated, and the new value of the foreign key is not NULL, the new value of the foreign key must match some value of the primary key in the parent table of the relationship. If any referential constraint is violated by an UPDATE operation, an error occurs and no rows are updated.
- A self-referencing row is never a violation of a self-referencing constraint. The following rules apply to the processing of self-referencing rows:
  - They can be inserted.
  - They can be deleted.
  - The primary key value cannot be updated. If you want to update the primary key, then you must write at least two SQL calls to:
    - Delete the row and reinsert it with the new primary and foreign key values, or
    - Update the foreign key value to another value or NULL (if permitted), and then update the primary key value.
  - The foreign key value can be updated only if it references a valid primary key value.
- INSERT and UPDATE operations using a view are subject to the same referential constraints as its base table. Likewise, if the base table of a view is a parent, DELETE operations using that view are subject to the same rules as DELETE operations on that base table.
- A referential constraint is implicitly dropped when the:

  – Parent or dependent table of the relationship is dropped.
  – Definition of the primary key of the parent table is dropped.

- SET NULL must not be specified if any nullable column of the foreign key is a column of the key of a partitioning index.

Example 3-2 shows the statement for creating and dropping foreign keys for some of the tables of the sample application.

*Example 3-2 CREATE, ALTER, and DROP statements with foreign keys*

```
WORKDEPT column of table EMP is defined as a foreign key whose parent table is DEPT.
   CREATE TABLE EMP(
      EMPNO    CHAR(6) NOT NULL,
      WORKDEPT CHAR(5),
      PRIMARY KEY (EMPNO),
      FOREIGN KEY (WORKDEPT) REFERENCES DEPT ON DELETE SET NULL
      )
   DELETE rule of this foreign key constraint is SET NULL
   Referential constraints also can be defined by the ALTER table statement after a table
   is created:
   ALTER TABLE EMP
     ADD FOREIGN KEY (WORKDEPT) REFERENCES DEPT ON DELETE SET NULL;

Creating the self-referencing constraint on the DEPT table
   1) CREATE TABLE DEPT(
      DEPTNO   CHAR(3) NOT NULL,
      ADMRDEPT  CHAR(3),
      PRIMARY KEY (DEPTNO)
      )
   2) CREATE UNIQUE INDEX IXDEPT
       ON DEPT
      (DEPTNO                 ASC) ...
   3) ALTER TABLE DEPT
        ADD FOREIGN KEY (ADMRDEPT) REFERENCES DEPT ON DELETE CASCADE;

Dropping foreign key of EMP table
   ALTER TABLE EMP
     DROP FOREIGN KEY WORKDEPT
```

### *Indexes on foreign keys*

An index on a foreign key is not required but is strongly recommended if rows of the parent table are often deleted. The validity of the delete statement and its possible effect on the dependent table, can be checked through the index.

To let an index on the foreign key be used on the dependent table for a delete operation on a parent table, the columns of the index on the foreign key must be identical to and in the same order as the columns in the foreign key.

A foreign key can also be the primary key; then, the primary index is also a unique index on the foreign key. In that case, every row of the parent table has at most one dependent row. The dependent table might be used to hold information that pertains to only a few of the occurrences of the entity described by the parent table.

The primary key can share columns of the foreign key if the first *n* columns of the foreign key are the same as the primary key's columns.

## 3.3.3 Plan, package, and trigger considerations

Adding or dropping referential constraints can change the validation of plans, packages, and trigger packages.

For these tests, some programs were created and packages and plans bound with DBRMs.

### Programs

- PGM001:

```
SELECT * FROM EMP
```

- PGM002:

```
SELECT * FROM EMP  DELETE FROM DEPT WHERE EMP = :HOST
```

- PGM003:

```
SELECT * FROM DEPT
```

- PGM004:

```
SELECT * FROM DEPT DELETE FROM EMP WHERE DEPT := HOST
```

### Packages

- PKG001, PKG002, PKG003, PGK004:

```
BIND PACKAGE(PKG001) MEMBER(PGM001) ...
BIND PACKAGE(PKG002) MEMBER(PGM002) ...
BIND PACKAGE(PKG003) MEMBER(PGM003) ...
BIND PACKAGE(PKG004) MEMBER(PGM004) ...
```

### Plans

- PLN001:

```
BIND PLAN(PLN001) MEMBER(PLN001) ...
BIND PLAN(PLN002) MEMBER(PLN002) ...
BIND PLAN(PLN003) MEMBER(PLN003) ...
BIND PLAN(PLN004) MEMBER(PLN004) ...
```

### Trigger packages

- INS_EMP:

```
CREATE TRIGGER INS_EMP
  AFTER INSERT ON EMP
  REFERENCING NEW AS NEMP
  FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
    UPDATE DEPT
      SET TOT_EMP= TOT_EMP + 1
        WHERE DEPT.DEPT = NEMP.DEPT ;
    END ?
```

Table 3-2 shows what can happen with plans.

*Table 3-2 Plan validation adding primary key and foreign key*

| Plan | ADD PK EMP | ADD PK DEPT | FK DEPT CASCADE | FK DEPT RESTRICT | FK DEPT SET NULL | FK DEPT NO ACTION |
|---|---|---|---|---|---|---|
| PLN001 | H | Y | A | A | A | A |
| PLN002 | H | H | N | A | N | A |

| Plan | ADD PK EMP | ADD PK DEPT | FK DEPT CASCADE | FK DEPT RESTRICT | FK DEPT SET NULL | FK DEPT NO ACTION |
|---|---|---|---|---|---|---|
| PLN003 | Y | H | N | Y | N | Y |
| PLN004 | H | H | N | A | N | A |

Notes:
Y: Plan is valid.
N: Plan is invalid.
A: Table descriptions changed, but plan is still valid.
H: Table descriptions changed. Plan is invalid for DB2 releases prior to V5.

Table 3-3 shows the packages' behavior.

*Table 3-3 Package validation adding primary key and foreign key*

| Package | ADD PK EMP | ADD PK DEPT | FK DEPT CASCADE | FK DEPT RESTRICT | FK DEPT SET NULL | FK DEPT NO ACTION |
|---|---|---|---|---|---|---|
| PKG001 | H | Y | A | A | A | A |
| PKG002 | H | H | N | A | N | A |
| PKG003 | Y | H | N | Y | N | Y |
| PKG004 | H | H | N | A | N | A |
| INS_EMP | Y | H | N | Y | N | Y |

Notes:
Y: Package is valid.
N: Package is invalid.
A: Table descriptions changed, but package is still valid.
H: Table descriptions changed. Package is invalid for DB2 releases prior to V5.

### 3.3.4 Maintaining RI when using data encryption

If you use encrypted data in a referential constraint, the primary key of the parent table and the foreign key of the dependent table must have the same encrypted value. The encrypted value should be extracted from the parent table (the primary key) and used for the dependent table (the foreign key). You can do this in one of the following two ways:

- Use the FINAL TABLE (INSERT statement) clause on a SELECT statement.
- Use the ENCRYPT_TDES function to encrypt the foreign key using the same password as the primary key. The encrypted value of the foreign key will be the same as the encrypted value of the primary key.

Similar considerations apply for the self-referencing row scenario.

The SET ENCRYPTION PASSWORD statement sets the password to use for the ENCRYPT_TDES function. See *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426, for more information about the SET ENCRYPTION PASSWORD statement and the ENCRYPT_TDES statement.

### 3.3.5 Informational referential constraint

An informational referential constraint is a referential constraint that is not enforced by DB2 during normal operations. DB2 ignores informational referential constraints during insert, update, and delete operations. Some utilities ignore these constraints; other utilities recognize them. For example, CHECK DATA and LOAD ignore these constraints. QUIESCE TABLESPACESET recognizes these constraints by quiescing all table spaces related to the specified table space set whether through enforced RI or informational RI.

You should use this type of referential constraint only when an application process verifies the data in a RI relationship. For example, when inserting a row in a dependent table, the application should verify that a foreign key exists as a primary or unique key in the parent table. To define an informational referential constraint, use the NOT ENFORCED option of the referential constraint definition in a CREATE TABLE or ALTER TABLE statement. For more information about the NOT ENFORCED option, see Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426.

*Example 3-3 CREATE and ALTER with NOT ENFORCED*

```
Creating an informational referential constraint:
   CREATE TABLE EMP(
      EMPNO    CHAR(6) NOT NULL,
      WORKDEPT CHAR(5),
      PRIMARY KEY (EMPNO),
      FOREIGN KEY (WORKDEPT) REFERENCES DEPT NOT ENFORCED
      )
Altering a table to include an informational referential foreign key
   ALTER TABLE EMP
     ADD FOREIGN KEY (WORKDEPT) REFERENCES DEPT NOT ENFORCED;
```

## 3.4 Functional implications

In order to provide good performance and satisfy the access path independence requirements, some restrictions of RI are implemented in DB2:

- The DELETE rule of a self-referencing constraint must be CASCADE or NO ACTION.
- A cycle involving two or more tables must not cause a table to be delete-connected to itself.
- If table T1 is delete-connected to table T2 through multiple paths, those relationships in which T1 is a dependent table and which form all or parts of those paths:
  - Must have the same delete rule.
  - Also, the delete rule must not be SET NULL.
- The CASCADE, SET NULL, and NO ACTION options on the UPDATE rule are not supported.
- INSERT, DELETE, and UPDATE restrictions.

### 3.4.1 DELETE rule for self-referencing tables

The restriction is that the DELETE rule of a self-referencing constraint must be CASCADE or NO ACTION.

Figure 3-5 shows the potential problem with a DELETE rule of RESTRICT or SET NULL for self-referencing constraints.

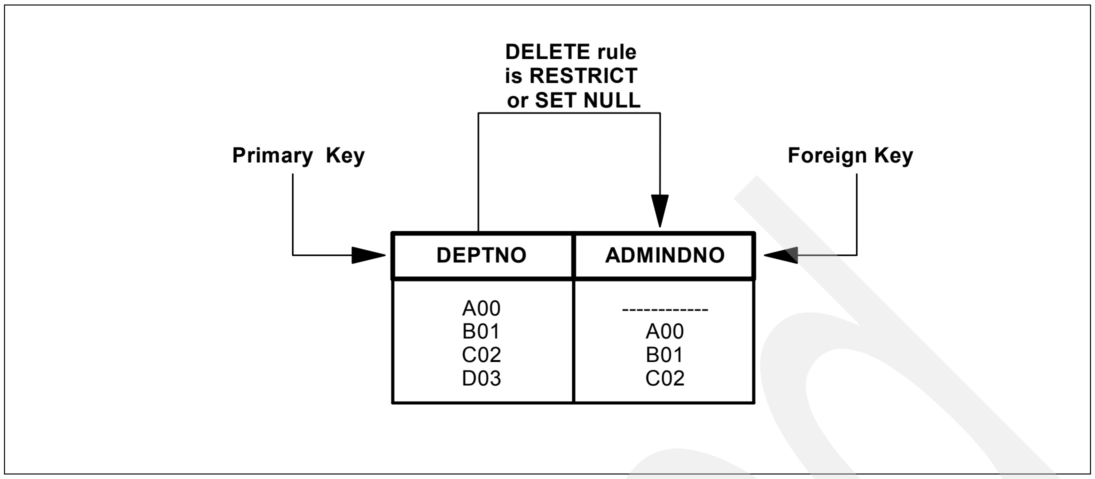


*Figure 3-5 Department table with self reference*

### 3.4.2 DELETE with RESTRICT

Statement:

```
DELETE FROM DEPT WHERE DEPTNO >= B01
```

The order in which rows are deleted is decided by the optimizer.

If the implementation deletes rows in the sequence of D03, C02, and B01, then the delete is successful. Any other sequence causes the statement to fail.

### 3.4.3 DELETE with SET NULL

Statement:

```
DELETE FROM DEPT WHERE ADMINDNO IS NULL
```

The result depends upon the sequence in which DB2 accesses table DEPARTMENT.

If DB2 accesses table DEPT in the sequence of rows D03, C02, B01, and A00, then the delete deletes only row A00.

If DB2 accesses table DEPT in the sequence of rows A00, B01, C02, and D03, then the delete deletes all rows in the table.

**Note:** The difference between NO ACTION and RESTRICT is when the referential constraint is enforced. RESTRICT (IBM SQL rules) enforces the rule immediately, and NO ACTION (SQL standard rules) enforces the rule at the end of the statement.

#### *Circumvention recommendations*

For applications that require the implementation of a DELETE RESTRICT or DELETE SET NULL type of function with self-referencing tables, two choices exist:

- Specify DELETE CASCADE rule on the relationship and implement the RESTRICT or the SET NULL rule through user code.
- Specify no rule at all and implement the RESTRICT or SET NULL rule through user code.

The recommendation is to specify the DELETE CASCADE rule on the relationship and implement the RESTRICT or SET NULL rule through user code if the following considerations are acceptable:

- DB2 can enforce the implicit insert rule when rows are being inserted.
- DB2 can enforce the update restrict rule if the application needs to implement such a function for this relationship.
- The CHECK utility can be run to verify that no rows in this table violate referential constraints, because the CHECK utility can only verify referential constraints defined in the DB2 catalog.

The following precautions must be taken when implementing the DELETE RESTRICT or SET NULL rule for this relationship through user code:

- All deletes to this table must be permitted only through user-written programs.
- Deletes should be inhibited through ad hoc environments such as QMF™ and SPUFI.
- Deletes should not be allowed to cascade to this table from other tables if the DELETE CASCADE option is chosen.
- For all programs that require the deletion of a row in this table, all deletes should be processed by a common single routine.

The appropriate SQL statements must be issued and the proper locks taken so that data integrity is not compromised.

## 3.4.4 Cycles should not cause a table to be delete-connected to itself

The restriction prevents a cycle involving two or more tables to cause a table to be delete-connected to itself. This means:

- In a two table cycle, neither delete rule can be CASCADE.
- In a cycle involving more than two tables, two or more delete rules must not be CASCADE.

Figure 3-6 shows a valid DB2 cycle structure; the arrows point from parent to dependent.

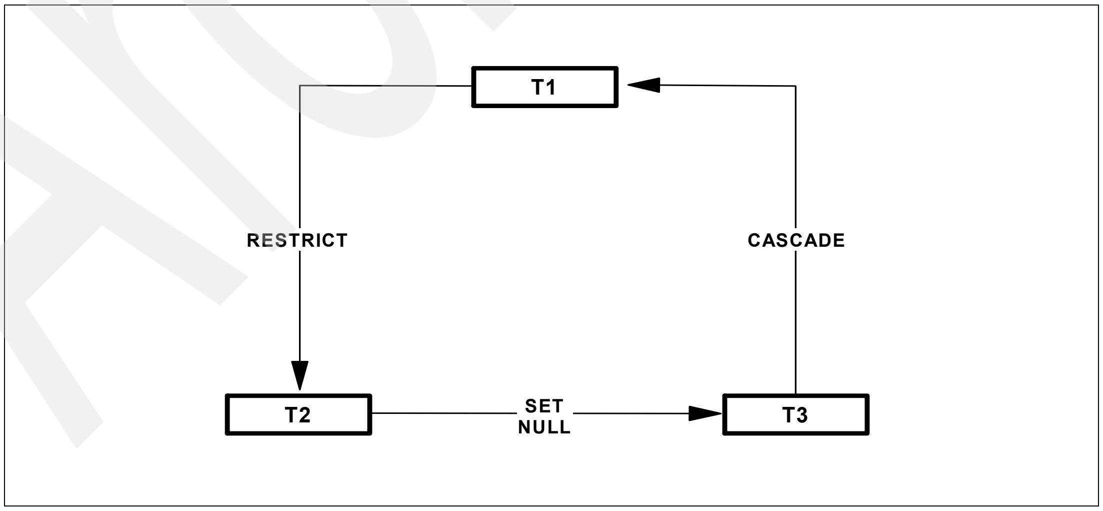


*Figure 3-6 Valid DB2 cycle structure*

Figure 3-7 shows an invalid DB2 cycle structure.

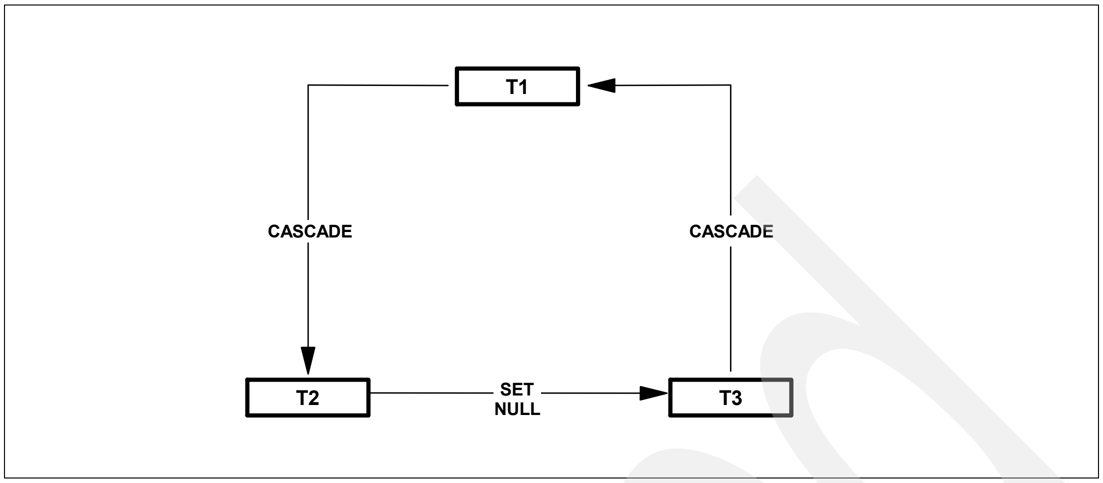


*Figure 3-7 Invalid DB2 cycle structure*

Figure 3-8 shows an example of a potential problem with a table that is delete-connected to itself.

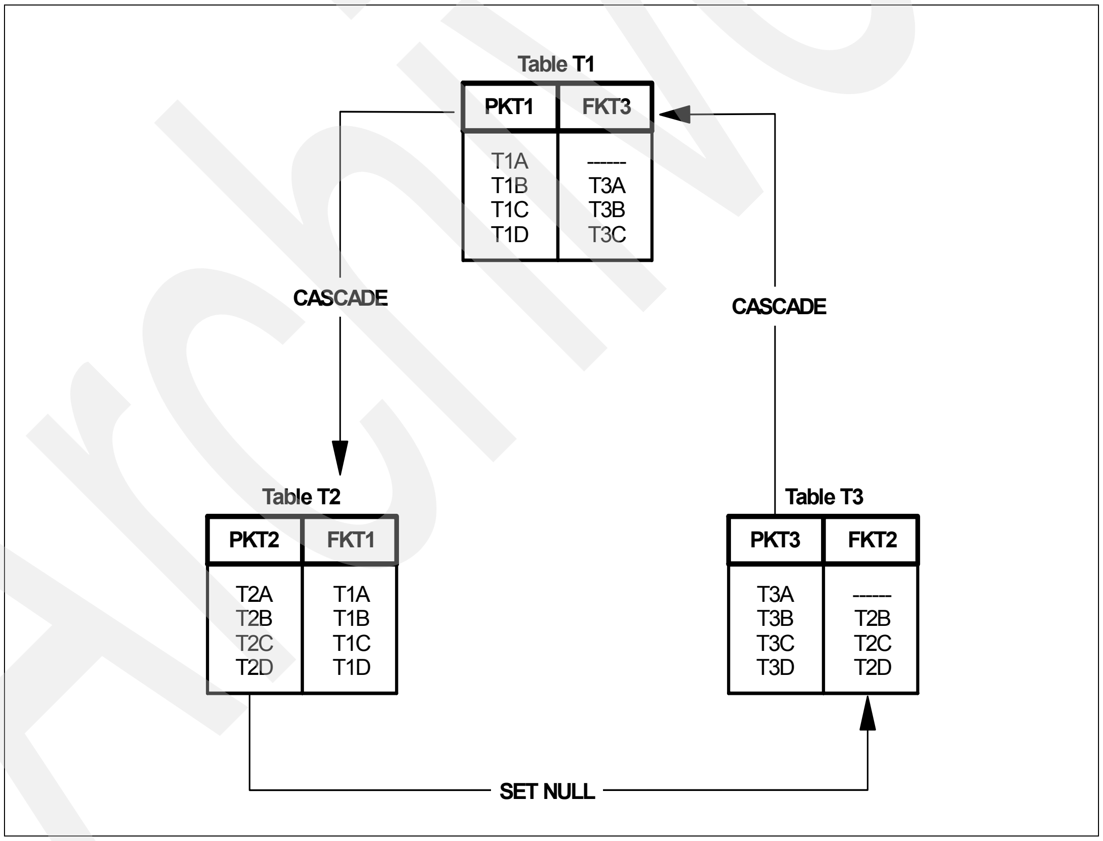


*Figure 3-8 Anomaly with table T3 that is delete-connected to itself*

The delete rule of CASCADE on the table T1 and table T2 relationship causes table T3 to be delete-connected to itself. Statement:

```
   DELETE FROM T3 WHERE FKT2 IS NULL
```

The result depends upon the sequence in which DB2 accesses table T3:

- If DB2 accesses table T3 in the sequence of rows T3D, T3C, T3B, and T3A, only row T3A would qualify for the delete.
- If DB2 accesses table T3 in the sequence of rows T3A, T3B, T3C, and T3D, all four rows T3A, T3B, T3C, and T3D would qualify for the delete.

### *Circumvention recommendations*

For any applications that cause a referential structure to be delete-connected to itself, the user must:

- Specify a DELETE RESTRICT or SET NULL rule on at least two of the relationships, so that a table does not delete-connect to itself.
- Write user code to satisfy the required functions.

Figure 3-6 on page 73 shows how a DELETE RESTRICT rule on the table T1 and T2 relationship and DELETE SET NULL on the table T2 and T3 relationship prevent table T3 from being delete-connected to itself. The recommendation is:

- Specify a DELETE RESTRICT rule rather than a SET NULL on one of the relationships. An accidental delete of a row with dependent rows will be inhibited with the RESTRICT option. The SET NULL option would cause the delete operation to complete successfully by changing the foreign key values to NULL. The DELETE SET NULL may be unacceptable, especially if deletes are permitted through a client environment.
- Simulate the DELETE CASCADE operation through user code by issuing the appropriate SQL statements and taking the proper locks so that data integrity is not compromised.

## 3.4.5 Table delete-connections through multiple paths

The restriction is that if a table T1 is delete-connected to table T2 through multiple paths, those relationships in which T1 is a dependent table and which form all or parts of those paths:

- Must have the same delete rule.
- The delete rule must not be SET NULL.

Figure 3-9 shows a valid delete-connect DB2 referential structure.

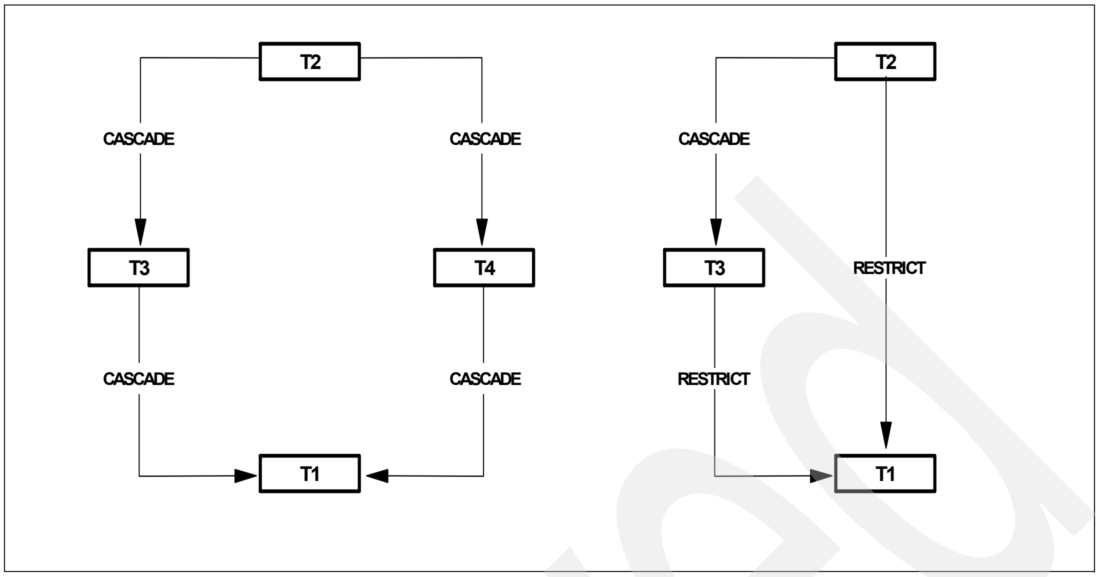


*Figure 3-9 Valid delete-connect DB2 referential structure*

Figure 3-10 shows an invalid delete-connect DB2 referential structure.

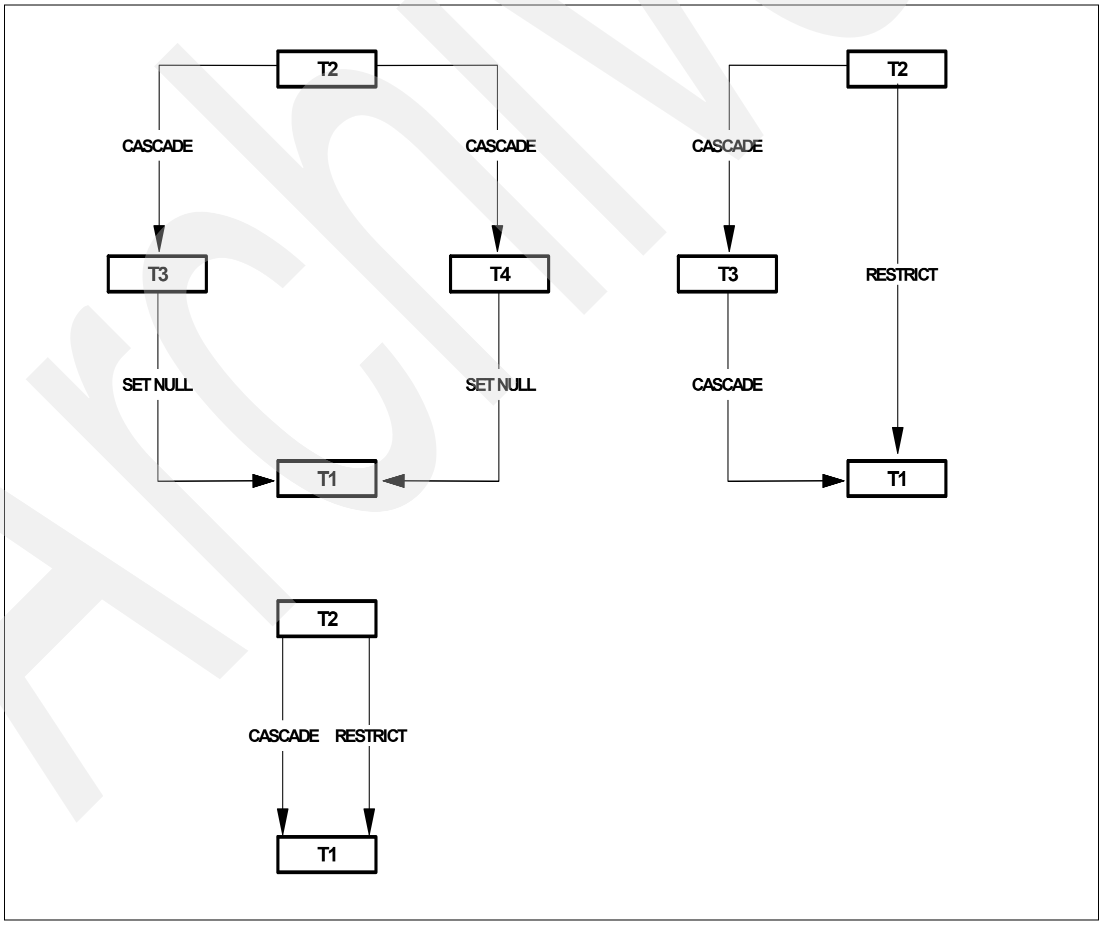


*Figure 3-10 Invalid delete-connect DB2 structure*

Figure 3-11 shows an example of a potential problem with a table that is delete-connected to another table through multiple paths with a delete rule that is not the same.

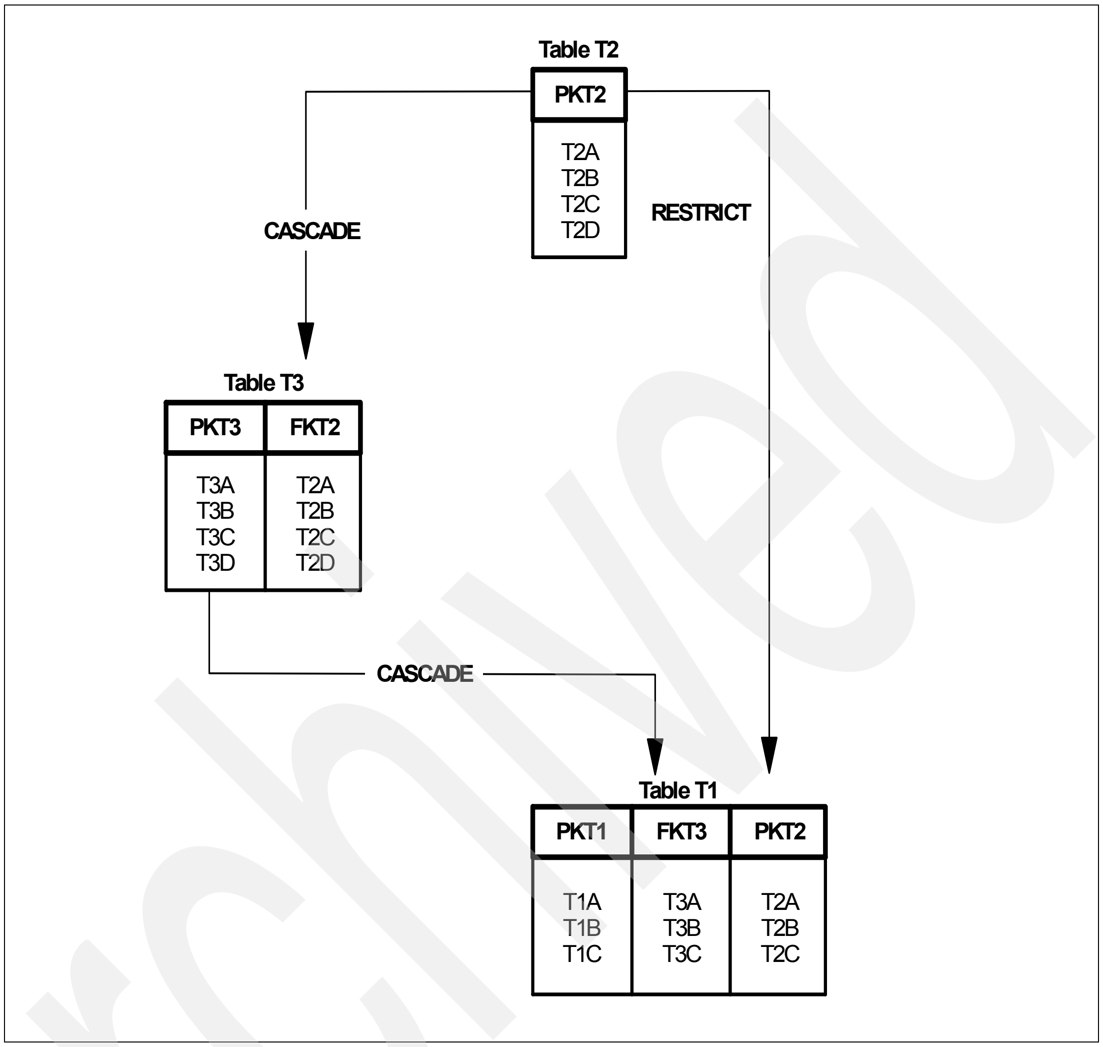


*Figure 3-11 Delete-connect with different rules*

Table T1 is delete-connected to table T2 through multiple paths with different delete rules. The delete rule of CASCADE on the table T1 and table T3 relationship is different from the delete rule of RESTRICT on the table T1 and table T2 relationship.

For the statement:

```
DELETE FROM T2 WHERE PKT2 = 'T2A'
```

The result depends upon the sequence in which DB2 accesses table T1 and T3:

- If DB2 accesses table T3 first, the SQL operation will be successful because rows T3A, T1A, and T2A will be deleted before the RESTRICT rule is checked on the table T2 and table T1 relationship.
- If DB2 accesses table T1 first, the SQL operation fails, because row T1A violates the RESTRICT rule for the table T1 and table T2 relationship.

### *Circumvention recommendations*

For any applications that cause a referential structure to violate this restriction, you must:

- Specify a DELETE RESTRICT or CASCADE rule on the relationships so that a table has the same<: delete rule when connected through multiple paths.
- Write user code to satisfy the required functions.

The recommendation is:

- Specify DELETE RESTRICT rule on both relationships, rather than CASCADE. An accidental delete of a row with dependent rows is inhibited with the RESTRICT option while the CASCADE option would cause the delete operation to complete successfully. The DELETE CASCADE option is likely to be unacceptable if deletions are permitted through an ad hoc environment, such as QMF, SPUFI, or AS.
- Simulate the DELETE CASCADE or SET NULL operation through user code by issuing the appropriate SQL statements and taking the proper locks so that data integrity is not compromised.

## 3.4.6 INSERT

If a self-referencing table is the object of an insert statement with a subquery, the subquery must not return more than one row; otherwise, the statement fails.

This violation is detected at runtime.

Figure 3-12 is used to show an example of a potential problem with a self-referencing table and an insert with subquery.

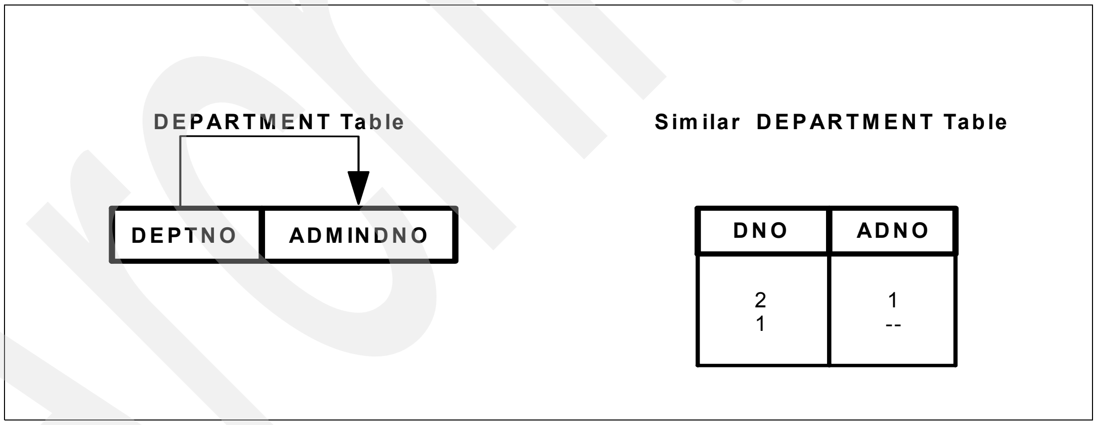


*Figure 3-12 Multi-row insert with self-relationship*

DEPT is a self-referencing table. SIMILARDEPARTMENT is a table that is identical to the DEPT table. Statement:

```
INSERT INTO DEPARTMENT
  SELECT * FROM SIMILARDEPARTMENT
```

The result depends upon the sequence in which DB2 accesses the SIMILARDEPARTMENT table:

- If the row with DNO of “2” is accessed first, the insert fails.
- If the row with DNO of “1” is accessed first, the insert succeeds.

If you have an application that requires this function, you might consider using the ALTER statement and CHECK utilities as follows:

1. Drop the self-referencing constraint using ALTER TABLE DROP FOREIGN KEY.
2. Issue the INSERT with SUBSELECT statement, which should execute successfully.
3. Recreate the self-referencing constraint. This puts the table space into a CHECK-pending status.
4. Run CHECK to determine if any referential constraint violations exist.

## 3.4.7 UPDATE

The restriction is that the CASCADE, SET NULL, and NO ACTION options on the UPDATE rule are unsupported. This restriction does not have anything to do with access path independence considerations.

The reason for this restriction is philosophical. Some people argue that by its very definition, a primary key value does not change. Once a primary key value is assigned, it remains the same for the lifetime of the entity occurrence.

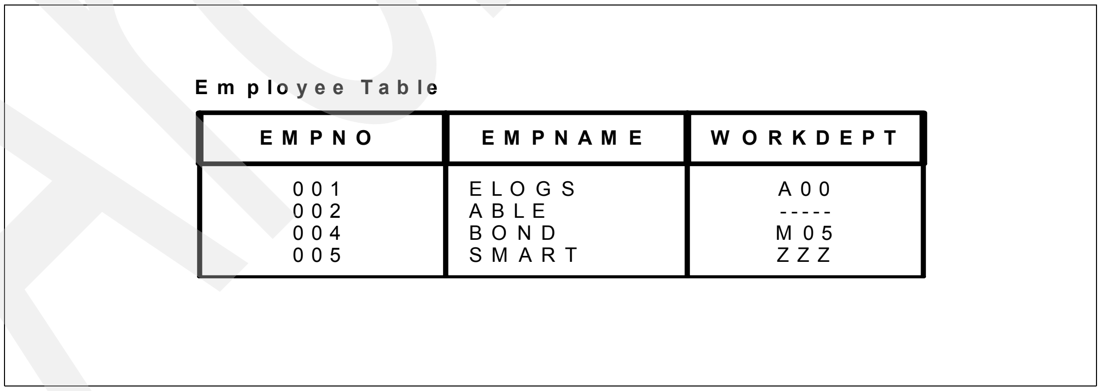
Therefore, the application of the UPDATE CASCADE rule or UPDATE SET NULL rule does not make sense. However, if the application requires an update of a primary key value, then the recommendation is to delete the entity occurrence and all its dependents, then reinsert them with the new value of the primary key.

DB2 permits UPDATE RESTRICT so that data entry types of mistakes can be corrected.

The recommendation for applications that need to implement the UPDATE CASCADE or SET NULL rule is to do so with user written code.

When the columns of a primary key are updated, the update statement should not apply to more than one row. Although DB2 allows massive update of the primary key, the update is access path dependent.

Figure 3-13 is used to explain a potential problem with an massive update of primary key.



Employee Table

| EMPNO | EMPNAME | WORKDEPT |
|---|---|---|
| 001 | ELOGS | A00 |
| 002 | ABLE | ----- |
| 004 | BOND | M05 |
| 005 | SMART | ZZZ |

*Figure 3-13 Employee table*

EMPNO is the primary key of the EMPLOYEE table. We issue the statement:

```
UPDATE EMPLOYEE SET EMPNO = EMPNO + 1
WHERE EMPNO =< 5
```

The result depends upon the sequence in which DB2 accesses the EMPLOYEE table:

- If DB2 accesses the table in the sequence of rows 005, 004, 002, and 001, the update would succeed.
- If DB2 accesses the table in the sequence of rows 001, 002, 004, and 005, the update would fail because the first update of 001 would fail with a duplicate on row 002.

## 3.4.8 DELETE

A DELETE statement which has a subquery referencing a table that can be affected by the deletion of rows from the target table is not permitted. Sometimes, a subquery with a correlated reference is executed once for each row. Because of this, the result of the deletion is access path dependent. This violation is detected at BIND time.

Figure 3-14 shows potential problems with DELETE with subquery. T2 denotes the object table of a DELETE statement and T1 denotes a table that is referenced in the FROM clause of a subquery of that statement. T1 must not be a table that can be affected by the DELETE on T2. Thus, the following rules apply:

- T1 and T2 must not be the same table.
- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL, unless the result of the subquery is materialized before the DELETE action is executed.
- T1 must not be a dependent of T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

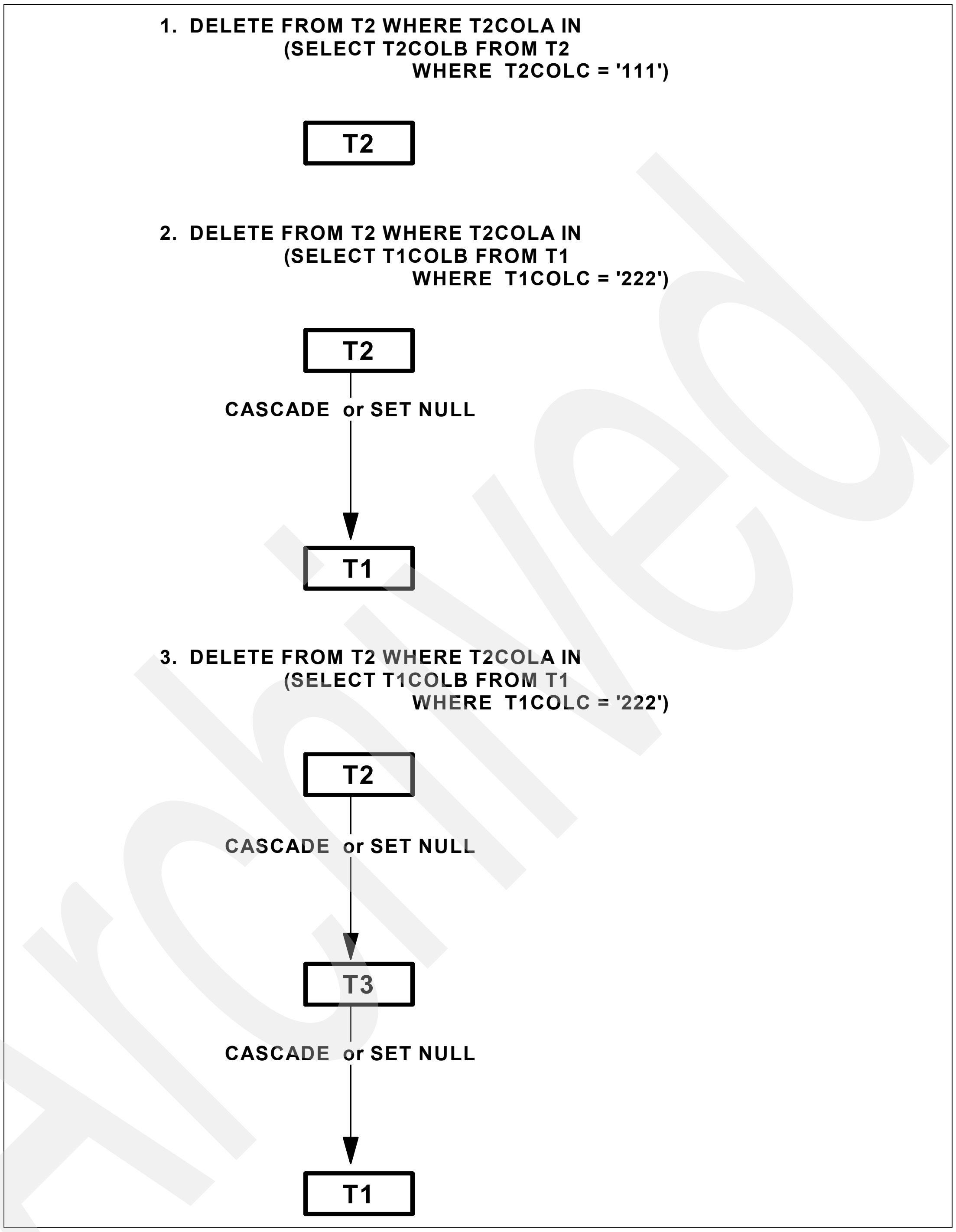


*Figure 3-14 Possible invalid deletes with subqueries*

Figure 3-15 shows a relationship with tables EMPLOYEE and DEPARTMENT that could have problems with a DELETE with subquery.

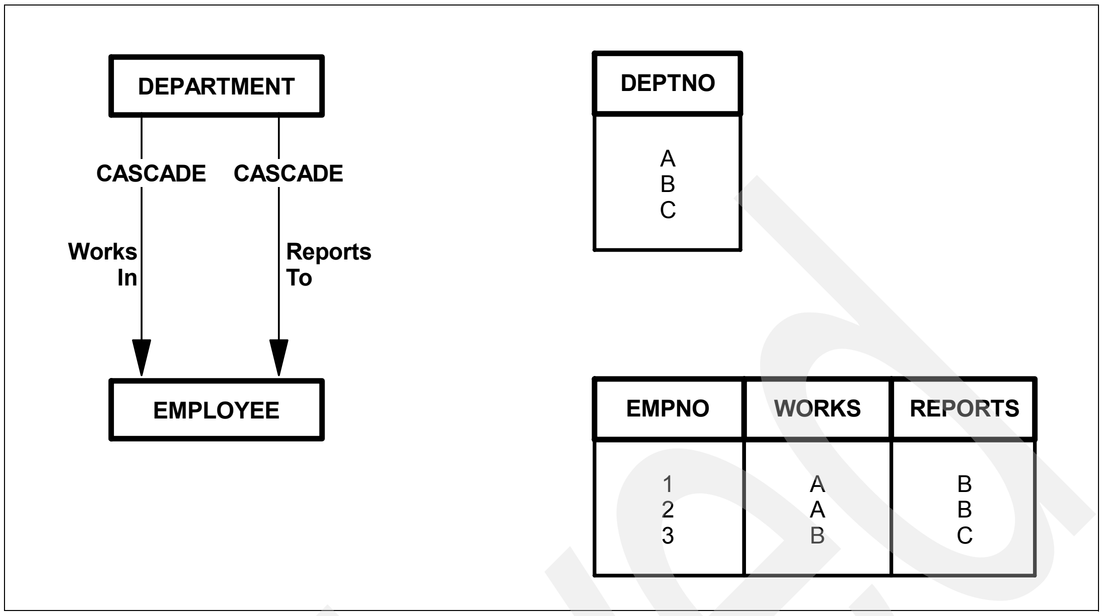


*Figure 3-15 Delete-connect sample*

This relationship shows to which department the employee reports, which can differ from the one in which the employee works.

The statement below is trying to delete all departments that do not have any employees reporting to them.

```
DELETE FROM DEPARTMENT
WHERE DEPTNO NOT IN (SELECT REPORTS FROM EMPLOYEE
WHERE EMPLOYEE.REPORTS = DEPARTMENT.DEPTNO)
```

The result depends upon the sequence in which rows in the DEPARTMENT table are accessed:

- If department “A” is accessed first:
  - The NOT IN clause is true; therefore, department “A” is deleted. This cascades resulting in employees “1” and “2” also being deleted.
  - Next, department “B” is accessed, and the NOT IN clause is again true because employees 1 and 2 have been deleted. Therefore, department “B” is now deleted, which results in employee “3” being deleted.
  - Finally, “C” is accessed and deleted.
- If department “B” is accessed first, the NOT IN clause is false and department “B” is not deleted. Similarly, department “C” is not deleted if it is accessed first. “A”, however, is always deleted no matter what the order.

## 3.5 Summary of design recommendations

In this section we summarize considerations and recommendations.

### 3.5.1 Primary key

- Assign a primary key for every table if the index overhead and the restriction on multi-row update of primary key columns is acceptable.

- Choose a primary key that cannot be updated.
- Primary key columns should be NOT NULL WITH DEFAULT only if the TIMESTAMP attribute is chosen.
- Choose a minimum number of columns to enforce uniqueness of the primary key.
- Recommend that all views include the primary key.

## 3.5.2 Foreign keys

- Consider carefully the semantics of columns that are part of foreign keys. Pay special attention to multi-column foreign keys, columns shared between multiple foreign keys, and columns shared with the primary key.
- Always supply a constraint name when defining a foreign key, and ensure that constraint names are easy to identify. A good naming standard for the objects defined will help identify parent and dependent quickly by just looking at the error message.

### Cycle considerations

A cycle involving two or more tables must not cause a table to be delete-connected to itself. Thus, if the relationship would form a cycle:

- The referential constraint cannot be defined if each of the existing relationships that would be part of the cycle have a delete rule of CASCADE.
- CASCADE must not be specified if T2 is delete-connected to T1.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. For example, assume that T1 is a dependent of T3 in a relationship with a delete rule of r and that one of the following is true:

- T2 and T3 are the same table.
- T2 is a descendent of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendents of the same table and the deletion of rows from that table cascades to both T2 and T3.

In this case, the referential constraint cannot be defined when r is SET NULL. When r is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as r.

If cycles are present in the referential structure, ensure that at least one of the foreign keys involved in the cycle is defined as being NULLable; otherwise the only way to insert data into the cycle is to use the LOAD utility.

A cycle with DELETE rules of RESTRICT can prevent delete operations, unless at least one of the foreign keys is defined as being NULLable.

Do not use referential constraints to maintain a one-to-one correspondence between the rows of the tables.

Keep the number of table spaces in a table space set reasonable. A group of 200 related tables is still reasonable, but 20,000 related tables is becoming no longer reasonable.

### 3.5.3 Circumventing DML restrictions

- To support RESTRICT or SET NULL options on the DELETE rule of self-referencing tables, we recommend you specify the DELETE CASCADE rule on the relationship and implement the RESTRICT or SET NULL rule through user code with some caveats.
- For any relationships that cause a referential structure to be delete-connected to itself, we recommend you specify a DELETE RESTRICT, SET NULL, or NO ACTION on at least two of the relationships, so that a table does not delete-connect to itself. Write user code to satisfy the required functions.
- For any applications that cause a referential structure to violate the restriction of a table being delete-connected through different paths with different DELETE rules or being SET NULL, we recommend you specify a DELETE RESTRICT rule on the relationships, so that a table has the same delete rule when connected through multiple paths. Write user code to satisfy the required functions.
- The UPDATE rule of RESTRICT can only be circumvented by user code that issues the appropriate SQL statements and takes the proper locks so that data integrity is not compromised.
- INSERT restrictions should be circumvented by using the ALTER statement and CHECK utility.

You can easily circumvent the DML restrictions by using SQL calls that only operate on one row at a time.

## 3.6 Code and look-up tables

A *code table* is a table where the domain of code columns is defined and the coded information is described. A l*ook-up table* is a type of code table that is used to replace one or more column values with a key or code that can be stored on the data table. The benefit of a look-up table is typically a reduction in space in the data table and the ability to update values in the look-up table without having to update each referencing row in the data table. In this section when we discuss "code tables", we are referring to both variations collectively.

Samples of coding tables are shown in Figure 3-16.

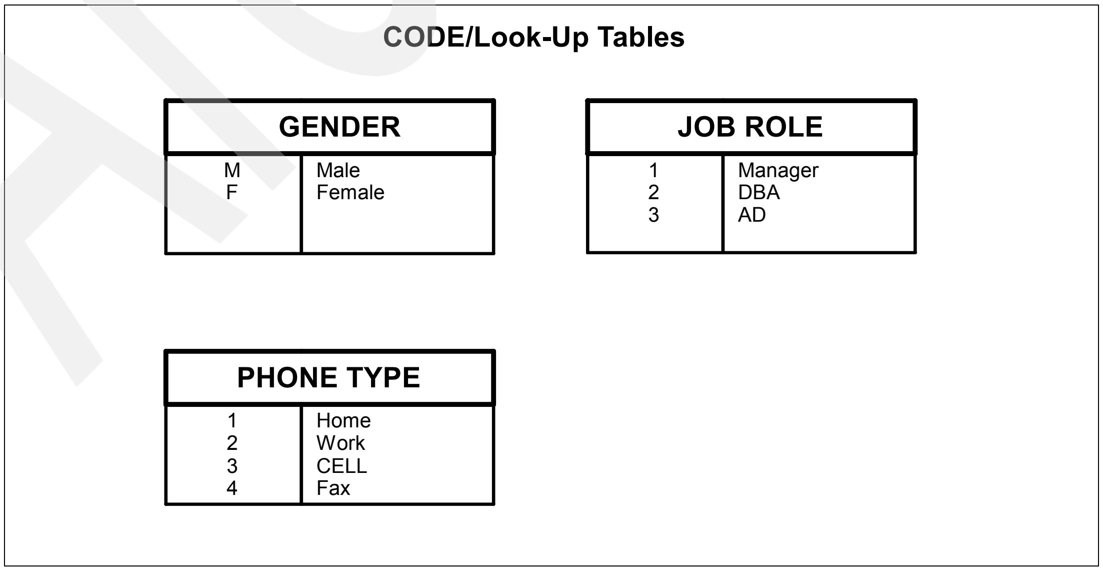
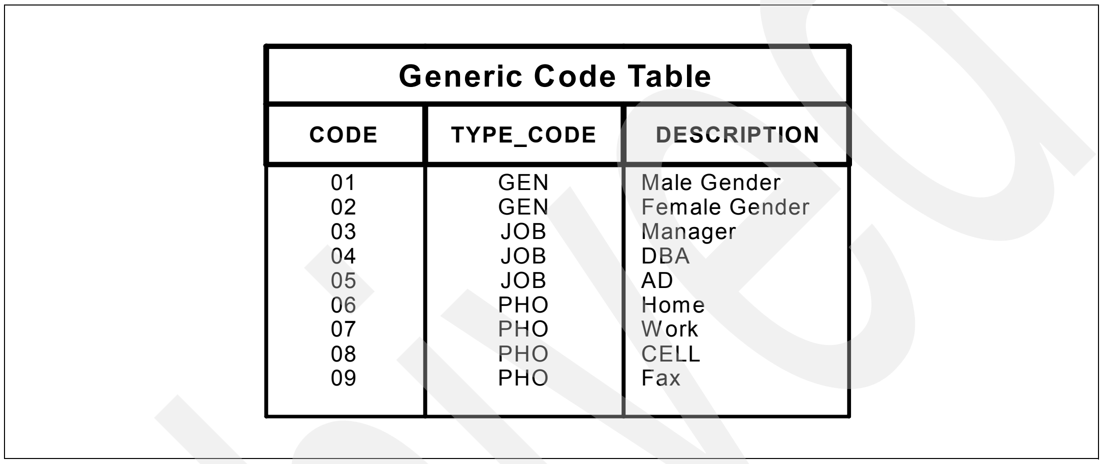


*Figure 3-16 Look-up tables*

A typical logical model might have numerous code tables that are used to defined the valid range of values that are allowed in a column on a data table. Implementing each different code table into the physical model with their own table, table space, and index may not be the most efficient approach. A very small code table with a unique index requires at least two physical data sets and three pages, since the index structure always begins with two levels.

Implemented individually, these code tables would require a minimum of six data sets and nine pages. Even if all three tables were placed into a segmented table space, they would require four data sets (one for data and three for indexes) and the same nine pages. By combining the three code tables into a single table as shown in Figure 3-17, we can store all of the code table data in two data sets and three pages.



**Generic Code Table**

| CODE | TYPE_CODE | DESCRIPTION |
|---|---|---|
| 01 | GEN | Male Gender |
| 02 | GEN | Female Gender |
| 03 | JOB | Manager |
| 04 | JOB | DBA |
| 05 | JOB | AD |
| 06 | PHO | Home |
| 07 | PHO | Work |
| 08 | PHO | CELL |
| 09 | PHO | Fax |

*Figure 3-17 Generic code table*

This approach would prevent database-enforced RI from being implemented between the code table and the data table. However, this is not a disadvantage because our recommendation is to not implement database-enforced RI on code tables.

In many relational database implementations, the number of accesses to code tables may equal or even exceed the number of accesses to the operational data tables. When RI is used to enforce relationships between code tables and data tables, a single insert into the data table may require accesses to multiple code tables. In addition, code tables are often accessed by applications performing edits or populating pick-lists or drop-down menus.

Because these tables are small and heavily accessed, they typically stay resident in the buffer pool, which should prevent I/O costs from becoming an issue. However, there is still a cost associated with every access to these tables, so options to reduce these costs should be considered.

In any case, the cost of the accesses to the code tables is not the only headache they cause. The problems are generally caused when RI to the code tables is enforced in the database.

## RI considerations

Whenever database-enforced RI is used, we recommend you index the foreign key columns on the dependent table. When relationships exist between code tables and operational data tables, the cardinality of the foreign key columns is typically very low relative to the cardinality of the table. Any index defined only on the foreign key would therefore provide very little filtering and would be of almost no use from a performance perspective, unless the application has access requirements using the foreign key and additional columns that can be added to the index to make it more unique.

Indexing the foreign keys of tables dependent on code tables might also make the performance of insert and delete activity on the dependent table too expensive because of the additional index maintenance.

A foreign key index will be used to identify dependent rows when a parent is deleted. However, this access path does not show up in the PLAN_TABLE when explaining the delete.

Finally, a load replace of a code table could result in multiple large operational tables being placed into CHECK-pending status. This may present an unacceptable risk to the availability of your database.

### 3.6.1 Code table alternatives

There are several alternatives that can be considered when implementing code tables:

- Check constraints can be used to enforce the domain of the foreign key in the database without accessing the code table.
- Code table values can be built into copybooks that can be included in programs that perform application-enforced RI checking or require look-up values.
- Batch programs can read code table values into working storage tables once and then use the values repeatedly.
- Code table values can be read from DB2 loaded into in-memory tables, such as CICS data tables, once and then used repeatedly to provide RI enforcement or look-up values.
- Denormalizing data to avoid accessing the lookup tables.
- Look-up table data that rarely or never changes can be stored redundantly on the data row to eliminate the need to access the look-up table.

With any of these alternatives, we recommend that you still implement the code tables as physical DB2 tables, and that these DB2 tables serve as the master copy of all allowed values.

## 3.7 DB2 versus application RI

In general, DB2-enforced RI is more efficient than application-enforced RI, because it can be performed at the time of the insert or delete without an additional trip to and from DB2. Therefore, our general recommendation is to use DB2-enforced RI wherever possible.

However, there are some factors that might result in DB2-enforced RI being less efficient than application-enforced RI.

- When multiple dependent rows are being inserted for a single parent, application-enforced RI can perform a single check, but DB2 will check for each dependent.
- DB2 RI checking cannot take advantage of index look-aside or sequential detection. If the access to the parent for RI checking is heavy and sequential, application-enforced RI, which can take advantage of both of these efficiency techniques, may be faster.
- If the foreign key of a partitioned table is indexed using a DPSI, DB2-enforced RI will require a probe into each partition looking for a dependent row when a parent row is deleted.

  If the partitioning key is derived from the foreign key so that the application can specify it in the WHERE clause, partition pruning can result in fewer partitions being probed.

There are also several circumstances under which DB2-enforced RI cannot be used, typically as a result of denormalization:

- DB2-enforced RI cannot be used if the entire primary key of the parent is not contained in the dependent table.
- For example, if a parent table contains a date component to allow the storage of history, and the date component was omitted from the dependent tables.
- DB2-enforced RI cannot be used if the foreign key of the dependent table can reference more than one parent, in an exclusive or relationship.

### Application-enforced RI recommendations

When performing application-enforced RI, either by choice or by necessity, consider the following recommendations:

- When inserting into a dependent table, be sure that the parent row is locked when the insert occurs.

  If the application reads the parent table using a singleton select and an isolation of CS, it is likely that no lock will be taken on the parent row or page. In this case, it is possible that another transaction can change or delete the parent row before the insert of the dependent row.

  Even with a cursor on the parent row, lock avoidance may leave the parent row exposed to deletion by other application processes.

  Be sure to read the parent with an isolation level that guarantees a lock. For more information about locking and isolation levels, see Chapter 12 of *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134.

- When inserting multiple dependent rows for a single parent row, read the parent once and hold the lock on the parent until all dependent inserts are complete.

  Avoiding redundant checks on the parent table is one of the ways that application-enforced RI can outperform DB2-enforced RI.

- When inserting multiple dependent rows, try to insert them in the order of the parent table.

  If application-enforced RI checks on the parent occur in a sequential manner, you can use index look-aside and sequential detection to reduce getpages and I/O.

- Even when rows are not ordered by the parent, consider the possibility that the foreign key has not changed.

  When inserting rows that have foreign keys to code or look-up tables, consider the possibility that multiple rows in sequence have the same value in the foreign key. A simple IF statement check comparing the current value to the previous value is cheaper than a database access or even the search of an in-memory table if the value is the same.

- Use informational referential constraints to let DB2 know that you are enforcing RI when accessing MQTs.

## 3.8 REPORT utility

The REPORT utility provides information about table spaces. There are two kinds of REPORT utility: REPORT TABLESPACESET and REPORT RECOVERY. In this chapter, we discuss the REPORT TABLESPACESET. For details, check *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

Use REPORT TABLESPACESET to find the names of all the table spaces and tables in a table space set, including LOB table spaces.

The output from REPORT TABLESPACESET consists of the names of all table spaces in the table space set that you specify. It also lists all tables in the table spaces and all tables that are dependent on those tables.

Let's assume SAMP as the database name and TSDEPT as the table space name for table DEPT.

Example 3-4 shows the output for the statement:

```
REPORT TABLESPACESET SAMP.TSDEPT
```

We see that TSEMP holds the EMP table and table space TSPROJ contains three other tables belonging to the referential structure. Test is the owner of the objects.

*Example 3-4 REPORT TABLESPACESET output*

```
DSNU050I    DSNUGUTC -  REPORT TABLESPACESET SAMP.TSDEPT
DSNU587I  -DB8A DSNUPSET - REPORT TABLESPACE SET WITH TABLESPACE SAMP.TSDEPT

TABLESPACE SET REPORT:


TABLESPACE       : SAMP.TSDEPT

   TABLE         : TEST.DEPT
      INDEXSPACE : SAMP.XDEPT1
           INDEX : TEST.XDEPT1
      INDEXSPACE : SAMP.XDEPT2
           INDEX : TEST.XDEPT2
      INDEXSPACE : SAMP.XDEPT3
           INDEX : TEST.XDEPT3
      DEP  TABLE : TEST.EMP
                   TEST.PROJ

TABLESPACE       : SAMP.TSEMP

   TABLE         : TEST.EMP
      INDEXSPACE : SAMP.XEMP1
           INDEX : TEST.XEMP1
      INDEXSPACE : SAMP.XEMP2
           INDEX : TEST.XEMP2
      DEP  TABLE : TEST.EMPPROJACT
                   TEST.PROJ

TABLESPACE       : SAMP.TSPROJ

   TABLE         : TEST.ACT
      INDEXSPACE : SAMP.XACT1
           INDEX : TEST.XACT1
      INDEXSPACE : SAMP.XACT2
           INDEX : TEST.XACT2
      DEP  TABLE : TEST.PROJACT

   TABLE         : TEST.EMPPROJACT
      INDEXSPACE : SAMP.XEMPPROJ
           INDEX : TEST.XEMPPROJACT1
      INDEXSPACE : SAMP.XEMP13ZK
           INDEX : TEST.XEMPPROJACT2

   TABLE         : TEST.PROJ
```

```
      INDEXSPACE : SAMP.XPROJ1
           INDEX : TEST.XPROJ1
      INDEXSPACE : SAMP.XPROJ2
           INDEX : TEST.XPROJ2
      DEP  TABLE : TEST.PROJACT

   TABLE         : TEST.PROJACT
      INDEXSPACE : SAMP.XPROJAC1
           INDEX : TEST.XPROJAC1
      DEP  TABLE : TEST.EMPPROJACT
DSNU580I    DSNUPORT - REPORT UTILITY COMPLETE - ELAPSED TIME=00:00:00
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

## 3.9 CHECK utility

When adding a referential constraint, unique constraint, or check constraint to a table, the table space of this dependent table is put in CHECK-pending status. For details, see “Altering a table for referential integrity” in the *DB2 UDB for z/OS Version 8 Administration Guide*, SC18-7413. This status is independent of the values within the table. There are three possible ways to resolve the CHKPEND status:

- DROP the constraint.
- Run the REPAIR utility with NOCHECKPEND.
- Run CHECK utilities.

Running the REPAIR utility can cause inconsistent data in the tables. The typical and most recommended way is to run the CHECK utility.

There are two types of CHECK:

- CHECK DATA
- CHECK INDEX

Before running CHECK DATA, you should run CHECK INDEX on primary key indexes and foreign key indexes to ensure that the indexes that CHECK DATA uses are valid. This action is especially important before using CHECK DATA with the DELETE YES or PART options.

You can add a foreign key with the NOT ENFORCED option to create an informational referential constraint. This action does not leave the table space in CHECK-pending status, and you do not need to execute CHECK DATA.

### 3.9.1 CHECK DATA

The CHECK DATA utility checks table spaces for violations of referential and table check constraints and reports information about violations that it detects. CHECK DATA also checks for consistency between a base table space and the corresponding LOB table space.

**Important:** The utility does not check informational referential constraints.

CHECK DATA optionally deletes rows that violate referential or table check constraints. CHECK DATA copies each row that violates one or more constraints to an exception table.

If a row violates two or more constraints, the row is copied only once. If the utility finds any violation of constraints, CHECK DATA puts the table space that it is checking in the CHECK-pending status.

You can specify that the CHECK-pending status is to be reset when CHECK DATA execution completes.

To avoid problems, you should run CHECK DATA with DELETE NO to detect the violations, before you attempt to correct the errors. If required, use DELETE YES after you analyze the output and understand the errors.

You can automatically delete rows that violate referential or table check constraints by specifying CHECK DATA with DELETE YES. However, you should be aware of the following possible challenges:

- The violation might be created by a non-RI error. For example, the indexes on a table might be inconsistent with the data in a table.
- Deleting a row might cause a cascade of secondary deletes in dependent tables. The cascade of deletes might be especially inconvenient within RI cycles.
- The error might be in the parent table.

CHECK DATA uses the primary key index and all indexes that exactly match a foreign key. Therefore, before running CHECK DATA, ensure that the indexes are consistent with the data by using the CHECK INDEX utility.

## Resetting the CHECK-pending status

CHECK DATA offers two ways to reset the CHECK-pending status:

- DELETE NO, when no tables contain rows that violate any kind of constraints (referential, unique, or check).
- DELETE YES to remove all rows that violate the constraints.

## Exception table

When using the DELETE YES option, an exception table for each dependent table or each table with table constraints that will be checked must be created. An exception table is a user-defined table with the same structure of a dependent table. The CHECK DATA with DELETE YES utility copies the violated rows to the the exception table and deletes them from the original table.

You can also include columns that identify the RID and the starting timestamp of the CHECK DATA. This is optional.

Example 3-5 shows how to create an exception table for the table DEPT shown in Figure 3-3 on page 58.

*Example 3-5 Exception table*

```
Create the exception table for DEPT

   CREATE EXP_DEPT LIKE DEPT IN SAMPLE.EXP_TBSP;

Adding the column that identifies the RID of the invalid row:
   ALTER TABLE EXP_DEPT
      ADD RID CHAR(4);

Adding the column that shows the starting DATE AND time of the CHECK DATA:
   ALTER TABLE EXP_DEPT
      ADD TIME_DATE TIMESTAMP NOT NULL WITH DEFAULT;
```

You have the capability to correct the data in the exception table using the update statement and reinsert the data into the original table using the insert statement.

## CHECK DATA output

CHECK DATA issues a message for every row that contains a referential or table check constraint violation. The violation is identified by:

- The RID of the row
- The name of the table that contains the row
- The name of the constraint that is being violated

Example 3-6 shows output for CHECK DATA for table DEPT. The DEPT is in CHECK-pending status, because of the addition of the foreign keys ADMRDEPT and MGRNO.

Input:

```
CHECK DATA TABLESPACE SAMPLE.TSDEPT
```

*Example 3-6 Output for CHECK DATA*

```
DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = CHKDEPT
DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNUGUTC - CHECK DATA TABLESPACE SAMPLE.TSDEPT SORTDEVT SYSDA SORTNUM 4
DSNUKDST - CHECKING TABLE TEST.DEPT
DSNUKDAT - CHECK TABLE SAMPLE.DEPT COMPLETE, ELAPSED TIME=00:00:00
DSNUK001 - CHECK DATA COMPLETE,ELAPSED TIME=00:00:00
DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

This case resets the CHECK-pending status.

Example 3-7 shows the output of the same utility on a copy of the table after an insertion of rows.

*Example 3-7 Output CHECK DATA with DELETE NO*

```
 DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = PAOLOR3.PAOLOR3C
 DSNUGTIS - PROCESSING SYSIN AS EBCDIC
 DSNUGUTC - CHECK DATA TABLESPACE FABRICIO.TSDEPT SORTDEVT SYSDA SORTNUM 4
 DSNUKDST - CHECKING TABLE PAOLOR3.DEPT
 DSNUGSOR - SORT PHASE STATISTICS -
           NUMBER OF RECORDS=7
           ELAPSED TIME=00:00:00
 DSNUKERK - ROW (RID=X'000000020F') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'000000020F') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000210') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'0000000210') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000211') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'0000000212') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000212') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKDAT - CHECK TABLE PAOLOR3.DEPT COMPLETE, ELAPSED TIME=00:00:00
DB8A DSNUGSRX - TABLESPACE FABRICIO.TSDEPT IS IN CHECK PENDING
 DSNUK001 - CHECK DATA COMPLETE,ELAPSED TIME=00:00:00
 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4
```

The table space FABRICIO.TSDEPT is still in CHECK-pending status. To remove the status, you can run the CHECK DATA utility with DELETE YES.

```
CHECK DATA TABLESPACE FABRICIO.TSDEPT
     FOR EXCEPTION IN DEPT USE EXP_DEPT
     DELETE YES
```

```
        SORTDEVT SYSDA SORTNUM 4
```

This utility deletes the invalid rows from the DEPT table and inserts them in the EXP_DEPT (exception) table. Example 3-8 shows the output for this case.

*Example 3-8 Output CHECK DATA with DELETE YES*

```
 DSNUKDST - CHECKING TABLE PAOLOR3.DEPT
 DSNUGSOR - SORT PHASE STATISTICS -
           NUMBER OF RECORDS=7
           ELAPSED TIME=00:00:00
 DSNUKERK - ROW (RID=X'000000020F') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'000000020F') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000210') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'0000000210') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000211') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKERK - ROW (RID=X'0000000212') HAS NO PARENT FOR PAOLOR3.DEPT.ADMRDEPT
 DSNUKERK - ROW (RID=X'0000000212') HAS NO PARENT FOR PAOLOR3.DEPT.MGRNO
 DSNUKDAT - CHECK TABLE PAOLOR3.DEPT COMPLETE, ELAPSED TIME=00:00:00
DB8A DSNUKRDY - 4 ROWS DELETED FROM TABLE PAOLOR3.DEPT
 DSNUK001 - CHECK DATA COMPLETE,ELAPSED TIME=00:00:01
 DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=4
```

The CHECK DATA utility resets the CHECK-pending status.

Example 3-9 shows the contents of the exception table.

*Example 3-9 Exception table for DEPT*

```
SELECT DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION, HEX(RID), TIME_DATE
FROM EXP_DEPT;

Result

DEPTNO  DEPTNAME  MGRNO   ADMRDEPT  LOCATION  HEX(RID)  TIME_DATE
----    --------  ------  --------  --------  --------  --------------------------
K01     DB2       12345   X01       -         0000020F  2005-10-26-14.57.13.329972
K02     IMS       23451   X32       -         00000210  2005-10-26-14.57.13.329972
K04     CICS      12345   D01       -         00000211  2005-10-26-14.57.13.329972
K03     MVS       00010   L01       -         00000212  2005-10-26-14.57.13.329972
```

## 3.9.2 CHECK INDEX

The CHECK INDEX online utility tests whether indexes are consistent with the data that they index and issues warning messages when it finds an inconsistency. We recommend you run CHECK INDEX before CHECK DATA, especially if you have specified DELETE YES. Running CHECK INDEX before CHECK DATA ensures that the indexes that CHECK DATA uses are valid.

With DB2 V8, CHECK INDEX now has two options:

- Read-only behavior as SHRLEVEL REFERENCE
- The new SHRLEVEL CHANGE

SHRLEVEL CHANGE allows you to have longer read-write access to applications during CHECK INDEX and reduces the read-only time for users. It satisfies the requirement of users with large tables who cannot afford outages caused by the read-only time of the normal CHECK INDEX SHRLEVEL REFERENCE. Add this new capability by applying APARs PQ92749 (PTF UK04683) and PQ96956 (UK08561) to DB2 V8.

If you specify SHRLEVEL REFERENCE, the default, the application can read from but cannot write to the index, table space, or partition that is to be checked. See Figure 3-18.

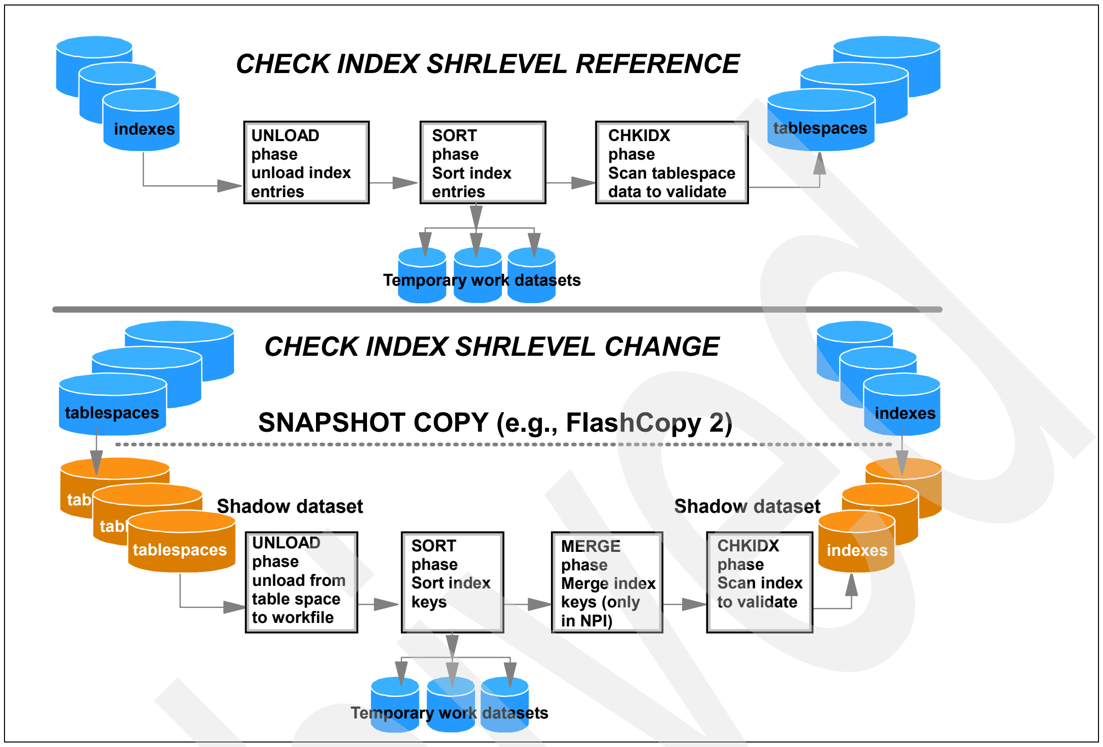

CHECK INDEX SHRLEVEL REFERENCE
indexes
UNLOAD phase unload index entries
SORT phase Sort index entries
CHKIDX phase Scan tablespace data to validate
tablespaces
Temporary work datasets
CHECK INDEX SHRLEVEL CHANGE
tablespaces
SNAPSHOT COPY (e.g., FlashCopy 2)
indexes
Shadow dataset
tablespaces
UNLOAD phase unload from table space to workfile
SORT phase Sort index keys
MERGE phase Merge index keys (only in NPI)
CHKIDX phase Scan index to validate
Shadow dataset
indexes
Temporary work datasets

*Figure 3-18 CHECK INDEX - SHRLEVEL REFERENCE and CHANGE*

If you specify SHRLEVEL CHANGE, the applications can read from and write to the index, table space, or partition that is being checked. The phases differ. In the case of SHRLEVEL REFERENCE, DB2 unloads the index entries, sorts the index entries, and scans the data to validate index entries. Alternatively, SHRLEVEL CHANGE first goes to the table space data set, unloads the data, sorts the index keys and merges them (if the index is non-partitioned), and then scans the index to validate.

In addition, CHECK INDEX provides a parallel processing structure similar to REBUILD INDEX.

When you specify SHRLEVEL CHANGE, DB2 performs the following actions:

- Drains all writers and forces the buffers to disk for the specified object and all of its indexes
- Invokes DFSMSdss™ to copy the specified object and all of its indexes to shadow data sets
- Enables read-write access for the specified object and all of its indexes
- Runs CHECK INDEX on the shadow data sets

DFSMSdss uses FlashCopy® Version 2, if available, to build the shadow data sets, reducing the unavailability to just the time to establish the relationship between the source and target copy.

Furthermore, SHRLEVEL CHANGE uses parallel tasks. With PI, the UNLOAD, SORT, and CHECKIDX tasks are processed in parallel on each partition of the TS. With NPI, the UNLOAD and SORT tasks are processed in parallel, just as with PI, but the CHECKIDX and MERGE phases are not.

SHRLEVEL CHANGE also provides extra checking of the index tree structure.

## 3.10 LOAD utility

By default, LOAD enforces referential constraints, except for informational referential constraints, which LOAD ignores. By enforcing referential constraints, LOAD provides you with several possibilities for error:

- Records that are to be loaded might have duplicate values of a primary key.
- Records that are to be loaded might have invalid foreign-key values, which are not values of the primary key of the corresponding parent table.
- The loaded table might lack primary key values that are values of foreign keys in dependent tables.

If you use the ENFORCE NO option, you tell LOAD not to enforce referential constraints. Sometimes you have good reasons for doing that, but the result is that the loaded table space might violate the constraints. Hence, LOAD places the loaded table space in CHECK-pending status. If you use REPLACE, all table spaces that contain any dependent tables of the tables that were loaded are also placed in CHECK-pending status. You must reset the status of each table before you can use any of the table spaces.

For more details about the LOAD utility, refer to *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

### 3.10.1 Loading tables involved in cycles

As a simple example, Figure 3-19 on page 94 shows the challenge of loading tables involved in cycles. In this example, each table resides in a separate table space.

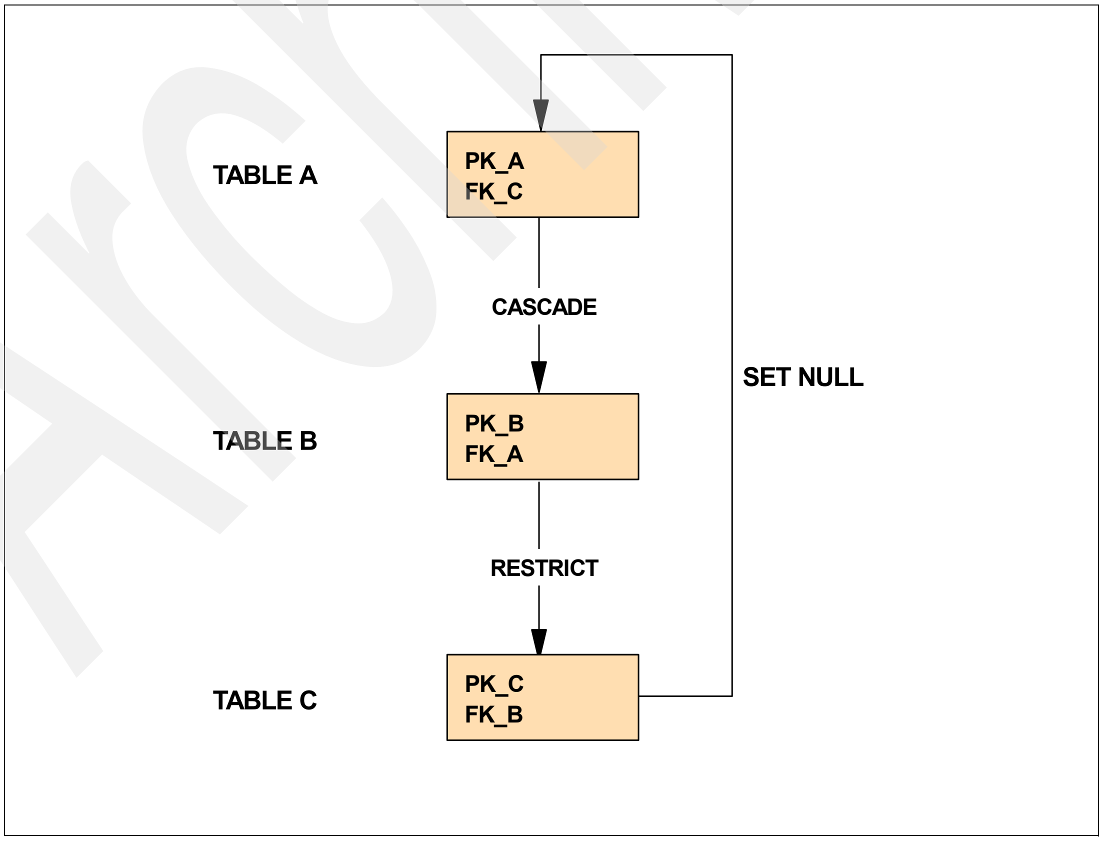


*Figure 3-19 Load tables involved in a cycle*

## Challenge of loading cycles

The challenges with loading cycles are:

- To load a table with ENFORCE CONSTRAINTS, the parent table must already have been loaded.

  In a cycle, this is not possible.
- If a table is loaded with ENFORCE NO, its table space is put in CHECK-pending status.
- When a table is in CHECK-pending status, it is impossible to read or update using normal SQL. Therefore, correcting errors is difficult.
- CHECK with DELETE(YES) always cascade deletes. In a cycle, this option could remove a lot of data which would have to be reinserted.

There are a number of ways to load cycles, we look at three options.

### *The first option*

If all the three tables involved in the cycle reside in the same table space, then all the tables can be loaded in a single load utility execution with ENFORCE CONSTRAINTS. This is possible because DB2 first loads all the input data in the RELOAD phase of the LOAD utility and verifies referential constraints in the ENFORCE phase.

The size of the tables in the referential structure and the availability of the new segmented table space organization may make this option a viable alternative. However, if there are many referential constraint violations in the data, many of the loaded rows will be deleted in the ENFORCE phase.

### *The second option*

If the three tables involved in the cycle reside in different table spaces:

1. Load all the tables in the referential structure, with ENFORCE NO. This results in all the table spaces being put into a CHECK-pending status.
2. Remove the CHECK-pending status of these table spaces as documented in "Resetting the CHECK-pending status" on page 90.

### *The third option*

- Allow all loading to be done with ENFORCE CONSTRAINTS.
- Ensure that a table does not have to be corrected when it is in CHECK-pending status.
- Ensure that no tables have to be reloaded.

This can be achieved by using the following technique:

1. Preprocess the input data to remove all the errors, if possible.
2. Remove one of the constraints so that the referential structure no longer has a cycle.

   This is done by dropping one of the foreign keys. Drop one of the foreign keys from the table with the fewest number of rows, so that the later phases have as little to do as possible.

   In Figure 3-19 on page 94, assume that table A has the fewest number of rows, and therefore, the foreign key from table A to table C is dropped:

   ```
   ALTER TABLE A
   DROP FOREIGN KEY FK_C
   ```

3. Load the data.

   The tables are now not in a cycle and can be loaded with ENFORCE YES in the following sequence:

   a. Table A is loaded first.
   b. Table B is loaded next.
   c. Finally, table C is loaded.

   This sequence ensures that the parent table rows are loaded first, so that dependent rows are not rejected due to the corresponding parent rows being absent.

4. Recreate the missing constraint so that the cycle is recreated.

   The foreign key between table A and table C can be recreated with statements as shown in Example 3-2 on page 68.

   This puts table A into CHECK-pending status.

5. Run the CHECK utility on the table spaces in CHECK-pending. Remove errors or reset the CHECK-pending status using the REPAIR utility if you are sure that no referential constraint violations actually exist in the data.

   Run CHECK utility with DELETE NO on table A.

   If there are no errors, then the load is completed.

   If there are errors, do the following:

   a. Break the cycle by dropping the foreign key FK-C again. This resets the CHECK-pending status on table A.
   b. Correct errors by modifying the foreign keys in table A or inserting corresponding parent rows in table C.
   c. Recreate the cycle, and then run CHECK utility to detect errors.
   d. Loop back to break the cycle until no more errors are detected.

The third option is preferable to the second option, because it potentially limits the amount of checking to be done by the CHECK utility to only one dependent table space. This is usually more efficient than loading all tables with no referential constraint enforcement followed by a CHECK utility execution against all the dependent table spaces.

## 3.11 Performance

As mentioned in “DB2 versus application RI” on page 86, most of the time, DB2-enforced RI performs better than an application-enforced RI. In “Triggers” on page 113, there are samples comparing insert and delete programs that use referential constraint, a trigger to enforce RI, and stored procedures. The results are generally much better using referential constraint.

But in fact, there are instances where application-enforced RI performs better, such as:

- Code tables

  Because these tables are small and heavily accessed, the cost associated with every access to these tables should be considered.

- Mass insert

  Application-enforced RI can check only once at the break of the parent table. DB2-enforced RI will check for each dependent row inserted.

- Mass delete

  With DB2-enforced RI, the advantage of mass delete performance with segmented table spaces is not available when all the rows in a parent table are deleted via a mass delete statement, such as:

  ```
  DELETE FROM T1
  ```

Our recommendation is to use DB2-enforced RI, wherever it is possible.

## 3.12 Migrating applications to RI

This section describes the considerations in migrating applications with application-enforced RI to DB2-enforced RI. This section is organized as follows:

- Planning considerations
- Application implementation considerations

### 3.12.1 Planning considerations

Migration of an existing application that has implemented application-enforced RI to DB2-enforced RI is a non-trivial task.

However, there are a number of reasons why the people in an installation may want to migrate an existing application to take advantage of DB2-enforced RI, such as:

- New application extensions are planned for the application, and both performance and application productivity improvements are expected.
- Updates are desirable from an ad hoc environment, such as QMF, SPUFI, or client tools.
- Maintenance costs are expected to be lower with DB2-enforced RI.
- Greater integrity of DB2 data with DB2-enforced RI.
- Increased automation of operations.

The DB2-enforced RI factors that should be taken into account when migrating existing applications are:

- DB2 restrictions implementing RI as described in 3.5.3, "Circumventing DML restrictions" on page 84
- Locking implications
- New SQL codes returned to applications on referential constraint violations
- Utility and operations impact

All these factors have a significant contribution in how easy or difficult it is for an application to be migrated to DB2-enforced RI.

Besides the DB2 considerations and the installation decision makers' desire to move to DB2-enforced RI, you must consider other factors, as follows:

- Does the migration process have to be phased over time or does it require being cut over in one step?

  A phased migration process is a process where some of the data and application programs are migrated to DB2-enforced RI, while other tables and application programs in the same application are not migrated. This is a very complex task because of the DB2-enforced RI considerations described earlier.

With a mix of DB2-enforced RI and application-enforced RI, care must be taken when backing up and restoring tables because DB2 is aware of a referential structure while the "application" reflects another.

A phased migration might be possible, but it is likely to take a significant portion of time compared to a migration in a single step.

- In addition, the installation must:
  - Understand how the existing applications are written, in particular the updating aspects which affect referential constraint. For example:
    - Does the current implementation enforce RI effectively?
    - Are all the updates in subroutines or spread throughout the code?
    - Would any of the statements violate DB2 restrictions?
    - How are unexpected SQLCODEs handled?

    This process will help define the amount of work necessary to migrate the application to DB2-enforced RI.
  - Understand how operations ensure RI for the existing application, such as:
    - How are tables loaded and RI-checked?
    - Are there any consistency check processes?
    - How is backup and recovery controlled across related tables?

    This process should highlight areas of concern when moving operations to DB2-enforced RI.

## 3.12.2 Application implementation considerations

The following is a checklist of the steps that must be taken in order to migrate an existing application to DB2-enforced RI in one procedure:

1. Revisit the conceptual model for the application and ensure that all the relevant RI-related information is included.
2. Design the DB2 referential structure.
3. On the basis of the existing DB2 implementation, if required, modify the DB2 referential structure designed in the previous step.
   - If the existing DB2 implementation structure is markedly different from the DB2 referential structure, a design compromise has to be made regarding which structure to choose.
   - If the existing structure is chosen, the performance of the migrated application could be affected.
   - If the DB2 referential structure is chosen, the migration process could become very complex.
4. Test the extended DDL on test tables to ensure that the referential constraints do not violate the DB2 restrictions.
5. REVOKE some of the authorities that were required with application-enforced RI, but are no longer required for DB2-enforced RI.
6. The privilege set must include a GRANT ALTER authority or the REFERENCES privilege on the columns of the parent key to creators of foreign keys.
7. ALTER TABLE to add unique key and primary key definitions, if needed.

8. ALTER TABLE ADD FOREIGN KEY for all the relationships of a dependent table. This will put the table space into CHECK-pending. This process could be done for all table spaces at once or a table space at a time. The choice depends on the complexity of the application and the level of confidence in the planning process. Choose useful names for the referential constraints.
9. Add FK index definitions to help when deleting parent rows. Note that these FK indexes are not shown in the PLAN_TABLE.
10. Run the CHECK utility. If it fails, drop the constraint and find out why the data is inconsistent. Make the necessary corrections and rerun the CHECK utility until no more errors are found.
11. Upgrade the LOAD, RECOVER, COPY, and CHECK jobs to use the facilities of DB2 and do a backup.
12. In the programs, modify the SQL statements that affect referential constraints, such as INSERT, UPDATE, and DELETE. Provide these SQL statements so people are aware of the constraint names and the related error message handling.
13. Rebind all plans.

You must do this with care to ensure that locking and back out considerations are not compromised.

## 3.13 DB2 catalog information and queries

This section highlights the information in the DB2 catalog to support RI and suggests some queries that you might use to obtain useful information for managing the RI environment.

This section is organized as follows:

- DB2 catalog extensions

  The extensions to specific tables in the DB2 catalog are described briefly.

- Sample catalog queries

  Some queries that might be useful in managing the RI environment are provided. These are sample queries. You are expected to build and extend upon the base provided to suit your environment.

### 3.13.1 DB2 catalog extensions

DB2 catalog tables are used specifically to support RI and constraint definitions, as well as existing tables impacted by such definitions and usage, are shown in Figure 3-20 and listed here:

- SYSIBM.SYSCHECKDEP
- SYSIBM.SYSCHECKS/SYSCHECKS2
- SYSIBM.SYSCOLUMNS
- SYSIBM.SYSCOPY
- SYSIBM.SYSFOREIGNKEYS
- SYSIBM.SYSINDEXES
- SYSIBM.SYSKEYS
- SYSIBM.SYSRELS
- SYSIBM.SYSTABLEPART
- SYSIBM.SYSTABLES
- SYSIBM.SYSTABLESPACE

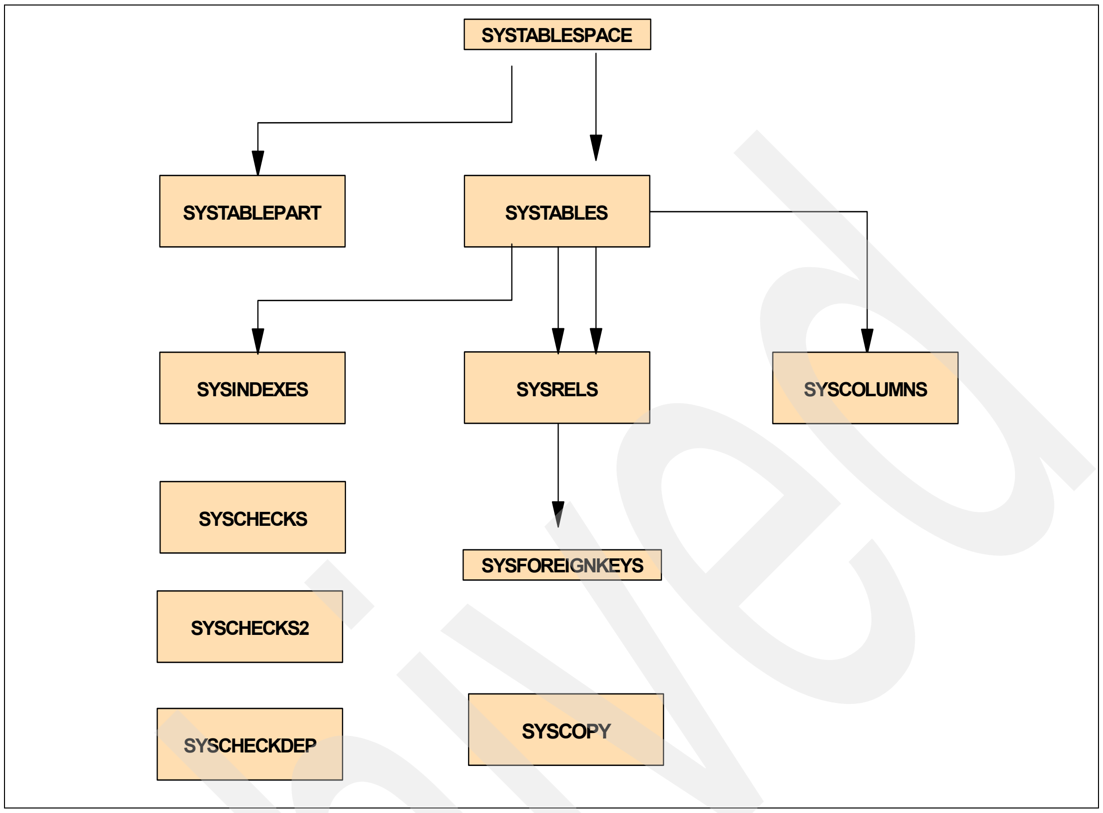


*Figure 3-20 DB2 catalog tables affected by RI*

The catalog tables used for check constraint definitions are listed here for completeness.

## SYSIBM.SYSCHECKS

SYSIBM.SYSCHECKS contains one row for each check constraint defined on a table. See Table 3-4.

*Table 3-4 SYSIBM.SYSCHECKS*

| Column name | Data type | Description |
|---|---|---|
| TBOWNER | VARCHAR(128) NOT NULL | Authorization ID of the owner of the table on which the check constraint is defined. |
| CREATOR | VARCHAR(128) NOT NULL | Authorization ID of the creator of the check constraint. |
| DBID | SMALLINTNOT NULL | Internal identifier of the database for the check constraint. |
| OBID | SMALLINTNOT NULL | Internal identifier of the check constraint. |
| TIMESTAMP | TIMESTAMP NOT NULL | Name of column. |
| RBA | CHAR(6) FOR BIT DATA NOT NULL WITH DEFAULT | The log RBA when the check constraint was created. |

| Column name | Data type | Description |
|---|---|---|
| IBMREQD | CHAR(1) NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |
| TBNAME | VARCHAR(128) NOT NULL | Name of the table on which the check constraint is defined. |
| CHECKNAME | VARCHAR(128) NOT NULL | Table check constraint name. |
| CHECKCONDITION | VARCHAR(7400) NOT NULL | Text of the table check constraint. |

## SYSIBM.SYSCHECKS2

SYSIBM.SYSCHECKS2 contains one row for each table check constraint for catalog tables created in or after Version 7. Check constraints for catalog tables created before Version 7 are not included in this table. See Table 3-5.

*Table 3-5 SYSIBM.SYSCHECKS2*

| Columns name | Data type | Description |
|---|---|---|
| TBOWNER | VARCHAR(128) NOT NULL | Authorization ID of the owner of the table on which the check constraint is defined. |
| TBNAME | VARCHAR(128) NOT NULL | Name of the table on which the check constraint is defined. |
| CHECKNAME | VARCHAR(128) NOT NULL | Table check constraint name. |
| PATHSCHEMAS | VARCHAR(2048) NOT NULL | SQL path at the time the check constraint was created. The path is used to resolve unqualified cast function names that are used in the constraint definition. |
| IBMREQD | CHAR(1) NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |

## SYSIBM.SYSCHECKDEP

SYSIBM.SYSCHECKDEP contains one row for each reference to a column in a check constraint. See Table 3-6.

*Table 3-6 SYSIBM.SYSCHECKDEP*

| Column name | Data type | Description |
|---|---|---|
| TBOWNER | VARCHAR(128) NOT NULL | Authorization ID of the owner of the table on which the check constraint is defined. |
| TBNAME | VARCHAR(128) NOT NULL | Name of the table on which the check constraint is defined. |
| CHECKNAME | VARCHAR(128) NOT NULL | Check constraint name. |

| Column name | Data type | Description |
|---|---|---|
| COLNAME | VARCHAR(128) NOT NULL | Name of column to which the check constraint refers. |
| IBMREQD | CHAR(1) NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |

The catalog columns used to support RI.

## SYSIBM.SYSCOLUMNS

SYSIBM.SYSCOLUMNS has two columns that pertain to RI. In the description of these columns in Table 3-7, C denotes a column described by a row of SYSCOLUMNS.

*Table 3-7 SYSIBM.SYSCOLUMNS columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| KEYSEQ | SMALLINT NOT NULL | The ordinality of C within the primary key of its table. Zero if C is not part of a primary key. |

## SYSIBM.SYSCOPY

SYSIBM.SYSCOPY has two existing columns that are used for RI, see Table 3-8.

*Table 3-8 SYSIBM.SYSCOPY columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| ICTYPE | CHAR(1) NOT NULL | If ICTYPE is Q, the row contains information about an invocation of the QUIESCE utility. |
| TIMESTAMP | TIMESTAMP | The date and time when the row was inserted. This is the date and time recorded in ICDATE and ICTIME. The use of TIMESTAMP is recommended over that of ICDATE and ICTIME, because the latter two columns may not be supported in later DB2 releases. For the COPYTOCOPY utility, this value is the date and time when the row was inserted for the primary local site or primary recovery site copy. |

## SYSIBM.SYSFOREIGNKEYS

SYSIBM.SYSFOREIGNKEYS contains one row for every column of every foreign key. See Table 3-9. The foreign key of SYSFOREIGNKEY is (CREATOR, TBNAME, RELNAME) and its values match the primary key of SYSRELS. Table 3-11 shows the columns of SYSRELS.

*Table 3-9 SYSIBM.SYSFOREIGNKEYS columns affected by RI*

| Columns name | Data type | Description |
|---|---|---|
| CREATOR | VARCHAR( 128) NOT NULL | Authorization ID of the owner of the table that contains the column. |

| Columns name | Data type | Description |
|---|---|---|
| TBNAME | VARCHAR( 128) NOT NULL | Name of the table that contains the column. |
| RELNAME | VARCHAR( 128) NOT NULL | Constraint name for the constraint for which the column is part of the foreign key. |
| COLNAME | VARCHAR( 128) NOT NULL | Name of column. |
| COLNO | SMALLINT NOT NULL | Numeric place of the column in its table. |
| COLSEQ | SMALLINT NOT NULL | Numeric place of the column in the foreign key. |
| IBMREQD | CHAR(1) NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |

## SYSIBM.SYSINDEXES

SYSIBM.SYSINDEXES has one column that is used for RI, see Table 3-10.

*Table 3-10 SYSIBM.SYSINDEXES column affected by RI*

| Column name | Data type | Description |
|---|---|---|
| UNIQUERULE | CHAR(1) NOT NULL | P: Yes, and it is a primary index (As in prior releases of DB2, a value of P is used for primary keys that are used to enforce a referential constraint.)<br>R: A non-PK unique key used as a parent key. |

## SYSIBM.SYSRELS

SYSIBM.SYSRELS has a row for every defined relationship. The primary key of SYSRELS is (CREATOR, TBNAME, RELNAME). The foreign keys of SYSRELS are (CREATOR, TBNAME) and (REFTBCREATOR, REFTBNAME). Both foreign keys reference the primary key of SYSTABLES which is (CREATOR, NAME). See Table 3-11.

*Table 3-11 SYSIBM.SYSRELS columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| CREATOR | VARCHAR(128) NOT NULL | Authorization ID of the owner of the dependent table of the referential constraint. |
| TBNAME | VARCHAR(128) NOT NULL | Name of the dependent table of the referential constraint. |
| RELNAME | VARCHAR(128) NOT NULL | Constraint name. |
| REFTBNAME | VARCHAR(128) NOT NULL | Name of the parent table of the referential constraint. |

| Column name | Data type | Description |
|---|---|---|
| REFTBCREATOR | VARCHAR(128)<br>NOT NULL | Authorization ID of the owner of the parent table. |
| COLCOUNT | SMALLINT<br>NOT NULL | Number of columns in the foreign key. |
| DELETERULE | CHAR(1)<br>NOT NULL | Type of delete rule for the referential constraint:<br>A NO ACTION<br>C CASCADE<br>N SET NULL<br>R RESTRICT |
| IBMREQD | CHAR(1)<br>NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |
| RELOBID1 | SMALLINT<br>NOT NULL<br>WITH DEFAULT | Internal identifier of the constraint with respect to the database that contains the parent table. |
| RELOBID2 | SMALLINT<br>NOT NULL<br>WITH DEFAULT | Internal identifier of the constraint with respect to the database that contains the dependent table. |
| TIMESTAMP | TIMESTAMP<br>NOT NULL<br>WITH DEFAULT | Date and time the constraint was defined. If the constraint is between catalog tables prior to DB2 Version 2 Release 3, the value is 1985-04-01-00.00.00.000000. |
| IXOWNER | VARCHAR(128)<br>NOT NULL | Owner of unique non-primary index used for the parent key. The values is 99999999 if the enforcing index has been dropped. Blank if the enforcing index is a primary index. |
| IXNAME | VARCHAR(128)<br>NOT NULL | Name of unique non-primary index used for a parent key. The value is 99999999 if the enforcing index has been dropped. Blank if the enforcing index is a primary index. |
| ENFORCED | CHAR(1) NOT NULL WITH DEFAULT | Enforced by the system or not:<br>Y Enforced by the system<br>N Not enforced by the system (trusted) |
| CHECKEXISTING DATA | CHAR(1) NOT NULL WITH DEFAULT | Option for checking existing data: I, Immediately check existing data. If ENFORCED = Y, this column will have a value of I. N, Never check existing data. If ENFORCED = N, this column will have a value of N. |

DSNDLX01 is an index on SYSLINKS that is not used by DB2 and is not required for RI. However, RI does require an index on SYSRELS. Thus, DSNDLX01 is redefined as an index on SYSRELS (REFTBCREATOR, REFTBNAME). This is a non-cluster, non-unique index with generic clustering on REFTBCREATOR. It is used by DB2 to determine all relationships in which a table is a parent. Furthermore, since it is an index on the foreign key of the relationship that is added to the catalog, it also used to cascade delete rows of SYSRELS when a parent table is dropped.

## SYSIBM.SYSTABCONST

SYSIBM.SYSTABCONST contains one row for each unique constraint (primary key or unique key) created in DB2 for OS/390 Version 7 or later. See Table 3-12.

*Table 3-12 SYSIBM.SYSTABCONST columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| CONSTNAME | VARCHAR(128) NOT NULL | Name of the constraint. |
| TBCREATOR | VARCHAR(128) NOT NULL | Authorization ID of the owner of the dependent table on which the constraint is defined. |
| TBNAME | VARCHAR(128) NOT NULL | Name of the dependent table on which the constraint is defined. |
| CREATOR | VARCHAR(128) NOT NULL | Authorization ID under which the constraint was created. |
| TYPE | CHAR(1) NOT NULL | Type of constraint:<br>P Primary key<br>U Unique key |
| IXOWNER | VARCHAR(128) NOT NULL | Owner of unique non-primary index used for the parent key. The value is 99999999 if the enforcing index has been dropped. Blank if the enforcing index is a primary index. |
| IXNAME | VARCHAR(128) NOT NULL | Name of unique non-primary index used for a parent key. The value is 99999999 if the enforcing index has been dropped. Blank if the enforcing index is a primary index. |
| CREATEDTS | TIMESTAMP NOT NULL | Time when the statement to create the constraint was executed. |
| IBMREQD | CHAR(1) NOT NULL | A value of Y indicates that the row came from the basic machine-readable material (MRM) tape. |
| COLCOUNT | SMALLINT NOT NULL | Number of columns in the constraint. |

## SYSIBM.SYSTABLEPART

SYSIBM.SYSTABLEPART has two columns that are used for RI. See Table 3-13.

*Table 3-13 SYSIBM.SYSTABLEPART columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| CHECKFLAG | CHAR(1) NOT NULL WITH DEFAULT | C .The table space partition is in a CHECK-pending status and there are rows in the table that can violate referential constraints, check constraints, or both.<br>Blank. The table space is not a partition or does not contain rows that may violate referential constraints, check constraints, or both. |
| CHECKRID5B | CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA | Blank. if the table or partition is not in a CHECK-pending status (CHECKFLAG is blank), or if the table space is not partitioned. Otherwise it is the RID of the first row of the table space partition that can violate referential constraints, check constraints, or both; or the value is X'0000000000', indicating that any row can violate referential constraints. |

## SYSIBM.SYSTABLES

SYSIBM.SYSTABLES has seven columns that pertain to RI. These columns apply only to tables and contain non-null default values in rows that describe views. See Table 3-14.

*Table 3-14 SYSIBM.SYSTABLES columns affected by RI*

| Column name | Data type | Description |
|---|---|---|
| PARENTS | SMALLINT NOT NULL | Number of relationships in which the table is a dependent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table. |
| CHILDREN | SMALLINT NOT NULL | Number of relationships in which the table is a parent. The value is 0 if the row describes a view, an alias, a created temporary table, or a materialized query table. |
| KEYCOLUMNS | SMALLINT NOT NULL | Number of columns in the table's primary key. The value is 0 if the row describes a view, an alias, or a created temporary table. |
| STATUS | CHAR(1) NOT NULL | Indicates the status of the table definition:<br>I .The definition of the table is incomplete. The TABLESTATUS column indicates the reason why the table definition is incomplete.<br>X. The table has a primary index and the table definition is complete.<br>Blank. The table has no primary index, the table is a catalog table, or the row describes a view or alias. The definition of the table, view, or alias is complete. |
| KEYOBID | SMALLINT NOT NULL | Internal DB2 identifier of the index that enforces uniqueness of the table's primary key; 0 if not applicable. |
| CHECKFLAG | CHAR(1) NOT NULL WITH DEFAULT | C. The table space that contains the table is in a CHECK-pending status because there are rows in the table that violate referential constraints, table check constraints, or both. |
| CHECKRID5B | CHAR(5) NOT NULL WITH DEFAULT FOR BIT DATA | Blank. if the table or partition is not in a CHECK-pending status (CHECKFLAG is blank), if the table space is not partitioned, or if the table is a created temporary table. Otherwise it is the RID of the first row of the table space partition that can violate referential constraints, check constraints, or both; or the value is X'0000000000', indicating that any row can violate referential constraints. |
| TABLESTATUS | SMALLINT NOT NULL | Indicates the reason for an incomplete table definition:<br>P. Definition is incomplete because the table lacks a primary index. |

### SYSIBM.SYSTABLESPACE

SYSIBM.SYSTABLESPACE has one existing column that is used for RI. See Table 3-15.

*Table 3-15 SYSIBM.SYSTABLESPACE column affected by RI*

| Column name | Data type | Description |
|---|---|---|
| STATUS | CHAR(1)<br>NOT NULL | Availability status of the table space:<br>A Available<br>P Table space is in a CHECK-pending status.<br>S Table space is in a CHECK-pending status with the scope less than the entire table space. |

## 3.13.2 Sample catalog queries

In this section, we show a set of sample queries. They are provided with the objective to help you check out the referential structures. They are meant to list:

- Tables without a primary key
- Tables by creator without a foreign key
- Dependent tables of a specific table by creator
- Parent tables of a specific table by creator
- All the foreign key columns for a specific creator
- Is the index a foreign key index?
- Foreign keys without indexes
- Columns in a foreign key without index
- Columns participating in primary key and foreign keys
- What are the quiesce points for the given table spaces?
- What is the scope of CHECK-pending for the tables in a table space?
- Which tables are the ancestors?
- Which tables are the descendents?
- The whole family

Many more queries can be identified to manage and control the RI environment, which are not described here, but you can expand on the basic queries given here to suit your own environment. In some cases, it becomes necessary to run the REPORT or CHANGE LOG INVENTORY utility to obtain information that is not readily or easily accessible from the DB2 catalog. For instance, SYSLGRNG entries for a table space can only be obtained by running the REPORT utility for a table space.

### Tables without a primary key

```
SELECT CREATOR. NAME
FROM SYSIBM.SYSTABLES
WHERE TYPE = 'T' AND
STATUS = ‘ ‘
ORDER BY NAME
```

### Tables by creator without a foreign key

```
SELECT CREATOR, NAME, DBNAME
FROM SYSIBM.SYSTABLES
WHERE CREATOR = &creator
AND TYPE = 'T'
AND CREATOR ||NAME
NOT IN ( SELECT DISTINCT CREATOR || TBNAME
FROM SYSIBM.SYSRELS )
ORDER BY NAME
```

## Dependent tables of a specific table by creator

```
SELECT CREATOR, TBNAME, RELNAME
FROM SYSIBM.SYSRELS
WHERE REFTBCREATOR = &creator
AND REFTBNAME = &tablename
ORDER BY CREATOR, TBNAME, RELNAME
```

This example shows how to get a list of dependent tables. Use this when considering a change to a table that could affect any children.

## Parent tables of a specific table by creator

```
SELECT REFTBCREATOR. REFTBNAME. RELNAME. DELETERULE
FROM SYSIBM.SYSRELS
WHERE CREATOR = &creator
AND TBNAME = &tablename
ORDER BY REFTBCREATOR, REFTBNAME, RELNAME
```

This example shows how to get a list of parent tables. Use this when considering a change to a table that could affect its parents.

## All the foreign key columns for a specific creator

```
SELECT F.CREATOR, F.TBNAME, F.COLNAME, F.RELNAME,
R.REFTBCREATOR, R.REFTBNAME, C.NAME
FROM
SYSIBM.SYSFOREIGNKEYS F,
SYSIBM.SYSRELS R,
SYSIBM.SYSCOLUMNS C
WHERE
F.CREATOR = &creator
AND F.CREATOR = R.CREATOR
AND F.TBNAME = R.TBNAME
AND F.RELNAHE = R.RELNAME
AND R.REFTBCREATOR = C.TBCREATOR
AND R. REFTBNAME = C. TBNAME
AND F.COLSEQ = C.KEYSEQ
ORDER BY F.CREATOR, F.TBNAME, F.RELNAME, C.NAME
```

This example lists the columns of each of the foreign keys in the dependent table and their corresponding primary key in the parent table. This query could be useful when developing complex queries joining various tables.

## Is the index a foreign key index?

```
SELECT I.NAME, R.CREATOR, R.TBNAME, R.RELNAME, R.COLCOUNT
FROM
   SYSIBM.SYSFOREIGNKEYS F,
   SYSIBM.SYSRELS R,
   SYSIBM.SYSINDEXES I,
   SYSIBM.SYSKEYS K
WHERE
       F.CREATOR = &creator
   AND F.CREATOR = R.CREATOR
   AND F.TBNAME = R.TBNAME
   AND F.RELNAME = R.RELNAME
   AND R.CREATOR = I.TBCREATOR
   AND R.TBNAME = I.TBNAME
   AND R.COLCOUNT <= I.COLCOUNT
   AND I.CREATOR = K.IXCREATOR
   AND I.NAME = K.IXNAME
```

```
      AND K.COLNAME = F.COLNAME
      AND K.COLSEQ = F.COLSEQ
   GROUP BY I.NAME, R.CREATOR, R.TBNAME, R.RETNAME, R.COLCOUNT
   HAVING COUNT(*) = R.COLCOUNT
   ORDER BY I.NAME, R.CREATOR, R.TBNAME, R.RELNAME
```

An index is an index on a foreign key if the foreign key has ‘*n’* columns and the first ‘*n’* columns of the index map onto the same columns of the table as the foreign key and in the same order.

The comparisons K.COLNO to F.COLNO and K.COLSEQ to F.COLSEQ ensure that the columns of the index map onto the columns of the foreign key.

The HAVING clause ensures that all the columns map.

The comparison R.COLCOUNT < = X.COLCOUNT is not necessary to get the correct answer but avoids checks of relationships that cannot qualify.

## Foreign keys without indexes

```
   SELECT REL.CREATOR, REL.TBNAME, REL.RELNAME
      FROM
         SYSIBM.SYSRELS REL
      WHERE
             REL.CREATOR = &creator
         AND REL.CREATOR ||REL.TBNAME || REL.RELNAME NOT IN
      ( SELECT R.CREATOR || R.TBNAME || R.RELNAME
         FROM
            SYSIBM.SYSFOREIGNKEYS F,
            SYSIBM.SYSRELS R,
            SYSIBM.SYSINDEXES I,
            SYSIBM.SYSKEYS K
         WHERE
                F.CREATOR = R.CREATOR
            AND F.TBNAME = R.TBNAME
            AND F.RELNAME = R.RELNAME
            AND R.CREATOR = I.TBCREATOR
            AND R.TBNAME = I.TBNAME
            AND R.COLCOUNT <= I.COLCOUNT
            AND I.CREATOR = K.IXCREATOR
            AND I.NAME = K.IXNAME
            AND K.COLNAME = F.COLNAME
            AND K.COLSEQ = F.COLSEQ
         GROUP BY I.NAME, R.CREATOR, R.TBNAME, R.RELNAME, R.COLCOUNT
         HAVING COUNT(*) = R.COlCOUNT )
      ORDER BY REL.CREATOR, REL.TBNAME, REL.RELNAME
```

## Columns in a foreign key without index

```
   SELECT         MISSINGIX.THE_TABLE   AS THE_TABLE
                , MISSINGIX.THE_RELNAME AS THE_RELNAME
                , FKDETAIL.COLSEQ  AS COLSEQ
                , FKDETAIL.COLNAME AS COLNAME
                , MISSINGIX.THE_COUNT
   FROM
        (SELECT  FKCOUNT.THE_RELNAME
               , FKCOUNT.THE_TABLE
               , FKCOUNT.THE_COUNT
         FROM
          (SELECT      FK.RELNAME             AS THE_RELNAME
                     , FK.TBNAME      AS THE_TABLE
                     , COUNT(*)       AS THE_COUNT
           FROM       SYSIBM.SYSFOREIGNKEYS  FK
           WHERE      FK.CREATOR            = '&creator'
           GROUP BY FK.RELNAME, FK.TBNAME
           )                    AS  FKCOUNT
         WHERE  THE_RELNAME||THE_TABLE||CHAR(THE_COUNT)  NOT IN
            (SELECT THE_RELNAME||THE_TABLE||CHAR(THE_COUNT) FROM
                 (SELECT    FK.RELNAME         AS THE_RELNAME
                                 ,  FK.TBNAME  AS THE_TABLE
                                 ,  IX.NAME
                                 ,  COUNT(*)   AS THE_COUNT
                  FROM       SYSIBM.SYSINDEXES       IX
                                  ,  SYSIBM.SYSKEYS   KY
                                  ,  SYSIBM.SYSFOREIGNKEYS  FK
                  WHERE      IX.CREATOR           =    '&creator'
                  AND           KY.IXCREATOR     =    '&creator'
                  AND           FK.CREATOR       =    '&creator'
                  AND           IX.NAME          =    KY.IXNAME
                  AND           IX.TBNAME        =    FK.TBNAME
                  AND           KY.COLNO         =    FK.COLNO
                  AND           KY.COLSEQ        =    FK.COLSEQ
                  GROUP BY FK.RELNAME, FK.TBNAME, IX.NAME
                 )                  AS  IXCOUNT
              WHERE    FKCOUNT.THE_RELNAME  = IXCOUNT.THE_RELNAME
              AND      FKCOUNT.THE_TABLE    = IXCOUNT.THE_TABLE
              AND      FKCOUNT.THE_COUNT    = IXCOUNT.THE_COUNT
            )
        ) AS MISSINGIX
       ,     SYSIBM.SYSFOREIGNKEYS FKDETAIL
   WHERE   FKDETAIL.CREATOR = '&creator'
   AND     FKDETAIL.TBNAME  = MISSINGIX.THE_TABLE
   AND     FKDETAIL.RELNAME  = MISSINGIX.THE_RELNAME
   ORDER BY
                    MISSINGIX.THE_TABLE
                  , MISSINGIX.THE_RELNAME
                  , FKDETAIL.COLSEQ
                  , FKDETAIL.COLNAME
```

## Columns participating in primary key and foreign keys

```
SELECT C.TBNAME, C.NAME, R.REFTBNAME, R.RELNAME
FROM
   SYSIBM.SYSCOLUMNS C,
   SYSIBM.SYSRELS R,
   SYSIBM.SYSFOREIGNKEYS F
WHERE
       C.TBCREATOR = &creator
   AND F.CREATOR = C.TBCREATOR
   AND F.TBNAME = C.TBNAME
   AND F.COLNAME = C.NAME
   AND F.CREATOR = R.CREATOR
   AND F.TBNAME = R.TBNAME
   AND F.RELNAME = R.RELNAME
   AND C.KEYSEQ > 0
ORDER BY C.TBNAME, C.NAME
```

## What are the quiesce points for the given table spaces?

```
SELECT TIMESTAMP
   FROM SYSIBM.SYSCOPY
   WHERE ICTYPE = 'Q'
```

```
        AND
        (DBNAME = &dbname1
        AND TSNAME IN (&tsname11, &tsname12, &tsname13, ...)
        OR
            DBNAME = &dbname2
        AND TSNAME IN C(&tsname21, &tsname22, &tsname23, ...)
        )
  GROUP BY TIMESTAMP
  ORDER BY TIMESTAMP
```

In this example, the query has a list of databases (dbname) and a list of table spaces for each of the databases (&tsname1, &tsname2, &tsname3, and so forth). The assumption is that these lists make up a complete table space set.

The query produces a list of timestamps of quiesce points that include all the table spaces of the set.

## What is the scope of CHECK-pending for the tables in a table space?

```
SELECT NAME. CHECKFLAG. HEX(CHECK5RID)
FROM SYSIBM.SYSTABLES
WHERE TSNAME = table space name
ORDER BY NAME
```

This example gives a list of all the tables in a table space with their CHECK-pending scope.

## Which tables are the ancestors?

This example gives the parents and the parent's parents for a specific table. Expect the response time to be dependent on the complexity of the RI structure.

```
 WITH RISET (LEVEL, TBNAME, REFTBNAME, ENFORCED) AS
   (
     SELECT 1, ROOT.TBNAME, ROOT.REFTBNAME, ROOT.ENFORCED
     FROM SYSIBM.SYSRELS ROOT
     WHERE ROOT.TBNAME = &yourtable
     AND   ROOT.ENFORCED IN ('Y','N')
     UNION ALL
     SELECT PARENT.LEVEL + 1, CHILD.TBNAME, CHILD.REFTBNAME,
     PARENT.ENFORCED
     FROM   RISET PARENT, SYSIBM.SYSRELS CHILD
     WHERE PARENT.REFTBNAME = CHILD.TBNAME
     AND   PARENT.ENFORCED IN ('Y','N')
     AND   PARENT.LEVEL < 15
   )
SELECT DISTINCT REFTBNAME AS PARENT , LEVEL, TBNAME AS CHILD
FROM RISET
```

## Which tables are the descendents?

This example gives the children and the child's children for a specific table.

```
WITH RISET (LEVEL, REFTBNAME, TBNAME, ENFORCED) AS
   (
     SELECT 1, ROOT.REFTBNAME, ROOT.TBNAME, ROOT.ENFORCED
     FROM SYSIBM.SYSRELS ROOT
     WHERE ROOT.REFTBNAME = &yourtable
     AND   ROOT.ENFORCED IN ('Y','N')
     UNION ALL
     SELECT PARENT.LEVEL + 1, CHILD.REFTBNAME, CHILD.TBNAME,
     PARENT.ENFORCED
     FROM   RISET PARENT, SYSIBM.SYSRELS CHILD
     WHERE PARENT.TBNAME = CHILD.REFTBNAME
     AND   PARENT.ENFORCED IN ('Y','N')
     AND   PARENT.LEVEL < 15
   )
SELECT DISTINCT TBNAME AS CHILD, LEVEL, REFTBNAME AS PARENT
FROM RISET
```

### The whole family

This example gives the parents and the parent's parent, and the child and child's child for a specific table. Expect the response time to be dependent on the complexity of the RI structure.

```
WITH RISET (LEVEL, REFTBNAME, TBNAME, ENFORCED) AS
   (
     SELECT 1, ROOT.REFTBNAME, ROOT.TBNAME, ROOT.ENFORCED
     FROM SYSIBM.SYSRELS ROOT
     WHERE ROOT.REFTBNAME = 'T16'
     AND   ROOT.ENFORCED IN ('Y','N')
     UNION ALL
     SELECT 1, ROOT.TBNAME, ROOT.REFTBNAME, ROOT.ENFORCED
     FROM SYSIBM.SYSRELS ROOT
     WHERE ROOT.REFTBNAME = &table
     AND   ROOT.ENFORCED IN ('Y','N')
     UNION ALL
     SELECT PARENT.LEVEL + 1, CHILD.REFTBNAME, CHILD.TBNAME,
     PARENT.ENFORCED
     FROM   RISET PARENT, SYSIBM.SYSRELS CHILD
     WHERE PARENT.TBNAME = CHILD.REFTBNAME
     AND   PARENT.ENFORCED IN ('Y','N')
     AND   PARENT.LEVEL <  &level
     UNION ALL
     SELECT PARENT.LEVEL + 1, CHILD.TBNAME, CHILD.REFTBNAME,
     PARENT.ENFORCED
     FROM   RISET PARENT, SYSIBM.SYSRELS CHILD
     WHERE PARENT.REFTBNAME = CHILD.TBNAME
     AND   PARENT.ENFORCED IN ('Y','N')
     AND   PARENT.LEVEL < &level
   )
SELECT DISTINCT REFTBNAME AS PARENT, LEVEL, TBNAME AS CHILD
FROM RISET
```

## 3.13.3 Constraints and multilevel security

Constraints operate in an multilevel-secure environment in the following ways:

- A unique constraint is allowed on a security label column.
- A referential constraint is not allowed on a security label column.
- A check constraint is not allowed on a security label column.

Multilevel security with row-level checking is not enforced when DB2 checks a referential constraint. Although a referential constraint is not allowed for the security label column, DB2 enforces referential constraints for other columns in the table that are not defined with a security label.

4

# Triggers

Triggers provide a very flexible and powerful mechanism for ensuring data integrity. In general, triggers provide everything that constraints, views with check option, and RI provide and much more. They are also generally more expensive.

If you are thoroughly familiar with the process of creating and maintaining triggers but are looking for some templates that may fit your business needs, 4.15, "Common business scenarios" on page 137 should be of special interest to you.

In this chapter, we discuss the following topics:

- Why use triggers for data integrity
- Trigger terminology
- Extending trigger functionality with UDFs and stored procedures
- Invoking UDFs and stored procedures
- Passing parameters to UDFs and stored procedures
- Raising error conditions
- Handling errors during execution
- Auditing versus mass replication
- Impact of LOAD utility
- Declarative RI versus triggers
- Execution sequence of multiple triggers
- Trigger cascading
- Interactions between triggers and other integrity checks
- Creating triggers to obtain consistent results
- Common business scenarios

## 4.1 Why use triggers for data integrity

Triggers are sets of SQL statements that execute when a certain event occurs in a DB2 table. Like constraints, triggers can be used to control changes in DB2 tables. However, they are more powerful because they can monitor a broader range of actions than constraints can.

Let us consider an example. A constraint can disallow an update to the salary column of the employee table if the new value exceeds a certain pre-specified amount. A trigger can monitor the amount by which the salary changes, as well as the salary value. It can do so conditionally depending on the value of a performance rating *stored in another table.* In addition, it can call user programs (stored procedures and user defined functions) that take additional action or modify the data before the initial update occurs. See "Extending triggers with UDFs and stored procedures" on page 114 for details.

## 4.2 Trigger terminology

In this section, we provide a quick brief summary of various terms related to triggers that are used in this chapter.

| | |
|---|---|
| **Trigger** | A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table. |
| **Trigger activation time** | An indication in the trigger definition of whether the trigger should be activated before or after the triggered event. |
| **Trigger body** | The set of SQL statements that is executed when a trigger is activated and its triggered action condition is true. |
| **Triggered action** | The SQL logic that is performed when a trigger is activated. |
| **Triggered action condition** | An optional part of the triggered action, it appears as a WHEN clause. |
| **Trigger granularity** | Determines whether the trigger is activated once per statement or once per row that is affected by that statement. |
| **Triggering event** | The specified operation in a trigger definition that causes the activation of that trigger. It is either INSERT, UPDATE, or DELETE. |
| **Transition variable** | A variable that contains a column value of the affected row. This can reference the set of old values or new values. |
| **Transition table** | A temporary table that contains all the affected rows of the subject table. These can reference the set of old values or new values. |

## 4.3 Extending triggers with UDFs and stored procedures

There are two common uses of triggers:

- Data validation
- Data propagation

### 4.3.1 Data validation

Data validation deals with invoking complex business logic to determine whether a certain action (INSERT, UPDATE, or DELETE) should be permitted.

This is typically achieved with a user defined function (UDF) invoked by a *before trigger*.

**Important:** For data validation, the trigger must act on the return code set by the invoked routine. This is **not** possible with stored procedures but only with UDFs.

Figure 4-1 shows how a UDF can be used for data validation.

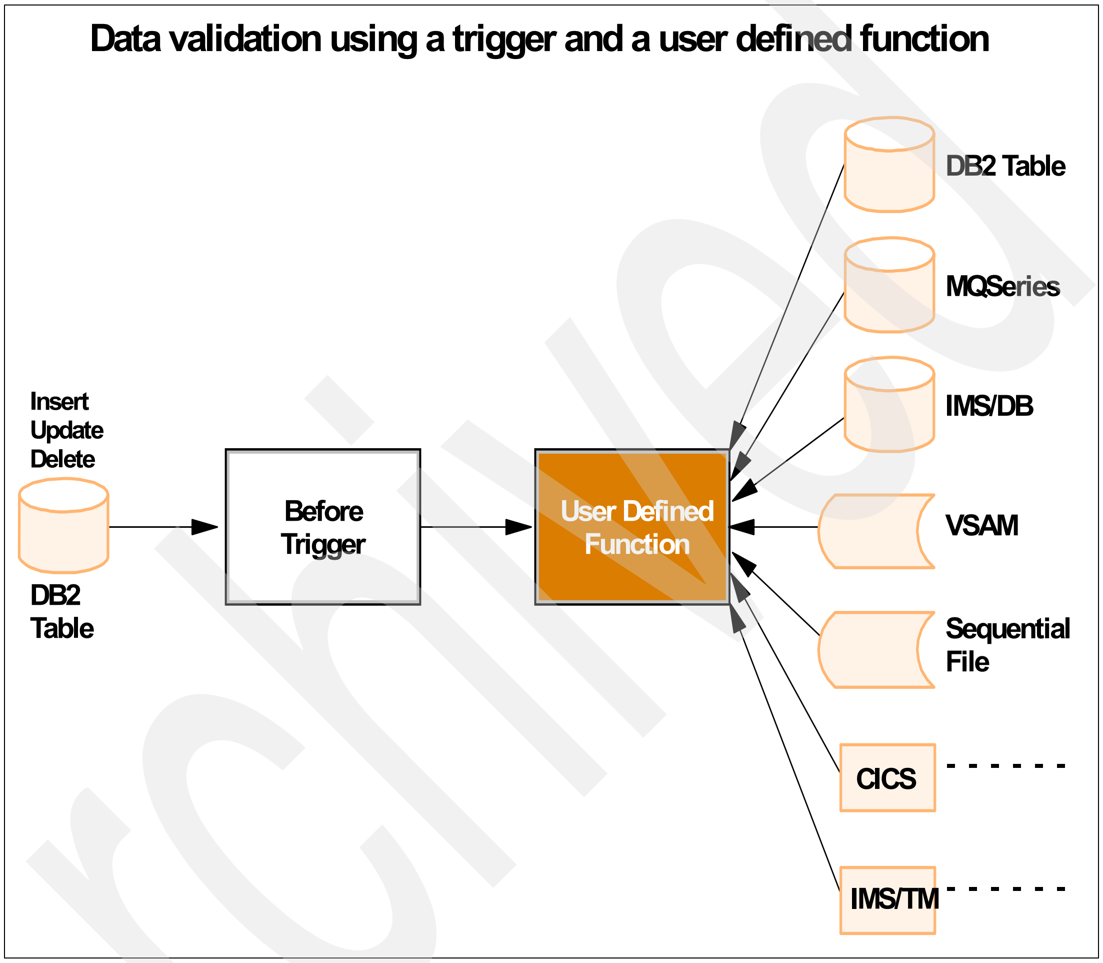


*Figure 4-1 Data validation using a trigger and a UDF*

### 4.3.2 Data propagation

Data propagation deals with invoking complex business logic after a certain action has taken place. The most efficient technique for data propagation is to use the log or check constraints, if they can do the job.

In the case where triggers are suitable, typically the solution is a stored procedure invoked by an after trigger.

Figure 4-2 shows how a stored procedure can be used for data propagation.

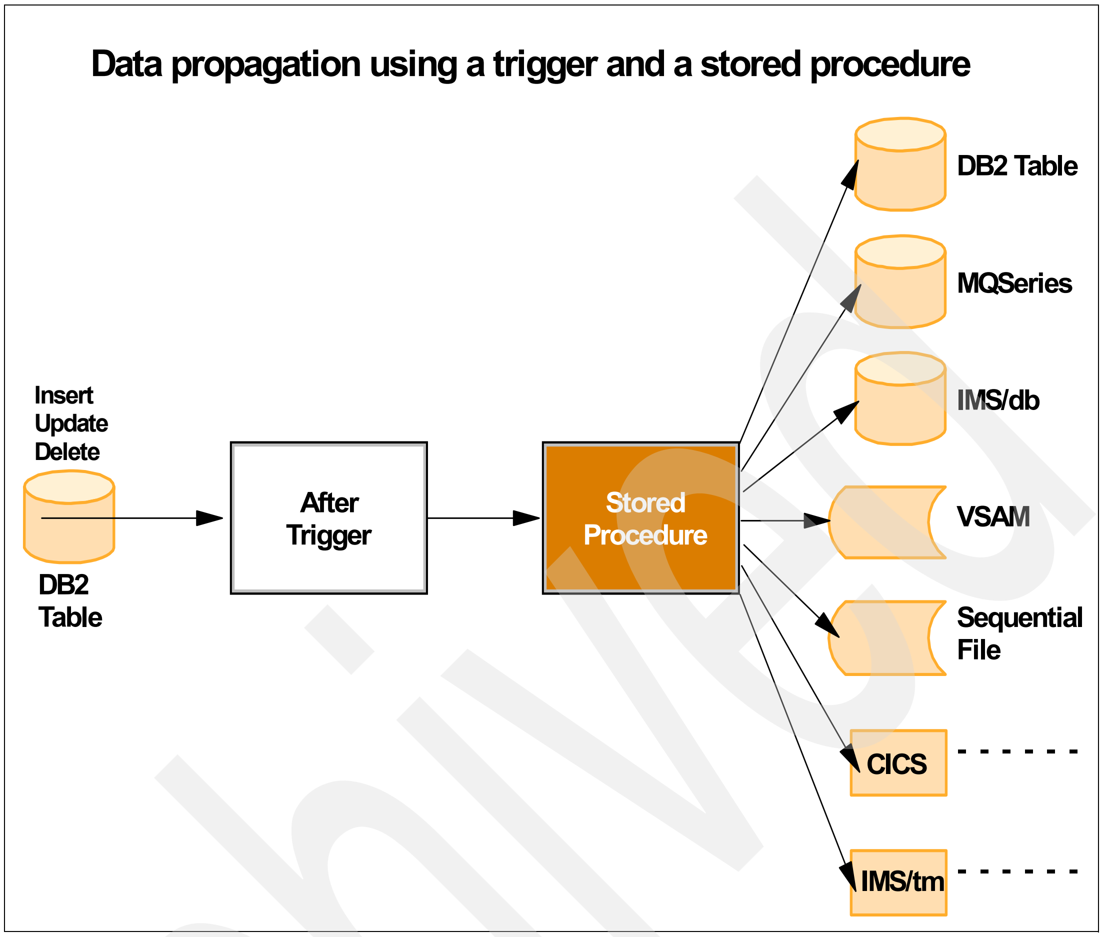


*Figure 4-2 Data propagation using a trigger and a stored procedure*

## 4.4 Invoking UDFs and stored procedures

In the context of data integrity, you can call UDFs and stored procedures for complex data validation. There are two means of invoking a UDF and only one way of invoking a stored procedure from within a trigger body. We discuss these ways in this section.

### 4.4.1 Using the VALUES statement

Example 4-1 shows how the VALUES statement invokes a UDF.

*Example 4-1 Using the VALUES statement*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
     VALUES(VALSAL(O.EMP_NO,O.SALARY,N.SALARY));
END #
```

When specifying a UDF as part of a VALUES expression, the UDF is invoked. However, if a negative SQLCODE is returned (from the UDF), DB2 stops executing the trigger and rolls back any triggered actions that were performed.

### 4.4.2 Using the SELECT statement

Example 4-2 shows how the SELECT statement invokes a UDF.

*Example 4-2 Using the SELECT statement*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  SELECT
    CASE
WHEN VALSAL(N.EMP_NO,O.SALARY,N.SALARY) = '1'
             THEN COALESCE(RAISE_ERROR
                   ('75001','SOME MESSAGE1'),' ')
....
    END
  FROM SYSIBM.SYSDUMMY1;
END #
.
```

Using the SELECT, you can also access the return codes and display the appropriate error message, as shown in 4.6, “Raising error conditions” on page 120. This type of conditional processing is impossible with VALUES.

### 4.4.3 Using the CALL statement

Example 4-3 shows how the CALL statement invokes a stored procedure.

*Example 4-3 Using the CALL statement*

```
CREATE TRIGGER DEVL7111.EMPTRIG1
AFTER UPDATE OF SALARY ON DEVL7111.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.10)
BEGIN ATOMIC
 CALL   DEVL7111.EMPAUDTS(N.EMPNO,O.SALARY,N.SALARY);
END#
```

## 4.5 Passing parameters to UDFs and stored procedures

The triggered action (stored procedure or UDF) can refer to the values in the set of affected rows. This is supported through the use of transition variables and transition tables. Transition variables refer to the values of a single row, and this is discussed in 4.5.1, “Using transition variables” on page 118. Transition tables refer to the complete set of values of all affected rows, and this is discussed in 4.5.2, “Using transition tables” on page 119.

Table 4-1 summarizes the allowable combinations of transition variables and transition tables that you can specify for the various trigger types.

*Table 4-1 Allowable combination of attributes in a trigger definition*

| **Granularity** | **Activation time** | **Triggering SQL operation** | **Transition variables allowed** | **Transition tables allowed** |
|---|---|---|---|---|
| FOR EACH ROW | BEFORE | INSERT | NEW | - |
| | | UPDATE | OLD, NEW | |
| | | DELETE | OLD | - |
| | AFTER | INSERT | NEW | NEW TABLE |
| | | UPDATE | OLD, NEW | OLD TABLE, NEW TABLE |
| | | DELETE | OLD | OLD TABLE |
| FOR EACH STATEMENT | BEFORE | INSERT | - | - |
| | | UPDATE | - | - |
| | | DELETE | - | - |
| | AFTER | INSERT | - | NEW TABLE |
| | | UPDATE | - | OLD TABLE, NEW TABLE |
| | | DELETE | - | OLD TABLE |

## 4.5.1 Using transition variables

Transition variables are similar to host variables in their behavior. A transition variable can be referenced in the search-condition of a triggered SQL statement of the triggered action wherever a host variable is allowed in the statement if the statement was issued outside the body of a trigger. They use the names of the columns in the subject table qualified by a specific name that identifies whether the reference is to the old value (before the update) or to the new value (after the update). In Example 4-4, we use "O" and "N" as the qualifiers to designate the before and after values.

*Example 4-4 Trigger with before and after values*

```
CREATE TRIGGER DEVL7111.EMPTRIG1
AFTER UPDATE OF SALARY ON DEVL7111.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.10)
BEGIN ATOMIC
 CALL   DEVL7111.EMPAUDTS(N.EMPNO,O.SALARY,N.SALARY);
END#
```

The list of parameters must be compatible with the parameter list defined in the linkage section of the stored procedure and the procedure division statement. For the sample stored procedure EMPAUDTS, the linkage section looks like this in COBOL:

```
   LINKAGE SECTION.
```

```
01  PEMPNO        PIC X(6).
01  POLDSALARY    PIC S9(7)V9(2) COMP-3.
01  PNEWSALARY    PIC S9(7)V9(2) COMP-3.
```

The procedure division for the stored procedure looks like this in COBOL:

```
PROCEDURE DIVISION USING PEMPNO, POLDSALARY, PNEWSALARY.
```

## 4.5.2 Using transition tables

Transition tales are also similar to host variables in their behavior. The name of the transition table can be referenced in a triggered SQL statement of the triggered action wherever a table name is allowed in the statement if the statement was issued outside the body of a trigger. The name of the table can be specified in the search condition of a triggered SQL statement of the triggered action wherever a column name is allowed in the statement if the statement was issued outside the body of a trigger.

Transition tables are read-only. Like transition variables, transition tables also use the names of the columns of the subject table, but have a name specified that allows the complete set of affected rows to be treated as a table.

In Example 4-5, we use “OT” and “NT” as the qualifiers to designate the before and after table values.

*Example 4-5 Trigger with transition tables*

```
CREATE TRIGGER DEVL7111.EMPTRIG3
AFTER  UPDATE OF SALARY ON DEVL7111.EMP
REFERENCING OLD TABLE AS OT
            NEW TABLE AS NT
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
 CALL DEVL7111.EMPPROPS(TABLE OT,TABLE NT);
END#
```

We now describe how to access transition tables in a stored procedure, but the same applies to a UDF.

To access transition tables in a stored procedure, use table locators which are pointers to the transition tables. You declare table locators as input parameters in the CREATE PROCEDURE statement using the TABLE LIKE *table-name* AS LOCATOR clause. See Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426, for more information.

The five basic steps to accessing transition tables in a stored procedure are:

1. Declare input parameters to receive table locators. You must define each parameter that receives a table locator as an unsigned 4-byte integer. This is shown in Example 4-6 for COBOL. This step is optional and it is required only if you plan to use the locator later in the program and need to save it. In general, for COBOL you can use the locators from the LINKAGE SECTION directly.

*Example 4-6 Declaring input variables for table locators*

```
01 WS-TRIG-TBL-ID-OLD SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
```

2. Declare table locators. The syntax varies with the application language. See Chapter 9, “Embedding SQL statements in host languages” of the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415, for information about the syntax for

C, C++, COBOL, and PL/I. See Chapter 6 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426, for information about the syntax for SQL procedures. This is shown in Example 4-7.

*Example 4-7 Declaring table locators*

```
LINKAGE SECTION.
01 TRIG-TBL-ID-OLD SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
01 TRIG-TBL-ID-NEW SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
```

3. Declare a cursor to access the rows in each transition table. This is shown in Example 4-8.

*Example 4-8 Declaring a cursor*

```
****      CURSOR FOR RETRIEVING “BEFORE” AND “AFTER” IMAGES
          EXEC SQL DECLARE C1
            CURSOR FOR
            SELECT
                OLDTAB.EMPNO
            , OLDTAB.SALARY
            , NEWTAB.SALARY
              FROM TABLE(:TRIG-TBL-ID-OLD LIKE EMP) AS OLDTAB
                 , TABLE(:TRIG-TBL-ID-NEW LIKE EMP) AS NEWTAB
              ORDER BY EMPNO
          END-EXEC.
```

4. Assign the input parameter values to the table locators. This is shown in Example 4-9.

*Example 4-9 Setting values of table locators*

```
PROCEDURE DIVISION USING TRIG-TBL-ID-OLD, TRIG-TBL-ID-NEW.
```

5. Access rows from the transition tables using the cursors that are declared for the transition tables. This is shown in Example 4-10.

*Example 4-10 Accessing the transition tables*

```
EXEC SQL
   OPEN  C1
END-EXEC.
...
EXEC SQL
  FETCH C1
  INTO  :WS-EMPNO
      , :WS-OLDSALARY
      , :WS-NEWSALARY
END-EXEC.
...
EXEC SQL
  CLOSE C1
END-EXEC.
```

## 4.6 Raising error conditions

You can return different SQLSTATEs and error messages to the calling application by using the RAISE_ERROR statement. This is done in the trigger body as shown in Example 4-11.

*Example 4-11 Generating error messages in a trigger body*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
   WHEN (N.SALARY > O.SALARY *
                     (SELECT MAX_RAISE
                      FROM DEPT D
                      WHERE D.DEPT_NO = O.DEPT_NO))
BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('CANNOT EXCEED DEPT LIMIT')
;
END
```

Example 4-12 shows how you can use the result of a UDF to return different error messages to the calling application.

*Example 4-12 Generating error messages in a trigger after calling a UDF*

```
CREATE TRIGGER DEVL7111.EMPTRIG2 NO CASCADE
BEFORE UPDATE OF SALARY ON DEVL7111.EMP
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
SELECT
  CASE
      WHEN DEVL7111.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '1' THEN COALESCE(RAISE_ERROR('75001','some message1'),' ')
      WHEN DEVL7111.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '2' THEN COALESCE(RAISE_ERROR('75002','some message2'),' ')
      WHEN DEVL7111.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '3' THEN COALESCE(RAISE_ERROR('75003','some message3'),' ')
      WHEN DEVL7111.EMPAUDTU(N.EMPNO,O.SALARY,N.SALARY)
          =  '4' THEN COALESCE(RAISE_ERROR('75004','some message4'),' ')
  END
FROM SYSIBM.SYSDUMMY1;

END#
```

The SQLSTATE value specified in the SIGNAL SQLSTATE statement must conform to the following rules:

- Each character must be numeric (0 through 9) or uppercase alphabetic (A through Z).
- The SQLSTATE class (first two characters) cannot be 00, 01, or 02, because these are not error classes.
- If the SQLSTATE class starts with 0 through 6, or A through H, then the subclass (last three characters) must start with a letter in the range I through Z.
- If the SLQSTATE class starts with 7 through 9, or I through Z, then the subclass (last three characters) can be any of 0 through 9, or A through Z.

Note that while the SQLSTATE is user-defined, the SQLCODE returned to the applications is always -438 as shown in Example 4-13.

Example 4-13   User-defined SQLSTATE and error message

```
DSNT408I SQLCODE = -438, ERROR:  APPLICATION RAISED ERROR WITH DIAGNOSTIC
         TEXT: TRIGGER4 DEPT NAME UPDATE NOT ALLOWED
DSNT418I SQLSTATE   = 7TR04 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXRTYP SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = 1 0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'00000001'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

## 4.7 Handling errors during execution

Severe errors that occur during the execution of a trigger SQL statement are returned with SQLCODE -901, -906, -911, and -913 (along with the corresponding SQLSTATEs). Non-severe errors raised by a triggered SQL statement through the SIGNAL SQLSTATE statement or an SQL statement containing a RAISE_ERROR function are returned with the specified SQLSTATE, and the SQLCODE is always -438. Other non-severe errors are returned with an SQLCODE -723 and SQLSTATE 09000. Warnings are not returned.

The ability to handle errors in a trigger is severely limited, especially when it calls a stored procedure. For this reason, you must decide carefully whether you use a stored procedure or a UDF. This has been discussed in 4.3, “Extending triggers with UDFs and stored procedures” on page 114.

## 4.8 Auditing versus mass replication

For auditing changes to sensitive data, the main options are audit traces and the use of the log data.

Triggers can be used effectively since they are fired regardless of the invoking application (that is, ad hoc updates are captured as well).

However, when the amount of data to be replicated is large, replication is generally the best way to address this need. The cost of capturing the change then becomes asynchronous (as opposed to triggers that are synchronous), since it is performed by the capture program. The subsequent invocation of the receiving process can be achieved by opting for the MQ-based replication architecture. Using the MQ-based configuration capture performs the put of the changes upon processing of the commit. Refer to Q Replication manuals and *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487, for information.

## 4.9 Impact of LOAD utility

The impact of running a LOAD utility against a table with triggers defined on it depends on the SHRLEVEL parameter of the utility. SHRLEVEL NONE (the default and most commonly used option) does not activate any triggers defined on the table. SHRLEVEL CHANGE load utility functions like a mass INSERT SQL operation. All triggers (before and after, row and statement) are activated when using this option.

You must be aware of how triggers and referential constraints interact since they impact the order of processing. See 4.13, “Interactions among triggers and other integrity checks” on page 130 for details.

## 4.10 DB2-enforced RI versus triggers

If you want to enforce RI in DB2, the most efficient means of doing so is by using the DB2-enforced declarative RI which is discussed in detail in Chapter 3, “Referential integrity” on page 49. You should consider using triggers for this purpose only if the straightforward means of doing so does not meet your business needs. For example, to implement an “either-or” relationship where the foreign key value must exist on at least one of a set of parent tables (see 4.15.8, “Enforcing multiple parent RI” on page 146), you must use triggers since the foreign key can reference a single parent table only. Triggers add complexity and should be used only when necessary for this purpose. In addition, they also add overhead compared to declarative RI.

In order to compare the performance of DB2-enforced RI, triggers, stored procedures, and UDFs, we create a test scenario. We create two tables and related indexes as shown in Example 4-14.

*Example 4-14 Table and index definition for test scenario*

```
CREATE TABLE BENDEPT
    (DEPTID   SMALLINT NOT NULL PRIMARY KEY,
     DEPTNAME CHAR(30),
     OTHERDATA1 CHAR(250),
     OTHERDATA2 CHAR(250))
IN DSNDB04.BENDEPT #

CREATE TABLE BENEMPL
    (EMPLID   INTEGER NOT NULL,
     DEPTID   SMALLINT,
     EMPLNAME CHAR(30),
     OTHERDATA1 CHAR(250),
     OTHERDATA2 CHAR(250))
   IN DSNDB04.BENEMPL #

CREATE UNIQUE INDEX BENDEPTK0
  ON BENDEPT
   (DEPTID               ASC)
...

CREATE UNIQUE INDEX BENEMPLK0
  ON BENEMPL
   (EMPLID               ASC)
...

CREATE        INDEX BENEMPLK1
  ON BENEMPL
   (DEPTID               ASC)
```

We populated the BENDEPT table with 1,000 departments and the BENEMPL with 500 employees in each department. We tested the insert to the child table (BENEMPL) and the the delete from the parent table (BENDEPT) under the following three cases:

- Using DB2-enforced RI
- Using a trigger to enforce the RI
- Using a trigger and a UDF or stored procedure to enforce the RI

The trigger DDL for verification during insert to the child table is shown in Example 4-15.

*Example 4-15 Trigger definition for insert verification*

```
CREATE TRIGGER TR2A NO CASCADE
BEFORE INSERT ON BENEMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (0 = (SELECT COUNT(*)
           FROM  BENDEPT
           WHERE DEPTID = N.DEPTID))
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR2A'
        ('TR2A VALID DEPT IS REQUIRED');
END #
```

When the same verification is made in a UDF, the DDL for the trigger and UDF is shown in Example 4-16.

*Example 4-16 UDF for insert verification*

```
CREATE TRIGGER TR3A NO CASCADE
BEFORE INSERT ON BENEMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  SELECT
    CASE
        WHEN UDFA(N.DEPTID) = 'E'
             THEN COALESCE(RAISE_ERROR
                   ('75001','DEPT DOES NOT EXIST'),' ')
    END
  FROM SYSIBM.SYSDUMMY1;
END #

CREATE FUNCTION UDFA (SMALLINT)
RETURNS CHAR(1)
EXTERNAL NAME UDFA
LANGUAGE COBOL
PARAMETER STYLE DB2SQL
NO DBINFO
COLLID PAOLOR4
WLM ENVIRONMENT DB8AWLM1 #
```

Similarly, for the delete from parent table, the trigger DDL is:

```
CREATE TRIGGER TR2C
AFTER  DELETE ON BENDEPT
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    DELETE FROM BENEMPL
    WHERE DEPTID = O.DEPTID ;
END #
```

When the same cascading is done via a stored procedure, the DDL for the trigger and stored procedure is:

```
CREATE TRIGGER TR3C
AFTER  DELETE ON BENDEPT
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
```

```
  CALL SPC(O.DEPTID);
END #

CREATE PROCEDURE SPC
(
 IN  PDEPTID       SMALLINT
)
DYNAMIC RESULT SETS 0
EXTERNAL NAME SPC
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
COLLID PAOLOR4
WLM ENVIRONMENT DB8AWLM1 #
```

The results of the benchmarks are summarized in Table 4-2.

*Table 4-2 DB2 RI versus triggers, UDFs, and stored procedures enforcement*

| Case # | Description | Enforcement method | CPU (sec.) | Elapsed (sec.) |
|---|---|---|---|---|
| 1 | Insert 500 rows in the BENEMPL table for each of 500 departments | DB2 RI | 3.07 | 28.80 |
| | | Trigger | 6.07 | 30.00 |
| | | Trigger + UDF | 37.36 | 450.00 |
| 2 | Successful delete of 500 rows from BENDEPT table each cascading to 500 BENEMPL table rows | DB2 RI | 0.02 | 3.00 |
| | | Trigger | 2.29 | 26.40 |
| | | Trigger + SP | 2.62 | 24.00 |

As is evident from this table, RI implementation via triggers is more expensive than DB2 RI and invocation of UDFs or stored procedures for this purpose degrades the performance further. Triggers, UDFs, or stored procedures should therefore be chosen only if standard DB2 RI cannot meet the business needs.

## 4.11 Execution sequence of multiple triggers

You can create multiple triggers for the same subject table, event, and activation time. The order in which those triggers are activated is the order in which the triggers were created. DB2 records the timestamp when each CREATE TRIGGER statement executes and DB2 uses this timestamp to determine which trigger to activate first.

DB2 always activates all *before triggers* that are defined on the table before the *after triggers* that are defined on that table, but the activation order is by timestamp within each set of before triggers followed by timestamp within each set of after triggers.

In several cases the order may not matter, but, as this example shows, the results can be quite different depending on the order.

Suppose you have created an EMPL table and a DEPT table, such as:

```
CREATE TABLE EMPL
    (EMPLID   CHAR(6) NOT NULL,
     DEPTID   CHAR(3),
     SALARY   DECIMAL(9,2),
     COMMISION DECIMAL(9,2) )
   IN DSNDB04.TRIGTEST #

CREATE TABLE DEPT
    (DEPTID   CHAR(3) NOT NULL,
     DEPTNAME CHAR(30),
     BUDGET   DECIMAL(9,2))
   IN DSNDB04.TRIGTEST #
```

Consider the two triggers TRIGGER1 and TRIGGER2 defined on the EMPL table, shown in Example 4-17. Notice that both of them have the same triggering event (INSERT), the same subject table (EMP), and the same activation time (AFTER).

*Example 4-17 Execution sequence of triggers*

```
CREATE TRIGGER TRIGGER1
AFTER INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
     UPDATE EMPL
     SET COMMISION = SALARY * 0.10
     WHERE EMPLID = N.EMPLID;
END #

CREATE TRIGGER TRIGGER2
AFTER INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
     UPDATE DEPT
     SET BUDGET = BUDGET +
     (SELECT SALARY + COMMISION
      FROM   EMPL
      WHERE EMPLID = N.EMPLID)
     WHERE DEPTID = N.DEPTID;
END #
```

Let us assume we insert a new employee row with a salary of 10,000. If TRIGGER1 is activated first, the employee commission is set to 1,000 and when TRIGGER2 is activated, the department's budget is incremented by 10,000 + 1,000 = 11,000. On the other hand, if TRIGGER2 is activated first, it is incremented by 10,000 only.

**Tip:** Consider having only one trigger with the same triggering event and trigger activation time on a table (for example, only one after UPDATE trigger on a table). Alternatively, always drop and recreate all such triggers when maintaining any one of them.

Another important thing to note is that when multiple rows are updated in one statement, TRIGGER1 is executed for all affected rows followed by TRIGGER2 for all affected rows. Even though the triggers are defined as row level, the sequence of execution is NOT TRIGGER1 followed by TRIGGER2 for each row. This is explored in more detail in 4.3, "Extending triggers with UDFs and stored procedures" on page 114.

## 4.12 Trigger cascading

An SQL statement that a trigger executes might modify the subject table or other tables with triggers, so DB2 also activates those triggers. A trigger that is activated as the result of another trigger can be activated at the same level as the original trigger or at a different level. We illustrate this concept with examples.

### 4.12.1 Triggers at the same level

#### Example 1

Table X has two triggers defined on it: a before trigger A and an after trigger B. An update to table X causes both triggers A and B to activate. This is shown in Example 4-3.

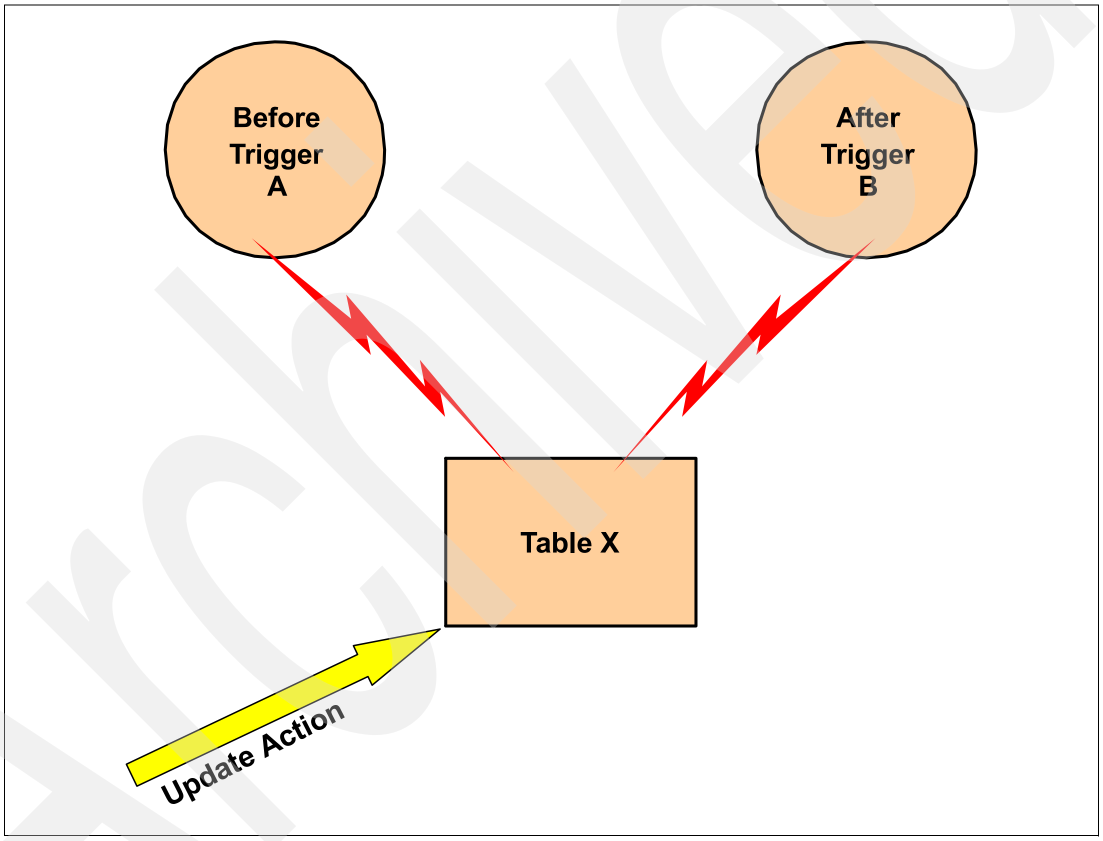


*Figure 4-3 Example one: Triggers at same level*

#### Example two

An update to table W activates after trigger A. Trigger A updates table X, which has a referential constraint with table Y, which has trigger B defined on it. The referential constraint causes table Y to be updated, which activates trigger B. This is shown in Example 4-4.

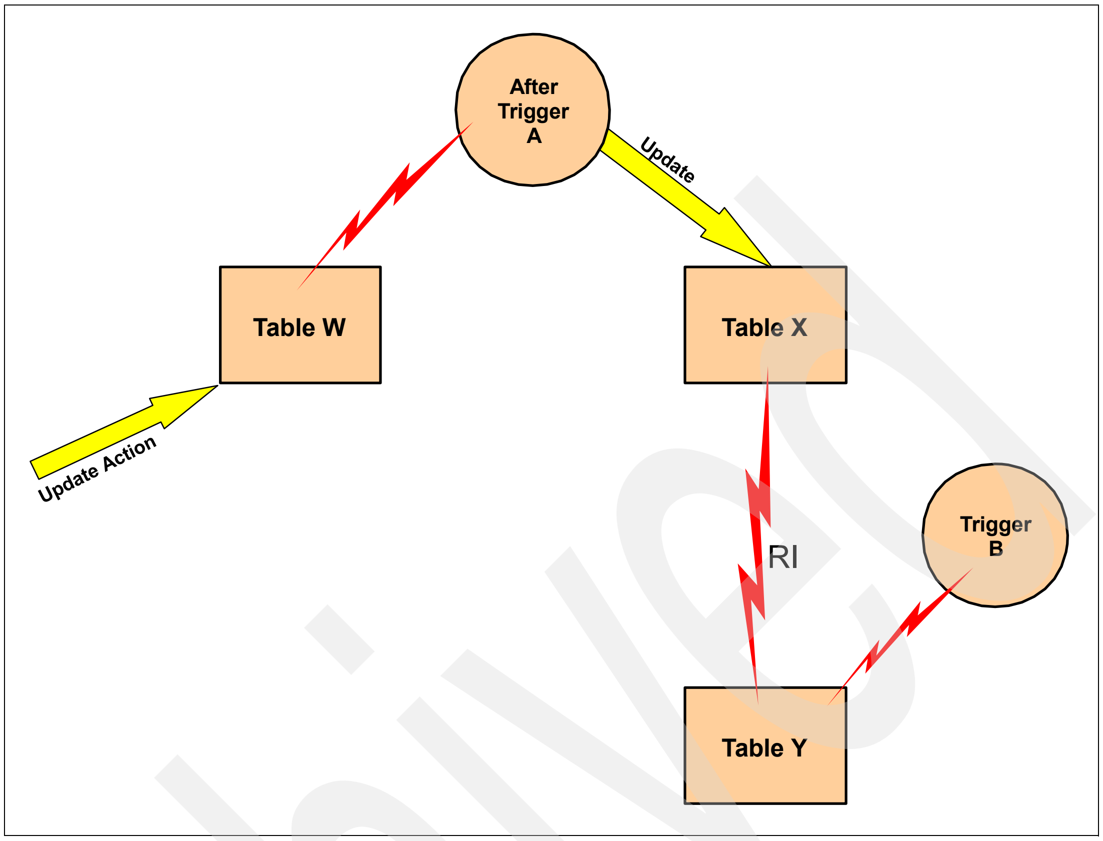


*Figure 4-4 Example two: Triggers at the same level*

### 4.12.2 Triggers at different levels

#### Example one

Trigger A is defined on table X, and trigger B is defined on table Y. Trigger B is an UPDATE trigger. An update to table X activates trigger A, which contains an UPDATE statement on table Y in its trigger body. This UPDATE statement activates trigger B. This is shown in Example 4-5.

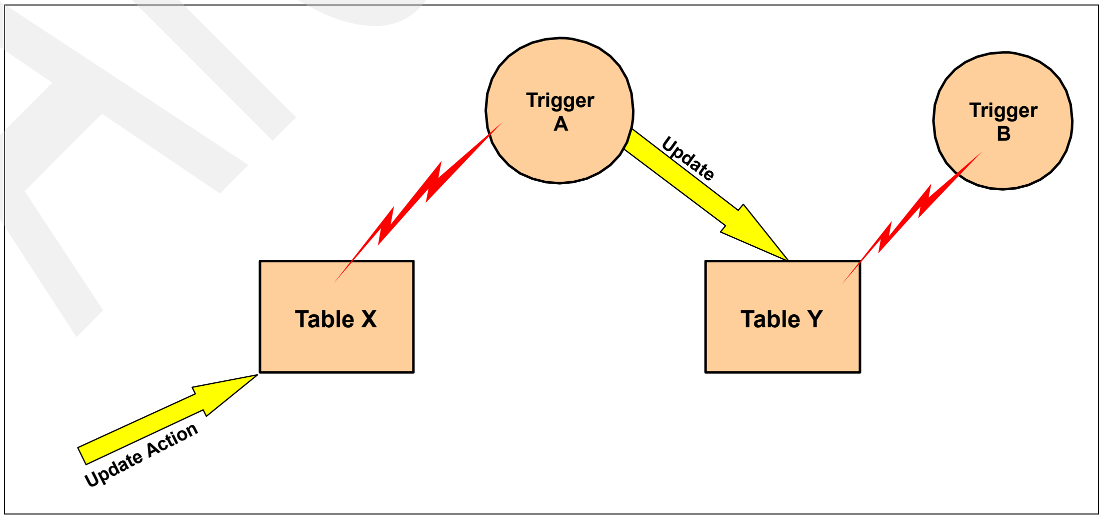


*Figure 4-5 Example one: Triggers at different levels*

## Example two

Trigger A on table W calls a stored procedure S. The stored procedure contains an INSERT statement for table X, which has an insert trigger B defined on it. When the INSERT statement on table X executes, trigger B is activated. This is shown in Example 4-6.

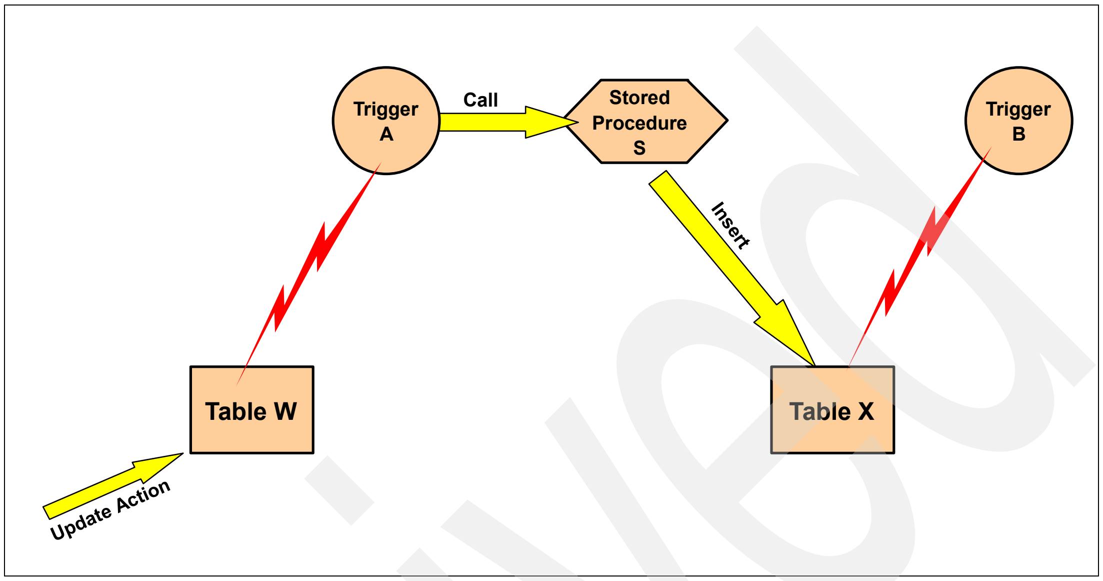


*Figure 4-6   Example two: Triggers at different levels*

When triggers are activated as different levels, it is called *trigger cascading*. Trigger cascading can occur only for after triggers because DB2 does not support cascading of before triggers as shown in Example 4-18.

*Example 4-18   Before triggers do not support cascading*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
   WHEN (N.SALARY > O.SALARY *
                     (SELECT MAX_RAISE
                      FROM DEPT D
                      WHERE D.DEPT_NO = O.DEPT_NO))
BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('CANNOT EXCEED DEPT LIMIT')
;
END
```

To prevent the possibility of an endless loop resulting from trigger cascading, DB2 supports only 16 levels of cascading. This includes triggers, stored procedures, and UDFs. If a trigger, stored procedure, or UDF at the 17th level is activated, DB2 returns SQLSTATE 54038 (SQLCODE -724) and rollbacks all SQL changes in the 16 levels of cascading. The error message is:

```
DSNT408I SQLCODE = -724, ERROR:  THE ACTIVATION OF THE TRIGGER OBJECT
         TRIGGER17 WOULD EXCEED THE MAXIMUM LEVEL OF INDIRECT SQL CASCADING
DSNT418I SQLSTATE   = 54038 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXESTS SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = -724 0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'FFFFFD2C'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

## 4.13 Interactions among triggers and other integrity checks

When you create triggers, you need to understand the interactions among the triggers and constraints on your tables, and the effect that the order of processing of those constraints and triggers can have on the results.

In general, the following steps occur when triggering SQL statement S1 performs an insert, update, or delete operation on table T1:

1. DB2 determines the rows of T1 to modify. We designate this set M1. The contents of M1 depend on the SQL operation:
   - For a delete operation, all rows that satisfy the search condition of the statement for a searched delete operation, or the current row for a positioned delete operation
   - For an insert operation, the row identified by the VALUES clause, or the rows identified by the result table of a SELECT clause within the INSERT statement
   - For an update operation, all rows that satisfy the search condition of the statement for a searched update operation, or the current row for a positioned update operation
2. DB2 processes all before triggers that are defined on T1 in order of creation. Each before trigger executes the triggered action once for each row of M1. If M1 is empty, the triggered action does not execute.

   If an error occurs when the triggered action executes, DB2 rolls back all changes that are made by S1.
3. DB2 makes the changes that are specified in statement S1 to table T1. If an error occurs, DB2 rolls back all changes that are made by S1.
4. If M1 is not empty, DB2 applies all the following constraints and checks that are defined on table T1:
   - Referential constraints
   - Check constraints
   - Checks that are due to updates of the table through view defined with the WITH CHECK OPTION clause

   Application of referential constraints with rules of DELETE CASCADE or DELETE SET NULL is activated before delete triggers or before update triggers on the dependent tables.

   If any constraint is violated, DB2 rolls back all changes that are made by constraint actions or by statement S1.
5. DB2 processes all after triggers that are defined on table T1 and all after triggers on tables that are modified as the result of referential constraint actions, in the order of creation.

   Each after row trigger executes the triggered action once for each row of M1. If M1 is empty, the triggered action does not execute.

   Each after statement trigger executes the triggered action once for each execution of S1, even if M1 is empty.

If any triggered actions contain SQL insert, update, or delete operations, DB2 repeats steps 1 through 5 for each operation.

This is summarized in Figure 4-7.

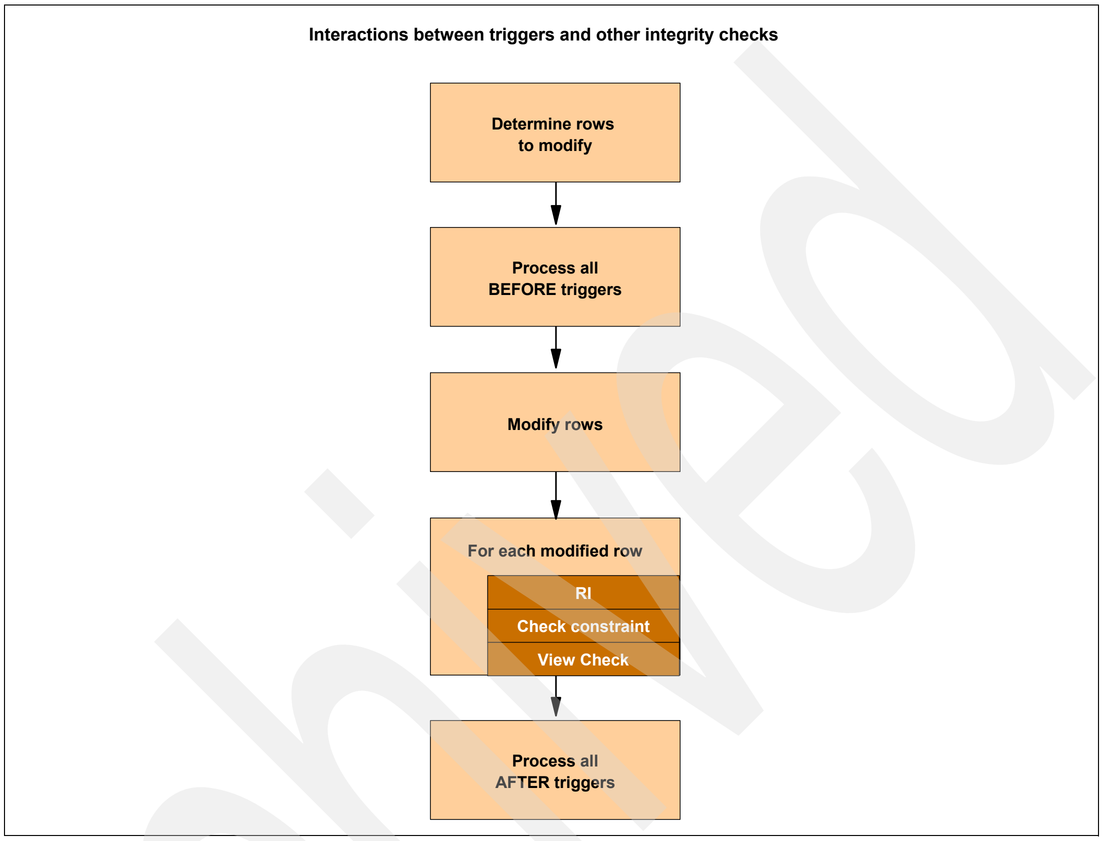

Interactions between triggers and other integrity checks
Determine rows to modify
Process all BEFORE triggers
Modify rows
For each modified row
RI
Check constraint
View Check
Process all AFTER triggers

*Figure 4-7 Interaction between triggers and other integrity checks*

## 4.14 Creating triggers to obtain consistent results

When you create triggers and write SQL statements that activate those triggers, you need to ensure that executing those statements on the same set of data always produces the same results. In this section, we discuss three common business scenarios where this does not hold true.

### 4.14.1 Effect of an uncorrelated subquery

DB2 evaluates an uncorrelated subquery only once per execution of the statement. This fact can create some undesirable effects when the table being updated has a trigger that modifies a table that is accessed in the subquery. The following example illustrates this case.

You have created a RESERVATION table and a SEATS table, such as:

```
   CREATE TABLE RESERVATION
       (REQUESTNO   INTEGER  NOT NULL,
        SEATNO      CHAR(4),
        CONFIRMFLAG CHAR(1))
      IN DSNDB04.TRIGTEST ;

   CREATE TABLE SEATS
       (SEATNO      CHAR(4)  NOT NULL,
        AVAILFLAG   CHAR(1))
      IN DSNDB04.TRIGTEST ;
```

You have also created a trigger on the RESERVATION table to mark the seat as unavailable once the reservation request has been processed, as follows:

```
CREATE TRIGGER TRIGGER1
AFTER UPDATE ON RESERVATION
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE SEATS
   SET AVAILFLAG = 'N'
   WHERE SEATNO = N.SEATNO ;
END #
```

Now an application processes the reservation requests one at a time and updates the CONFIRMFLAG to Y to indicate that the seat has been reserved. The cursor declaration for this process looks similar to this:

```
DECLARE C1 CURSOR FOR
SELECT * FROM RESERVATION
WHERE SEATNO IN
          (SELECT SEATNO
           FROM   SEATS
           WHERE  AVAILFLAG = 'Y') ;
```

Since DB2 evaluates an uncorrelated subquery only once, the results of the trigger are not known as we process through each row of the cursor after it is fetched. This means that a second request for the same seat will also be processed and *two passengers will be assigned the same seat*. You should be aware of this fact and change the application to use a correlated subselect, which looks like this:

```
SELECT * FROM RESERVATION R
WHERE EXISTS
          (SELECT 1
           FROM SEATS
           WHERE SEATNO = R.SEATNO
           AND   AVAILFLAG = 'Y')  ;
```

While it is true that this effect of uncorrelated subqueries is not specific to triggers (for example, if the application updates the rows as it processes the cursor, the same issue arises), what is important to remember with triggers is that the application may not be aware of the underlying processing, which makes it more error-prone.

## 4.14.2 Effect of row processing order

The order of processing can change the outcome of an after row trigger when the trigger logic includes set-oriented operations. The following example illustrates this case.

Tables T1, T2, and T3 look like this:

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
| 1 | | |
| 2 | | |

You create the following trigger on T1:

```
CREATE TRIGGER TRIGGER1
AFTER UPDATE ON T1
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   INSERT INTO T2 VALUES(N.A1);
   INSERT INTO T3 (SELECT B1 FROM T2);
END #
```

Now, assume that the program executes the following update statement:

```
   UPDATE T1 SET A1 = A1 + 1;
```

The contents of tables T2 and T3, after the update statement executes, depend on the order in which DB2 updates the rows of T1.

If DB2 updates the rows of T1 first, after the update statement and the trigger execute for the first time, the values in the three tables are:

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
| 2 | 2 | 2 |
| 2 | | |

After the second row of T1 is updated, the values in the three tables are:

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
| 2 | 2 | 2 |
| 3 | 3 | 2 |

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
|  |  | 3 |

However, if DB2 updates the second row of T1 first, after the update statement and trigger execute for the first time, the values in the three tables are:

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
| 1 | 3 | 3 |
| 3 |  |  |

After the first row of T1 is updated, the values in the three tables are:

| T1 | T2 | T3 |
|---|---|---|
| **A1** | **B1** | **C1** |
| 2 | 3 | 3 |
| 3 | 2 | 3 |
|  |  | 2 |

The simple example shows how the order of processing can affect the final result. You should be aware of this fact. When designing multiple triggers on a table, avoid set-oriented operations in any of the triggers that can cause this order dependency.

### 4.14.3 Effect of set update with row triggers

In 4.13, “Interactions among triggers and other integrity checks” on page 130, we discussed the fact that DB2 makes the changes that are specified in the statement (step 3) and then DB2 processes all after triggers that are defined on the table (step 5). Since the processing does not occur one row at a time (update followed by the after trigger), some undesirable results can occur. The following example illustrates this case.

Assume you have created a TRANSACTION table and a BALANCE table, such as:

```
CREATE TABLE TRANSACTION
    (ACCTNO      INTEGER  NOT NULL,
     TRANNO      SMALLINT NOT NULL,
     AMOUNT      DECIMAL(7,2))
   IN DSNDB04.TRIGTEST ;

CREATE TABLE BALANCE
    (ACCTNO      INTEGER  NOT NULL,
     CURRBAL     DECIMAL(9,2))
   IN DSNDB04.TRIGTEST ;
```

You have also created a trigger on the TRANSACTION table to process any updates so that a difference in amounts is reflected in the current balance on the BALANCE table as follows.

```
CREATE TRIGGER TRIGGER1
AFTER UPDATE ON TRANSACTION
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE BALANCE
   SET CURRBAL = CURRBAL + (N.AMOUNT - O.AMOUNT)
   WHERE ACCTNO = O.ACCTNO ;
END #
```

In order to prevent a direct update to the BALANCE table, you also create another trigger to block any updates where the balance amount is not in sync. See 4.15.6, “Maintaining summary data” on page 142 for further discussion about this topic.

```
CREATE TRIGGER TRIGGER2 NO CASCADE
BEFORE UPDATE ON BALANCE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.CURRBAL <>
                   (SELECT SUM(AMOUNT)
                    FROM TRANSACTION T
                    WHERE T.ACCTNO = O.ACCTNO))
BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('BALANCE DOES NOT MATCH') ;
END #
```

Assume the TRANSACTION table looks like this:

| ACCTNO | TRANNO | AMOUNT |
|---|---|---|
| 100 | 1 | 1000 |
| 100 | 2 | 2000 |
| 100 | 3 | 3000 |
| 200 | 1 | 500 |
| 200 | 2 | 400 |

The BALANCE table looks like this:

| ACCTNO | CURRBAL |
|---|---|
| 100 | 6000 |
| 200 | 900 |

You execute the following update statements one at a time:

```
UPDATE TRANSACTION
   SET AMOUNT = AMOUNT + 10
   WHERE ACCTNO = 100
   AND   TRANNO  = 1     ;
UPDATE TRANSACTION
   SET AMOUNT = AMOUNT + 10
   WHERE ACCTNO = 100
   AND   TRANNO  = 2     ;
UPDATE TRANSACTION
   SET AMOUNT = AMOUNT + 10
   WHERE ACCTNO = 100
   AND   TRANNO  = 3     ;
```

For each row, DB2 executes TRIGGER1 to update the BALANCE table. Since TRIGGER2 exists on the BALANCE table, DB2 executes and verifies that the balance is in sync. The TRANSACTION table now looks like this:

| ACCTNO | TRANNO | AMOUNT |
|---|---|---|
| 100 | 1 | 1010 |
| 100 | 2 | 2010 |
| 100 | 3 | 3010 |
| 200 | 1 | 500 |
| 200 | 2 | 400 |

The BALANCE table now looks like this:

| ACCTNO | CURRBAL |
|---|---|
| 100 | 6030 |
| 200 | 900 |

Instead, you execute the following update statement that updates all three rows in one statement.

```
UPDATE TRANSACTION
   SET AMOUNT = AMOUNT + 10
   WHERE ACCTNO = 100 ;
```

When DB2 processes the first rows (after all updates to TRANSACTION have been applied) in TRIGGER1, DB2 attempts to set the balance amount to 6010, which is different from the sum *as of now* since the second and third rows have already been updated by DB2. This results in an error message:

```
DSNT408I SQLCODE = -438, ERROR:  APPLICATION RAISED ERROR WITH DIAGNOSTIC
         TEXT: BALANCE DOES NOT MATCH
DSNT418I SQLSTATE   = 75001 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXRTYP SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = 1 0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'00000001'  X'00000000'  X'00000000'  X'FFFFFFFF'
         X'00000000'  X'00000000' SQL DIAGNOSTIC INFORMATION
```

## 4.15 Common business scenarios

In this section, we provide practical examples of how to use triggers (possibly in combination with stored procedures or UDFs) to ensure data integrity. We begin with a description of the business problem and provide a solution.

**Important:** Note that several scenarios require a combination of triggers. For example, when maintaining redundant data, it is *not* sufficient to correctly populate the changes. It is also mandatory that you prohibit independent changes to the replicated data.

### 4.15.1 Data validation

Before allowing data to be inserted or updated to a table, you may want to impose strict business rules on the data. We illustrate how triggers can be used for this purpose with an example.

**Business requirements: Verify that the raise given to an employee does not exceed the department's maximum.**

Assume you have created an EMPLOYEE table and a DEPT table, such as:

```
CREATE TABLE EMPLOYEE
    (EMP_NO   CHAR(6) NOT NULL,
     DEPT_NO  CHAR(3),
     SALARY   DECIMAL(9,2))  IN DSNDB04.TRIGTEST ;

CREATE TABLE DEPT
    (DEPT_NO  CHAR(3) NOT NULL,
     DEPT_NAME CHAR(30),
     MAX_RAISE DECIMAL(9,2) ) IN DSNDB04.TRIGTEST ;
```

A trigger can be used to verify, before adjusting an employee's salary, that the percentage increase does not exceed a limit set for the department (on a *different* table, which is something check constraints or view cannot do). This is shown in Example 4-19.

*Example 4-19 Trigger for data validation*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
   WHEN (N.SALARY > O.SALARY *
                     (SELECT MAX_RAISE
                      FROM DEPT D
                      WHERE D.DEPT_NO = O.DEPT_NO))
BEGIN ATOMIC
    SIGNAL SQLSTATE '75001' ('CANNOT EXCEED DEPT LIMIT')
;
END
```

## 4.15.2 Complex data validation with a UDF

We discussed how strict business rules can be imposed on the data in 4.15.1, “Data validation” on page 137. When these rules are complex or require access to non-DB2 resources, you must invoke an external UDF first.

**Business requirements: Verify that the raise given to an employee follows the policy laid out by the Human Resources department. This involves access to VSAM files, IMS databases, and has complex business logic.**

Suppose you have created an employee table:

```
CREATE TABLE EMPLOYEE
    (EMP_NO   CHAR(6) NOT NULL,
     DEPT_NO  CHAR(3),
     SALARY   DECIMAL(9,2))  IN DSNDB04.TRIGTEST ;
```

You have also created an external UDF for validating updates to the salary as follows. This function accepts the employee number, old salary, and new salary and returns an error code with a value of 1, 2, 3, or 4 if it detects an error. If no error is found, it returns a blank return code. You also have to code this external function in a host language of your choosing, for example, COBOL, compile, link, and bind it.

```
CREATE FUNCTION VALSAL (CHAR(6),DEC(9,2),DEC(9,2))
RETURNS CHAR(1)
EXTERNAL NAME VALSAL
LANGUAGE COBOL
PARAMETER STYLE DB2SQL
NO DBINFO
COLLID PAOLOR4
WLM ENVIRONMENT DB8AWLM1
```

We illustrate how triggers can be used for this purpose with Example 4-20.

*Example 4-20 Trigger for complex data validation (with UDF)*

```
CREATE TRIGGER TRIGTEST NO CASCADE
BEFORE UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  SELECT
    CASE
WHEN VALSAL(N.EMP_NO,O.SALARY,N.SALARY) = '1'
             THEN COALESCE(RAISE_ERROR
                   ('75001','SOME MESSAGE1'),' ')
        WHEN VALSAL(N.EMP_NO,O.SALARY,N.SALARY) = '2'
             THEN COALESCE(RAISE_ERROR
                   ('75002','SOME MESSAGE2'),' ')
        WHEN VALSAL(N.EMP_NO,O.SALARY,N.SALARY) = '3'
             THEN COALESCE(RAISE_ERROR
                   ('75003','SOME MESSAGE3'),' ')
        WHEN VALSAL(N.EMP_NO,O.SALARY,N.SALARY) = '4'
             THEN COALESCE(RAISE_ERROR
                   ('75004','SOME MESSAGE4'),' ')
    END
  FROM SYSIBM.SYSDUMMY1;
END #
```

## 4.15.3 Maintaining redundant data

Your database may contain data stored redundantly. In several cases, you may design a denormalized database for performance reasons. Keeping the redundant data in sync with the base table is a challenge that triggers can meet efficiently.

**Business requirements: Ensure the data integrity of the department name, stored redundantly on the employee table.**

On the surface, this may appear to be a scenario similar to auditing discussed in 4.15.12, “Auditing” on page 152. However, it is more complex than that. We also need to make sure that this redundant data is *not* updated independently. Simply preventing updates to the redundant data is insufficient. This will block the updates issued by the trigger on the source table itself.

In Figure 4-8, we show the combination of triggers that is necessary to maintain data integrity.

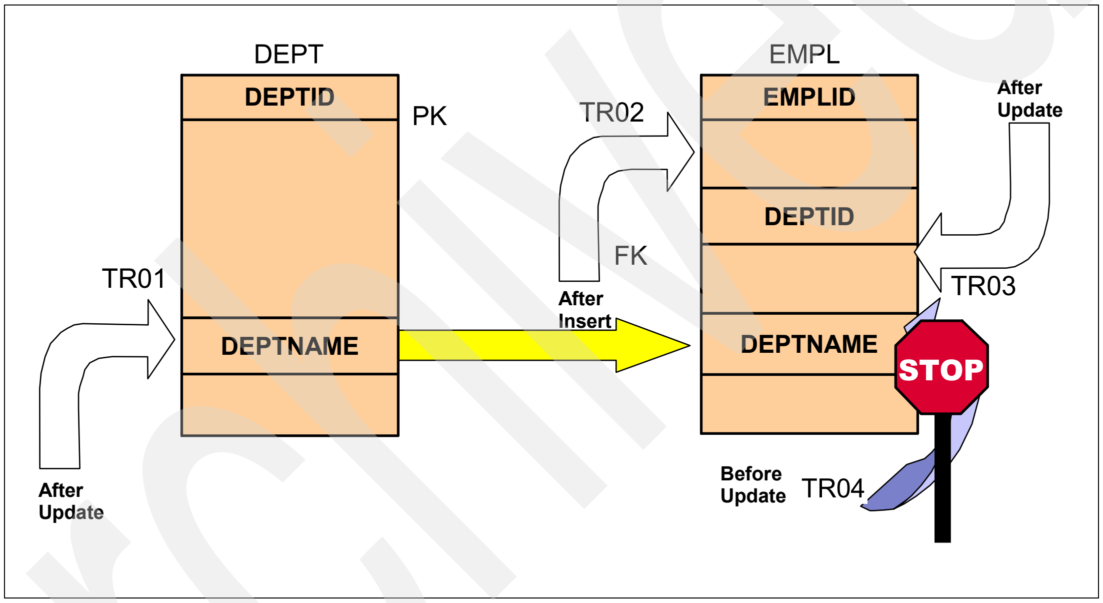


*Figure 4-8 Maintaining redundant data*

Suppose you have created an EMPL table and a DEPT table, such as:

```
CREATE TABLE EMPL
    (EMPLID   CHAR(6) NOT NULL,
     DEPTID   CHAR(3),
     DEPTNAME CHAR(30))
   IN DSNDB04.TRIGTEST ;

CREATE TABLE DEPT
    (DEPTID   CHAR(3) NOT NULL,
     DEPTNAME CHAR(30) )  IN DSNDB04.TRIGTEST ;
```

We illustrate how triggers can be used for this purpose with Example 4-21. Trigger1 automatically updates all employees in the department name when the department name changes. Trigger2 ensures that only the correct department name is inserted and Trigger3 ensures this on an update. Trigger4 prevents a direct update to the employee table with an incorrect value. Stopping all updates will cause Trigger1, Trigger2, and Trigger3 to fail (since we have no means of allowing DB2-generated updates, but prevent user-generated updates).

*Example 4-21 Triggers for maintaining redundant data*

```
-- Trigger1 - AFTER UPDATE ON DEPT
CREATE TRIGGER TRIGGER1
AFTER UPDATE OF DEPTNAME ON DEPT
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE EMPL
   SET    DEPTNAME = N.DEPTNAME
   WHERE  DEPTID = O.DEPTID;
END #

-- Trigger2 - AFTER INSERT ON EMPL

CREATE TRIGGER TRIGGER2
AFTER INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE EMPL
   SET    DEPTNAME =
         (SELECT DEPTNAME
          FROM DEPT
          WHERE DEPTID = N.DEPTID)
   WHERE EMPLID = N.EMPLID;
END #

-- Trigger3 - AFTER UPDATE ON EMPL

CREATE TRIGGER TRIGGER3
AFTER UPDATE OF DEPTID ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE EMPL
   SET    DEPTNAME =
         (SELECT DEPTNAME
          FROM DEPT
          WHERE DEPTID = N.DEPTID)
   WHERE EMPLID = O.EMPLID;
END #

-- Trigger4 - BEFORE UPDATE ON EMPL

CREATE TRIGGER TRIGGER4 NO CASCADE
BEFORE UPDATE OF DEPTNAME ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.DEPTNAME <>
           (SELECT DEPTNAME
            FROM DEPT
            WHERE DEPTID = O.DEPTID))
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR04'
        ('TRIGGER4 DEPT NAME UPDATE NOT ALLOWED');
END #
```

### 4.15.4 Complex redundant data maintenance with a stored procedure

When converting a large application from existing structures to new structures, it is common to have a stop-gap “bridging” solution in which programs are changed to access the new structures one at a time rather than as a complete rewrite. When the existing data resides in non-DB2 resources which are not accessible by the trigger, calling a stored procedures extends the trigger functionality and allows the data to be kept in sync. We illustrate how to use triggers for this purpose with Example 4-22.

**Business requirements: Every update to the employee table should create an audit trail, provided various other complex business conditions are met. This may include access to VSAM files, IMS databases, and other logic.**

Suppose you have created an EMPLOYEE and a SALARY_AUDIT table, such as:

```
CREATE TABLE EMPLOYEE
    (EMP_NO   CHAR(6) NOT NULL,
     SALARY   DECIMAL(9,2))  IN DSNDB04.TRIGTEST #
COMMIT #
CREATE TABLE SALARY_AUDIT
    (EMP_NO   CHAR(6) NOT NULL,
     OLD_SALARY DECIMAL(9,2),
     NEW_SALARY DECIMAL(9,2) ) IN DSNDB04.TRIGTEST #
```

You have also created a stored procedure. You also have to code this external function in a host language of your choosing, for example, COBOL, compile, link, and bind it:

```
CREATE PROCEDURE AUDSAL
(
 IN  PEMPNO       CHAR(6)
,IN  POLDSALARY   DEC(9,2)
,IN  PNEWSALARY   DEC(9,2)
)
DYNAMIC RESULT SETS 0
EXTERNAL NAME AUDSAL
LANGUAGE COBOL
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
NO DBINFO
COLLID PAOLOR4
WLM ENVIRONMENT DB8AWLM1 ;
```

You can use the trigger created in Example 4-22 for this purpose.

*Example 4-22   Trigger for complex redundant data maintenance (with stored procedure)*

```
CREATE TRIGGER TRIGTEST
AFTER UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   CALL AUDSAL(O.EMP_NO, O.SALARY, N.SALARY);
END #
```

### 4.15.5 Bidirectional data maintenance

When converting a large application from existing structures to new structures, it is common to have a stop-gap “bridging” solution in which programs are changed to access the new structures one at a time rather than as a complete rewrite. In these cases, updates to old DB2 tables as well as new DB2 tables are possible, and there is a business need to keep these in sync. This bidirectional data maintenance presents special challenges that we discuss here.

#### INSERT

As long as the key values generated on each side are know to be unique, all inserts can be safely propagated to the other side. Collisions can occur; however, and you must deal with these by deciding which transaction is processed and which transaction is treated as an error. One method for resolving conflicts is to use the timestamp of the transaction, allowing the first one to process.

#### UPDATE

Conflict resolution in this case is especially challenging, since, in theory, each transaction may change values for different columns. In general, the second transaction will override the effects of the first transaction. You must have adequate controls in place to make sure this does not cause undesirable effects, for example, the employee's address is changed by the existing HR system and the salary is changed by the new payroll system.

#### DELETE

This is relatively straightforward, but you must deal with an attempt to delete a row that has already been deleted.

In all these cases, you must be aware of the dependencies among the values. For example, the salary of an employee may be adjusted to a value based on the current performance rating. If the performance rating is later modified by the other application system, the salary needs to be reevaluated.

Keep in mind that when the amount of data to be replicated is large, replication is generally the best way to address this need. We discussed this in 4.8, "Auditing versus mass replication" on page 122.

### 4.15.6 Maintaining summary data

Summary data is a special form of redundant data discussed in 4.15.3, "Maintaining redundant data" on page 139.

**Business requirements: Ensure the data integrity of the total salary for the department is stored redundantly.**

In Figure 4-9, we show the combination of triggers that are necessary to maintain data integrity.

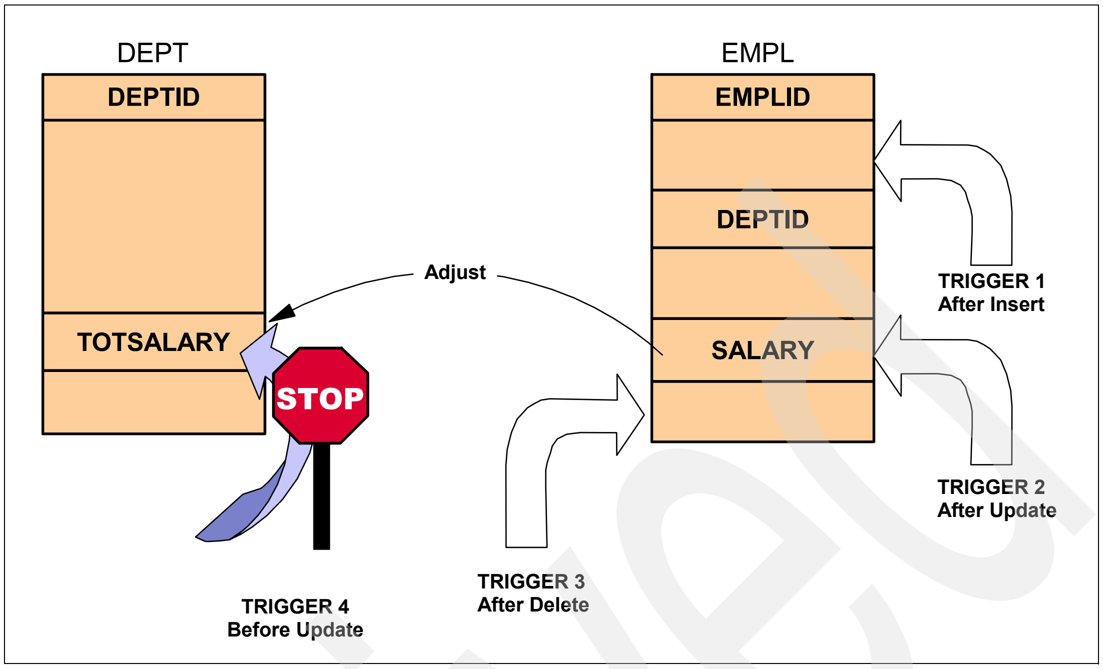


*Figure 4-9 Maintaining summary data*

Suppose you have created an EMPL table and a DEPT table, such as:

```
   CREATE TABLE EMPL
       (EMPLID   CHAR(6) NOT NULL,
        DEPTID   CHAR(3),
        SALARY   DECIMAL(9,2))
      IN DSNDB04.TRIGTEST #

   CREATE TABLE DEPT
       (DEPTID   CHAR(3) NOT NULL,
        DEPTNAME CHAR(30),
        TOTSALARY DECIMAL(15,2))
   IN DSNDB04.TRIGTEST #
```

We illustrate this with Example 4-23.

*Example 4-23 Triggers for maintaining summary data*

```
-- Trigger1 - AFTER INSERT ON EMPL

CREATE TRIGGER TRIGGER1
AFTER INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE DEPT
   SET    TOTSALARY =
                     (SELECT SUM(SALARY)
                      FROM EMPL
                      WHERE DEPTID = N.DEPTID)
   WHERE  DEPTID = N.DEPTID;
END #

-- Trigger2 - AFTER UPDATE ON EMPL
```

```
CREATE TRIGGER TRIGGER2
AFTER UPDATE OF SALARY ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE DEPT
   SET    TOTSALARY =
         (SELECT SUM(SALARY)
          FROM EMPL
          WHERE DEPTID = N.DEPTID)
   WHERE DEPTID = N.DEPTID;
END #

-- Trigger3 - AFTER DELETE ON EMPL

CREATE TRIGGER TRIGGER3
AFTER DELETE ON EMPL
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE DEPT
   SET    TOTSALARY =
         (SELECT SUM(SALARY)
          FROM EMPL
          WHERE DEPTID = O.DEPTID)
   WHERE DEPTID = O.DEPTID;
END #

-- Trigger4 - BEFORE UPDATE ON DEPT

CREATE TRIGGER TRIGGER4 NO CASCADE
BEFORE UPDATE OF TOTSALARY ON DEPT
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.TOTSALARY <>
           (SELECT SUM(SALARY)
            FROM EMPL
            WHERE DEPTID = O.DEPTID))
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR04'
        ('TRIGGER4 UPDATE OF TOTAL SALARY NOT ALLOWED');
END #
```

### 4.15.7 Maintaining existence flags

*Existence flags* are another special form of redundant data that is discussed in 4.15.3, “Maintaining redundant data” on page 139.

**Business requirements: Ensure the data integrity of the existence flag on the employee table.**

In Figure 4-10, we show the combination of triggers that are necessary to maintain data integrity.

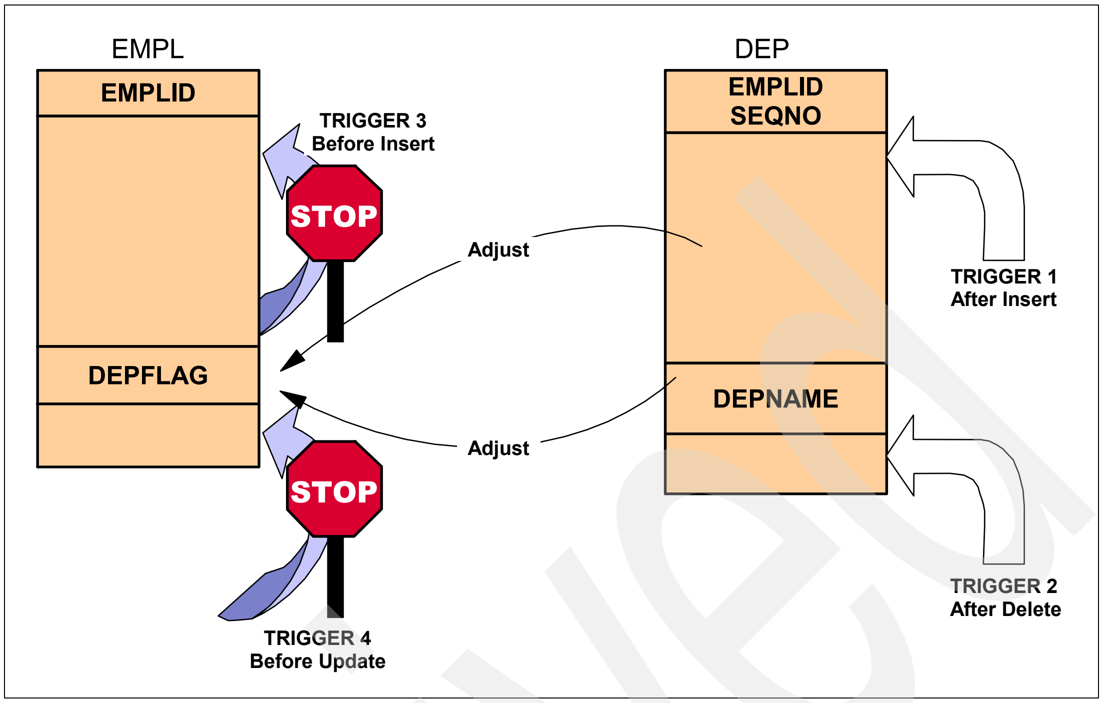


*Figure 4-10 Maintaining existence flags*

Suppose you have created an employee table EMPL and a dependent table DEP, such as:

```
CREATE TABLE EMPL
    (EMPLID      CHAR(6) NOT NULL,
     DEPFLAG     CHAR(1))
   IN DSNDB04.TRIGTEST #

CREATE TABLE DEP
    (EMPLID   CHAR(6) NOT NULL,
     SEQNO    SMALLINT,
     DEPNAME  CHAR(30))
IN DSNDB04.TRIGTEST #
```

We illustrate this with Example 4-24.

*Example 4-24 Triggers for maintaining existence flags*

```
-- Trigger1 - AFTER INSERT ON DEP
CREATE TRIGGER TRIGGER1
AFTER INSERT ON DEP
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE EMPL
   SET    DEPFLAG = 'Y'
   WHERE  EMPLID = N.EMPLID;
END #

-- Trigger2 - AFTER DELETE ON DEP

CREATE TRIGGER TRIGGER2
AFTER DELETE ON DEP
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
```

```
BEGIN ATOMIC
   UPDATE EMPL
   SET    DEPFLAG =
         (SELECT
            CASE COUNT(*)
                WHEN 0 THEN 'N'
                ELSE        'Y'
            END
          FROM DEP
          WHERE EMPLID = O.EMPLID)
   WHERE EMPLID = O.EMPLID;
END #

-- Trigger3 - BEFORE INSERT ON EMPL

CREATE TRIGGER TRIGGER3 NO CASCADE
BEFORE INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.DEPFLAG <> 'N')
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR03'
        ('TRIGGER3 DEP FLAG MUST BE N ON INSERT');
END #

-- Trigger4 - BEFORE UPDATE ON EMPL

CREATE TRIGGER TRIGGER4 NO CASCADE
BEFORE UPDATE OF DEPFLAG ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN ((0 < (SELECT COUNT(*)
            FROM DEP
WHERE EMPLID = O.EMPLID)
      AND N.DEPFLAG <> 'Y')
      OR
      (0 = (SELECT COUNT(*)
            FROM DEP
            WHERE EMPLID = O.EMPLID)
      AND N.DEPFLAG <> 'N'))
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR04'
        ('TRIGGER4 UPDATE OF DEP FLAG NOT ALLOWED');
END #
```

### 4.15.8 Enforcing multiple parent RI

Declarative RI, discussed in Chapter 3, “Referential integrity” on page 49, allows the foreign key to reference only one parent. In some situations you may have a need to reference more than one parent. Existence of the foreign key in any one of the parent tables is sufficient, which is termed a “mom-or-dad” trigger since a permission slip from either one is OK.

**Business requirements: Ensure the data integrity of the employee table, which must contain a valid department or a valid plant.**

In Figure 4-11, we show the combination of triggers that are necessary to maintain data integrity.

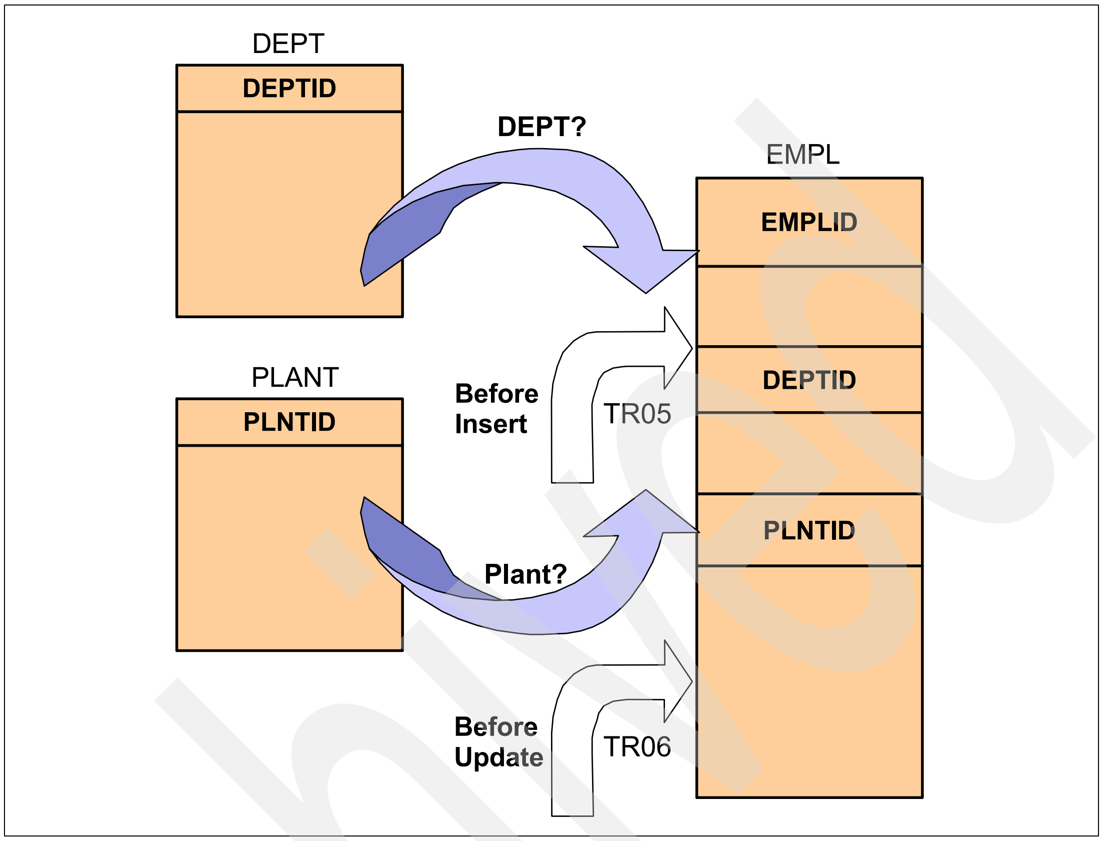


*Figure 4-11 Maintaining multiple parent RI*

Suppose you have created EMPL, DEPT, and PLANT tables, such as:

```
CREATE TABLE EMPL
    (EMPLID   CHAR(6) NOT NULL,
     DEPTID   CHAR(3),
     PLANTID  CHAR(3))
   IN DSNDB04.TRIGTEST ;

CREATE TABLE DEPT
    (DEPTID   CHAR(3) NOT NULL,
     DEPTNAME CHAR(30) )  IN DSNDB04.TRIGTEST ;

CREATE TABLE PLANT
    (PLANTID  CHAR(3) NOT NULL,
     PLANTNAME CHAR(30) )  IN DSNDB04.TRIGTEST ;
```

We illustrate this with Example 4-25.

*Example 4-25 Triggers for multiple parent RI*

```
-- Trigger5 - BEFORE INSERT ON EMPL
CREATE TRIGGER TRIGGER5 NO CASCADE
BEFORE INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (0 = (SELECT COUNT(*)
           FROM  DEPT
           WHERE DEPTID = N.DEPTID)
   AND
      0 = (SELECT COUNT(*)
           FROM  PLANT
           WHERE PLANTID = N.PLANTID) )
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR05'
        ('TRIGGER5 VALID DEPT OR PLANT IS REQUIRED');
END #

-- Trigger6 - BEFORE UPDATE ON EMPL

CREATE TRIGGER TRIGGER6 NO CASCADE
BEFORE UPDATE ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (0 = (SELECT COUNT(*)
           FROM  DEPT
           WHERE DEPTID = N.DEPTID)
   AND
      0 = (SELECT COUNT(*)
           FROM  PLANT
           WHERE PLANTID = N.PLANTID) )
BEGIN ATOMIC
     SIGNAL SQLSTATE '7TR06'
        ('TRIGGER6 VALID DEPT OR PLANT IS REQUIRED');
END #
```

## 4.15.9 Enforcing “Empty-nest-last-child-gone” rule

In some applications, you may need to extend the declarative RI discussed in Chapter 3, “Referential integrity” on page 49. While declarative RI allows you to specify the action to take when the parent row is deleted, it does not allow you specify any action on the deletion of the child row. When the last order for a customer is deleted; for example, you may wish to delete the customer also (this will make sure they never buy from you again).

**Business requirements: When a department has no employees in it, delete it.**

In Figure 4-12, we show the combination of triggers that are necessary to maintain data integrity.

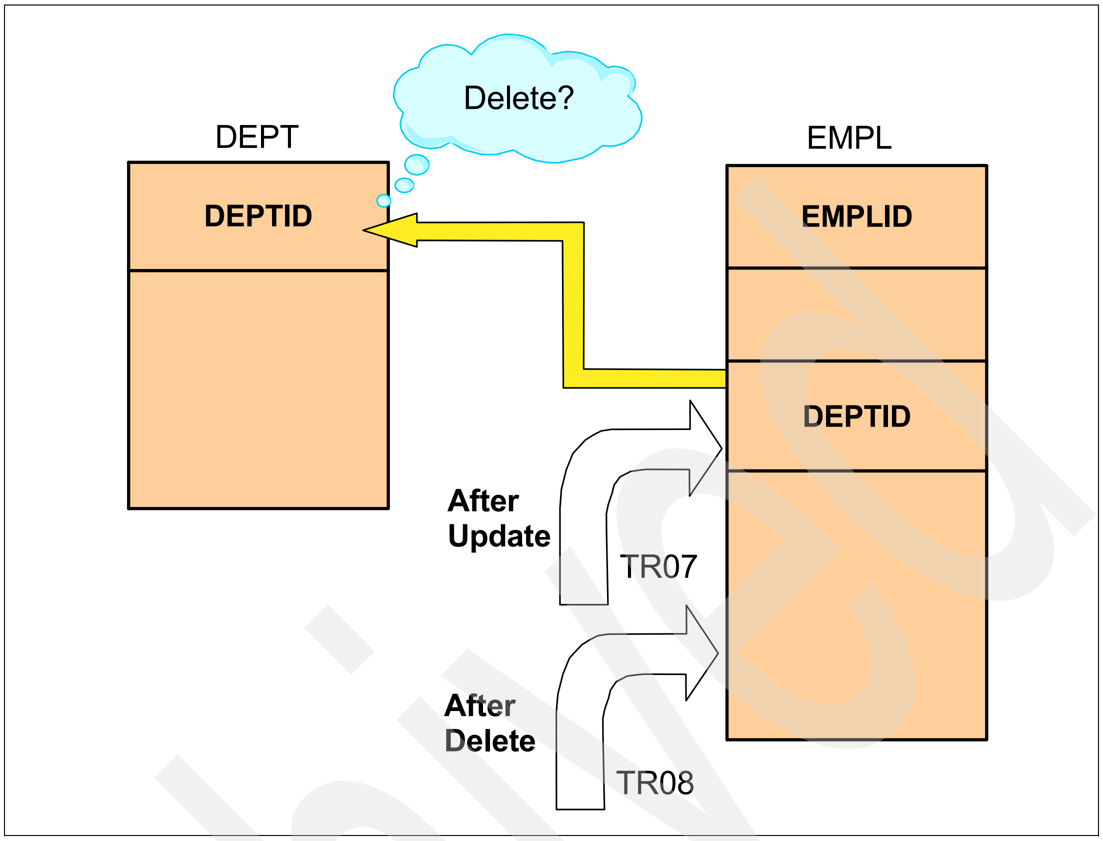


*Figure 4-12 Enforcing "empty-nest" last-child-gone rule*

Suppose you have created EMPL and DEPT tables, such as:

```
   CREATE TABLE EMPL
       (EMPLID   CHAR(6) NOT NULL,
        DEPTID   CHAR(3))
   IN DSNDB04.TRIGTEST ;

   CREATE TABLE DEPT
       (DEPTID   CHAR(3) NOT NULL,
        DEPTNAME CHAR(30) )
   IN DSNDB04.TRIGTEST ;
```

We illustrate this with Example 4-26.

*Example 4-26 Triggers for "Last-child-gone" rule*

```
-- Trigger7 - AFTER UPDATE ON EMPL
CREATE TRIGGER TRIGGER7
AFTER UPDATE OF DEPTID ON EMPL
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
WHEN (0 = (SELECT COUNT(*)
           FROM  EMPL
           WHERE DEPTID = O.DEPTID) )
BEGIN ATOMIC
     DELETE FROM DEPT
     WHERE DEPTID = O.DEPTID;
END #

-- Trigger8 - AFTER DELETE ON EMPL
```

```
CREATE TRIGGER TRIGGER8
AFTER DELETE ON EMPL
REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
WHEN (0 = (SELECT COUNT(*)
           FROM  EMPL
           WHERE DEPTID = O.DEPTID) )
BEGIN ATOMIC
     DELETE FROM DEPT
     WHERE DEPTID = O.DEPTID;
END #
```

## 4.15.10 Generating alerts

If you want to be notified via an alert (a page, e-mail, and so forth) when a certain critical condition occurs on a table, you can do so by using triggers that call a stored procedure.

**Business requirements: When someone increases the salary of any employee by more than 20%, send an e-mail to the HR director.**

Since the trigger body can include only SQL statements, you must call a stored procedure as shown in the following examples. You can code the stored procedure to invoke a program that interfaces with your e-mail system. Since this depends on your installation, we provide no examples of the installation-specific programs used.

We illustrate this with Example 4-27.

*Example 4-27 Trigger for generating an e-mail*

```
CREATE TRIGGER EMPTRIG1
AFTER UPDATE OF SALARY ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.20)
BEGIN ATOMIC
 CALL EMPAUDTS(N.EMPLID,O.SALARY,N.SALARY);
-- this stored procedure generates the e-mail--
END#
```

## 4.15.11 Writing an MQ message

If you want to write an MQ message when a certain condition occurs on a table, you can use Q replication or Q publish/subscribe. An alternative is do so by using triggers. Depending on your installation, you can do so directly in the trigger or call a stored procedure.

**Business requirements: When someone makes any changes to the employee table, generate an MQ message so that the changes can be replicated on the shadow system later.**

As before, you can call a stored procedure from the trigger for this purpose, as shown in Example 4-28.

*Example 4-28 Trigger to write calling a stored procedure for an MQ message*

```
CREATE TRIGGER EMPTRIG1
AFTER UPDATE OF SALARY ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN (N.SALARY >< O.SALARY)
BEGIN ATOMIC
 CALL EMPAUDTS(N.EMPNO,O.SALARY,N.SALARY);
-- this stored procedure writes the MQ message--
END#
```

The Application Messaging Interface (AMI) is a commonly used application programming interface (API) for WebSphere MQ that is available in various high-level languages. In addition to the AMI, DB2 provides its own API to the WebSphere MQ message handling system through a set of external user-defined functions. Using these functions in SQL statements allows you to combine DB2 database access with WebSphere MQ message handling. For further information, see Chapter 33, “Using WebSphere MQ with DB2” of the *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

You can use one of the two external UDFs, MQSEND and MQPUBLISH, for this purpose, which we discuss in this section. The examples use the DB2MQ2C schema for two-phase commit, with the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY.

## Using MQSEND

Example 4-29 shows how you can use MQSEND.

*Example 4-29 Using MQSEND*

```
CREATE TRIGGER EMPTRIG1
AFTER UPDATE OF SALARY ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.20)
BEGIN ATOMIC
SELECT DB2MQ2C.MQSEND(N.EMPLID CONCAT ' ' CONCAT
                      DIGITS(O.SALARY) CONCAT ' 'CONCAT
                      DIGITS(N.SALARY))
FROM SYSIBM.SYSDUMMY1;
END#
```

## Using MQPUBLISH

Example 4-30 shows how you can use MQPUBLISH. Any users or applications that subscribe to the HR_INFO_PUB service with a registered interest in the SAL_CHANGE topic will receive a message that contains the date, employee number, old salary, and new salary when rows are updated in the employee table.

*Example 4-30 Using MQPUBLISH*

```
CREATE TRIGGER EMPTRIG2
AFTER UPDATE OF SALARY ON EMPL
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW
MODE DB2SQL
WHEN ((N.SALARY - O.SALARY) > O.SALARY * 0.20)
BEGIN ATOMIC
SELECT DB2MQ2C.MQPUBLISH('HR_INFO_PUB',
CHAR(CURRENT DATE) CONCAT N.EMPLID
CONCAT DIGITS(O.SALARY) CONCAT DIGITS(N.SALARY), 'SAL_CHANGE')
FROM SYSIBM.SYSDUMMY1;
END#
```

## 4.15.12 Auditing

For some applications, you are required to create an audit trail for any changes to sensitive data. We illustrate how triggers can meet the need with an example.

**Business requirements: Every update to the employee table should create an audit trail.**

Suppose you have created an EMPLOYEE table and a table SALARY_AUDIT to audit all updates made to the salary.

```
   CREATE TABLE EMPLOYEE
       (EMP_NO   CHAR(6) NOT NULL,
        SALARY   DECIMAL(9,2))  IN DSNDB04.TRIGTEST ;

   CREATE TABLE SALARY_AUDIT
       (EMP_NO   CHAR(6) NOT NULL,
        OLD_SALARY DECIMAL(9,2),
        NEW_SALARY DECIMAL(9,2)
        SALTIST TIMESTAMP NOT NULL WITH DEFAULT) IN DSNDB04.TRIGTEST ;
```

The trigger shown in Example 4-31 can be used for this purpose. Every update to the employee table automatically creates an audit trail showing the old salary, new salary, and timestamp. You can create similar triggers for insert and delete as well.

*Example 4-31 Trigger for auditing*

```
CREATE TRIGGER TRIGTEST
AFTER UPDATE ON EMPLOYEE
REFERENCING OLD AS O
            NEW AS N
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   INSERT INTO SALARY_AUDIT
   VALUES (O.EMP_NO, O.SALARY, N.SALARY)
;
END
```

5

# Other integrity features

Besides the CHECK DATA utilities that DB2 offers for validating referential integrity and check constraints, DB2 has also functions that check the physical page structure.

Furthermore, DB2 for z/OS V8 provides several new features that help with data integrity from the application's point of view. In this chapter, we focus on these new functions.

This chapter contains the following:

- Data structure validation
- Insert within select
- Atomic versus not atomic on multi-row insert and update
- Sequence objects
- Informational RI
- Locking

## 5.1 Data structure validation

We have seen in 3.9, "CHECK utility" on page 89, the CHECK INDEX and CHECK DATA utilities that DB2 offers for validating referential integrity and check constraints. DB2 also has some built-in functions that automatically check the physical page structure on a page by page basis. This structure checking is also provided by:

- DSN1COPY with CHECK option
- DSN1PRNT with FORMAT option
- COPY with CHECKPAGE option

### 5.1.1 DSN1COPY with CHECK option

With the DSN1COPY standalone utility, you can copy:

- DB2 VSAM data sets to sequential data sets
- DSN1COPY sequential data sets to DB2 VSAM data sets
- DB2 image copy data sets to DB2 VSAM data sets
- DB2 VSAM data sets to other DB2 VSAM data sets
- DSN1COPY sequential data sets to other sequential data sets

Using DSN1COPY, you can also:

- Print hexadecimal dumps of DB2 data sets and databases.
- Check the validity of data or index pages (including dictionary pages for compressed data).
- Translate database object identifiers (OBIDs) to enable moving data sets between different systems.
- Reset to 0 the log RBA that is recorded in each index page or data page.

DSN1COPY is compatible with LOB table spaces when you specify the LOB keyword and omit the SEGMENT and INLCOPY keywords.

The CHECK option of DSN1COPY checks each page from the SYSUT1 data set for validity. The validity checking operates on one page at a time and does not include any cross-page checking. If an error is found, a message is issued describing the type of error, and a dump of the page is sent to the SYSPRINT data set. If you do not receive any messages, no errors were found. If more than one error exists in a given page, the check identifies only the first of the errors. However, the entire page is dumped. DSN1COPY does not check *system* pages for validity.

The OBIDXLAT option specifies that OBID translation must be done before the DB2 data set is copied. This parameter requires additional input from the SYSXLAT file by using the DD statements. DSN1COPY can translate only up to 1000 (increased from 500 with APAR PK05758) record OBIDs. If you specify OBIDXLAT, CHECK processing is performed, regardless of whether you specify the CHECK option.

### 5.1.2 DSN1PRNT with FORMAT option

With the DSN1PRNT standalone utility, you can print:

- DB2 VSAM data sets that contain table spaces or index spaces (including dictionary pages for compressed data)
- Image copy data sets
- Sequential data sets that contain DB2 table spaces or index spaces

Using DSN1PRNT, you can print hexadecimal dumps of DB2 data sets and databases. If you specify the FORMAT option, DSN1PRNT formats the data and indexes for any page that does not contain an error that would prevent formatting. If DSN1PRNT detects such an error, it prints an error message just before the page and dumps the page without formatting. Formatting resumes with the next page. Compressed records are printed in compressed format. DSN1PRNT is especially useful when you want to identify the contents of a table space or index. You can run DSN1PRNT on image copy data sets as well as table spaces and indexes. DSN1PRNT accepts an index image copy as input when you specify the FULLCOPY option. You cannot run DSN1PRNT on concurrent copies.

DSN1PRNT is compatible with LOB table spaces when you specify the LOB keyword and omit the INLCOPY keyword.

The FORMAT option causes the printed output to be formatted. Page control fields are identified, and individual records are printed. Empty fields are not displayed.

- EXPAND specifies that the data is compressed and causes DSN1PRNT to expand it before formatting. This option is intended to be used only under the direction of your IBM Support Center. SWONLY causes DSN1PRNT to use software to expand the compressed data, even when the compression hardware is available. This option is intended to be used only under the direction of your IBM Software Support.
- NODATA suppresses printing of table row data. The row headers are formatted and printed. This keyword is ignored for indexes. Specify NODATA to reduce the volume of the output when the content of the rows is not important.
- NODATPGS suppresses all data pages of a table space. This keyword is ignored for indexes. Specify NODATPGS to format and print only non-data pages to reduce the volume of output when only certain page types are of interest (for example, LOB space map pages). Alternatively, you can specify NODHDR.

DSN1PRNT cannot format a leaf or nonleaf page for an index page set that contains keys with altered columns. When it encounters this situation, DSN1PRNT generates the following message:

```
*KEY WITH ALTERED COLUMN HAS BEEN DETECTED-UNABLE TO FORMAT PAGE*
```

DSN1PRNT generates unformatted output for the page. FORMAT might generate unformatted output for certain system pages.

## 5.1.3 COPY with CHECKPAGE option

The COPY utility is used to take secure images of table spaces and index spaces to allow for the recovery of data in the event of data loss or corruption.

The COPY utility also checks the integrity of data pages as it is copying the table space or index space. Before V6, any additional checking had to be undertaken using DSN1COPY with the CHECK option specified.

Starting with V6, additional checking can be undertaken using the CHECKPAGE option of the COPY utility. This option is equivalent to the checking undertaken by DSN1COPY with CHECK. By specifying this option, DB2 will validate each page of the table space or index page as it is processed.

CHECKPAGE option indicates that each page in the table space or index space is to be checked for validity. The validity checking operates on one page at a time and does not include any cross-page checking. If it finds an error, COPY issues a message that describes the type of error. If more than one error exists in a given page, only the first error is identified.

COPY continues checking the remaining pages in the table space or index space after it finds an error.

The benefit of using this option is that any errors in the pages are reported at the time that the backup is taken and not when the image copy is required by the RECOVER utility. The risk of any errors existing when taking the image is low, but the impact of finding out at recover time is high.

Without the CHECKPAGE option, a rare but possible hardware error could cause defective image copies that are invalid for recovery. A periodic COPY with the CHECKPAGE option would report the error and then create an opportunity to rectify the error, either using REPAIR or recovery back to a valid image copy. After repairing the error, a new valid image copy should then be taken.

## 5.2 Insert within select

The INSERT within SELECT (also called the SELECT FROM INSERT) feature provides you the ability to:

- Find the values of automatically generated columns.
- Retrieve the default values for columns.
- Retrieve column values changed by a BEFORE INSERT trigger.
- Retrieve all values for an inserted row without specifying individual column names.
- Retrieve all values inserted through a multiple-row INSERT.

With the new syntax of having an INSERT statement within the SELECT statement, the rows inserted into the table are considered to be a result table. The select list of the query can reference all columns in this result table. The keywords FINAL TABLE refer to the result table of the INSERT statement.

The result table contains all of the rows inserted and includes all of the columns requested in the SELECT list. Triggers, constraints, and values generated by DB2 affect the result table in the following ways:

- If the INSERT activates a before trigger, the values in the result table include any changes that are made by the trigger. After triggers do not affect the values in the result table.
- DB2 verifies check constraints, unique index constraints, and RI constraints before generating the result table and issuing error messages on violations.
- The result table includes generated values for identity columns, ROWID columns, and columns based on expressions.

DB2 enforces check constraints, unique index constraints, and RI constraints before it generates the result table.

See Chapter 5 of *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426, for more information.

We provide examples of how you can use this feature.

### 5.2.1 Generated values example

Suppose you have created a table EMPL, as shown below:

```
CREATE TABLE EMPL
    (EMPNO       INTEGER GENERATED ALWAYS AS IDENTITY,
     EMPNAME     CHAR(20),
     CITY        CHAR(20) NOT NULL DEFAULT 'KANSAS CITY',
     SALARY      DECIMAL(9,2))
   IN DSNDB04.TRIGTEST #
```

Notice that column EMPNO is an identity column (value to be generated by DB2) and the CITY column has a default value of Kansas City.

If you run the following SQL:

```
SELECT * FROM FINAL TABLE
(INSERT INTO EMPL(EMPNAME, SALARY)
VALUES ('SURESH SANE',10000.00))
```

The result will be similar to this:

```
---------+---------+---------+---------+---------+---------+---------
      EMPNO  EMPNAME              CITY                       SALARY
---------+---------+---------+---------+---------+---------+---------
          1  SURESH SANE          KANSAS CITY              10000.00
```

Notice the values assigned to the columns EMPNO and CITY are included.

Instead, if you supply a value for CITY by issuing this:

```
SELECT * FROM FINAL TABLE
(INSERT INTO EMPL(EMPNAME, CITY, SALARY)
VALUES ('PAOLO BRUNI','SAN JOSE', 20000.00)) ;
```

This would result in the following:

```
---------+---------+---------+---------+---------+---------+---------
      EMPNO  EMPNAME              CITY                       SALARY
---------+---------+---------+---------+---------+---------+---------
          1  PAOLO BRUNI          SAN JOSE                 20000.00
```

## 5.2.2 Multiple-row inserts example

You can also use the INSERT within SELECT when you insert multiple rows.

Suppose you have created tables EMPL and NEWHIRE as shown below:

```
CREATE TABLE EMPL
    (EMPNO       INTEGER GENERATED ALWAYS AS IDENTITY,
     EMPNAME     CHAR(20),
     CITY        CHAR(20) NOT NULL DEFAULT 'KANSAS CITY',
     SALARY      DECIMAL(9,2))
   IN DSNDB04.TRIGTEST ;
```

and

```
CREATE TABLE NEWHIRE
    (EMPNAME     CHAR(20) NOT NULL,
     SALARY      DECIMAL(9,2))
   IN DSNDB04.TRIGTEST ;
```

Assume you have populated NEWHIRE with the data shown:

```
---------+---------+---------+----
EMPNAME                    SALARY
---------+---------+---------+----
SURESH SANE              10000.00
PAOLO BRUNI              20000.00
```

Issuing the following statement:

```
SELECT * FROM FINAL TABLE
(INSERT INTO EMPL(EMPNAME, SALARY)
SELECT * FROM NEWHIRE)
```

results in the following:

```
---+---------+---------+---------+---------+---------+---------
EMPNO  EMPNAME              CITY                      SALARY
---+---------+---------+---------+---------+---------+---------
    1  SURESH SANE          KANSAS CITY             10000.00
    2  PAOLO BRUNI          KANSAS CITY             20000.00
```

Notice the values assigned to the EMPNO and CITY columns for each of the two rows inserted. This example of multiple-row insert uses another table (NEWHIRE), but the results would be identical if a host variable array is used instead.

## 5.2.3 Trigger example

When you use the INSERT within SELECT, the values in the result table include changes made by a before trigger.

Suppose you have created tables EMPL and NEWHIRE as shown below:

```
CREATE TABLE EMPL
    (EMPNO       INTEGER GENERATED ALWAYS AS IDENTITY,
     EMPNAME     CHAR(20),
     DEPT        CHAR(4),
     SALARY      DECIMAL(9,2))
   IN DSNDB04.TRIGTEST ;
```

and

```
CREATE TABLE NEWHIRE
    (EMPNAME     CHAR(20) NOT NULL,
     DEPT        CHAR(4),
     SALARY      DECIMAL(9,2))
   IN DSNDB04.TRIGTEST #
```

Suppose you have populated NEWHIRE with the data shown below:

```
---+---------+---------+---------+---------+---
EMPNO  EMPNAME              DEPT       SALARY
---+---------+---------+---------+---------+---
    1  SURESH SANE          ARCH     10000.00
    2  PAOLO BRUNI          ITSO     20000.00
```

You have also created a trigger on the EMPL table that increases the salary of anyone in the ARCH department by 5000 as follows:

```
CREATE TRIGGER TRIGGER1 NO CASCADE
BEFORE INSERT ON EMPL
REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
WHEN (N.DEPT = 'ARCH')
BEGIN ATOMIC
    SET N.SALARY = N.SALARY + 5000;
END #
```

Issuing the following query:

```
SELECT * FROM FINAL TABLE
```

```
(INSERT INTO EMPL(EMPNAME, DEPT, SALARY)
SELECT * FROM NEWHIRE) ;
```

results in:

```
---+---------+---------+---------+---------+---
EMPNO  EMPNAME              DEPT       SALARY
---+---------+---------+---------+---------+---
    1  SURESH SANE          ARCH     15000.00
    2  PAOLO BRUNI          ITSO     20000.00
```

Notice the salary is increased by 5000 for EMPNO 1 in department ARCH.

## 5.3 Atomic versus not atomic on multi-row insert and update

The ATOMIC or NOT ATOMIC CONTINUE ON SQLEXCEPTION clause of multiple-row insert allows you to specify whether you want the multiple-row insert to succeed or fail as a unit, or if you want DB2 to proceed despite a partial failure of one or more rows.

- ATOMIC

  Specifies that if the insert of any row fails, all changes made to the database by any of the inserts of this statement, including changes made by successful inserts, are undone. This is the default and provides a "all-or-nothing" capability, which is easier to restart or reposition.

- NOT ATOMIC CONTINUE ON SQLEXCEPTION

  Specifies that regardless of the failure of any particular insert of a row, the insert statement will not undo any changes made to the database by the successful inserts of other rows. The clause can be specified for static SQL only and cannot be used with the INSERT within SELECT (discussed in 5.2, "Insert within select" on page 156).

  An important consideration of this clause is its effect on triggers when triggers are processed with a multiple row insert statement.

- ATOMIC

  The inserts are processed as a single statement. Any statement level triggers are invoked once for the statement and transition tables include all of the rows inserted.

- NOT ATOMIC CONTINUE ON SQLEXCEPTION

  The inserts are processed separately. *Any statement level triggers are processed for each inserted row,* and transition tables include the individual row inserted. When errors are encountered, processing continues, and some of the specified rows may not get inserted. If an insert trigger is defined on the underlying base table, the trigger transition table includes only rows that are successfully inserted.

**Attention:** PTF UK05890 for APAR PQ96775 fixes a problem of incorrect output when using NOT ATOMIC multi-row insert SQL with triggers.

## 5.4 Sequence objects

Sequences are completely standalone objects and have no connection to a table. They can be used by multiple applications in different ways. A sequence is a stored object that simply generates the next ascending or descending value when requested by an application. They provide an excellent way for an application to obtain unique values for use in key structures.

There are two important considerations when using sequence objects: They may have gaps, and, in a data sharing environment, they may not be in strict sequential order. We discuss these next. These considerations also apply to identity columns.

## 5.4.1 Generated values may have gaps

There are various reasons why an application requesting a sequence number may not obtain the next sequential number. These are:

- Transaction advances sequence and then rolls back.
- SQL statement leading to generation of next value fails after the value is generated.
- NEXTVAL used in SELECT statement of cursor in DRDA where client uses block-fetch and not all retrieved rows are FETCHed.
- Sequence (or identity column) associated with sequence is altered and then ALTER rolled back.
- Sequence (or identity column) table DROPped and then DROP rolled back.
- SYSIBM.SYSSEQ table space is stopped, leading to loss of unused cache values.
- DB2 system failure or shut-down leading to loss of unassigned cache values causing gap in sequence.

You need to be aware of these possibilities, which are possible in any environment (data sharing or otherwise).

## 5.4.2 Generated values may not be in strict sequential order

In a data sharing environment, strict sequential order cannot be guaranteed when sequence numbers are cached on multiple members.

Before explaining how this is possible, we discuss two keywords: CACHE/NO CACHE and ORDER/NO ORDER that impact how the sequence object behaves:

- CACHE/NO CACHE

  The CACHE/NO CACHE keywords specify whether or not sequenced values will be preallocated in memory. In a data sharing environment, each member has its own cache.
- ORDER/NO ORDER

  The ORDER/NO ORDER keywords specify whether or not the sequence numbers must be generated in order of request.

### Impact of caching in a data sharing environment

Note that each member has its own cache and there is only one SYSIBM.SYSSEQUENCES table from which to request values. Assume an application is on two members of a 2-way data sharing system at the same time. If caching is active, each application can alternately request a sequence on each member and the sequence would be satisfied from that member's set of cached sequence numbers. For example, if CACHE is set to 20, each member would have a set of 20 cached values: member 1 could have cached 1 through 20 and member 2 could have cached 21 through 40. If the application alternated between members, the sequence order would be 1, 21, 2, 22, 3, 23, 4, and so forth. These numbers are clearly not in sequence.

### Summary and recommendations

Table 5-1 summarizes the previous discussion and offers recommendations.

*Table 5-1 Sequence objects: Impact of caching on order*

| Data Sharing Environment? | ORDER keyword | CACHE keyword | Impact |
|---|---|---|---|
| Data Sharing | ORDER | CACHE | Order is assured, but it may disable the caching of values (OK with single application process). |
| | | NO CACHE | Order assured. |
| | NO ORDER | CACHE | Order not assured (application must be able to tolerate), good performance. |
| | | NO CACHE | Order not assured (application must be able to tolerate), no reason to choose since performance improvement possible. |
| Non-Data Sharing | ORDER | CACHE | DB2 changes CACHE to NO CACHE. |
| | | NO CACHE | Order is assured and integrity dictates that you sacrifice performance. |
| | NO ORDER | CACHE | Order not assured (application must be able to tolerate), good performance. |
| | | NO CACHE | Order not assured (application must be able to tolerate), no reason to choose since performance improvement possible with CACHE. |

## 5.5 Informational RI

Informational RI constraints are constraints that are ignored by some aspects of DB2 and recognized by others. We discuss this in 5.5.1, "What is informational RI" on page 161. In 5.5.2, "Impact on utilities" on page 162, we discuss how each utility is affected by informational RI, and, in 5.5.3, "Impact on MQT usage" on page 164, we offer recommendations about the appropriate use of informational RI.

### 5.5.1 What is informational RI

An informational referential constraint is a referential constraint that is not enforced by DB2 during SQL operations: insert, update, and delete. Some utilities ignore these constraints (those that verify integrity) while other utilities recognize them (those that report only).

This example shows how an informational RI constraint is created. Suppose you have created a DEPT and EMPL tables, as follows:

```
CREATE TABLE DEPT
    (DEPTID   CHAR(3) NOT NULL PRIMARY KEY,
     DEPTNAME CHAR(30) )
IN DSNDB04.RIDEPT ;
```

and

```
CREATE TABLE EMPL
    (EMPLID   CHAR(6) NOT NULL,
     DEPTID   CHAR(3) NOT NULL,
     EMPLNAME CHAR(30))
   IN DSNDB04.RIEMPL ;
```

You have also created an index on the DEPT table to support the primary key as shown below:

```
CREATE UNIQUE INDEX DEPTKO
  ON DEPT
   (DEPTID                ASC)
  USING STOGROUP SG247111
  PRIQTY 48 SECQTY 48
  ERASE  YES
  FREEPAGE 10 PCTFREE 10
  GBPCACHE CHANGED
  NOT CLUSTER
  BUFFERPOOL BPO
  CLOSE YES
  COPY NO
  PIECESIZE 2 G ;
```

To create a foreign key where DB2 enforces the RI constraints, you use:

```
ALTER TABLE EMPL
  ADD FOREIGN KEY (DEPTID) REFERENCES DEPT ON DELETE RESTRICT;
```

However, to create the same constraint as an informational constraint, you issue:

```
ALTER TABLE EMPL
  ADD FOREIGN KEY (DEPTID) REFERENCES DEPT NOT ENFORCED ;
```

Informational RI could be useful in a phased migration approach. For example, a LISTDEF with keyword RI could help in the scenario described at 3.12.1, “Planning considerations” on page 97.

## 5.5.2 Impact on utilities

As mentioned earlier, some utilities ignore informational RI constraints while others recognize them. We discuss the details, with examples, below.

- LOAD and CHECK DATA do not enforce informational RI constraints.
- REPORT TABLESPACESET also reports all tables spaces related to the named table space through informational RI constraints.

  Here is an example of REPORT TABLESPACESET.

  ```
  REPORT TABLESPACESET DSNDB04.RIDEPT
  ```

  produces the following:

  ```
  DSNU050I    DSNUGUTC -  REPORT TABLESPACESET DSNDB04.RIDEPT
  DSNU587I  -DB8A DSNUPSET - REPORT TABLESPACE SET WITH TABLESPACE DSNDB04.RIDEPT

  TABLESPACE SET REPORT:


  TABLESPACE       : DSNDB04.RIDEPT

     TABLE         : PAOLOR4.DEPT
        INDEXSPACE : DSNDB04.DEPTKO
             INDEX : PAOLOR4.DEPTKO
        DEP  TABLE : PAOLOR4.EMPL

  TABLESPACE       : DSNDB04.RIEMPL

     TABLE         : PAOLOR4.EMPL
  DSNU580I    DSNUPORT - REPORT UTILITY COMPLETE - ELAPSED TIME=00:00:00
  ```

```
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

which includes the EMPL related via an informational RI constraint.

- QUIESCE TABLESPACESET also reports all tables spaces related to the named table space through informational RI constraints.

  An example of QUIESCE TABLESPACESET is shown below:

  ```
  QUIESCE TABLESPACESET  DSNDB04.RIDEPT
  ```

  produces the following:

  ```
  DSNU050I    DSNUGUTC -  QUIESCE TABLESPACESET DSNDB04.RIDEPT
  DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACESET DSNDB04.RIDEPT
  DSNU477I  -DB8A DSNUQUIA -    QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIDEPT
  DSNU477I  -DB8A DSNUQUIA -    QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIEMPL
  DSNU474I  -DB8A DSNUQUIA - QUIESCE AT RBA 0002105DA1DF AND AT LRSN 0002105DA1DF
  DSNU475I    DSNUQUIB - QUIESCE UTILITY COMPLETE, ELAPSED TIME= 00:00:00
  DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
  ```

  which includes the EMPL related via an informational RI constraint.

- LISTDEF also includes all table spaces related through informational RI constraints when the keyword RI is specified.

  A LISTDEF example is shown below:

  ```
  OPTIONS PREVIEW
  LISTDEF MYTABLES INCLUDE TABLESPACE  DSNDB04.RIDEPT RI
  ```

  produces:

  ```
  DSNU1000I   DSNUGUTC - PROCESSING CONTROL STATEMENTS IN PREVIEW MODE
  DSNU1035I   DSNUILDR - OPTIONS STATEMENT PROCESSED SUCCESSFULLY
  DSNU050I    DSNUGUTC -  LISTDEF MYTABLES INCLUDE TABLESPACE DSNDB04.RIDEPT RI
  DSNU1035I   DSNUILDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
  DSNU1020I -DB8A DSNUILSA - EXPANDING LISTDEF MYTABLES
  DSNU1021I -DB8A DSNUILSA -   PROCESSING INCLUDE CLAUSE TABLESPACE DSNDB04.RIDEPT
  DSNU1022I -DB8A DSNUILSA - CLAUSE IDENTIFIES 2 OBJECTS
  DSNU1023I -DB8A DSNUILSA - LISTDEF MYTABLES CONTAINS 2 OBJECTS
  DSNU1010I   DSNUGPVV - LISTDEF MYTABLES EXPANDS TO THE FOLLOWING OBJECTS:
              LISTDEF MYTABLES -- 00000002 OBJECTS
                INCLUDE TABLESPACE DSNDB04.RIDEPT
                INCLUDE TABLESPACE DSNDB04.RIEMPL
  DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
  ```

  which includes the EMPL related via an informational RI constraint.

  Alternatively, you can use the LISTDEF utility to generate the control cards needed for another utility, such as QUIESCE. An example is shown below:

  ```
  LISTDEF MYTABLES INCLUDE TABLESPACE  DSNDB04.RIDEPT RI
  QUIESCE LIST MYTABLES
  ```

  produces:

  ```
  DSNU050I    DSNUGUTC -  LISTDEF MYTABLES INCLUDE TABLESPACE DSNDB04.RIDEPT RI
  DSNU1035I   DSNUILDR - LISTDEF STATEMENT PROCESSED SUCCESSFULLY
  DSNU050I    DSNUGUTC -  QUIESCE LIST MYTABLES
  DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIDEPT
  DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIEMPL
  DSNU474I  -DB8A DSNUQUIA - QUIESCE AT RBA 000210C7F8BE AND AT LRSN 000210C7F8BE
  DSNU475I    DSNUQUIB - QUIESCE UTILITY COMPLETE, ELAPSED TIME= 00:00:00
  DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
  ```

  which includes the EMPL related via an informational RI constraint.

The list generated by LISTDEF can be saved for later use. This is illustrated in Example 6-18 on page 222.

### 5.5.3 Impact on MQT usage

Automatic query rewrite is a general process within DB2 where a query is evaluated and possibly rewritten for optimization purposes. For MQTs, the process examines a submitted query that references source tables and, if appropriate, rewrites the query so that performance is optimized by accessing MQTs. This process uses informational RI constraints to determine whether or not it can use a materialized query table and can result in significant reduction in query runtime, especially for decision-support queries that operate over huge amounts of data.

This example shows how an original query is rewritten.

Suppose you have created tables DEPT, EMPL, and EMPLHIST as shown:

```
CREATE TABLE DEPT
    (DEPTID   SMALLINT NOT NULL PRIMARY KEY,
     DEPTNAME CHAR(30) )
IN DSNDB04.DEPT ;

CREATE TABLE EMPL
    (EMPLID   INTEGER NOT NULL PRIMARY KEY,
     DEPTID   SMALLINT NOT NULL,
     EMPLNAME CHAR(30))
   IN DSNDB04.EMPL ;

CREATE TABLE SALHIST
    (EMPLID   INTEGER NOT NULL,
     SEQNO    SMALLINT NOT NULL,
     RAISE    DECIMAL(7,2))
   IN DSNDB04.SALHIST ;
```

In addition, you have created the indexes to support the primary and implied foreign keys as follows:

```
CREATE UNIQUE INDEX DEPTK0
  ON DEPT
   (DEPTID                ASC)
...
```

and

```
CREATE UNIQUE INDEX EMPLK0
  ON EMPL
   (EMPLID                ASC)
...
```

You have created a materialized query table that contains the department summary as shown:

```
CREATE TABLE DEPTSUM AS
   (SELECT D.DEPTID     AS SUMDEPT,
           SUM(S.RAISE) AS TOTRAISE
    FROM   DEPT    D,
           EMPL    E,
           SALHIST S
    WHERE  D.DEPTID = E.DEPTID
    AND    E.EMPLID = S.EMPLID
    GROUP BY D.DEPTID)
```

```
DATA INITIALLY DEFERRED REFRESH DEFERRED ;
```

Notice that the MQT is a join among three tables.

Now, you run the following query:

```
SELECT    E.DEPTID,
          SUM(S.RAISE)
FROM      EMPL     E,
          SALHIST  S
WHERE     E.EMPLID   = S.EMPLID
GROUP BY  E.DEPTID
ORDER BY  E.DEPTID ;
```

The resulting access path displayed by Visual Explain, shown in Figure 5-1, indicates that both base tables are accessed.

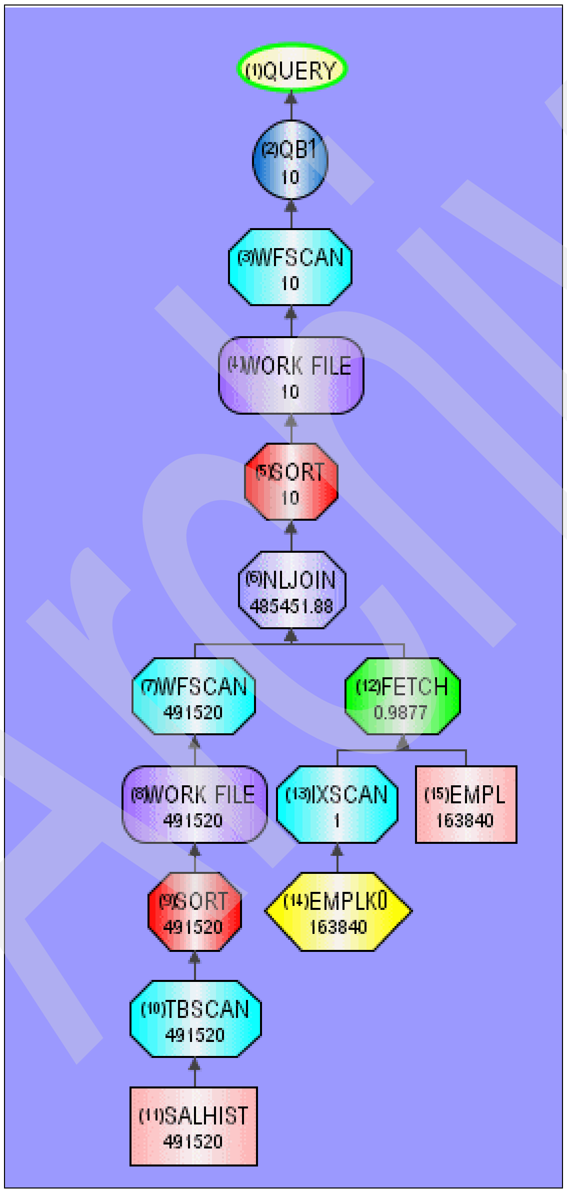


*Figure 5-1   Access path without informational RI*

Now suppose you create an informational RI between the DEPT and EMPL table as shown:

```
ALTER TABLE EMPL
  ADD FOREIGN KEY (DEPTID) REFERENCES DEPT NOT ENFORCED
```

This lets DB2 know that the definition of the MQT, which contains the DEPT table, in addition to the EMPL and SALHIST tables, can be used to satisfy this query since the join to DEPT is a “lossless” join. This means no rows are eliminated by including the DEPT table. Since the MQT is substantially smaller than the base tables (10 rows versus 80,000 and 240,000 rows for the base tables in our study), DB2 chooses to use the MQT instead. The resulting access path displayed by Visual Explain is shown in Figure 5-2. Needless to say, this performs substantially better.

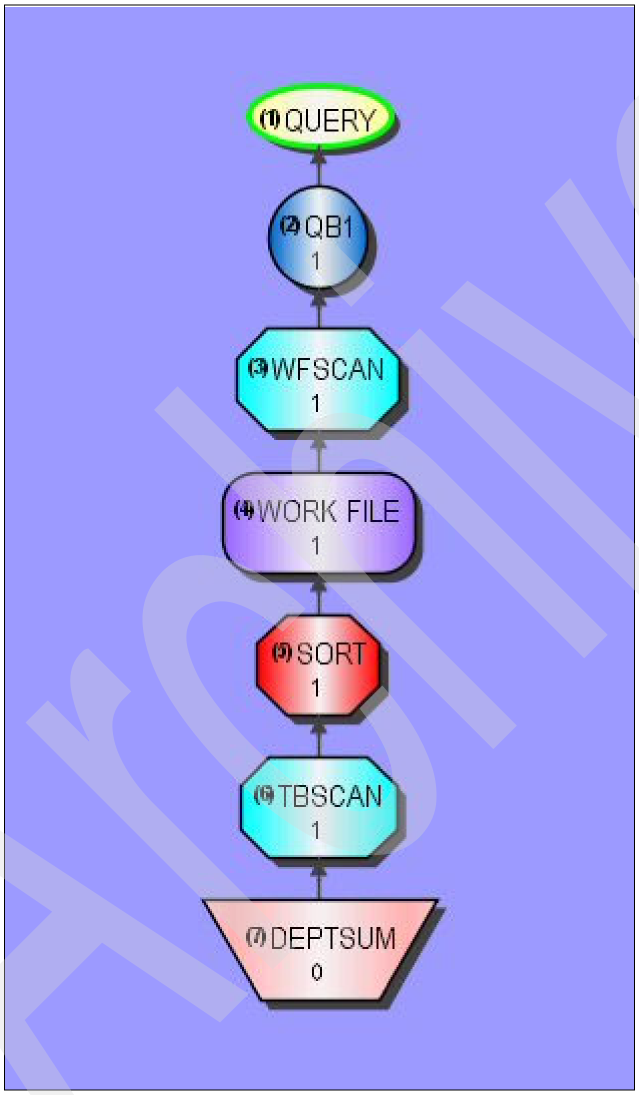


Figure 5-2 Access path with informational RI

**Important:** Keep in mind that, for the join to be “lossless”, the definition of the column must not allow NULLs. In addition, you must issue (or set the proper option in Visual Explain) as follows:

SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = ALL ;

and

SET CURRENT REFRESH AGE = ANY ;

In this example, we created EMPL table as follows:

```
CREATE TABLE EMPL
    (EMPLID   INTEGER NOT NULL PRIMARY KEY,
     DEPTID   SMALLINT NOT NULL,
     EMPLNAME CHAR(30))
RI   IN DSNDB04.EMPL ;
```

If we created the DEPTID to allow nulls, the join is NOT “lossless” and an MQT will *not* be considered.

### 5.5.4 Usage recommendations

These are some of the business scenarios where informational RI constraints may be useful:

- Application-enforced RI is in effect.

  While you do not want to have DB2 enforce the rules, you can still take advantage of utilities like REPORT TABLESPACESET and QUIESCE TABLESPACESET which allow you to identify a set of application-related objects and to create a consistency point among objects that are linked via application-enforced RI. The DB2 catalog also provides a means of documenting these rules.
- Enforcement may be unnecessary in a data warehouse.

  Typically, data in a data warehouse environment has been extracted from other sources and cleansed. RI may already be guaranteed. In this case, informational RI constraints can be safely used as a means of documenting the business rules.
- To encourage automatic query rewrite.

  *Automatic query rewrite* is a process that examines a submitted query that references source tables and, if appropriate, rewrites the query so that it executes against a materialized query table. We discussed this in 5.5.3, “Impact on MQT usage” on page 164.

## 5.6 Locking

The term *locking* refers to the collective set of serialization techniques used by DB2 to ensure the integrity of data in the database. It is important for the application developer to understand the impact of locking on concurrent accesses to D2 data. This topic is covered in detail in Chapter 12, “What you need to know about locking” in the redbook, *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134. In this chapter, we only provide some further information related to data sharing.

## 5.6.1 Data sharing implications

DB2 data sharing introduces additional locks to manage the data concurrency and data integrity when more than one member of the data sharing group post interest in the same DB2 object. We briefly outline the concepts and usage of local, global, logical, physical, and parent and child locks. Figure 5-3 shows the usage of the coupling facility by DB2 and the three structures. The lock structure is the repository of the global locks and it is the communication area for all the members to get information and negotiate global locks. We do not discuss these locks in detail but just refresh the terminology for the description of protocol level (2) locking in 5.6.2, “Locking protocol level 2” on page 170.

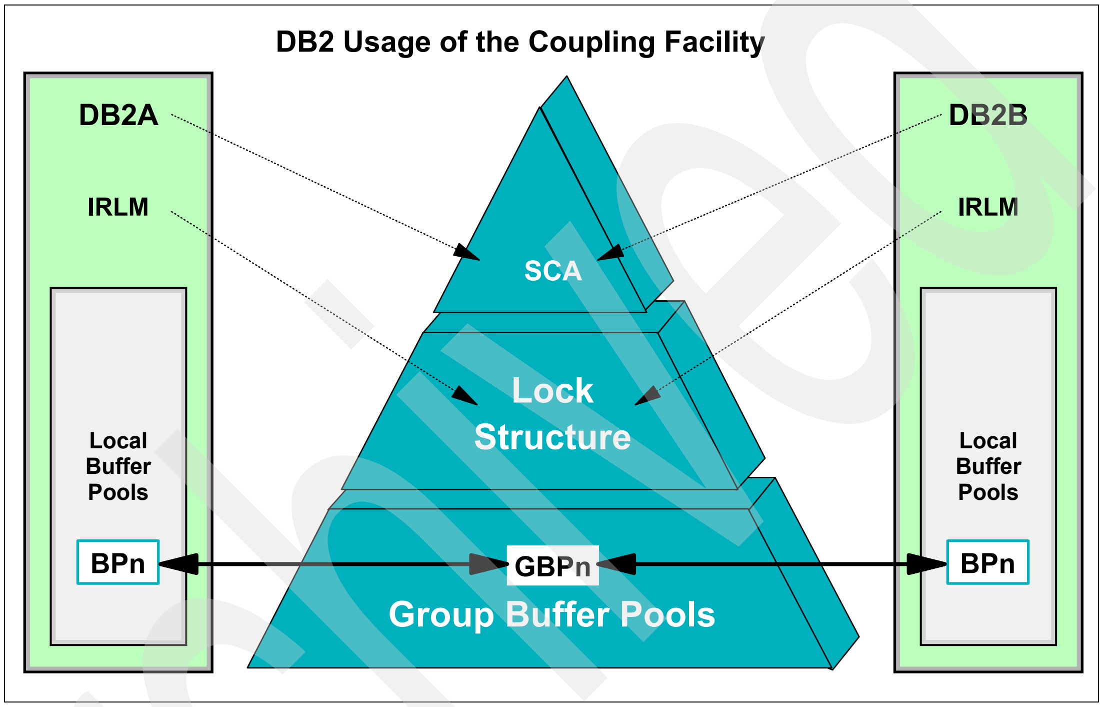


*Figure 5-3 DB2 and coupling facility*

Refer to 11.1 of *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More,* SG24-6079, for a general review of locking in a data sharing environment.

These are the types of locks:

- Local locks

  These are locks that we get in non-data sharing subsystems that we continue to get in data sharing groups.

- Global locks

  A global lock is a lock that the DB2 subsystems in a data sharing group have to make known to other member subsystems in the group. The lock structure in the coupling facility, part of the Parallel Sysplex, is the communication agent for the global locks. The DB2 member requesting the global lock sends the request to the DB2 lock structure. All other members in the data sharing group automatically have access to the information. The global lock communication is also referred to as *global lock propagation*.

- Logical locks (*L-Locks*)

  Logical locks are also referred to as *transaction locks*. L-Locks are used to serialize access to data to ensure data consistency.

L-Locks are owned by a transaction, and the lock duration is controlled by the transaction. For example, the lock is generally held from the time the application issues an update until the time it issues a commit. (Exceptions are share locks associated with cursors defined WITH HOLD and table space and partition locks acquired by SQL associated with plans and packages bound using RELEASE(DEALLOCATE).) The locks are controlled locally per member by each member's IRLM.

- Parent lock and child lock

  Parent locks and child locks refer to the L-Locks hierarchy. The parent locks are at table space and partition level while child locks are at page and row level. Parent locks are propagated to global lock structure. The child lock propagation to global structure is determined by the parent lock's conflict.

- Physical locks (*P-Locks*):

  Physical locks are used to do many things. We discuss the two most commonly used P-Locks: page set P-Locks and page P-Locks. Other types of P-Locks include DBD, castout, GBP structure, index tree, and repeatable read tracking P-Locks.

  – Page set physical locks:

    Page set P-Locks are used to track inter-DB2 read-write interest, therefore, determining when a page set has to become GBP-dependent.

    When a DB2 member requires access to a page set or partition, a page set P-Lock is taken. This lock is always propagated to the lock table in the coupling facility and is owned by the member. No matter how many times the resource is accessed through the member, there will always be only one page set P-Lock for that resource for a particular member. This lock will have different modes depending on the level (read or write) of interest the member has in the resource.

    The first member to acquire a page set P-Lock on a resource takes the most restrictive mode of lock possible, that is, an S page set P-Lock for read or an X page set P-Lock for write interest. An X page set P-Lock indicates that the member is the only member with interest (read/write) in the resource. Once another member becomes interested in the resource, the page set P-Lock mode can be negotiated; that is, it can be made less restrictive if the existing page set P-Lock is incompatible with the new page set P-Lock request.

    The negotiation always allows the new page set P-Lock request to be granted, except when there is a retained X page set P-Lock. A retained P-Lock cannot be negotiated. (Retained locks are locks that must be kept to protect possibly uncommitted data left by a failed DB2 member.) Page set P-Lock negotiation signifies the start of GBP dependence for the resource.

    Although it may seem strange that a lock mode can be negotiated, remember that page set P-Locks do not serialize access to a resource; they are used to track which members have interest in a resource and to determine when a resource must become GBP-dependent.

    Page set P-Locks are released when a page set or partition data set is closed. The mode of page set P-Locks is downgraded from R/W to R/O when the page set or partition is not updated within an installation-specified time period or a number of system checkpoints. When page set P-Locks are released or downgraded, GBP dependency is reevaluated.

  – Page physical locks:

Page P-Locks are used to ensure the physical consistency of a page across members of a data sharing group in much the same manner as latches do in a non-data sharing environment. A page P-Lock protects the page while the structure is being modified. Page P-Locks are used when row locking is in effect, or when changes are being made to GBP-dependent space map pages. Page physical locks are also used to read and update index pages.

The lock information for P-Locks and L-Locks is stored in the same places: the IRLM, XES, and the coupling facility.

### Explicit hierarchical locking

Conceptually, all locks taken in a data sharing environment are global locks; that is, they are effective group-wide, even though all locks do not have to be propagated to the lock structure in the coupling facility.

DB2 data sharing has introduced the concept of *explicit hierarchical locking* to reduce the number of locks that must be propagated to the coupling facility.

Within IRLM, a hierarchy exists between certain types of L-Locks, where a parent L-Lock is the lock on a page set (table space or partition) and a child L-Lock is the lock held on either the table, data page, or row within that page set. For partitioned table spaces, if you use LOCKPART YES in DB2 V7, each locked partition is a parent of the child locks held for that partition. If you use LOCKPART NO (the default in V7), the last data partition is the parent lock for all child locks.

By using explicit hierarchical locking, DB2 is able to reduce the number of locks that must be propagated to the lock structure in the coupling facility. The number of locks propagated to the lock structure for a page set or partition is determined by the number of DB2 members interested in the page set and whether their interest is read or write. Wherever possible, locks are granted locally and not propagated to the coupling facility.

If an L-Lock has already been propagated to XES, protecting a particular resource for this member, subsequent lock requests for the same lock do not have to be sent to XES by the same member for the same resource. They can be serviced locally. In addition, a parent L-Lock is propagated only if it is more restrictive than the current status that XES knows about for this resource from this member.

Parent L-Locks are released either when the transaction commits, or when the thread terminates, depending on the value you have specified for the RELEASE parameter on the bind. Child L-Locks are propagated to the lock table in the coupling facility only when there is inter-DB2 interest for the page set.

Child L-Locks (page and row locks) are propagated to XES and the coupling facility based on inter-DB2 interest on the parent (table space or partition) lock. If all the table space locks are IS, then no child locks are propagated. If there is a parent S lock on the table space or partition, then all the child locks must be propagated. If there is a parent X lock on the table space or partition, then only the X child locks must be propagated.

## 5.6.2 Locking protocol level 2

Locking protocol level 2 remaps parent IX L-Locks from XES-X to XES-S locks. Data sharing locking performance benefits because this allows IX and IS parent global L-Locks to be granted without invoking global lock contention processing to determine that the new IX or IS lock is compatible with existing IX or IS locks.

The purpose of this enhancement is to avoid the cost of global contention processing whenever possible. It will also improve availability due to a reduction in retained locks following a DB2 subsystem or MVS system failure.

XES contention caused by parent L-Locks is reasonably common in a data sharing environment. On page set open (an initial open or open after a pseudo close), DB2 normally tries to open the table space in RW. (DB2 rarely opens a table space in RO.) To do this, DB2 must ask for an L-Lock. This L-Lock is normally an IS or IX L-Lock, depending on whether or not a SELECT or UPDATEable SQL statement caused the table space to open. If any other DB2 member already has the table space open for RW, global lock contention generally occurs.

DB2 V8 attempts to reduce this global contention by remapping parent IX L-Locks from XES-X to XES-S locks. Data sharing locking performance benefits because parent IX and IS L-Locks are now both mapped to XES-S locks and are therefore compatible and can now be granted locally by XES. DB2 no longer needs to wait for global lock contention processing to determine that a new parent IX or IS L-Lock is compatible with existing parent IX or IS L-Locks.

The majority of parent L-Lock contention occurs when the table space is opened by at least one DB2 member in RW:

- IS-IS: No contention. We want to execute some read-only SQL against a table space and there are some other members who currently have some read-only SQL active against the same table space.
- IS-IX: We want to execute some read-only SQL against a table space and there are some other members who currently have some update SQL active against the same table space.
- IX-IS: We want to execute some update SQL against a table space and there are some other members who currently have some read-only SQL active against the same table space.
- IX- IX: We want to execute some update SQL against a table space and there are some other members who currently have some update SQL active against the same table space.

Parent lock contention with parent S L-Locks is less frequent than checking for contention with parent IS and IX L-Locks. Parent S L-Locks are only acquired when DB2 is about to issue read-only SQL against a table space opened for RO.

To ensure that parent IX L-Locks remain incompatible with parent S L-Locks, S table and table space L-Locks are remapped to XES-X locks. This means that additional global contention processing will now be done to verify that an S L-Lock is compatible with another S L-Lock, but this is a relatively rare case (executing read-only SQL against a table space, where at least one other DB2 member has the table space open in RO and currently has some read-only SQL active against the same table space).

Another impact of this change is that child L-Locks are no longer propagated based on the parent L-Lock. Instead, child L-Locks are propagated based on the held status of the table space P-Lock. If the table space P-Lock is negotiated from X to SIX or IX, then child L-Locks must be propagated.

It may be that some child L-Locks are acquired before the page set P-Lock is obtained. In this case, child L-Locks will automatically be propagated. This situation occurs because DB2 always acquires locks before accessing the data. In this case, DB2 acquires that L-Lock before opening the table space to read the data. It can also happen during DB2 restart.

An implication of this change is that child L-Locks will be propagated for longer than they are needed; however, this should not be a concern. There will be a short period from the time where there is no intersystem read/write interest until the table space becomes non-GBP-dependent, that is, before the page set P-Lock reverts to X. During this time, child L-Locks will be propagated unnecessarily.

It is now important that L-Locks and P-Locks are maintained at the same level of granularity. Remember that page set P-Locks now determine when child L-Locks must be propagated to XES.

For partitioned table spaces defined with LOCKPART NO, prior versions of DB2 lock only the last partition to indicate we have a lock on the whole table space. There are no L-Locks held on each of the partition page sets. So, when should we propagate the child L-Locks for the various partitions that are being used? We cannot tell by looking at the table space parent L-Locks that we need to propagate the child locks since there no longer is a lock contention conflict that can trigger child lock propagation, and we cannot determine how each partition page set is being used by looking at the page set P-Locks that are held at the partition level, while the parent L-Lock is at the table space level with LOCKPART NO.

To overcome this problem, DB2 V8 obtains locks at the part level. LOCKPART NO behaves the same as LOCKPART YES.

In addition, LOCKPART YES is not compatible with LOCKSIZE TABLESPACE. However, if LOCKPART NO and LOCKSIZE TABLESPACE are specified, then we will lock every partition, just as every partition is locked today when LOCKPART YES is used with ACQUIRE(ALLOCATE). With this change, you may see additional locks being acquired on individual partitions even though LOCKPART(NO) is specified.

This change to the LOCKPART parameter behavior applies to both data sharing and non-data sharing environments.

Since the new locking protocol cannot coexist with the old, the new protocol will only take effect after the first group-wide shutdown and restart after the data sharing group is in new-function mode (NFM). You do not have to delete the lock structure from the coupling facility prior to restarting DB2 in order to trigger DB2 on group restart to build a new lock structure. In addition, Protocol Level 2 will not be enabled if you merely ask DB2 to rebuild the lock structure while any DB2 member remains active.

The new mapping takes effect after the restart of the first member, after successful quiesce of all members in the DB2 data sharing group. So, a group-wide outage is required to enable this feature.

No other changes are required to take advantage of this enhancement.

You can use the -DISPLAY GROUP command to check whether the new locking protocol is used (mapping IX IRLM L-Locks to an S XES lock). Example 5-1 shows the output from a -DISPLAY GROUP command which shows Protocol Level 2 is active.

*Example 5-1 Sample -DISPLAY GROUP output*

```
DSN7100I  -DT21 DSN7GCMD
 *** BEGIN DISPLAY OF GROUP(DSNT2   ) GROUP LEVEL(810) MODE(N)
                   PROTOCOL LEVEL(2)  GROUP ATTACH NAME(DT2G)
 --------------------------------------------------------------------
 DB2                                   DB2 SYSTEM    IRLM
 MEMBER   ID  SUBSYS CMDPREF   STATUS   LVL NAME      SUBSYS IRLMPROC
 -------- --- ----   --------  -------- --- --------  ----   --------
 DT21       1 DT21   -DT21     ACTIVE   810 STLABB9   IT21   DT21IRLM
 DT22        3 DT22   -DT22     FAILED   810 STLABB6   IT22   DT22IRLM
```

The use of locking Protocol Level 2 requires that the PTFs for the following APARs are applied: PQ87756, PQ87168, and PQ86904 (IRLM).

See *DB2 UDB for z/OS Version 8 Performance Topics,* SG24-6465, for the performance advantages of Protocol Level 2 locking.

# 6 Recovery

In this chapter, we look at the characteristics of recoverability of DB2 applications by analyzing the DB2 recoverability functions.

We begin with the basic concepts of the DB2 subsystem structure and the definitions needed to understand DB2's unit of recovery. We then look at the recovery of transaction data by the DB2 subsystem and when the subsystem restarts after an abnormal failure. We also look at the recovery of application data after an I/O subsystem failure. In the rest of the chapter, we discuss the processes involved in ensuring the recovery of application data by application programmers and DBAs.

This chapter contains the following:

- DB2 attachment facilities
- DB2 commit process
- Unit of work
- Data integrity
- Scrollable cursors
- DB2 subsystem restart after abend
- Recovery of objects in error
- Application recovery process
- Preparing to recover to a point of consistency

We do not provide here disaster recovery considerations. For details about recovering DB2 subsystems and user data, refer to *Disaster Recovery with DB2 UDB for z/OS,* SG24-6370.

## 6.1 DB2 attachment facilities

An attachment facility can be thought of as a required portal or gateway when a user request to connect to a DB2 subsystem originates from certain environments. In order to communicate with DB2 for z/OS from a Customer Information Control System (CICS), Information Management System (IMS), Time Sharing Option (TSO), batch or WebSphere (when using RRS) environment, a DB2 attachment facility is required to establish a session. Attachment facilities are subcomponents of DB2 that run in the user's address space.

### CICS

The CICS attachment facility, language interface module DSNCLI, is provided by the CICS product. It receives CICS application requests and passes them to DB2. Example 6-1 shows the standard CICS command-level services that you can use.

*Example 6-1 CICS command-level services used with the CICS attachment facility*

```
EXEC CICS WAIT
EXEC CICS ABEND
```

By issuing DB2 commands, an authorized CICS terminal operator is able to control and monitor DB2 and the attachment facility.

### IMS

The IMS attachment facility, program interface DFSLI000, is required to access DB2 from an IMS environment. It receives and interprets requests for access to DB2 databases using exits provided by IMS subsystems. The IMS attachment facility also allows an authorized IMS terminal operator the ability to issue DB2 commands such as starting and stopping DB2 databases.

### TSO

The TSO attachment facility offers unique functionality. It is required for binding application plans and packages and executing online DB2 functions. In fact, the CICS and IMS attachment facilities depend on the TSO attachment facility for these functions.

Access to DB2 in the foreground through a TSO terminal or in batch mode through the TSO terminal monitor program (TMP) can be accomplished with the TSO attachment facility. In addition, two command processors are provided:

- DSN command processor: Primarily used for batch jobs, uses the TSO attachment facility and runs as a TSO command processor
- DB2 Interactive (DB2I): An interactive connection to DB2 allowing users to run SQL statements, issue DB2 commands, and run DB2 utilities using Interactive System Productivity Facility (ISPF) panels

With the TSO attachment facility, load module DSNELI, access to DB2 is fairly simple. On the other hand, this also means that an application has less control over the status of the connections and, when using DSN services, applications running under the control of DSN.

### Call attachment facility (CAF)

As the alternative attachment facility for TSO and batch applications, CAF supplies additional connection functionality and greater control over the execution environment including:

- Explicit control over the status of the connection to DB2.
- Used with or without TSO TMP.

- DB2 version verification.
- Translation of DB2 reason codes, return codes, and abend codes into customized messages.
- Establish an implicit connection to DB2 (with a default subsystem identifier and a default plan name) by using SQL statements or instrumentation facility interface (IFI) calls without first calling CAF (available in all currently supported DB2 versions).

CAF uses language interface module DSNALI.

## Resource Recovery Services (RRS)

RRS is a component of the z/OS operating system that provides system-wide coordination of commit processing (including two-phase commit) over the life of a transaction. With the additional functionality included in z/OS RRS, the RRS attachment facility (RRSAF) has become the "successor" to CAF. With RRSAF, it is possible to:

- Coordinate DB2 updates with updates made by all other resource managers that also use z/OS RRS in an z/OS system.
- Establish an implicit connection to DB2 (with a default subsystem identifier and a default plan name) by using SQL statements or instrumentation facility interface (IFI) calls without first calling RRSAF (in DB2 V8).

z/OS RRS is a started task that can be started and stopped independently of DB2. RRS must be started before application programs using RRSAF connect to DB2. Applications must load the RRSAF language interface module DSNRLI. An application can connect to DB2 using RRSAF in two ways:

- Implicitly, by SQL statements or IFI calls
- Explicitly, by using CALL DSNRLI statement to invoke RRSAF functions

  To use RRSAF, load module DSNRLI provided by DB2 must be available, and the z/OS RRS component must be started.

Programming information for RRSAF can be found at:

http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.doc.apsg/bjnqmstr632.htm

and in Part 6 of *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415.

Figure 6-1 illustrates the logical flow of data when a request is made from a z/OS or distributed environment. Depending on the originating environment, a z/OS request uses the associated attachment facility to establish communication with the DB2 for z/OS subsystem. For example, a CICS request would use the CICS attachment facility to establish a connection. Once the connection is established, the general flow of data begins in the MSTR address space. This is where the request is initially validated and the necessary output messages, if any, are generated. Once the MSTR address space declares the request to be valid, data flow continues to the DBM1 address space. The DBM1 address space does much of the legwork. However, the IRLM address space must provide the necessary locking before and data is ultimately passed back to the requester to insure data integrity. Once the locks have been acquired, the data is retrieved and passed back to the requester.

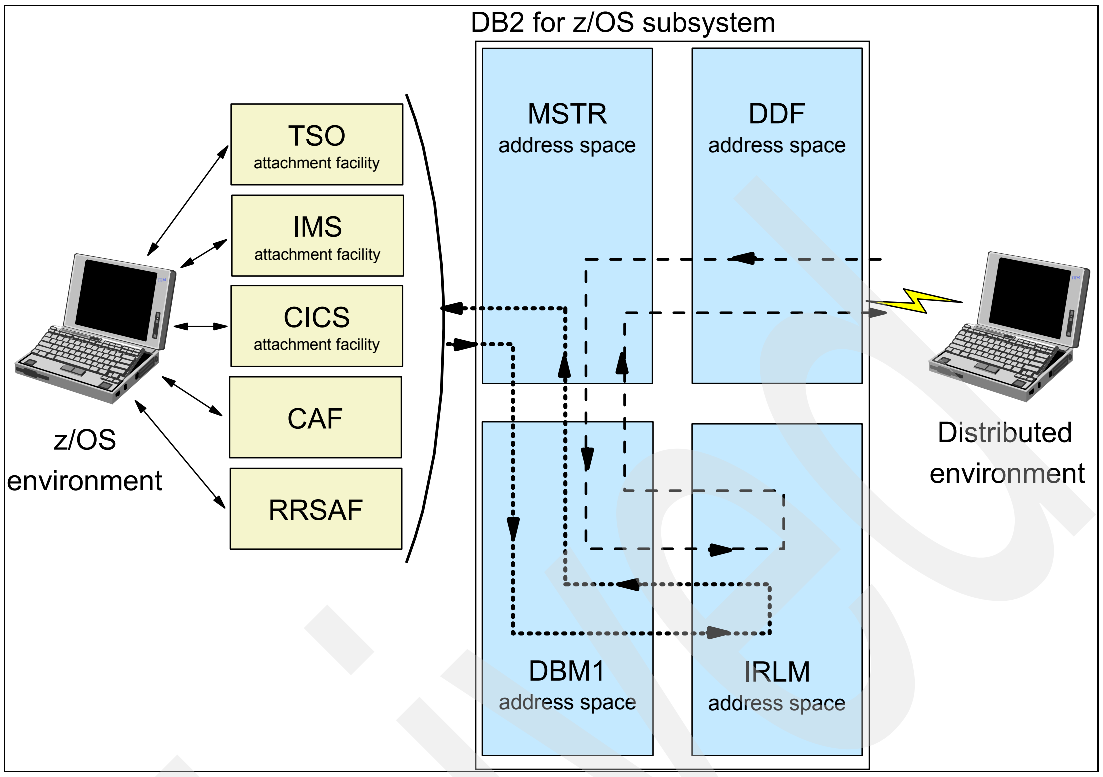


*Figure 6-1 Basic data flow in a DB2 subsystem from z/OS and distributed environments*

A request from a distributed environment is very similar to the data flow originating from within a z/OS environment (local attachment). The significant difference is using the DDF address space to establish a connection as opposed to an attachment facility.

This is also the case when using JDBC Type 4 connectivity provided by the IBM DB2 Universal JDBC Driver to connect to a DB2 for z/OS subsystem. In this scenario, the Universal Driver is acting as a DRDA application requester to establish a connection into the DDF address space (acting as a DRDA application server) of the DB2 for z/OS subsystem.

Commands can connect to DB2 in all of these environments and more. Stored procedures can be local or remote. Utilities use another connection and can also run as stored procedures.

## 6.2 DB2 commit process

Before we begin looking at the unit of work (UOW) and unit of recovery (UR), we briefly discuss the DB2 commit process in terms of subcomponent interactions and explain when recovery information is written to the log media. An understanding of the status of the system is useful if the system terminates abnormally.

We discuss the DB2 commit process as shown on Figure 6-2 and the role of DB2 as a coordinator and participant in the single-phase and two-phase commit processes.

Refer to chapter 19, “Maintaining consistency across multiple systems”, *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413, for details.

An application using DB2 notifies that it has reached a point of consistency using a commit point. DB2 does not proceed beyond a commit point until it has ensured that all work done before the commit point is fully protected against loss or invalidation.

- TSO applications notify a commit point using SQL COMMIT statement.
- IMS and CICS applications establish a sync point to establish a point of consistency.
- To commit work in RRSAF applications, you can use the CPIC SRRCMIT function or the DB2 COMMIT statement. To roll back work, you use the CPIC SRRBACK function or the DB2 ROLLBACK statement. See *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-03.
- All applications can use the rollback function to intentionally back out all data changes since the last commit point or application savepoint.

When DB2 restarts after an abnormal termination (see also 6.7, "DB2 subsystem restart after abend" on page 211):

- IN DOUBT threads are resolved.
- All uncommitted database changes are backed out.
- Committed changes that might not have been applied to disk are reapplied to ensure consistency of data.
- All committed database changes cannot be backed out.

## 6.3 Unit of recovery

A unit of recovery is the work, done by a single DB2 DBMS for an application, that changes DB2 data from one point of consistency to another. A unit of recovery begins with the first change to the data after the beginning of the job or following the last point of consistency and ends at a later point of consistency.

*Unit of recovery (UR)* is the information required for DB2 to backout all the application database changes since the last commit point. DB2 records all recovery information in a DB2 recovery log, which is made up of the active log and the archive log. These logs contain chronological entries that record significant DB2 activities of all users and jobs. The only exceptions are database changes done by utilities with LOG NO where the backup copies done with the COPY utility are absolutely essential for recovery to a consistency point prior to the execution of the utility.

Dual active logs and dual archive logs are strongly recommended to avoid recovery failures due to media failures. Changes are not updated to disk but queued in the log buffer until either commit or a buffer pool write threshold has been reached. If the same disk record is updated before the media is updated, the buffer is altered to reflect both changes. The archive logs are recorded in the bootstrap data set (BSDS). The total number of BSDS archive log entries is limited by the DSNZPARM parameter MAXARCH. Other DSNZPARM parameters related to active and archive logs are TWOACTV and TWOARCH.

In the data sharing environment, DB2 uses "a force-at-commit" policy for updates against page sets that are dependent on group buffer pool GBP. See 5.6.1, "Data sharing implications" on page 168 for the DB2 structures in the coupling facility. GBP caches data and maintains data consistency for group members.

DB2's role in a commit process as coordinator or participant is determined by the application:

- In TSO applications, DB2 acts as the commit *coordinator,* see 6.3.1, "Commit processing for TSO applications" on page 180.

- In a CICS and IMS commit process, DB2 is the *participant* and CICS or IMS is the coordinator. See 6.3.2, “Commit processing for CICS, IMS, or RRSAF applications” on page 181.
- In commit processing for RRSAF applications, DB2 is the participant and RRS is the coordinator.
- DB2 takes on the role of both coordinator and participant in environments where there are more than two systems.

## 6.3.1 Commit processing for TSO applications

In a TSO application, DB2 is the commit coordinator. The commit process goes through the following phases. Refer to Figure 6-2.

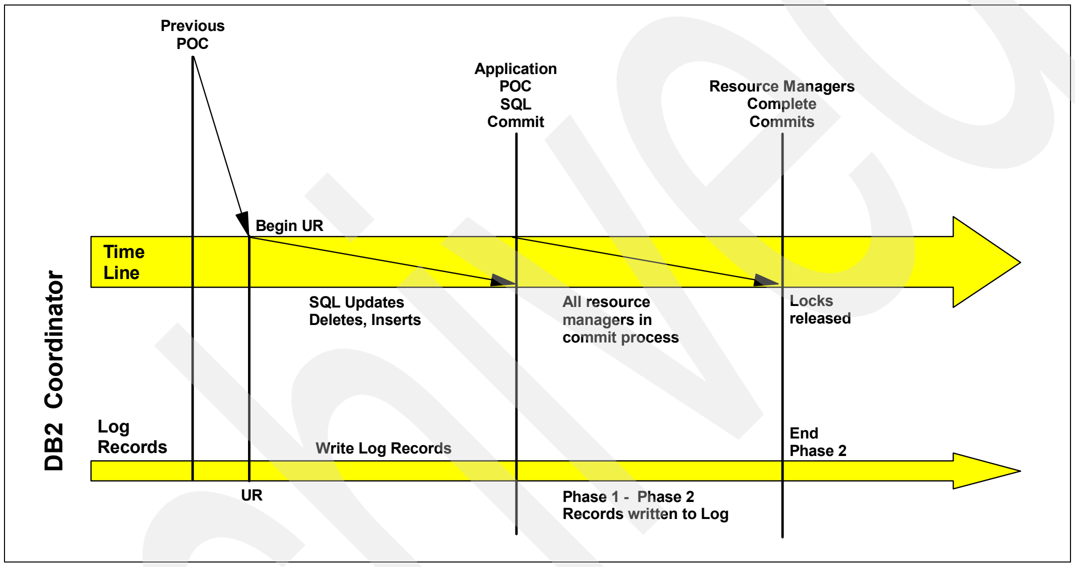

Previous POC
Application POC SQL Commit
Resource Managers Complete Commits
Begin UR
Time Line
SQL Updates Deletes, Inserts
All resource managers in commit process
Locks released
DB2 Coordinator
Log Records
Write Log Records
End Phase 2
UR
Phase 1 - Phase 2 Records written to Log

*Figure 6-2 DB2 as Coordinator: TSO Application*

- The unit of recovery begins with begin-UR log record.
- All application updates (INSERT, DELETE, and UPDATE) log records are queued and written to the log media immediately or later.
- The application reaches a point of consistency (POC) and issues a SQL COMMIT or normal termination.
- DB2 polls each resource manager involved in the commit process, and each responds about its ability to commit.

### Case one: Normal commit process

- All resource managers reply affirmatively.
- A phase 1-phase 2 transition record is queued to be written to the recovery log.
- DB2 waits for all of the queued output log records to be written to the log media.
- When all the resource managers have completed their commit processing:
  - An end-phase-2 record is queued for writing to the log.
  - All locks held on database objects are released except where the cursor is defined as WITH HOLD.

- If DB2 abends before completing the commit process, then log records for unit of recovery are redone when DB2 restarts, see 6.7, “DB2 subsystem restart after abend” on page 211 for details.

### Case two: Abnormal commit process

- If one or more of the resource managers fail in its commit process, then DB2 converts the commit operation into an *abort* (“undo”) operation.
- DB2 initiates a ROLLBACK and undoes all updates to the last point of consistency.
- If DB2 abends before completing the remainder of the commit process, then log records for unit of recovery are redone when DB2 restarts. See 6.7, “DB2 subsystem restart after abend” on page 211 for details.

## 6.3.2 Commit processing for CICS, IMS, or RRSAF applications

DB2 acts as the participant when CICS, IMS, or RRS is the coordinator in the commit processing. The unit of recovery begins with begin-UR record and the subsequent log records are queued for writing onto the log media.

### Single-phase commits

- DB2 is the only resource manager that updates and creates log records.
- DB2 communicates with the coordinator CICS or IMS, and the coordinator, CICS or IMS, does no update logging.
- If the coordinator is RRS and DB2 is the only recoverable resource manager that is holding updates in the unit of recovery.

### Two-phase commits

- More than one resource manager communicates with IMS, CICS, or RRS in the same unit of work.
- DB2 updates and write log records.
- There is at least another resource manager that updates and writes log records.

### Read-only commits

- More than one resource manager communicates with CICS or RRS in the same unit of work.
- DB2 access is read-only.
- There is another resource manager that updates.

For a brief description of two-phase commit, see Figure 6-3. This figure illustrates a two-phase commit process where the coordinator is CICS, IMS, RRS, or even DB2 (DDF communication). The coordinator’s role is in the upper line and the participant’s role is in the lower line. The numbers in the following discussion are keyed to those in the figure.

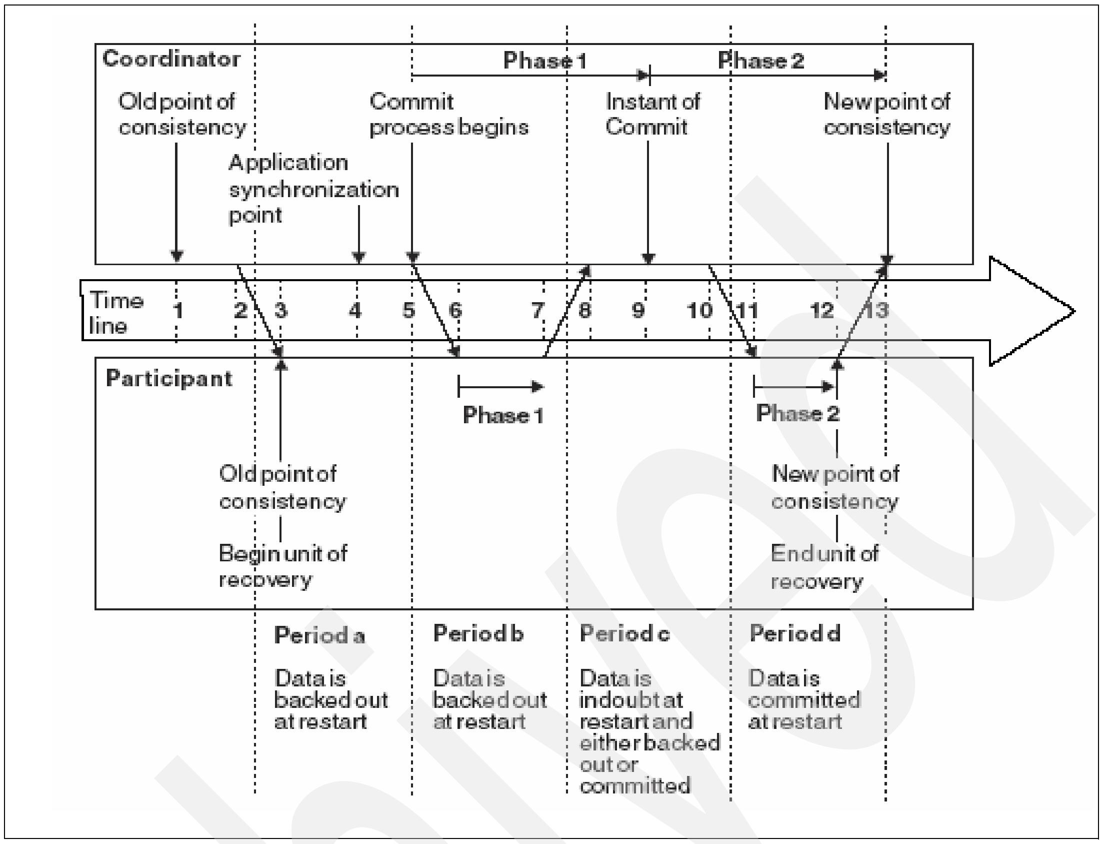


*Figure 6-3 Illustration of two-phase commit*

1. The data in the coordinator is at point of consistency (POC).
2. The application program in the coordinator calls the participants to update data by executing SQL statements.
3. The participants commence the unit of recovery. DB2 writes a begin-UR log record.
4. Processing continues in the coordinator until the application program reaches an application synchronization point.
5. The coordinator starts the commit processing. CICS uses SYNCPOINT. DB2 applications use SQL COMMIT statement or normal termination. Phase 1 of commit processing begins.
6. The coordinator informs the participants that it is prepared to commit. The participants begin phase 1 processing.
7. Participants successfully complete phase 1, update their log records, and notify the coordinator.
8. Coordinator receives the notification from participants.
9. Coordinator successfully completes phase 1 processing and records the instant of commit in its log. Phase 2 processing starts the actual commitment.
10. Coordinator notifies the participants to begin phase 2.
11. Participants log the start of phase 2 in their log.
12. Participants complete phase 2, establish a new point of consistency, and notify coordinator that they are finished with phase 2.

13. Coordinator finishes its phase 2 processing. The data controlled by all participants and coordinators is now available to other applications.

Figure 6-3 also describes what happens to the data when a DB2 failure occurs during a particular period.

## 6.3.3 Consistency across multiple DBMSs

The principles and methods for maintaining consistency across more than two systems are similar to those used to ensure consistency across two systems. The main difference involves a system's role as coordinator or participant when a unit of work spans multiple systems.

The coordinator of a unit of work that involves two or more other DBMSs must ensure that all systems remain consistent. After the first phase of the two-phase commit process, the DB2 coordinator waits for the other participants to indicate that they can commit the unit of work. If all systems are able, the DB2 coordinator sends the commit decision and each system commits the unit of work.

If even one system indicates that it cannot commit, then the DB2 coordinator sends out the decision to roll back the unit of work at all systems. This process ensures that data among multiple DBMSs remains consistent. When DB2 is the participant, it follows the decision of the coordinator, whether the coordinator is another DB2 or another DBMS.

DB2 can play the role of participant and the coordinator in a unit of recovery where more than two subsystems are involved. DB2 is the participant for CICS, IMS, and RRS coordinators. It becomes the coordinator for other DBMSs or DB2 subsystems in the same unit of work. Consider DB2A in Figure 6-4. DB2A is the participant and CICS or IMS is the coordinator, but DB2A becomes the coordinator for the two database servers (AS1 and AS2), DB2B, and its respective DB2 servers (DB2C, DB2D, and DB2E).

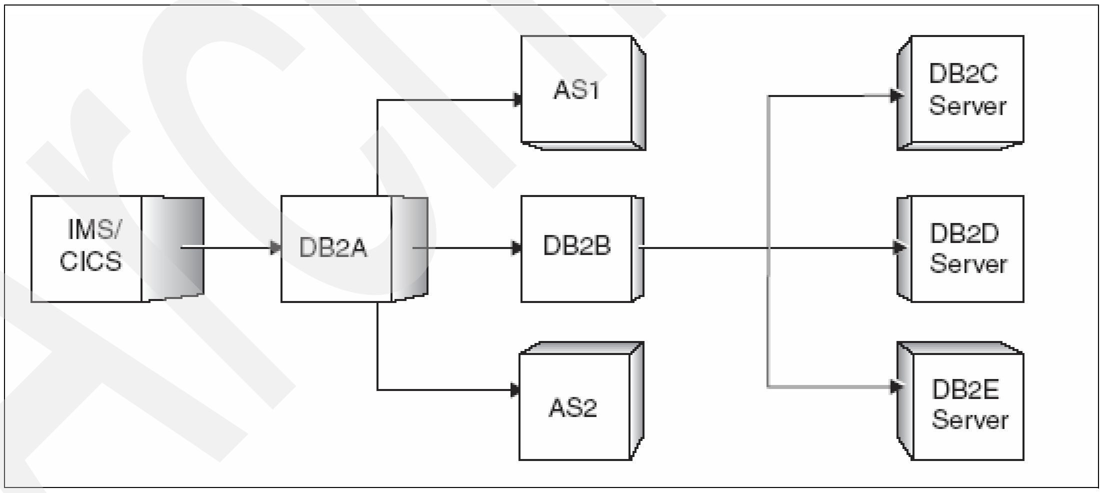


*Figure 6-4 DB2's role as participant and coordinator*

If the connection goes down between DB2A and the coordinating CICS or IMS system, with pending commit, the connection becomes an indoubt thread. However, DB2A's connections to the other systems are still waiting and are not considered indoubt. Wait for automatic recovery to occur to resolve the indoubt thread. When the thread is recovered, the unit of work commits or rolls back and this action is propagated to the other systems involved in the unit of work.

Refer to Chapter 19, “Maintaining consistency across multiple systems”, *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413, for details.

## 6.4 Unit of work

A logical unit of work in an application is a set of SQL statements that makes up a business process. At any point during the logical unit of work, a failure will require the rollback of all previous updates in that logical unit of work in order to keep the database in a status that is consistent with the business rules. At the end of the logical unit of work, the data manipulated by the business process should be in a status that is consistent with the business rules.

In DB2, a unit of recovery is the set of UPDATE, DELETE, and/or INSERT statements that are performed from one point of consistency to another by a single application process. A unit of recovery begins with the first change to the data after the beginning of the job or following the last point of consistency and ends at a commit point.

In order to ensure that the data in a database remains in a status that is consistent with the business rules, the logical unit of work should be no larger than the DB2 unit of recovery. However, in some cases the logical unit of work cannot be completed within a single DB2 unit of recovery. An example would be a process that requires three online transactions to complete a business function that is considered a single logical unit of work. Because each online transaction ends with a commit, the rows that have been added or altered in the first transaction are exposed to other processes without being locked. This may also be the case when multiple batch job steps or even multiple batch jobs are required to complete a logical unit of work. In either of these cases, there is also the dilemma of how to handle an abend in a subsequent unit of recovery after previous units of recovery have already been committed.

One way of solving the problem is by adding application-managed locking information to the table which indicates that the current data is inconsistent. This solution has two disadvantages:

- Every SQL statement that uses the table must verify the status of the application lock even when just retrieving the data.
- It does not solve the problem of the abend situation, because you do not have a before image of the data.

A better solution is to use the techniques described in 6.5.3, “Work-in-progress tables” on page 196 and 6.5.4, “Restricting other applications’ data access” on page 197 to control access to the data until the logical unit of work completes.

For best performance, it may be advisable to group multiple logical units of work into a single DB2 unit of recovery. When this is done, savepoints (which are discussed in 6.4.4, “Savepoints” on page 189) can be used to allow the application to roll back a single logical unit of work.

The diagram in Figure 6-5 shows the relationship between a DB2 unit of recovery and a logical unit of work.

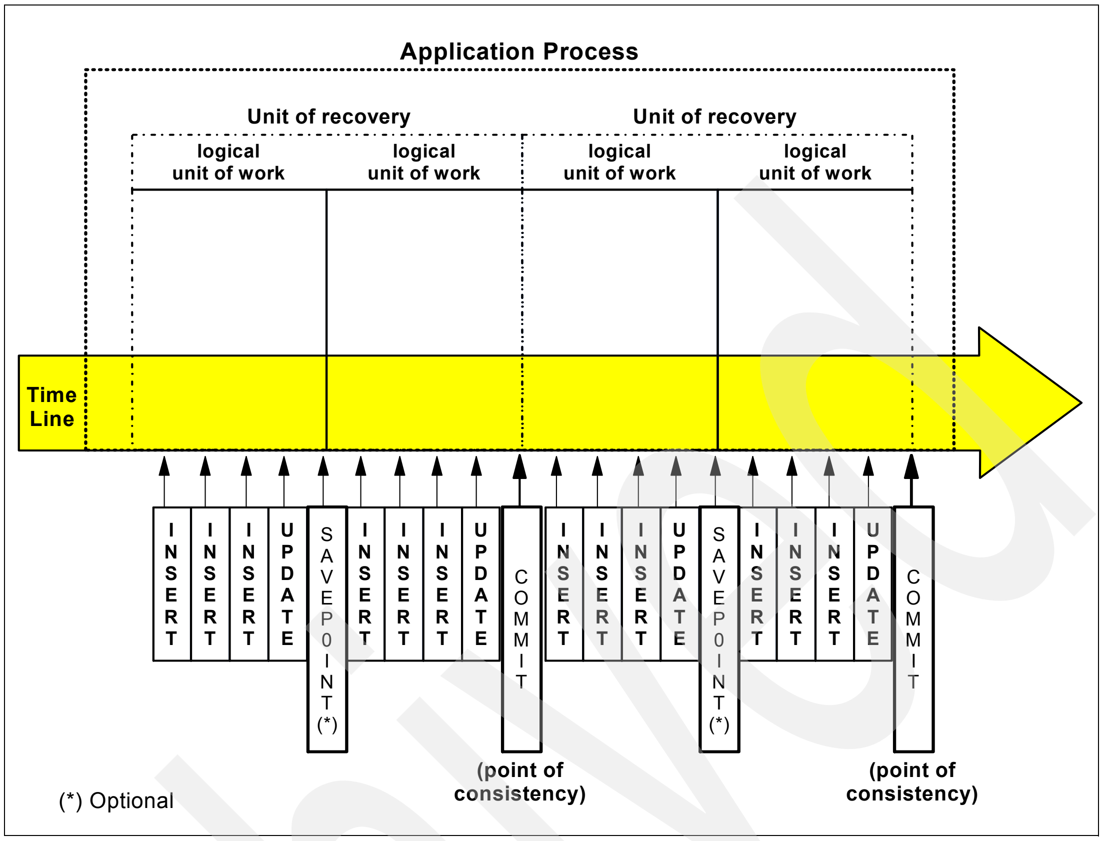


*Figure 6-5 Logical units of work and DB2 units of recovery*

## 6.4.1 Commit

All batch application processes, as well as any other application processes that acquire locks and run for more than a few seconds, should perform commits periodically. We discuss commit frequency in 6.4.2, “Commit frequency” on page 187. The following points summarize the reasons for performing commits:

- Any processes that are updating DB2 are acquiring locks on objects that may prevent other processes from accessing these objects. If these locks are held for an unnecessarily long period of time, concurrency will suffer. Excessive long lock duration leads to deadlock and timeout.
- Any processes that are accessing DB2, including read-only applications, are acquiring claims on objects that may prevent drains from completing. Drains are typically issued by utilities that must be run on the database objects in order to keep them continuously available.
- In a well designed environment, most read-only applications will not acquire any page or row level locks because of lock avoidance. An increase in the number or duration of X locks on pages and rows in the system will reduce the amount of lock avoidance. Acquiring additional locks will add to the elapsed time of read-only transactions and increase the workload in the IRLM address space leading to overall system performance degradation.
- Whenever a batch program abends or a system failure occurs, DB2 backs out data changes that have not been committed. If no commits are taken then the amount of data to be backed out may be large and may take a long time. In addition, upon restart the

program will have to reapply all of the updates. More frequent commits reduce the amount of data that must be rolled back and then reapplied on restart.

  Committing in a batch process requires that the application be designed for restart. See 6.5.7, “Restart using sequential input and output files” on page 201 for detailed information regarding how to design an application for restart.

- When updating a table space that has a LOCKMAX other than 0 specified, it is important to commit before reaching the maximum number of locks on the table space to avoid lock escalation. You also want to commit before reaching the NUMLKUS threshold, which is the maximum number of locks per user.

Issuing a COMMIT statement ends a unit of recovery and commits all database updates that were made in that unit of recovery. DB2 performs the following functions when a commit occurs:

- All page and row locks are released, except those held by an open cursor that was declared WITH HOLD.
- If the RELEASE(COMMIT) bind option was specified, all table, table space, and partition locks will be released, except for those held by an open cursor that was declared WITH HOLD (depending also on RELCURHL).
- All claims are released, except those held by an open cursor that was declared WITH HOLD.
- If the RELEASE(COMMIT) bind option was specified, the statistics used by the index look-aside and sequential detection processes are discarded.
- If the RELEASE(COMMIT) bind option was specified, any IPROCs or UPROCs that had been created by the thread are discarded. For information about IPROCs and UPROCs, see Appendix A of *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134.
- A write is forced out of the log buffer to both active logs.

  The length of time required for the immediate write of the log buffers depends upon the amount of data that has been updated since the last write of the log buffer. Spreading updates throughout a unit of recovery increases the chances that your log data has been already written before you commit.

  **Note:** While saving all update, insert, and delete activity until right before you commit can reduce lock duration. It can also increase the amount of log data that must be written at commit and therefore may cause the commit to take longer.

- In data sharing, all changed pages are forced to the global buffer pool (GBP) at commit.

For more information on the effect of the use of cursors defined WITH HOLD on locking, see 6.4.3, “Cursors WITH HOLD” on page 187.

The major cost component of a commit is two synchronous I/O operations for the write of the log buffer to both copies of the active log. In addition, commit may destroy any IPROCs or UPROCs that were built to improve application performance within the unit of recovery and throw away the information used for index look-aside and sequential detection.

DB2 commits updates when a COMMIT is issued either explicitly or implicitly as a result of a termination of a unit of work. The type of DB2 attachment influences when a unit of work is terminated. See Table 6-1.

*Table 6-1 Termination of a unit of recovery*

| Attachment | Termination of a unit of recovery |
|---|---|
| CAF | The program issues a COMMIT/ROLLBACK SQL statement.<br>The program terminates. |
| RRSAF | The program issues a SRRCMIT/SRRBACK statement or COMMIT or ROLLBACK.<br>The program terminates. |
| IMS | The program issues a CKPT, ROLB, or SYNC call.<br>The program issues a GU for the I/O PCB for single-mode transactions.<br>The program terminates. |
| CICS | The program issues a CICS SYNCPOINT command explicitly.<br>The program terminates signaled by CICS RETURN.<br>The program issues a DL/I checkpoint or TERM call command. |

When designing an application that is going to issue COMMIT statements, be sure to place the COMMIT in the program that controls the logical unit of work. Issuing a COMMIT statement in a subprogram may lead to inadvertently committing in the middle of a logical unit of work. If data access modules are used, the COMMIT statement can be placed in a module that only performs the COMMIT statement, and it should be called only by the program that controls the scope of the logical unit of work.

## 6.4.2 Commit frequency

The commit frequency of an application is different depending upon the purpose of the commits. We do not advise setting the commit frequency of all applications to a single value, because the commit frequency that satisfies the requirements of the most highly accessed and critical DB2 objects is probably too frequent for the majority of other processes.

The primary reason for read-only programs to commit is to release the claims that may prevent utilities from draining. The maximum time that a utility will wait for a claim is calculated by multiplying the IRLMRWT DSNZPARM value times the UTIMOUT DSNZPARM value. An appropriate commit frequency for these processes would be half of that time.

For update processes that need to allow concurrent read activity at the row or page level, a general commit frequency would be 25% of the IRLMRWT DSNZPARM value. However, for update processes that access critical, highly accessed rows or pages, a lower commit frequency might be required.

**Recommendation:** When determining your application's commit frequency, take into consideration the IRLMRWT timeout value and the ability of concurrent processes to wait.

A general rule is to start by committing no more than three times a second, but at least every 3-5 seconds. Then commit less often if there is no contention, more if contention is high.

For information about how to design your application to have variable commit frequencies, see 6.5.6, "Error handling control table" on page 199.

## 6.4.3 Cursors WITH HOLD

The WITH HOLD clause of a DECLARE CURSOR statement allows the cursor to remain open across commit boundaries. From an application's point of view, this option simplifies the use of commits since cursor repositioning is not necessary. However, as mentioned, using WITH HOLD *may* prevent locks and claims from being released at commit.

Whether or not the page or row locks of a cursor defined WITH HOLD are held across a commit is controlled by the RELCURHL DSNZPARM. If a value of "YES" is specified, then the lock on the currently positioned row is released at commit. If a value of "NO" is specified, then the lock on the currently positioned row is held across commit. However, if the lock held on the currently positioned row is an X or U lock, the lock is demoted to an S lock.

While either value of RELCURHL may be acceptable to achieve application concurrency and batch restartability, neither option will affect the releasing of claims held by the cursor. However, whether or not the cursor materializes will affect the releasing of claims held by the cursor.

### The effect of materialization on cursors WITH HOLD

If the cursor does not materialize, the claims on the base DB2 objects held by the cursor WITH HOLD will be held until the first commit after the cursor is closed.

If the cursor does materialize, the claims on the base DB2 objects held by the cursor WITH HOLD will be released by the first commit following materialization (OPEN CURSOR). This is because the data from the base DB2 objects is read into a work file when the OPEN CURSOR is executed. The cursor will continue to hold claims on the work file, but all locks and claims on the base objects are released. Since we never run utilities on the work files, these claims do not threaten our concurrency and availability requirements.

#### *To hold or not to hold*

The following criteria should be used to decide when it is appropriate to code WITH HOLD on a cursor:

- If you are certain that the cursor will always materialize, you should leave the WITH HOLD clause on the cursor.

  By definition, the materialized cursor has already read all of the qualifying rows and placed them in a work table. Removing the WITH HOLD clause and reopening the cursor after each commit would result in the unnecessary overhead of repeatedly materializing all or part of the results set.

  If you want to guarantee materialization, you might consider the use of a static scrollable cursor. However, in most cases it is not possible to materialize very large result sets because of work file storage limitations.

- If the cursor does not materialize and is easy and inexpensive to reposition, you should consider removing the WITH HOLD from the cursor and reopening it after each commit.

- If the cursor does not materialize, but the repositioning logic is expensive, consider the following techniques to improve the performance of the repositioning so that the WITH HOLD clause can be removed.

  The typical cause of poor performance in the repositioning of a non-materialized cursor is a high number of columns used in the ORDER BY to uniquely identify each row and a low number of matching columns (usually one matching column) on the index.

  The first technique to improve repositioning performance is to create a separate cursor for each subset of rows that can be returned. The number of cursors required will be equal to the number of columns used in the ORDER BY to uniquely identify a row in the results set.

  Another option to improve repositioning performance is to commit only after a change in the value of the high order column in the index. This will improve the performance of the repositioning because the values in the non-high order repositioning columns should be blank, and the first row qualified through the index should be the first row processed on restart.

We only recommend this technique if the maximum number of qualifying rows for each value of the high order column can be processed within a reasonable commit scope.

- If the cost of the cursor repositioning cannot be reduced to an acceptable level the WITH HOLD should be left on the cursor to maintain position.

  Be aware of the effect of the WITH HOLD clause on lock and claim retention and schedule the process at a time when it will not be in contention with utilities that require drains or other applications.

- In a distributed environment, using a WITH HOLD cursor will prevent DB2 from separating the connection from the DBAT at COMMIT time when using CMTSTAT=INACTIVE. This will prevent efficient thread pooling in a distributed environment.

## 6.4.4 Savepoints

A *savepoint* represents the status of data at some point in time during a unit of work. Savepoints are set by the application through the use of the SAVEPOINT statement. If subsequent logic dictates that only a portion of the unit of recovery should be rolled back, the application can perform a rollback to the savepoint. Figure 6-6 illustrates the concept of rolling back to a savepoint. In this example, all DB2 changes made between time T2 and time T4 would be rolled back.

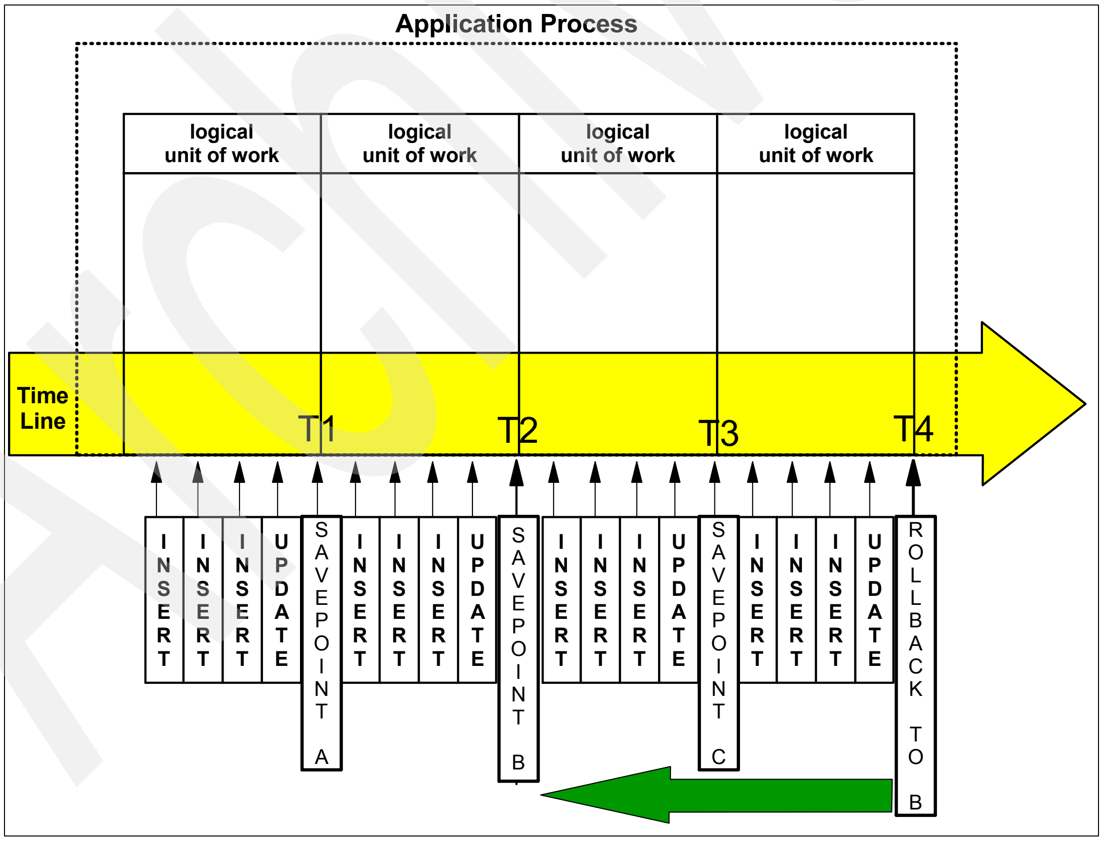


*Figure 6-6 Rollback to a savepoint*

Savepoints provide applications with a tool to achieve a more granular level of data recoverability without the expense of frequent commits. Issuing a SAVEPOINT command is much less expensive than issuing a COMMIT because there is no forced write of the log buffer.

The ROLLBACK TO SAVEPOINT command performs a partial rollback of the unit of recovery. If a savepoint name is not specified, rollback is to the last active savepoint. If a rollback to a named savepoint occurs, all savepoints set after the named savepoint are released. The savepoint to which rollback is performed is not released.

A rollback to a specific savepoint backs out all DB2 changes that were made after the savepoint was set. Changes that were made to created temporary tables are not logged and are not backed out, but changes to declared temporary tables are logged and rolled back. A rollback to savepoint has no effect on the opening or closing of cursors, cursor positioning, the acquisition or release of any locks or claims, and the caching of SQL statements.

**Restriction:** A rollback to a SAVEPOINT only backs out changes made to DB2 objects. It has no effect on changes made to data in any other databases or file management systems, such as VSAM, MQ queues, or flat files.

Use SAVEPOINT rather than COMMIT to create a point to which your application can roll back for logic purposes. However, be aware that neither SAVEPOINT nor ROLLBACK TO SAVEPOINT end a unit of recovery. All locks and claims that are acquired during the unit of recovery are held until commit.

## 6.4.5 More on read only COMMIT

Many application programmers and DBAs have often wondered, “Why do I need to COMMIT?” for a read only application. We recommend you COMMIT (see “Commit frequency” on page 187) even for a “dirty read” with ISOLATION UR. Isolation Uncommitted Reads (UR) do take a MASS DELETE lock on a table if a mass delete is involved. An IX lock is taken if work file database (ORDER BY, GROUP BY) is involved. The most “destructive” of them all is the CLAIM on the object. We review a few cases where read only COMMIT can improve concurrency.

### Utilities

The CLAIM on the tablespace or partition is only released when the thread terminates or when a COMMIT is issued. A parallel running utility, such as REORG, requires DRAIN locks on the object for its last log apply and switch phases. The REORG utility will wait for the drain of all claims on the object until it succeeds, controlled by DRAIN WAIT and RETRY options of REORG, and will abend if the utility timeout limits are reached. The CLAIM and DRAIN requirements of the REORG utility can cause other jobs to fail with a “`resource unavailable`” error.

Consider the following scenario:

- Job 1 running program A with ISOLATION UR enters the job queue and holds claim on an object.
- Job 2 running REORG utility enters the job queue after job 1 and issues a drain on the object and waits.
- Job 3 running program C enters the job queue and wants to access data from the same object. Since there is a drain on the object by the REORG utility, it has to wait.
- Eventually program C in job 3 will fail with TIMEOUT -904 SQLCODE - `resource unavailable`.

The FASTSWITCH YES, DRAIN WAIT, and RETRY options of REORG utility can be used to prevent the REORG from timing out. Job 3 can time out if the DRAIN Wait is shorter than the DB2 IRLM timeout value. Nevertheless, using COMMIT more frequently at logical unit of work in read-only programs will improve concurrency and availability.

### Auto REBIND

A package can be invalid if an index used by the package is dropped and recreated. The package is automatically rebound on first reference if the DSNZPARM parameter ABIND is set to YES. Consider the following simple case:

Program A uses package A and calls program B which uses package B. Package B is invalid when an index was dropped but the usually capable DBA has failed to perform a manual REBIND. Programs A and B are read-only programs.

Now program A executes and calls program B. AUTO REBIND binds package B. The execution continues in program B but fails with -911. Although program B did not perform any updates to application data, the AUTO REBIND updates the catalog. Due to the -911 error a rollback was done and the package B is reset to invalid.

One possible solution to avoid an AUTO REBIND on subsequent executions of program B is to issue a COMMIT as the first executable statement in program B. This will commit all the catalog updates, including those of the prior program A, and set the VALID and OPERATIVE values of SYSIBM.SYSPACKAGE to “Y”.

In a complex environment, we recommend you do manual, planned REBINDs.

### Temporary table

The created temporary table (CTT) is created in the work database DSNDB07. The work database DSNDB07 is also used to sort work files, sort out files, and, when necessary, view materialization. The space in DSNDB07 is almost static and the work files can extend to get additional disk space if they are defined with non-zero secondary extent by the system programmer. The declared temporary table is defined in the TEMP database.

CTTs are deallocated according to the value of the BIND RELEASE parameter. A value of COMMIT releases the CTT and the space in DSNDB07 when the program COMMITs. CTTs are released by an explicit DROP TABLE statement.

### ISOLATION RR and RS

Plans or packages that are bound with ISOLATION RR or RS obtain S lock on the page or row, depending on the LOCKSIZE, release these locks on COMMIT or at the termination of the thread. Resources taken to manage these locks in the Internal Resource Lock Manager (IRLM) V2 include 540 bytes per lock (254 bytes per lock with IRLM V1) and are also released at COMMIT point. A long running read only application without a COMMIT can exhaust the IRLM resource and therefore can cause other parallel applications to fail. Alternatively, a lock escalation to the next level can lock the whole table, tablespace, or partition and therefore denying access to other applications requiring the same object by failing with -911.

## 6.5 Data integrity

As numbers of concurrent users and parallel transactions increase over time, it is even more important to ensure data integrity even when different applications and transactions manipulate the same data. In the following sections, we describe how to make sure your data does not enter an inconsistent status by concurrently running applications, regardless of accessing data online or batch.

To guarantee data consistency, you should consider the following in your application design:

- Protect the same data being updated by concurrently running processes (see 6.5.1, “Concurrent update” on page 192).
- Roll back the data to a point where the data is consistent when an abnormal termination occurs.
- COMMIT even in read-only applications to avoid rollback of parallel running applications or utilities due to timeout on lock waits and DRAIN failures in utilities.

## 6.5.1 Concurrent update

When running applications concurrently, it is even more important to make sure those applications follow basic concurrency rules when accessing the same data. In the worst cases, updates can get lost by applications by violating integrity rules across multiple columns or rows.

In Figure 6-7, we describe a scenario of two online updating processes running concurrently. Both applications have read CUSTNO 18 at nearly the same time, so the represented data is consistent and still the same.

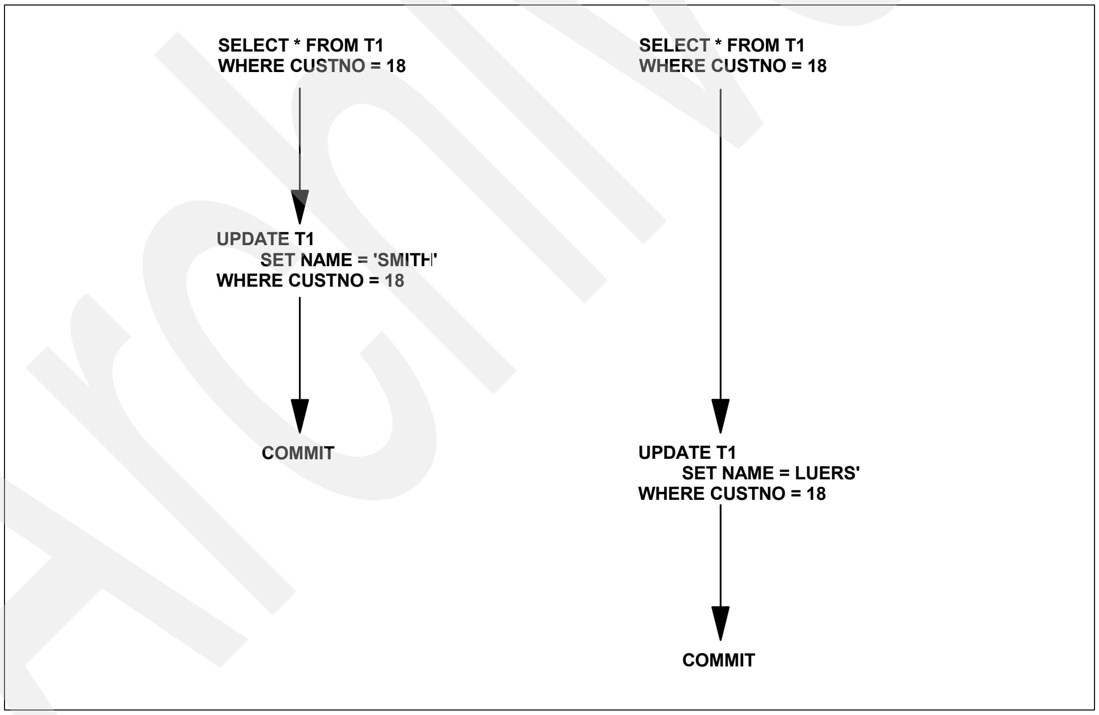


*Figure 6-7 Concurrently updating processes*

Assume that the UPDATE statements in the applications are coded as shown, the data in DB2 depends on the user who first saves his changes, to be more specific on the elapsed time between both select and update statements. The second update will be the update you can find in the database, since the first update is only visible between both updates.

In the following sections, we describe how you can prepare your applications for concurrency and easily prevent these problems.

## 6.5.2 Last update column

A common approach to dealing with concurrent updates for certain rows is to add a timestamp column to a table which contains the timestamp when a certain row has been last updated. The idea is to reference the previously read value in your UPDATE statement.

See Example 6-2 for the DDL we assume in Figure 6-8, containing a unique index on CUSTNO.

*Example 6-2 DDL for table T1*

```
CREATE TABLE QUALIFIER.T1
( CUSTNO      DECIMAL(10,0) NOT NULL
, NAME        CHAR(32)      NOT NULL WITH DEFAULT
, COLa        CHAR(10)      NOT NULL WITH DEFAULT
, ...    ...            ...
, TS_LAST_CH TIMESTAMP NOT NULL )                  -- Timestamp of last update
 IN DBNAME.TSNAME;
```

For a visual explanation of application programming techniques for ensuring concurrency, see Figure 6-8.

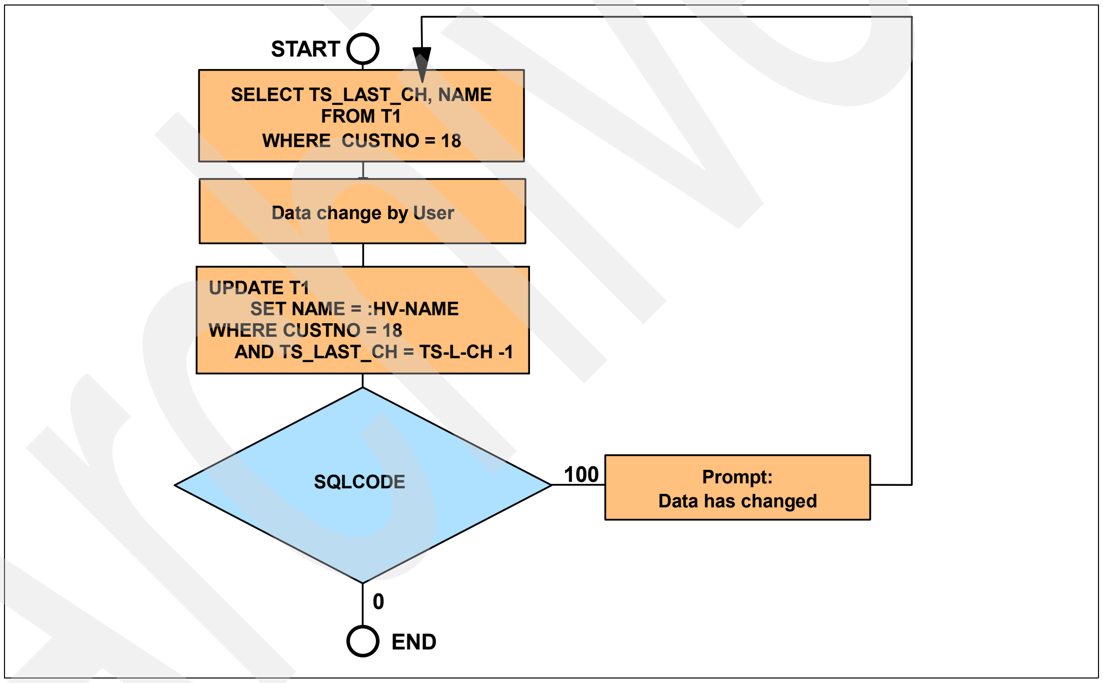


*Figure 6-8 Coding for online concurrency*

Right at the beginning of the business case, the application reads data for CUSTNO 18 and displays it on the screen while the timestamp when the data was read is stored in host-variable TS-LAST-CH-1. After an unspecified amount of time, the user changes the data and sends it back to the application. To ensure no concurrent process has updated the data, include the previously retrieved value for TS-LAST-CH-1 in the WHERE clause of your UPDATE statement.

To provide the previously read timestamp to the updating transaction, you can consider two techniques:

- Timestamp becomes an additional output parameter for read-transactions and an input-parameter for updating transactions.

- Timestamp is kept in temporary storage among transactions.
  - Scratch pad area for IMS transactions
  - Transient data queue for CICS transactions

In case you use a change logging table and have no plans to keep the timestamp of most recent data changes in the same table as your operational data, you can consider using change logging tables and perform the check if a change has occurred on the change logging table.

If available, storing values as user-identification, terminal-number, table name, key values of affected rows, and type of DML (insert, update, or delete) can help you in error situations because they can provide information about who has changed a certain value in the database.

However, if you implement change logging tables, keep in mind that additional tables in your transaction increase elapsed time, the amount of I/Os, and CPU consumption for each executed transaction.

### Pessimistic locking

In case of a high probability of previously read rows being updated by concurrently running processes, you can flag the previously read rows as "update pending" using application logic. Concurrent processes are not allowed to update these rows. Be aware that using this technique enormously decreases the availability of your data.

Normally, a record lock is taken on a read and this lock is only released when the application code has finished with the record and releases the lock. This form of locking is known as *pessimistic locking* because your application assumes that previously accessed data will change during processing time.

### Optimistic locking

Applications using optimistic locking expect successful updates and no concurrent data updates during processing time. The approach of *optimistic locking* compares previously read column values to actual column values in the WHERE clause of your UPDATE statement. Assume you are going to update COLb of table T1 WHERE COLa = 10 and the old value of COLb is stored in host-variable HV-COLb-OLD. To ensure that no concurrent application has already updated the value of COLb, you can code the update as shown in Example 6-3.

*Example 6-3 Update using optimistic locking*

```
UPDATE T1
   SET COLb = :HV-COLb-NEW
      ,COLc = :HV-COLc-NEW
      ,COLd = :HV-COLd-NEW
 WHERE COLa = 10
   AND COLb = :HV-COLb-OLD
   AND COLc = :HV-COLc-OLD
   AND COLd = :HV-COLd-OLD;
```

Unless you do not list all old column values in your WHERE clause, the context in which an update is applied might have changed. It depends on your business process if concurrent updates on different columns of one single row are allowed or not. Assuming COLa is unique, Example 6-4 shows two update statements, both of them affecting the same row, manipulating different columns. If values for COLb and COLc depend on each other, the old values for those columns have to be included in the WHERE clause, too.

*Example 6-4 Concurrent update on independent columns*

```
Transaction 1:

UPDATE T1
   SET COLb = 20
 WHERE COLa = 10
   AND COLb = 15;

Transaction 2:

UPDATE T1
   SET COLc = 30
 WHERE COLa = 10
   AND COLc = 25;
```

If all previous column values are checked in a WHERE clause as shown in Example 6-5, the first update statement will succeed, while the second statement will receive SQLCODE 100 since COLb or COLc has changed.

*Example 6-5 Concurrent update on dependent columns*

```
Transaction 1:

UPDATE T1
   SET COLb = 20
 WHERE COLa = 10
   AND COLb = 15
   AND COLc = 25;

Transaction 2:

UPDATE T1
   SET COLc = 30
 WHERE COLa = 10
   AND COLb = 15
   AND COLc = 25;
```

You can consider the technique mentioned above using a TS-LAST-CH instead of providing old column values as a special case of optimistic locking.

**Note:** DB2 V9 for z/OS introduces enhancements to optimistic locking. It provides an easier and more efficient approach for detecting a change of a row. An application does not need to know all the old values which are marked for update.

## USE AND KEEP ... LOCKS options of the WITH clause

If you use the WITH RR or WITH RS clause, you can use the USE AND KEEP EXCLUSIVE LOCKS, USE AND KEEP UPDATE LOCKS, and USE AND KEEP SHARE LOCKS options in SELECT and SELECT INTO statements.

They are specified as shown in the following example:

```
SELECT ... WITH RS USE KEEP UPDATE LOCKS;
```

By using one of these options, you tell DB2 to acquire and hold a specific mode of lock on all the qualified pages or rows. The table shows which mode of lock is held on rows or pages when you specify the SELECT using the WITH RS or WITH RR isolation clause.

Table 6-2 shows which mode of lock is held on rows or pages when you specify the SELECT using the WITH RS or WITH RR isolation clause Option Value Lock Mode.

*Table 6-2 Option Value Lock Mode*

| Statement | Lock mode | Lock type |
|---|---|---|
| USE AND KEEP | EXCLUSIVE LOCKS | **X** |
| USE AND KEEP | UPDATE LOCKS | U |
| USE AND KEEP | SHARE LOCKS | S |

## 6.5.3 Work-in-progress tables

Work-in-progress tables are considered if you have to package more than a single unit of work from a DB2 point of view together in one business unit of work by "tying" different DB2 units of work together. If you apply data changes after every single unit of work to your operational data other applications can read data that might be inconsistent. An additional issue is the case of an abend. How do you undo the changes made by the first two transactions if the third fails? See Figure 6-9 for a visual explanation of work-in-progress tables.

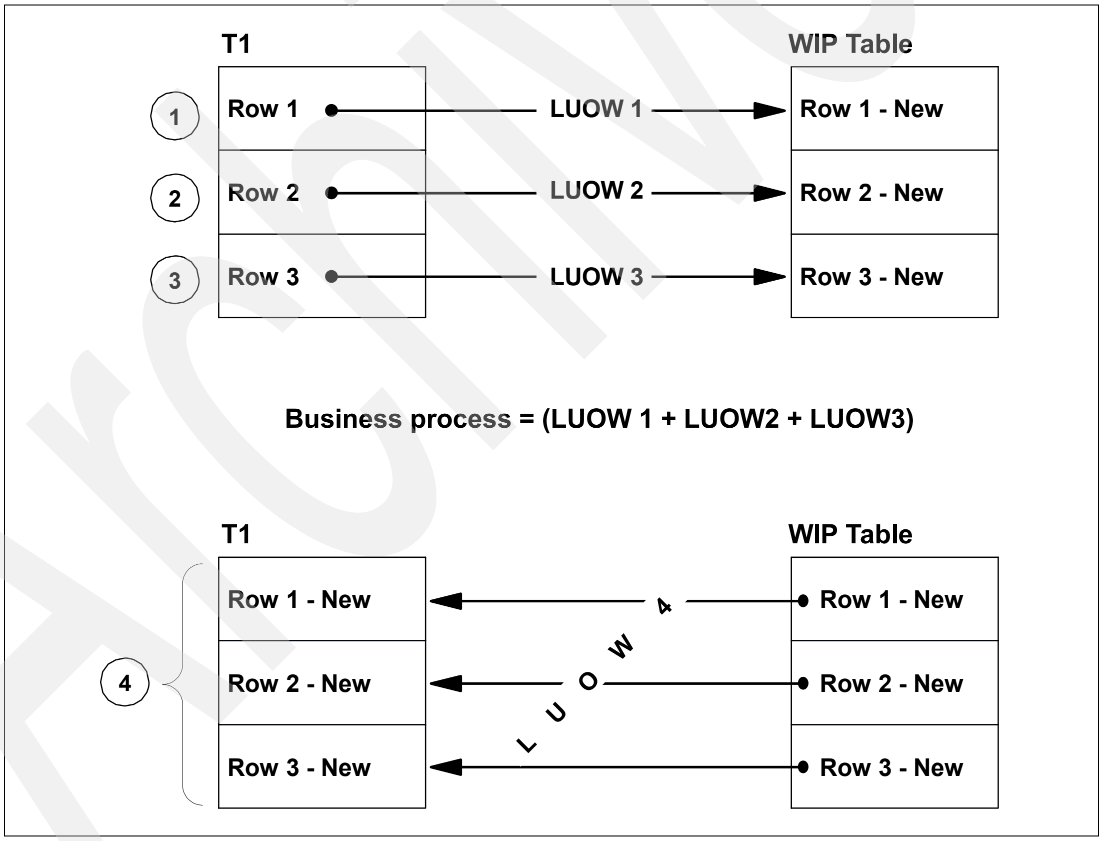


*Figure 6-9 Work-in-progress table*

Upon invocation of the first unit of work, the information that is subject to change is copied from the actual table into a work table, and a logical application lock is taken in the actual table to signal this event to other transactions that also want to update the same information. The other transactions that are part of the same application logical unit of work interact with the information in the work-in-progress table. Once the application logical unit of work completes, the updated information from the work-in-progress table can be stored in the actual table. This process is within the scope of a DB2 logical unit of work, so DB2 locking mechanisms can be used to guarantee consistency.

A business process requiring multiple transactions can be implemented by tying transactions together using artificial keys: A logical transaction identifier can be passed from the front end (which is aware of a logical unit of work containing *n* physical units of work) to application processes in the back end.

The work-in-progress table solves the consistency issues of the application but might create another challenge if multiple transactions running in parallel use the same table. The work-in-progress table becomes a hot spot.

There are several ways we can deal with the hot spot. Using UR isolation can be a solution for those statements that retrieve data. UR isolation does not, however, provide a solution for UPDATE and DELETE. The use of row locking is appropriate unless this table is used in a data sharing environment. In a data sharing environment, consider using MAXROWS 1.

To remove any contention problem, it is best to add a column that uniquely identifies your application logical unit of work. Adding this column to all WHERE clauses guarantees that you do not really need UR, although there might be a performance benefit, provided that the additional column is a matching predicate in the index.

A work-in-progress table can vary in size dramatically, starting with zero rows, growing during the day, and containing again zero rows at the end of the day (assuming your business cases are finished during online hours). This makes it even harder to collect proper statistics to ensure appropriate access paths. Consider using the “VOLATILE” attribute for a work-in-progress table to guarantee index access even if no column statistics are available. Also try to keep a minimum number of indexes (one if possible) on a work-in-progress table.

## 6.5.4 Restricting other applications’ data access

If your application needs to protect data from being accessed by other applications, consider locking tables or sets of tables using logical application locking. In this context, when we talk about logical application locking we assume a control table containing different rows for critical tables as mentioned earlier. In case a table is not allowed to be updated by other processes, the control table contains at least one entry per affected table (or set of tables). Example 6-6 shows table T1 is not allowed to be updated by other applications and table T2 is not allowed to be read by other applications.

*Example 6-6 Logical locking table*

```
TABLE_NAME   JOB_NAME   TIMESTAMP                    LOCK_TYPE
----------   --------   --------------------------   ---------
T1           JOB1       2005-07-12-22.29.41.362282   UPDATE
T2           JOB2       2005-07-13-08.14.20.825462   READ
```

Using this method implicitly forces applications accessing tables contained as rows in a control table to read this table first to access the operational data. If no row can be found, all access is granted. Depending on the lock type, only reads or updates are currently allowed on the specified table.

This technique might only be used for critical tables where retrieval of inconsistent data cannot be tolerated (for example, as a result of application problems) or exclusive access is needed to ensure data consistency during data updates (for example, job step scope unit of work). You can also use logical application locking in case you encounter application problems corrupting your data and you decide not to allow users to update but to only view data any longer until the problem is fixed.

Note that locks taken by DB2 do not survive the end of a thread, and most of them do not survive even at COMMIT.

## 6.5.5 Applications to switch between isolation levels

In the following paragraphs, we describe a technique an application can use to switch between more than one isolation level without affecting the program logic. A common use might be dynamically switching between using isolation levels CS and UR. See Example 6-7 for the required BIND PACKAGE statements for the application.

*Example 6-7 BIND PACKAGE statements*

```
BIND PACKAGE (COLLI001) -
  MEM (PGM1)           -
  QUALIFIER (QUALI001) -
  ACT (REP)            -
  CURRENTDATA (NO)     -
  ISOLATION (CS)       -
  RELEASE (COMMIT)     -
  VALIDATE (BIND)      -
  EXPLAIN (YES)        -

BIND PACKAGE (COLLU001) -
  MEM (PGM1)           -
  QUALIFIER (QUALI001) -
  ACT (REP)            -
  CURRENTDATA (NO)     -
  ISOLATION (UR)       -
  RELEASE (COMMIT)     -
  VALIDATE (BIND)      -
  EXPLAIN (YES)        -
END
```

The only difference between both BIND PACKAGE statements shown is the name of the collection and the isolation level. You can use the special register CURRENT PACKAGESET to switch between both packages, both accessing the same tables. The necessary variable declarations needed for package switching are shown in Example 6-8.

*Example 6-8 Host variables for SET CURRENT PACKAGESET*

```
01 PARM-PACKAGESET                  PIC X(18) VALUE 'COLLx001'.
01 FILLER                           REDEFINES PARM-PACKAGESET.
  05 FILLER                         PIC X(4).
  05 PARM-PACK-ISL                  PIC X(1).
  05 FILLER                         PIC X(3).

01 PACKAGE-OF-CALLER                PIC X(18).
```

A change of special register CURRENT PACKAGESET is necessary to get advantage from switching isolation levels from within the application. It has to be set before your first data access using SQL inside your application. You only need to change the special register if it differs from the CURRENT PACKAGESET which is already used. Only in this case, you have to switch it back at the end of your program. See Example 6-9 for a short pseudo-code description of when to set CURRENT PACKAGESET.

*Example 6-9 Setting CURRENT PACKAGESET*

```
Program entry:

EXEC SQL
  SET :PACKAGE-OF-CALLER = CURRENT PACKAGESET
END-EXEC

For ISOLATION (CS):    MOVE ‘I’ TO PARM-PACK-ISL

For ISOLATION (UR):    MOVE ‘U’ TO PARM-PACK-ISL

IF PARM-PACKAGESET NOT = PACKAGE-OF-CALLER THEN
  EXEC SQL
    SET CURRENT PACKAGESET = :PARM-PACKAGESET
  END-EXEC
END-IF

Program exit:

IF PARM-PACKAGESET NOT = PACKAGE-OF-CALLER THEN
  EXEC SQL
    SET CURRENT PACKAGESET = :PACKAGE-OF-CALLER
  END-EXEC
END-IF
```

The host-variable for assigning the new value for CURRENT PACKAGESET must use the same format as the register itself which is CHAR(18).

**Important:** If you change special register CURRENT PACKAGESET, either make sure all other programs involved in a certain thread support the same collection or reset CURRENT PACKAGESET before exiting your program. Otherwise, DB2 issues SQLCODE -805 at the time when the next SQL is invoked inside your thread.

If you change CURRENT PACKAGESET special register, ensure that PKLIST entries in your BIND PLAN statement support the used packages.

The information regarding which isolation level you have to use can be retrieved from the application that calls your module.

## 6.5.6 Error handling control table

An *error handling control table* (EHCT) can be used for applications to determine their own correct commit frequencies and behavior on exception situations depending on various parameters. In Table 6-3, we provide you with an idea of possible attributes of an EHCT.

*Table 6-3 Error handling parameters*

| Attribute | Content |
|---|---|
| APPLICATION_NAME | Error handling instructions for named application |
| START_TIME | Starting time for described behavior |
| END_TIME | Ending time for described behavior |
| COMMIT_FREQUENCY | Commit every *n* logical units of work or number of rows or elapsed time |

| Attribute | Content |
|---|---|
| SQLCODE | SQLCODE requiring special handling |
| NUMBER_RETRIES | Number of retries for SQLCODE |

Table 6-3 gives you ideas which parameters you can use to determine how to force the correct behavior for your applications.

Example 6-10 gives you an idea of possible descriptions for an application.

*Example 6-10 Error handling table*

```
APPLICATION_NAME  START_TIME  END_TIME  COMMIT#ROWS  COMMIT#SECS  SQLCODE  NUMBER_RETRIES
----------------  ----------  --------  -----------  -----------  -------  --------------
APP1              05:00:00    20:29:59  50           3            NULL     NULL
APP1              05:00:00    20:29:59  NULL         NULL         -904     0
APP1              05:00:00    20:29:59  NULL         NULL         -911     1
APP1              20:30:00    04:59:59  1000         8            NULL     NULL
APP1              20:30:00    04:59:59  NULL         NULL         -904     5
APP1              20:30:00    04:59:59  NULL                      -911     5
```

As shown in the example, application APP1 is forced to behave differently during online and batch times regarding commit frequency and the number of possible retries in certain exception situations caused by unexpected SQLCODEs.

Application APP1 has to COMMIT after manipulating 50 rows or at least every three seconds during online hours, whatever comes first. The number of retries for SQLCODE -904 is zero. The minimum checking for SQLCODE -911 and -913 is to check the reason code. If it is a deadlock, do not retry, since that just elongates the problem. During batch hours, APP1 commits every 1000 rows or 8 seconds. The number of retries for each SQLCODE -904 and -911 is five.

The table should just give you an idea of how to set up an error handling table. Of course, table design can be different. For example, you can also consider using a column for 904#RETRIES instead of using different rows.

Keep in mind that SQLCODE -911 includes a rollback to solve the deadlock situation and issuing the failing statement again is probably not the solution you might want. In this case, the rolled back unit of work from a DB2 point of view has to be completely repeated usually requiring more application logic to handle this situation automatically using an error handling table.

Independent from using an error handling control table, make sure you provide all necessary information you need to solve the problem in SYSOUT. Always use DSNTIAR and consider using GET DIAGNOSTICS for cases where you need the full explanation for the received SQLCODE.

For online environments, avoid abends in online transactions after receiving exceptional SQLCODEs whenever possible.

In some designs, the application itself can heuristically determine the commit frequency based on the activity in the system.

## 6.5.7 Restart using sequential input and output files

An important factor to consider when designing a time-consuming batch application is the time required for the backout and restart operation in case of abnormal termination. The application should be composed of many units of work. COMMIT should be performed frequently with respect to both application logic and performance. Checkpoint interval is important in determining the time for restart. Commit interval should be shorter than checkpoint interval.

If you use WITH HOLD, there is no need to reposition after each COMMIT. However, if a job is restarted, you always need to reposition. In general, you need special application logic to handle cursor repositioning and restart situations including file repositioning to the last commit point. When designing your batch application, you should favor accessing data in clustering order so you can be sure to benefit from sequential prefetch algorithms as well as from proper indexing in the case of repositioning your cursor during restart. For repositioning, save the last key before committing your changes as you do on all needed variables. You can externalize those values in a DB2 table or use external files. See Figure 6-10 for a brief overview looking at restart implementations.

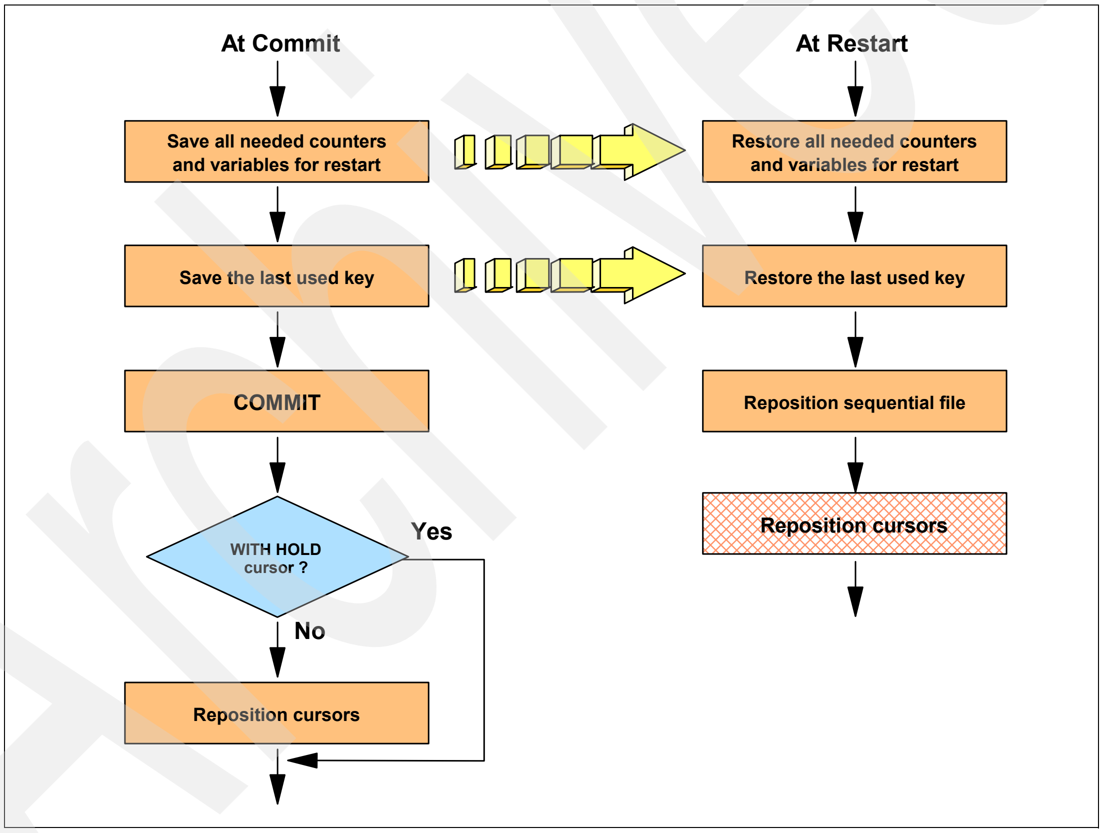

At Commit
At Restart
Save all needed counters and variables for restart
Restore all needed counters and variables for restart
Save the last used key
Restore the last used key
COMMIT
Reposition sequential file
Reposition cursors
WITH HOLD cursor ?
Yes
No
Reposition cursors

*Figure 6-10 Program logic for commit and restart*

In the following sections, we provide checkpoint and restart considerations concerning sequential input/output files.

Different types of sequential files are commonly used as input/output files in DB2 batch programs. The common file types used are:

- QSAM files are the most commonly used sequential files. QSAM manages all aspects of I/O buffering automatically. This is convenient for application development but implies more logic in case of restarts.

- Basic sequential access method (BSAM) files require the program to manage its own input and output buffers. It must pass a buffer address to the READ macro and fill its own output buffer before issuing a WRITE macro. BSAM is usually used for specific requirements and may take advantage of the NOTE and POINT macro statements for repositioning. BSAM is more complex to handle than QSAM.
- GSAM files are part of IMS and therefore require the existence of IMS/DB. They may be used in IMS batch and BMP jobs. Because of the two-phase commit protocol implemented between DB2 and IMS, the commitment of updates is synchronized in both systems. Also, running the job under BMP batch allows the use of the IMS checkpoint and restart facility, in which case GSAM files can be automatically repositioned at restart time. There are special considerations for repositioning GSAM files.
- Recoverable VSAM using RRS (recoverable resource services) manages backouts on behalf of DB2. RRS uses a two-phase commit mechanism to ensure that all updates to all resources are committed.

When dealing with sequential files, techniques must be implemented to:

- Control the buffer management of output files and synchronize it with DB2 commit phases
- Reposition the input/output files when restarting
- Handle potential duplicate output records as a result of restart

Figure 6-11 illustrates the need for repositioning sequential files unaffected by COMMIT and ROLLBACK.

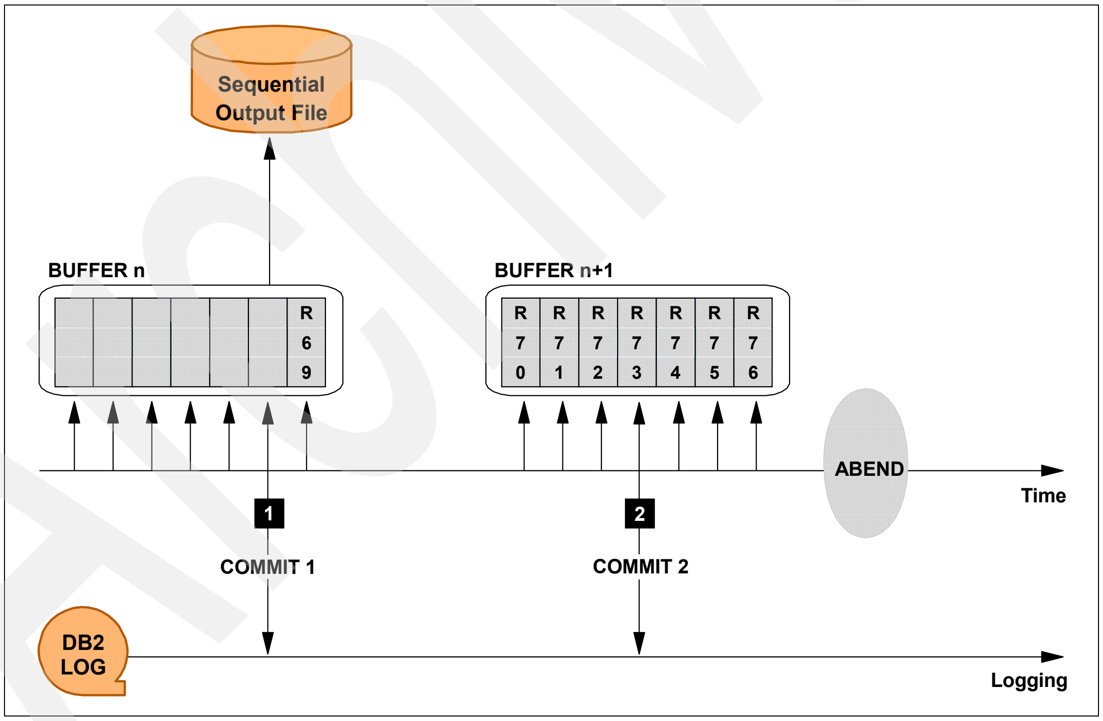


*Figure 6-11 Sequential output files*

The application program issues two COMMIT statements and writes sequential data to QSAM files. All DB2 changes up to these points in time are externalized to the DB2 log. The QSAM buffers are not externalized at the same point in time, depending on your buffer management. For this instance, it is most likely in abend scenarios that your QSAM data does not correspond with your DB2 data. Therefore, QSAM files can contain data which is associated to DB2 rows affected by ROLLBACK. A QSAM file might not contain all data up to the last commit point if an abend has occurred directly after a COMMIT.

Besides closing and reopening the data set (which consumes lots of elapsed time for the unlikely event of an abend, decreasing your performance dramatically) to ensure externalization of your allocated QSAM buffers, you can think of ESTAE (Extended Specify Task Abnormal Exit) routines to clean up your environment (for example, closing files to force buffer externalization) in case of an abnormal termination.

If your application abends with abend code B37 (not enough space available), you can lose your output buffers, regardless of an established ESTAE routine.

You will need to talk to your z/OS systems programmer to implement ESTAE routines.

Instead of repositioning on your output data sets, you can consider using GDG (generation data group) data sets. At each restart, you create a new generation of the GDG. When you are concerned about eliminating duplicate records in sequential data sets as a result of a restart, the elimination of those records can be postponed until the end of all application phases by running an additional DFSORT™ for suppressing records with duplicate control fields, thus involving all created generations. Example 6-11 shows the required DFSORT statement.

*Example 6-11 Eliminating duplicate records using DFSORT*

```
SORT FIELDS=(1,3,CH,A)
SUM FIELDS=NONE
```

## 6.5.8 Restart using DB2 tables for input and output files

Since sequential file handling during restart operations can be challenging, data consistency can be guaranteed if you use the following technique:

1. LOAD your input files into DB2 table A without indexes.
2. Read your input data from DB2 table A using a FOR UPDATE OF cursor.
3. Write your output records to DB2 table B.
4. Delete rows you have already processed from table A using WHERE CURRENT OF CURSOR.

In case of a restart, your cursor can simply reposition on table A since only non-processed rows are in the table.

There will be an additional overhead of doing INSERTS to a DB2 table instead of writing to a sequential file, but the special checkpoint and restart considerations for sequential files will be eliminated. To reduce overhead for round-trips to DB2 for each insert statement, consider using multi-row insert if your commit frequency contains a reasonable number of output records. Often an application does one write to a sequential file after having executed several SQL statements. The overhead of the INSERTs may then be small.

## 6.6 Scrollable cursors

Scrollable cursors were initially introduced in DB2 V7. They provide support for:

- Fetching a cursor result set in backward and forward directions
- Using a relative number to fetch forward or backward
- Using an absolute number to fetch forward or backward

The technique used in DB2 V7 is based on declared temporary tables automatically created by DB2 at runtime. The actual database and table space for the DTT are created by the user, however DB2 will create the DTT. Copying rows to declared temporary tables means that those rows are not sensitive to subsequent inserts which qualify for the result set. DB2 V8 has introduced dynamic scrollable cursors. They operate directly on the base tables and allow you to view inserts of concurrently running processes.

## 6.6.1 Static scrollable cursors

Prior to DB2 V8, you could declare a scrollable cursor using the following keywords:

- INSENSITIVE, causing the result set to be static. The cursor is read only and is not sensitive (insensitive) to updates, deletes, and inserts to the base table.
- SENSITIVE STATIC, causing the result set not to be static during scrolling the result set. The behavior of the cursor depends on the FETCH statement. You can specify on the FETCH statement either:
  - INSENSITIVE, meaning that its own changes are visible to the cursor. They are visible to the application because DB2 updates both the base table and the result table when a positioned update or delete is issued by the application.
  - SENSITIVE, which is the default for cursors declared as SENSITIVE STATIC. Its own changes are visible to the cursor. Furthermore, committed deletes and updates from other applications are also visible. However, as part of a SENSITIVE FETCH, the row is verified against the underlying table to make sure it still exists and qualifies. Inserts are not visible for the cursor.
  - Inserts are never visible when using any form of static scrollable cursors.

All scrollable cursors mentioned above use a declared temporary table which is populated at OPEN CURSOR time. You can refer to those cursors as static scrollable cursors. See Figure 6-12 for a visual explanation of result sets for static scrollable cursors.

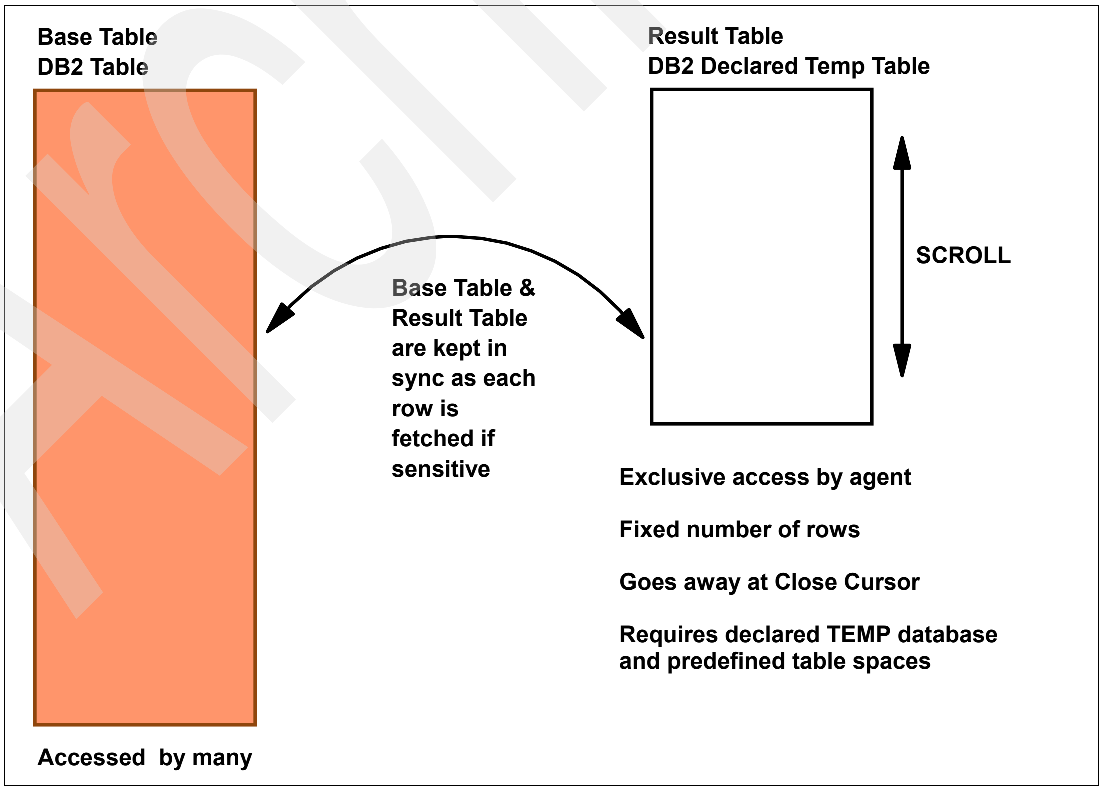

Base Table
DB2 Table
Result Table
DB2 Declared Temp Table
SCROLL
Base Table & Result Table are kept in sync as each row is fetched if sensitive
Exclusive access by agent
Fixed number of rows
Goes away at Close Cursor
Requires declared TEMP database and predefined table spaces
Accessed by many

*Figure 6-12 Temporary result set for static scrollable cursors*

**Note:** Scrollable cursors support large objects (LOBs), but no temporary LOB table space is created. Data consistency is ensured by storing the LOB descriptor in the result table; therefore, the associated row in the LOB base table cannot be deleted.

## 6.6.2 Dynamic scrollable cursors

This functionality was enhanced in DB2 V8, allowing scrollable cursors to view inserts of concurrently running processes, using SENSITIVE DYNAMIC SCROLL keywords.

A dynamic scrollable cursor does not materialize the result table at any time. Instead, it scrolls directly on the base table and is therefore sensitive to all committed INSERTs, UPDATEs, and DELETEs. Dynamic scrollable cursors can use index scan and table space scan access paths. Since DB2 supports backwards index scan, there is no need for creating indexes in an inverted order to support backwards scrolling.

Dynamic scrollable cursors also support multi-row fetch.

Two new keywords were introduced in DB2 V8:

- ASENSITIVE, which is the default

  DB2 determines the sensitivity of the cursor. If the cursor is not read-only, SENSITIVE DYNAMIC is used for maximum sensitivity.

  Using the keyword ASENSITIVE allows DB2 to decide whether a cursor is either INSENSITIVE or SENSITIVE DYNAMIC, bearing in mind that the cursor should always be as sensitive as possible. A cursor meeting the following criteria is considered to be a read-only cursor having an effective sensitivity of INSENSITIVE; otherwise, the effective sensitivity is SENSITIVE DYNAMIC:

  - The first FROM clause identifies or contains:
    - More than one table or view
    - A catalog table with no updatable columns
    - A read-only view
    - A nested table expression
    - A table function
    - A system-maintained materialized query table (MQT)
  - The first SELECT clause specifies the keyword DISTINCT, contains an aggregate function, or uses both.
  - The SELECT statement of the cursor contains an INSERT statement.
  - The outer subselect contains a GROUP BY clause, a HAVING clause, or both clauses.
  - It contains a subquery where the base object of the outer subselect and of the outer subquery is the same table.
  - Any of the following operators or clauses are specified:
    - A UNION or UNION ALL operator
    - An ORDER BY clause (except when the cursor is declared as SENSITIVE STATIC scrollable)
    - A FOR READ ONLY clause
  - It is executed with isolation level UR and a FOR UPDATE is not specified.

  The cursor is implicitly going to be INSENSITIVE if the SELECT does not allow it to be sensitive (for example, if UNION, UNION ALL, FOR FETCH ONLY, or FOR READ ONLY is used).

- SENSITIVE DYNAMIC

  Specifies that the size of the result table is not fixed at OPEN cursor time. This is different for static scrollable cursors because inserts are not visible, hence the maximum amount of rows are known at Open cursor time. A FETCH statement is always executed against the base table since no temporary result set is created at OPEN time. Therefore, the cursor has complete visibility of changes:

  – All committed inserts, updates, and deletes by other application processes
  – All positioned updates and deletes within cursor
  – All inserts, updates, and deletes by the same application process, but outside of the cursor

The declaration of a SENSITIVE DYNAMIC scrollable cursor is shown in Example 6-12.

*Example 6-12 Declaring a SENSITIVE DYNAMIC scrollable cursor*

```
DELCARE C1 SENSITIVE DYNAMIC SCROLL CURSOR FOR
  SELECT COLa, COLb, COLc
    FROM T1
   WHERE COLa > 10;
```

For maximum concurrency, we recommend using ISOLATION(CS). Note that the isolation level is promoted to CS even if you use BIND option ISOLATION(UR) and the SELECT statement contains the FOR UPDATE OF clause.

Benefits of dynamic scrollable cursors are:

- Ability to view inserted data
- No temp table and work file
- Enhances usability of SQL
- Enhances portability
- Conforms to SQL standards

## 6.6.3 FETCHing options for scrollable cursors

In general, the sensitivity to changes can be specified in two ways: on DECLARE CURSOR statements and on FETCH statements.

Depending on your cursor definition, there can be several implications for your FETCH syntax on SENSITIVE DYNAMIC scrollable cursor. Table 6-4 lists the dependencies on DECLARE and FETCH statements for scrollable cursors.

*Table 6-4 Dependencies on DECLARE and FETCH statements for scrollable cursors*

| Specification on DECLARE | Specification on FETCH | Comment | Visibility of changes |
|---|---|---|---|
| INSENSITIVE | INSENSITIVE | Default INSENSITIVE | None |
| INSENSITIVE | SENSITIVE | Not allowed | |
| SENSITIVE | INSENSITIVE | Allowed | See own updates and deletes |
| SENSITIVE | SENSITIVE | Default SENSITIVE | See own changes and others' committed updates and deletes |

| Specification on DECLARE | Specification on FETCH | Comment | Visibility of changes |
|---|---|---|---|
| SENSITIVE DYNAMIC | INSENSITIVE | Not allowed, even if ASENSITIVE and DB2 decides for SENSITIVE DYNAMIC | |
| SENSITIVE DYNAMIC | SENSITIVE | | See own and others' committed changes, including inserts |

You can fetch data from your current positioning using the FETCH orientation keywords as shown find in Table 6-5:

*Table 6-5 FETCH orientation keywords for scrollable cursors*

| Keyword in FETCH statement | Cursor position when FETCH is executed |
|---|---|
| BEFORE | Before the first row |
| FIRST or ABSOLUTE +1 | On the first row |
| LAST or ABSOLUTE -1 | On the last row |
| AFTER | After the last row |
| ABSOLUTE | On an absolute row number, from before the first row forward or from after the last row backward |
| RELATIVE | On the row that is forward or backward a relative number of rows from the current row |
| CURRENT | On the current row |
| PRIOR or RELATIVE -1 | On the previous row |
| NEXT | On the next row (default) |

For further details about fetch orientation keywords, refer to *DB2 for z/OS Version 8 SQL Reference,* SC18-7426.

Table 6-6 compares different kinds of scrollable cursors looking at change-visibility and materialization of the result table.

*Table 6-6 Comparing scrollable cursors*

| Cursor type | Result table | Visibility of own changes | Visibility of others' changes | Updatability |
|---|---|---|---|---|
| Non-scrollable (SQL contains a join or sort, etc.) | Fixed, work file | No | No | No |
| Non-scrollable | No work file, base table access | Yes | Yes | Yes |
| INSENSITIVE SCROLL | Fixed, declared temp table | No | No | No |
| SENSITIVE STATIC SCROLL | Fixed, declared temp table | Yes (INSERTs not allowed) | Yes (no INSERTs) | Yes |

| Cursor type | Result table | Visibility of own changes | Visibility of others' changes | Updatability |
|---|---|---|---|---|
| SENSITIVE DYNAMIC SCROLL | No declared temp table, base table access | Yes | Yes | Yes |

## 6.6.4 Updating using scrollable cursors

In general, positioned updates and deletes using scrollable cursors are possible and are based on the temporary result table for static scrollable cursors and on the underlying base table for dynamic scrollable cursors. A positioned UPDATE or DELETE is always allowed if the cursor is not read-only and the page or row lock was acquired successfully.

But how does DB2 perform updates and deletes? Let's look at DB2's processing.

For packages and plans containing updatable static scrollable cursors, ISOLATION (CS) lets DB2 use optimistic locking. DB2 can use optimistic concurrency control to shorten the amount of time that locks are held in the following situations:

- Between consecutive FETCH operations
- Between FETCH operations and subsequent positioned UPDATE or DELETE statements

Optimistic locking consists of the following steps:

- When the application opens the static scrollable cursor, DB2 fetches the qualifying rows into the DTT. When doing so, DB2 will try to use lock avoidance to minimize the amount of locking required.
- When the application requests a positioned UPDATE or DELETE operation on a row, DB2 finds and locks the corresponding base table row and reevaluates the predicate to verify that the row still satisfies the search condition.
- For columns that are in the result table, compares current values in the row to the values of the row when the result table was built. DB2 performs the positioned update or delete operation only if the values match.
- An UPDATE or DELETE is disallowed if either the row fails to qualify the WHERE clause, or the values do not match.

If the row passes the above two conditions, the following actions are performed for UPDATE statements:

- Update the base table.
- Re-FETCH from the base table to reevaluate the predicate.
- Update the result table.
  - If the search condition fails, mark row as an update hole.
  - If the search condition satisfies, update row with latest values.

For DELETE statements, DB2 performs the following actions:

- Delete the base table row.
- Update the result table.
  - Mark row as a delete hole.

However, optimistic locking cannot be used for dynamic scrollable cursors. For dynamic scrollable cursors, each lock is acquired on the underlying base table while fetching, similar to non-scrollable cursors.

## 6.6.5 Change of underlying data for scrollable cursors

In general, when your application scrolls on a result set and not on the underlying table, data in the base table can be changed by concurrent application processes. You can refer to these data changes as update or delete holes in the context of scrollable cursors.

Static scrollable cursors declared as SENSITIVE have created the necessity for detecting update and delete holes. Changes performed by others are only visible if your FETCH statement uses SENSITIVE keyword:

- DELETE HOLE

  A delete hole is created when the underlying base table row has been deleted. Delete holes are not prefetched.

  It is possible to undo a delete hole by using savepoints and rollbacks.

- UPDATE HOLE

  An update hole is created when the corresponding base table row has been modified such that the values of the rows do not qualify the row for the query any longer. Every SENSITIVE FETCH reevaluates the row against the predicate. If the evaluation fails, the row is marked as an update hole and a SQLCODE +222 is returned.

  An update hole can turn into a row again on a subsequent FETCH SENSITIVE of an update hole only if a process reverses the update and returns the values as prior to the update.

Since cursors declared with SENSITIVE DYNAMIC scroll directly on the base table, in general no update or delete holes might occur. The only special case is if an application issues a FETCH CURRENT or FETCH RELATIVE +0 statement to fetch the previously fetched row again, but the row was deleted or updated so that it no longer satisfies your WHERE clause, DB2 returns SQLCODE +231. For example, this can occur if you use lock avoidance or uncommitted read, and so no lock is taken for a row your application retrieves.

Note that the order of your result set is always maintained. If a column for an ORDER BY clause is updated, then the next FETCH statement behaves as if the updated row was deleted and reinserted into the result table at its correct location. At the time of a positioned update, the cursor is positioned before the next row of the original location and there is no current row, making the row appear to have moved.

## 6.6.6 Using multi-row FETCH with scrollable cursors

A new value of -3 for an indicator variable indicates that values were not returned for the row because a hole was detected. The value of -3 is only used for multiple-row FETCH statements. You need to provide an indicator variable array for at least one column, even if there are no nullable columns in the result table. If multiple indicator variable arrays are provided, then the indication of the hole is reflected in each indicator array.

The purpose of an indicator variable is to indicate when the associated value is the null value, or that values were not returned because a hole was detected. The value is:

- -1, if the value selected was the null value, as in prior versions.
- -2, if the null value was returned due to a numeric conversion or arithmetic expression error that occurred in the SELECT list of an outer SELECT statement, as in prior versions.
- -3, if the null value was returned because a hole was detected for the row on a multiple row FETCH, and values were not returned for the row. In cases where -3 is set to indicate a hole, SQLSTATE 02502, SQLCODE +222, is also returned for that row.

If no indicator variable arrays are provided for a multiple-row FETCH statement, and a hole is detected, an error is returned (SQLSTATE 24519, SQLCODE -247).

## 6.6.7 SQLCODEs for scrollable cursors

DB2 issues the SQLCODEs listed in Table 6-7 when your application deals with scrollable cursors:

*Table 6-7 SQLCODEs for scrollable cursors*

| SQLCODE | Description |
|---|---|
| +222 | Update or delete hole detected. |
| +231 | Cursor position invalid. |
| -222 | Update or delete attempted against an update or delete hole. |
| -224 | Result table does not agree with base table. |
| -225 | FETCH statement incompatible with non-scrollable cursor. |
| -228 | FOR UPDATE OF clause specified for read-only cursor. |
| -243 | Sensitive Cursor cannot be defined for specific SELECT statements. |
| -244 | Sensitivity option specified on FETCH conflicts with the sensitivity option in effect for your cursor. |

## 6.6.8 Summary on scrollable cursors

Scrollable cursors are not going to work in IMS environments (except for BMPs), because IMS transactions implicitly force a DB2 thread to end after the transaction completes. In CICS environments, you may keep affected threads alive using conversational transactions over several scroll up and down operations, which is usually not the case. The reason is that if the thread ends, DB2 cleans up the temporary tables being used to hold the entire result set and loses cursor positions as well for dynamic scrollable cursors.

If you have no need to scroll backwards, choose forward only cursors. If you have to maintain your cursor position to go back and forth, choose scrollable cursors in non-CICS and non-IMS environments. If you only need a snapshot of data in your tables, use INSENSITIVE cursors. In case you need actual data, choose the SENSITIVE DYNAMIC option for your scrollable cursor to receive the most actual data.

Whenever you need a scrollable cursor to scroll backwards, provide an ORDER BY that exactly matches the reverse sequence of the available index to use backward index scan and avoid sort.

**Note:** The prerequisite of scrollable cursors is keeping a thread alive for maintaining the current cursor position. As soon as a thread terminates (which is the case for most IMS and CICS transactions), cursor position is lost and the application has to reposition.

## 6.7 DB2 subsystem restart after abend

DB2 subsystem can abend due to different factors, the most frequent are software bugs in the DB2 code or other dependent software such as the operating system, I/O subsystem, third party software, power failure, operator error, and others. In every instant, all units of work that are in progress at the time of the abend can be in any status of completion.

When DB2 is restarted either manually by the operator or by any automation tool such as Automation Restart Manager (ARM), DB2 determines the status of all units of recovery (UR) and takes the appropriate action to recover the inflight transactions. There are a number of DSNZPARM parameters that determine how and when the URs should be recovered. For a detailed discussion of DB2 subsystem restart and recovery, refer to 3.1, “Improving Recovery and Restart” in *DB2 UDB for OS/390 and Continuous Availability*, SG24-5486, and Chapter 22, “Validation and performance”, of the more recent *Disaster Recovery with DB2 UDB for z/OS,* SG24-6370.

In Example 6-13, we show a sample DB2 subsystem restart after an abnormal termination. It provides a count of URs in each category and briefly lists the details of the URs that are inflight. In this particular case, we have a utility in COMMIT status. DB2 COMMITs the utility and resets all the status to 0. The message DSNR002I indicates the completion of the restart process.

*Example 6-13 DB2 MSTR address space*

```
DSNR001I  -DB2H RESTART INITIATED
DSNR003I  -DB2H RESTART...PRIOR CHECKPOINT RBA=B055F92C7A0B
DSNR004I  -DB2H RESTART...UR STATUS COUNTS  081
IN COMMIT=1, INDOUBT=0, INFLIGHT=0, IN ABORT=0, POSTPONED ABORT=0
DSNR007I  -DB2H RESTART...STATUS TABLE  082
T  CON-ID    CORR-ID     AUTHID    PLAN   S     URID      DAY   TIME
- -------- ------------ -------- -------- - ------------ --- --------
B UTILITY  DSNTEJ1A     DB2ADM   DSNUTIL  C B055F94DEB58 277 14:13:12
DSNR005I  -DB2H RESTART...COUNTS AFTER FORWARD  090
RECOVERY
IN COMMIT=0, INDOUBT=0
DSNR006I  -DB2H RESTART...COUNTS AFTER BACKWARD  091
RECOVERY
INFLIGHT=0, IN ABORT=0, POSTPONED ABORT=0
DSNR002I  -DB2H RESTART COMPLETED
-DB2HRECOVER POSTPONED
```

The status of a unit of recovery after a termination or failure depends upon the moment at which the incident occurred as listed here and referencing Figure 6-3 on page 182:

- Inflight

  The participant or coordinator failed before finishing phase 1 (period a or b); during restart, both systems back out the updates.

- Indoubt

  The participant failed after finishing phase 1 and before starting phase 2 (period c); only the coordinator knows whether the failure happened before or after the commit (point 9). If it happened before, the participant must back out its changes; if it happened afterward, it must make its changes and commit them. After restart, the participant waits for information from the coordinator before processing this unit of recovery.

- In-commit

  The participant failed after it began its own phase 2 processing (period d); it makes committed changes.

- In-abort

  The participant or coordinator failed after a unit of recovery began to be rolled back, but before the process was complete (not shown in Figure 6-3). The operational system rolls back the changes; the failed system continues to back out the changes after restart.

- Postponed abort

  If LIMIT BACKOUT installation option is set to YES or AUTO, any backout not completed during restart is postponed. The status of the incomplete URs is changed from inflight or in-abort to postponed abort.

The DSNZPARM parameter LBACKOUT is set by the DB2 system programmer. It has three possible values, NO, YES, and AUTO (default). If the value is NO, then the Unit of Recovery URs will not be postponed and DB2 processes the backouts. A value of YES will postpone the backouts of some unit of work until the command RECOVER POSTPONED is explicitly issued by the DBA. With YES or AUTO, backout processing runs concurrently with new work. Page sets or partitions with backout work pending are unavailable until their backout work is complete.

The DSNZPARM parameter BACKODUR is set to a value that indicates how many log records are to be read during restart's backward log scan. The BACKOUT DURATION field is a multiplier of the value you specify for the number of log records or the time interval of the checkpoint frequency (panel DSNTIPN of the installation).

For detailed description of recovery in each status category, refer to Chapter 18, "Restarting DB2 after termination*", DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413.

## 6.8 Recovery of objects in error

The following command can be used to list all objects that are in restrict mode.

```
-DISPLAY DB(*) SPACE(*) LIMIT(*) RESTRICT
```

This command lists all objects that are in restrict mode including COPY, LPL, WEPR, and GRECP. For more information about the DISPLAY DATABASE command, Chapter 22 of *DB2 UDB for z/OS Version 8 Command Reference,* SC18-7416, lists of all restrictive states of objects.

The command:

```
-DISPLAY DB(*) SPACE(*) LIMIT(*) ADVISORY
```

will show the objects that are in advisory status ICOPY, AUXW, and AREO*. The objects in this status do not require immediate attention, since all accesses are still permitted on these objects.

We discuss several types of status that are common for application recovery. For full details of the status and the recovery processes, refer to Appendix C. "Advisory or restrictive states", in the *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

### 6.8.1 LPL recovery

A page can be in Logical page list (LPL) if its problem can be fixed without redefining new disk tracks and volumes. This generally occurs when DB2 loses connectivity to the disk while trying to write a page to the disk. This page is marked as logically in error and placed in LPL.

The command:

```
-DISPLAY DB(*) SPACE(*) LPL ONLY
```

lists all table spaces and index spaces that are in LPL with the page ranges.

DB2 V8 will try to recover LPL pages automatically. If the automatic LPL recovery fails, DB2 V8 issues message DSNI005I to indicate the failure of the automatic LPL recovery. A new message, DSNB357I, is issued to inform you about the fact that pages have been added to LPL, which might not be automatically recovered.

If the automatic recovery is unsuccessful, then a manual recovery is required. Pages in LPL can be recovered either with the -START DATABASE command or the RECOVER utility. The error message DSNB250E has been enhanced to indicate the reasons that pages are added to the LPL.

If LPL entries exist, you need to manually issue the START DATABASE command with the SPACENAM option to initiate LPL recovery, for example:

```
-DB1G STA DB(db) SPACENAM(ts) ACCESS(RW)
```

DB2 will then read the DB2 log and apply any changes to the page set. Prior to V8, the DB2 -START DATABASE command drains the entire page set or partition, therefore, making the entire page set or partition unavailable for the duration of the LPL recovery process, even if only one page is in the LPL for that page set or partition. The "drain" means that the command must wait until all current users of the table space or partition reach their next commit point.

The RECOVER and LOAD utilities can also be used to recover LPL pages. If the START DATABASE command fails to successfully recover the LPL pages, you are forced to recover the whole page set using the RECOVER utility.

All users requesting new access to the table space or partition are also suspended and must wait until the recovery completes (or until the user times out). Therefore, the drain operation can be very disruptive to other work that is running in the system, especially in the case where only one or a few pages are in LPL.

In DB2 V8, the locking and serialization schemes in the -START DATABASE command have changed when doing the LPL recovery. In prior versions of DB2, the -START DATABASE command acquires a DRAIN ALL lock on the table space or partition when doing the LPL recovery. DB2 V8 makes a WRITE CLAIM on the table space or partition. By acquiring a WRITE CLAIM instead of a DRAIN ALL, the “good” pages can still be accessed by SQL while the -START DATABASE is recovering the LPL pages. A new "LPL recovery” lock type is also introduced to enforce that only one LPL recovery process is running at a time for a given table space or partition.

This less disruptive locking strategy potentially enhances both the performance and availability of your applications, as more data can potentially be available to the application while the pages in the LPL are being recovered.

DB2 V8 also automatically attempts to recover pages that are added to the LPL at the time they are added to the LPL, if DB2 determines that automatic recovery has a reasonable chance of success. Automatic LPL recovery is not initiated by DB2 in the following situations:

- Disk I/O error
- During DB2 restart or end of start time
- GBP structure failure
- GBP 100% loss of connectivity

Automatic LPL recovery improves the availability of your applications, because the pages in the LPL can be recovered sooner. In many cases, you do not have to issue the -START DATABASE commands yourself to recover LPL pages.

In addition, DB2 V8 provides more detailed information in message DSNB250E to explain why a page has been added to the LPL. The different reason types are:

- Disk: DB2 encountered a disk I/O error when trying to read or write pages on disk.
- LOGAPPLY: DB2 cannot apply log records to the pages.
- GBP: DB2 cannot successfully read or write the pages from or to the group buffer pool due to link or structure failure, GBP in rebuild, or GBP was disconnected.
- LOCK: DB2 cannot get the required page latch or page P-lock on the pages.
- CASTOUT: The DB2 castout processor cannot successfully cast out the pages.
- MASSDEL: DB2 encountered an error in the mass delete processor during phase 2 of commit.

These extra diagnostics help you to quickly identify and hopefully resolve why pages are being placed into the LPL, therefore, increasing the availability of your applications.

For additional information on LPL recovery, refer to 11.3 “Improved LPL Recovery”, *DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More,* SG24-6079.

#### *Objects in LPL after a DB2 restart*

As we indicated earlier, DB2 does not recover objects in LPL status when it is restarted after an abnormal failure. This can cause unnecessary outage for applications which access the object in LPL and will fail with -904, “`resource unavailable`” error. In order to prevent this failure and improve availability, the -START DATABASE command can be issued by an automation product, such as NETVIEW or an independent software vendor (ISV) product. One such technique is to:

- Capture the DB2 normal start message DSN9022I.
- Issue a -DISPLAY DATABASE(*) SPACE(*) LPL ONLY command within a REXX exec and scan for any objects in LPL.
- If objects are found in LPL, then issue the -START DB(db) SPACE(ts) ACCESS(RW).
- If the command is successful and the object is recovered by the -START command, then the object becomes available to applications.
- If the command fails with DSNI005, then alert the DBA to take manual action.

Such a procedure can be invoked for both normal and abnormal starts of the DB2 subsystem.

### 6.8.2 CHECK-pending

The CHECK-pending (CHKP) restrictive status indicates that an object might be in an inconsistent status and must be checked.

The following utilities set the CHECK-pending status on a table space if RI constraints are encountered:

- LOAD with ENFORCE NO.
- RECOVER to a point in time.
- CHECK LOB CHECK-pending status can also affect a base table space or a LOB table space.

DB2 ignores informational RI constraints and does not set CHECK-pending status for them.

Table spaces and base table spaces can be reset by using the CHECK DATA utility which checks RI constraint failures and in the case of bad FKs, deletes the offending rows from the table and stores them in an exception table. If a table space is in both REORG-pending and CHECK-pending status (or auxiliary CHECK-pending status), run REORG first, and then use CHECK DATA to clear the respective states.

A partitioning index, non-partitioning indexes, and indexes on the auxiliary tables can all be placed in check pending status if they are recovered to a point in time - PIT (either RBA or LRSN) and the logs have been applied but the corresponding table space is not recovered to the same PIT. The same happens if the table space and the index space are not recovered to the same QUIESCE point or COPY SHRLEVEL REFERENCE point. To reset the CHKP status, you run the CHECK INDEX utility. If errors are found, then you can use the REBUILD INDEX utility to rebuild the index.

For a LOB table space, run the CHECK LOB utility and if errors are found, then you can correct the defects using the REPAIR utility. Rerun the CHECK LOB utility to reset the status.

## 6.8.3 Write Error Page Range recovery

A page is physically in error if there are physical errors caused by disk errors. The page is then placed in 59B9E34, Write Error Page Range (WEPR). The range has low and high pages, which are the same if only one page has errors. The command:

```
-DISPLAY DB(*) SPACE(*) WEPR ONLY
```

will list all table spaces and index spaces in physical error. Pages in WEPR can be recovered with the RECOVER and LOAD utilities.

Applications attempting to access the pages in LPL or WEPR will receive an SQLCODE error “`resource unavailable`.”

## 6.8.4 COPY utility

An object can be placed in COPY-pending status if:

- A LOAD or REORG utility with LOG NO is executed on the objects without the inline COPYDDN option.
- A MODIFY is done resulting in zero entries in SYSIBM.SYSCOPY for the object.
- The object has been RECOVERed to current.

We strongly recommend that a backup of the object is made, either with the COPY utility or the INLINE option of the REORG utility. We strongly discourage using the REPAIR utility with SET table space spec NOCOPYPEND or -START DB(db) SPACE(ts) ACCESS(FORCE) to maintain data integrity of the object.

Table 6-8 contains a list of all types of object status from a -DISPLAY DATABASE command. Appendix C. “Advisory or restrictive states” in the *DB2 for z/OS V8 Utility and Reference Guide*, SC18-7427, contains a list of the types of restrictive status and the required steps to correct each status for a particular object.

*Table 6-8 DB2 object status from DISPLAY command*

| Status | Description |
|---|---|
| AREO* | The table space, index, or partition is in Advisory REORG-pending status.<br>The object should be reorganized to improve performance. This status is new as of DB2 V8. |
| ACHKP | When the LOB table space is recovered to any previous point in time, the base table space is placed in auxiliary CHECK-pending (ACHKP) status, and the index space containing an index on the auxiliary table is placed in REBUILD-pending (RBDP) status.<br>Update or delete invalid LOBs using SQL.<br>Run the CHECK DATA utility with the appropriate SCOPE option to verify the validity of LOBs and reset ACHKP status. |
| AREST | The table space, index space, or partition is in Advisory Restart Pending status.<br>If backout activity against the object is not already underway, either issue the RECOVER POSTPONED command or recycle the specifying LBACKOUT=AUTO. |
| AUXW | Either the base table space or the LOB table space is in the Auxiliary Warning status. This warning status indicates an error in the LOB column of the base table space or an invalid LOB in the LOB table space.<br>Update or delete invalid LOBs using SQL.<br>For base table space, run CHECK DATA utility to verify the validity of LOBs and reset AUXW status.<br>For LOB table space, run CHECK LOB utility to verify the validity of LOBs and reset AUXW status. |
| CHKP | The Check Pending status has been set for this table space or partition, index (partition, non-partition, index on auxiliary table), or LOB table space.<br>Refer to 6.8.2, "CHECK-pending" on page 214 for reset of CHKP status. |
| COPY | The COPY-pending flag has been set for this table space or partition.<br>Using the COPY utility, generate an image copy for the table space or partition.<br>Refer to 6.8.4, "COPY utility" on page 215. |
| DEFER | Deferred restart is required for the object. |
| GRECP | The table space, table space partition, index, index partition, or logical index partition is in the group buffer pool Recovery Pending status.<br>Recover the object, or use START DATABASE to recover the object. |
| ICOPY | The index is in Informational COPY-pending status.<br>Run COPY utility on the index. |
| INDBT | In-doubt processing is required for the object. |
| LPL | The table space, table space partition, index, index partition, or logical index partition has logical page errors.<br>Refer to 6.8.1, "LPL recovery" on page 212. |
| LSTOP | The logical partition of a non-partitioning index is stopped. |
| OPENF | The table space, table space partition, index, index partition, or logical index partition had an open data set failure. |
| PSRCP | Indicates Page Set Recovery Pending status for an index (non-partitioning indexes). |
| RBDP | The physical or logical index partition is in the REBUILD-pending status. |
| RBDP* | The logical partition of a non-partitioning index is in the REBUILD-pending status, and the entire index is inaccessible to SQL applications. However, only the logical partition needs to be rebuilt. |

| Status | Description |
|---|---|
| RECP | The Recover Pending flag has been set for this table space, table space partition, index, index partition, or logical index partition.<br>A successful LOAD REPLACE or RECOVER will reset the status. |
| REFP | The table space, index space, or index is in Refresh Pending status.<br>A successful LOAD REPLACE or RECOVER will reset the status. |
| RELDP | The object has a release dependency. |
| REORP | The data partition is in a REORG-pending status.<br>For reset of REORGP status, Refer to Table 171. Resetting REORG-pending status, Appendix C. “Advisory or restrictive states”, *DB2 UDB for z/OS Version 8 Utility Guide and Reference*, SC18-7427. |
| RESTP | The table space or index is in Restart Pending status. Restart processing has been initiated for the table space, table space partition, index, index partition, or logical index partition. |
| RO | The table space, table space partition, index, index partition, or logical index partition has been started for read-only processing. |
| RW | The table space, table space partition, index, index partition, or logical index partition has been started for read and write processing. |

## 6.9 Application recovery process

In this section, we concentrate on application recovery process only. The recovery of the entire DB2 subsystem in a disaster recovery scenario is covered in the redbook, *Disaster Recovery with DB2 UDB for z/OS*, SG24-6370.

DB2 logs all SQL updates to table spaces and index spaces in the active log. LOAD and REORG utilities have the option not to log the changes with the LOG NO option. When these utilities are run with LOG NO option, the table space is set to COPY-pending status. The COPY-pending status must be reset in order to allow update activities on the table space. Refer to 6.8.2, “CHECK-pending” on page 214 for details about resetting the status.

The active log is switched and copied to archive log when the active log becomes full. The bootstrap data set (BSDS) records the current status of the active logs and records the data set details of all archive logs with the START and END-RBA (LRSN in data sharing). The maximum number of archive log entries in BSDS is limited by the DSNZPARM option MAXARCH which can range from 10 to 10000 in DB2 V8. To understand more about the DB2 logging and archiving of logs environment, refer to Chapter 17, “Establishing the logging environment”, in *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413.

### Expiry of archive logs

One of common problems encountered in many DB2 subsystem recovery processes is the availability of the required archive logs. Although the archive logs are recorded in the BSDS, the data set may be missing in the ICF catalog.

For tape-based archive log data sets, the problem is with the RETPD value for the archive log. The storage system programmer configures the tape management system RMM or other third-party product to retain the tapes for a limited period, which may override the DB2 DSNZPARM parameter ARCRETN. Either ARCRETN or the RETPD parameter can expire a tape data set while it still exists in the BSDS repository. In such an environment, recovery of the DB2 object will fail if no frequent image copies exist for the object.

Similar problems may exist for disk-based archive logs which are SMS-managed and the retention period of the data set is set in the management class. Based on the management class rules, the disk data set can be migrated and deleted based on the creation date.

The reverse problem may exist where the physical archive log dataset exists in the z/OS platform, but it is not longer listed in the BSDS. This may occur when there is a combination of:

- Active log size is too small.
- High logging activity in the DB2 subsystem which triggers high archive logging.
- The MAXARCH value in DSNZPARM (DSNTIPA installation panel) is too low.

In all cases, the recovery of the table space and index space may be affected. Close consultation with the storage administrators and DB2 system programmers will assist in setting the DSNZPARM values and the storage data set expiry values. Note that in DB2 V8 new-function mode, you can increase the MAXARCH value to retain up to 10,000 archive log entries in the BSDS.

## 6.9.1 Rolling back work

If failure occurs within a unit of recovery, DB2 backs out any changes to data, returning the data to its status at the start of the unit of recovery; that is, DB2 undoes the work. The events are shown in Figure 6-13. The SQL ROLLBACK statement, deadlocks, and timeouts (reported as SQLCODE -911, SQLSTATE 40001) cause the same events.

The effects of inserts, updates, and deletes to large object (LOB) values are backed out along with all the other changes made during the unit of work being rolled back, even if the LOB values that were changed reside in a LOB table space with the LOG NO attribute.

An operator or an application can issue the CANCEL THREAD command with the NOBACKOUT option to cancel long running threads without backing out data changes. DB2 backs out changes to catalog and directory tables regardless of the NOBACKOUT option. As a result, DB2 does not read the log records and does not write or apply the compensation log records. After CANCEL THREAD NOBACKOUT processing, DB2 marks all objects associated with the thread as refresh pending (REFP) and puts the objects in a logical page list (LPL). For information about how to reset the REFP status, see *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

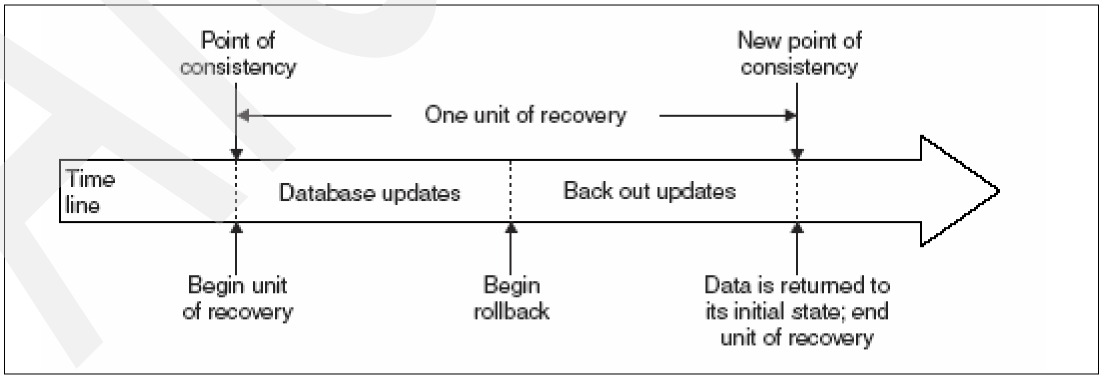


*Figure 6-13 Unit of recovery (rollback)*

The NOBACKOUT request might fail for either of the following reasons:

- DB2 does not completely back out updates of the catalog or directory (message DSNI032I with reason 00C900CC).
- The thread is part of a global transaction (message DSNV439I).

## 6.10 Preparing to recover to a point of consistency

Backups of the application-related objects must exist in a consistent status in order to recover the objects to a point of consistency (POC) to overcome either application errors or physical errors caused by I/O subsystems, software errors, and others. Prior to running the COPY utility to back up the objects, use the DISPLAY DATABASE RESTRICT command to determine whether the data is in an exception status. Refer to Table 6-8 on page 216 for status and the reset process.

A method must be established to identify a group of related objects prior to the COPY utility which does the backups and the QUIESCE utility to establish a point of consistency. The same methodology can be used to recover the objects with the RECOVER utility.

### Establishing naming standards

Design the application and establish a naming standard to identify related objects. As an example, if the payroll application can be identified as PAY, then the databases naming standard can include PAY in the database name, such as PAYDB*nnn* where *nnn* is a sequence number; similarly, name the table spaces (PAYTS*nnn*), index spaces (PAYIX*nnn*), and tables (PAYTB*nnn*). Then, using the LISTDEF utility statement, all table spaces and index spaces belonging to the PAYROLL application can be obtained as:

```
LISTDEF PAYROLL INCLUDE table space PAYDB*.*
```

COPY and QUIECE utilities can utilize the list of objects generated by the LISTDEF utility to create the backups and to establish a point of consistency as in Example 6-14.

*Example 6-14 Sample LISTDEF, COPY, and QUIESCE*

```
LISTDEF PAYROLL INCLUDE table space PAYDB*.*
TEMPLATE COPY1 ................
TEMPLATE COPY2 ................
COPY LIST PAYROLL SHRLEVEL CHANGE PARALLEL
      COPYDDN     COPY1
      RECOVERYDDN  COPY2
QUIESCE LIST PAYROLL
```

### RI and table space set

Design the application and create RI between the tables. Refer to Chapter 3, “Referential integrity” on page 49 for discussions about RI. The LISTDEF utility statement, REPORT utility, and the QUIESCE utility can be used to generate all objects that are RI-related. The table space set can be constructed from the job output of these utilities.

#### *REPORT utility TABLESPACESET*

The REPORT utility with TABLESPACESET option can be used to list all table spaces that are RI-related and informational RI-related for table space RIDEPT. In Example 6-15, a table space report lists RIDEPT, RIEMPL, and RIHIST that are RI-related. The report includes the associated indexes and tables. The utility does not update any catalog tables.

*Example 6-15 REPORT with TABLESPACESET option*

```
//*
//* GET A LIST OF TABLE SPACE SET
//*
//REPORT   EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLOR1
//SYSIN    DD   *
  REPORT TABLESPACESET DSNDB04.RIDEPT

TABLESPACE SET REPORT:


TABLESPACE       : DSNDB04.RIDEPT

   TABLE         : PAOLOR4.DEPT
      INDEXSPACE : DSNDB04.DEPTKO
           INDEX : PAOLOR4.DEPTKO
      DEP  TABLE : PAOLOR4.EMPL

TABLESPACE       : DSNDB04.RIEMPL

   TABLE         : PAOLOR4.EMPL
      INDEXSPACE : DSNDB04.EMPLKO
           INDEX : PAOLOR4.EMPLKO
      DEP  TABLE : PAOLOR4.EMPHIST

TABLESPACE       : DSNDB04.RIHIST

   TABLE         : PAOLOR4.EMPHIST
DSNU580I    DSNUPORT - REPORT UTILITY COMPLETE - ELAPSED TIME=00:00:00
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

### QUIESCE with TABLESPACESET

The QUIESCE utility can be used to obtain a list of all table spaces that are RI-related to a single table space. In Example 6-16, the utility was run for RIDEPT table space. The job output lists RIEMPL and RIHIST with RIDEPT. Note that the QUIESCE utility takes a quiesce point and records the value in SYSCOPY.

*Example 6-16 QUIESCE with TABLESPACESET option*

```
//*
//* GET A LIST OF TABLE SPACE SET USING QUIESCE UTILITY
//*
//REPORT   EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLOR1
//SYSIN    DD   *
  QUIESCE TABLESPACESET DSNDB04.RIDEPT
//
DSNU000I    DSNUGUTC - OUTPUT START FOR UTILITY, UTILID = PAOLOR1
DSNU1044I   DSNUGTIS - PROCESSING SYSIN AS EBCDIC
DSNU050I    DSNUGUTC -  QUIESCE TABLESPACESET DSNDB04.RIDEPT
DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACESET DSNDB04.RIDEPT
DSNU477I  -DB8A DSNUQUIA -    QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIDEPT
DSNU477I  -DB8A DSNUQUIA -    QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIEMPL
DSNU477I  -DB8A DSNUQUIA -    QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIHIST
DSNU474I  -DB8A DSNUQUIA - QUIESCE AT RBA 000212ED1A43 AND AT LRSN 000212ED1A43
DSNU475I    DSNUQUIB - QUIESCE UTILITY COMPLETE, ELAPSED TIME= 00:00:00
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
******************************* BOTTOM OF DATA ********************************
```

### LISTDEF utility with RI option

The LISTDEF utility was introduced in DB2 V7 and can be used to generate table spaces and index spaces with a wild card. The utility can also be used to generate objects that are RI-related.

The command:

```
LISTDEF PAYROLL INCLUDE TABLESPACE DSNDB04.RIDEPT RI
```

generates a list of all table spaces that are RI-related to table space DSNDB04.RIDEPT. The COPY and QUIESCE utilities can be set up in the same jobstep to back up and quiesce all the table spaces in the list as in Example 6-17.

*Example 6-17 Sample LISTDEF with RI, COPY, and QUIESCE*

```
//*
//*   LISTDEF, COPY AND QUIESCE UTILITY
//*
//COPY     EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLO,
//         LIB='DB8A8.SDSNLOAD'
//SYSIN    DD   *
   LISTDEF  PAOLO
            INCLUDE TABLESPACE DSNDB04.RIDEPT RI
   TEMPLATE COPY1
            DSN(DB8ALDS.&DB..&TS..D&DATE..T&TIME.)
            DISP(NEW,CATLG,DELETE) UNIT SYSDA SPACE(10,5) TRK
   COPY     LIST PAOLO
            SHRLEVEL CHANGE    PARALLEL
            COPYDDN COPY1
   QUIESCE  LIST PAOLO
//
Sample Job output
DSNU1038I   DSNUGDYN - DATASET ALLOCATED.  TEMPLATE=COPY1
                       DDNAME=SYS00001
                       DSN=DB8ALDS.DSNDB04.RIHIST.D2005298.T182326
DSNU1038I   DSNUGDYN - DATASET ALLOCATED.  TEMPLATE=COPY1
                       DDNAME=SYS00002
                       DSN=DB8ALDS.DSNDB04.RIEMPL.D2005298.T182326
DSNU1038I   DSNUGDYN - DATASET ALLOCATED.  TEMPLATE=COPY1
                       DDNAME=SYS00003
                       DSN=DB8ALDS.DSNDB04.RIDEPT.D2005298.T182326
DSNU400I    DSNUBBID - COPY PROCESSED FOR TABLESPACE DSNDB04.RIHIST
                      NUMBER OF PAGES=2
                      AVERAGE PERCENT FREE SPACE PER PAGE =  0.00
                      PERCENT OF CHANGED PAGES =  0.00
                      ELAPSED TIME=00:00:00
DSNU428I    DSNUBBID - DB2 IMAGE COPY SUCCESSFUL FOR TABLESPACE DSNDB04.RIHIST
DSNU400I    DSNUBBID - COPY PROCESSED FOR TABLESPACE DSNDB04.RIEMPL
                      NUMBER OF PAGES=2
                      AVERAGE PERCENT FREE SPACE PER PAGE =  0.00
                      PERCENT OF CHANGED PAGES =  0.00
                      ELAPSED TIME=00:00:00
DSNU428I    DSNUBBID - DB2 IMAGE COPY SUCCESSFUL FOR TABLESPACE DSNDB04.RIEMPL
DSNU400I    DSNUBBID - COPY PROCESSED FOR TABLESPACE DSNDB04.RIDEPT
                      NUMBER OF PAGES=2
                      AVERAGE PERCENT FREE SPACE PER PAGE =  0.00
                      PERCENT OF CHANGED PAGES =  0.00
                      ELAPSED TIME=00:00:00
DSNU428I    DSNUBBID - DB2 IMAGE COPY SUCCESSFUL FOR TABLESPACE DSNDB04.RIDEPT
DSNU050I    DSNUGUTC -  QUIESCE LIST PAOLO
DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIDEPT
```

```
DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIEMPL
DSNU477I  -DB8A DSNUQUIA - QUIESCE SUCCESSFUL FOR TABLESPACE DSNDB04.RIHIST
DSNU474I  -DB8A DSNUQUIA - QUIESCE AT RBA 00021A65645F AND AT LRSN 00021A65645F
DSNU475I    DSNUQUIB - QUIESCE UTILITY COMPLETE, ELAPSED TIME= 00:00:00
DSNU010I    DSNUGBAC - UTILITY EXECUTION COMPLETE, HIGHEST RETURN CODE=0
```

Alternatively, the list can be passed in a data set which can be allocated using the OPTIONS utility. In Example 6-18, the LISTDEF definition is saved in a dataset and passed to the RUNSTATS and QUIESCE utilities with the OPTIONS LISTDEFDD option.

*Example 6-18 LISTDEF list passed through OPTIONS utility*

```
//*
//LISTDEF  EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=*
//SYSIN    DD   DUMMY
//SYSUT2   DD   DSN=PAOLOR1.PAOLO.LISTDEF,DISP=(,PASS),
//         UNIT=SYSDA,SPACE=(TRK,(1,1),RLSE),
//         DCB=(RECFM=FB,LRECL=80,BLKSIZE=0)
//SYSUT1   DD   *
   LISTDEF PAOLO
           INCLUDE TABLESPACE DSNDB04.RIDEPT RI
//*
//*   RUNSTATS UTILITY
//*
//LIST     EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLO,
//         LIB='DB8A8.SDSNLOAD'
//LISTLIB  DD   DSN=PAOLOR1.PAOLO.LISTDEF,DISP=(OLD,PASS)
//SYSIN    DD   *
   OPTIONS LISTDEFDD LISTLIB
   RUNSTATS TABLESPACE LIST PAOLO
            SHRLEVEL CHANGE
            UPDATE(ALL) HISTORY(ALL)
//*
//*   QUIESCE UTILITY
//*
//QUIESCE  EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLO,
//         LIB='DB8A8.SDSNLOAD'
//LISTLIB  DD   DSN=PAOLOR1.PAOLO.LISTDEF,DISP=(OLD,DELETE)
//SYSIN    DD   *
   OPTIONS LISTDEFDD LISTLIB
   QUIESCE LIST PAOLO
//
```

We advise using the LISTDEF utility rather than the QUIESCE table space set option. The list generated by LISTDEF can be passed to many other DB2 utilities in the same jobstep.

### 6.10.1 Copying the data

You can copy the data and also establish a point of consistency for a list of objects, in one operation, by using the COPY utility with the option SHRLEVEL REFERENCE. That operation allows read-only access to the data while it is copied. The data is consistent at the moment when copying starts and remains consistent until copying ends. The advantage of the method is that the data can be restarted at a point of consistency by restoring the copy only, with no need to read log records. The disadvantage is that updates cannot be made throughout the entire time that the data is being copied.

Copies of data created with COPY utility SHRLEVEL CHANGE do not provide a point of consistency, since updates are allowed during the utility. The QUIESCE utility must be run immediately after the COPY utility to establish the point of consistency. The RECOVER utility will restore the backup copy and apply all log records to the point of consistency when recovered with TORBA. The RBA (LRSN in data sharing) can be found in SYSCOPY, ICTYPE=Q, and START_RBA.

Inline copies created with REORG utility SHRLEVEL CHANGE cannot be used to recover to TOCOPY. Inline copies of REORG may have duplicates pages appended to the end of the copy data set. These duplicate pages are created during the last log apply phase of REORG. The inline copies can be used to recover to a RBA if the RBA is established with the QUIESCE utility.

### Frequency of image copies

Many factors influence the frequency of image copies. If the recovery of your application is at near real time (within minutes or less than an hour), then consider “mirroring” all the DB2 objects at the remote site, using products such as XRC or PPRC. Refer to *Disaster Recovery with DB2 UDB for z/OS*, SG24-6370, for details.

To recover from an “application disaster recovery”, image copies and the point of consistency must be available prior to the application disaster. The frequency of image copies to recover from an “application disaster recovery” is determined by:

- The elapsed time of recovery
- The elapsed time of COPY utility
- The characteristics for your active and archive logs
- The required DB2 restart time, including the recovery of data

Even if the organization has disaster recovery policies and processes in place, we still recommend that frequent image copies of the application are taken at regular intervals for application recovery. It will reduce the elapsed time of the recovery and obviously reduce the application outage.

The DBA can use customer and application knowledge to determine the frequency of image copies. Generally, application objects are image-copied at the end of an online session and after a mass batch update. The trick is to find a “quiet” time to image copy with SHRLEVEL REFERENCE and QUIESCE the objects. Alternatively, image copy with SHRLEVEL CHANGE and QUIESCE the objects during the quiet window. DB2 tools, such as the Log Analysis Tool, can be used to scan the log to identify a quiet time of the day to ensure that the QUIESCE will complete successfully. You might also consider using  incremental copies to reduce the execution time.

### QUIESCE utility

As discussed, the QUIESCE utility is used to obtain a point of consistency for an object or a group of objects. Generally, the utility will succeed but in some cases it will not when it fails to drain the writers within a specified time limit. For applications such as SAP and PeopleSoft, which use large numbers of DB2 objects, it may not be practical to list all the objects for the QUIESCE utility. There is an alternative method to obtain a system-wide quiesce point. The command ARCHIVE LOG MODE(QUIESCE) has a wait parameter which can be used to obtain a system-wide quiesce point. In Example 6-19, an ARCHIVE LOG command with MODE(QUIESCE) was issued. The command waits for 600 seconds to drain all writers to establish the system-wide quiesce point. It it fails after a 600 seconds wait, then a normal ARCHIVE LOG command is issued to switch the active log and produce an archive log.

*Example 6-19 Archive log command*

```
//*
//STEP001  EXEC DB2CMD,SYSTEM=DB2D
//SYSTSIN  DD *
 DSN SYSTEM(DB2D)
 -ARCHIVE LOG MODE(QUIESCE) WAIT(YES) TIME(600)
 END
//*
//         IF (RC > 4) THEN
//*
//******************************************************************
//* SWITCH WITHOUT QUIESCE IF THE PRIVIOUS ATTEMPT FAILS.
//*******************************************************************
//*
//STEP002  EXEC DB2CMD,SYSTEM=DB2D
//SYSTSIN  DD *
 DSN SYSTEM(DB2D)
 -ARCHIVE LOG
 END
/*
//         ENDIF
```

A successful execution of the ARCHIVE LOG command registers an entry in the BSDS as in Example 6-20. The MODE column indicates if the ARCHIVE LOG was successful with MODE(QUIESCE). The BSDS can be listed using DSNJU004 utility.

*Example 6-20 BSDS listing of archive log command*

```
ARCHIVE LOG COMMAND HISTORY
                      08:02:12 OCTOBER 19, 2005
   DATE         TIME         RBA        MODE   WAIT   TIME
------------  ----------  ------------   -------  ----   -----
OCT 18, 2005  22:06:48.5  00D92360314A   QUIESCE  YES    600
OCT 17, 2005  22:02:35.0  00D91850DB14   QUIESCE  YES    600
OCT 16, 2005  22:02:25.2  00D90ECD6EDC   QUIESCE  YES    600
OCT 15, 2005  22:02:35.0  00D903D69C53   QUIESCE  YES    600
OCT 14, 2005  22:06:50.5  00D8FF2E8490   QUIESCE  YES    600
OCT 13, 2005  22:02:31.8  00D8EFDEDA72   QUIESCE  YES    600
OCT 12, 2005  22:08:45.1  00D8E508F984   QUIESCE  YES    600
OCT 11, 2005  22:05:09.3  00D8DB3ACBB7   QUIESCE  YES    600
OCT 10, 2005  22:04:59.1  00D8CFEF4CEB   QUIESCE  YES    600
OCT 09, 2005  22:02:00.9  00D8C7384EDC   QUIESCE  YES    600
OCT 08, 2005  22:02:37.9  00D8BC56C4EA   QUIESCE  YES    600
OCT 07, 2005  22:02:20.0  00D8B795CB20   QUIESCE  YES    600
```

DB2 also provides assistance to the DBA in determining when to image copy. The statistics generated by RUNSTATS utility, Real-Time Statistics (RTS), DB2 Administration tool, and DB2 Automation Tool can assist the DBA in determining when image copies are required. DB2 Automation Tool is the strategic tool for generating utilities.

## RUNSTATS statistics

The RUNSTATS utility with HISTORY(YES) updates the SYSTABLEPART_HIST catalog table. Table SYSTABLEPART_HIST has two fields, CARDF and SPACEF, which contain the values for total number of rows and total amount of disk space allocated to the table space or partition space respectively. Similar fields also exist for index space in SYSINDEXPART_HIST. CARDF and SPACEF can be tracked regularly for space growth of an object. Assuming constant growth over time, these numbers can be used to derive growth trends to plan for future needs.

Consider the following sample SQL:

```
SELECT MAX(CARDF), MIN(CARDF), ((MAX(CARDF)-MIN(CARDF))*100)/MIN(CARDF),
(DAYS(MAX(STATSTIME))-DAYS(MIN(STATSTIME)))
FROM SYSIBM.SYSTABLEPART_HIST
WHERE DBNAME='DB' AND TSNAME='TS';
```

Assuming that the number of rows is constantly increasing, so that the highest number is the latest, the query shows the percentage of rows added over a specific time period. This could be extrapolated to scale on a monthly or yearly basis.

The space growth and SYSCOPY entries of a DB2 object can be used to determine the requirement for an image copy. If the space growth is greater than a predetermined percentage growth value since the last full image copy, then the COPY utility can be triggered to take a full image copy of the object.

## Real-Time Statistics (RTS)

The statistics are collected in real time, kept in memory, and periodically written to user defined DB2 tables from which applications and tools can query the statistics.

### *Table definition*

The statistics are contained within a user-created database called DSNRTSDB and a segmented table space called DSNRTSTS. The RTS tables are:

- SYSIBM.SYSTABLESPACESTATS

  It contains table space statistics, one row per table space or partition.
- SYSIBM.SYSINDEXSPACESTATS

  It contains index space statistics, one row per index space or index partition.
- The tables must be defined with row level locking and CCSID EBCDIC. A dedicated buffer pool will improve the efficiency while updating statistics. Refer to Appendix E, “Real-time statistics tables”, in the *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427, for details.

### *DSNACCOR stored procedure*

This stored procedure queries the new DB2 RTS tables to determine which DB2 objects should be reorganized, should have their statistics updated, should be image-copied, have exceeded number of extents, or are in a restricted status. The default scope for this stored procedure is to scan "all" data in the RTS tables and provide recommendations for "any" of the conditions mentioned.

For example, here is the formula that DSNACCOR uses to query the RTS tables to determine if a full image copy should be run on a table space:

```
((QueryType='COPY' OR QueryType='ALL') AND
   (ObjectType='TS' OR ObjectType='ALL') AND
   ICType='F') AND
   (COPYLASTTIME IS NULL OR
   REORGLASTTIME>COPYLASTTIME OR LOADRLASTTIME>COPYLASTTIME OR
    (CURRENT DATE-COPYLASTTIME)>CRDaySncLastCopy OR
(COPYUPDATEDPAGES*100)/NACTIVE>CRUpdatedPagesPct OR
    (COPYCHANGES*100)/TOTALROWS>CRChangesPct)
```

- CRDaySncLastCopy has a default value of “7” (seven days since last copy).

- CRUpdatedPagesPct has a default value of “20” (if the ratio of updated pages to preformatted pages is at least 20%).
- CRChangesPct has a default value of 10 (if the ratio of the number of INSERTs, UPDATEs, and DELETEs since the last image copy to the total number of rows or LOBs in a table space or partition, expressed as a percentage, is greater than 10%).

Refer to Appendix B, “The DB2 real-time statistics stored procedure”, *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427, for details about this stored procedure.

## DB2 Administration Tool

The DB2 Administration Tool can be used to query the catalog for table spaces that need to be image-copied based on the CRDaySncLastCopy value of the DSNACCOR stored procedure. Example 6-21 shows a sample of table spaces in DSNDB04 that require full image copy with CRDaySncLastCopy = 1. The tool also recommends REORG and RUNSTATS for the table spaces.

*Example 6-21 Recommendations from the DB2 Administration Tool*

```
DB2 Admin ---------- DB8A Table Space Maintenance ---------------- Row 1 of 19
Command ===>                                                     Scroll ===> CSR

Commands:       C - Full Copy  CI - Inc Copy  O - Reorg  R - Runstats
Line commands:  C - Full Copy  CI - Inc Copy  O - Reorg  R - Runstats
               AL - Resize
                                          Pct    Num   <---Recommendations--->
Sel TSname   DBname     Part   Space(KB) Used    Ext   Copy Reorg Runst Resize
    *        *             *           *    *      *   *    *     *     *
--- -------- -------- ------ ----------- ---- ------   ---- ----- ----- ------

    PLANRTAB DSNDB04       0          48  100      1   FUL  YES   YES   NO
    PLAN1QDX DSNDB04       0         720  100      1   FUL  YES   YES   NO
    ROAD     DSNDB04       0         720  100      1   FUL  YES   YES   NO
    TRIGTEST DSNDB04       0       23232   96      1   FUL  YES   YES   NO
******************************* END OF DB2 DATA *******************************
```

## REORG and LOAD utility

Both REORG and LOAD utilities can take an image copy of the table space if the COPYDDN and RECOVERYDDN options are specified as part of the utility execution. The utilities also write log records if the LOG YES option is specified. Generally, the utilities are run with LOG NO for performance.

The object is placed in COPY status if the COPYDDN option is not coded as part of the utility run. We strongly advise you to take an image copy of the object using the COPY utility, SHRLRVEL REFERENCE or CHANGE immediately after the successful completion of the utility. Resetting the table space status to RW with -START DB(db) SPACE(ts) ACCESS(FORCE) or running REPAIR utility with SET NOCOPYPEND is strongly discouraged for recoverability.

The inline image copies taken with REORG and LOAD utilities are not suitable for RECOVER utility TOCOPY, TOLASTCOPY, or TOLASTFULLCOPY options.

## MODIFY utility

The MODIFY utility with the RECOVERY option deletes records from the SYSIBM.SYSCOPY catalog table, related log records from the SYSIBM.SYSLGRNX directory table, and entries from the DBD, and recycles version numbers for reuse. The AGE or DATE options of the utility can be used to selectively delete all rows from the catalog tables older than a certain number of days or to a given date. The utility can delete all rows in the catalog tables if AGE(*) or DATE(*) is specified. Both the later options will place the table space in COPY-pending status.

We advise you to run the MODIFY utility regularly to remove "old" entries from the catalog tables, SYSCOPY and SYSLGRNX, for performance. A DB2 subsystem-wide policy may be developed to maintain a limited number of rows in the catalog for each DB2 object.

### 6.10.2 Recovery of the data to the previous point of consistency

The RECOVER utility recovers table spaces and index spaces from the corresponding image copies and log records. The REBUILD utility can be used to rebuild the indexes if the index is not copied using the COPY utility. Standalone utilities, such as DSN1COPY and any similar third-party vendor software, can be used to restore the table space data but the restore is done outside of the control of DB2. You must take appropriate action to ensure that the data is intact and consistent.

In this section, we discuss the recovery of table spaces that are logically related to a table space set. For a complete description of the recovery process, refer to Chapter 20, *DB2 Administration Guide*, SC18-7413. The relationship between tables can be defined either using the RI constraint of table definition, application programs, or a combination of both. Similarly, a LOB table space and its associated base table space are also part of a table space set. In all cases, the DBA must be able to identify the table space set for recovery.

You can use the REPORT table space set utility or the LISTDEF utility with RI option to determine all the page sets that belong to a single table space set and then restore those page sets that are related. Refer to the LISTDEF utility examples and discussion in "Preparing to recover to a point of consistency" on page 219. However, if page sets are logically related outside of DB2 in application programs, you are responsible for identifying all the page sets on your own.

#### Obtain the point of consistency

A point of consistency can be obtained from:

- SYSCOPY
- BSDS
- DSN1LOGP standalone utility
- REPORT utility

#### *SYSCOPY*

The START_RBA attribute in SYSCOPY is assigned a 48-bit positive integer that contains the LRSN of a point in the DB2 recovery log. The LRSN is the RBA equivalent in the data sharing environment. A value for START_RBA is recorded in SYSCOPY whenever the COPY, QUIESCE, LOAD, REORG, and RECOVER utilities are executed on table spaces and index spaces. The value of the attribute START_RBA can be used to recover to a point in time if:

- The START_RBA is created with the QUIESCE utility (ICTYPE=Q).
- The COPY utility is run on the table space with SHRLEVEL REFERENCE (ICTYPE=F, SHRLEVEL=R).

In both instances, the table space can be recovered with the RECOVER utility and to a TOLOGPOINT. Although the table space may be consistent, there is no guarantee that the table space set is consistent.

Consider an example where DEPT and EMP are part of a simple application. Figure 6-14 shows a timeline for the recovery of table spaces TOLOGPOINT. All image copies of DEPT and EMP were done with SHRLEVEL REFERENCE. If DEPT is recovered at time T4 to the last image copy (with TOCOPY or TOLOGPOINT= R1) then all updates to DEPT will be lost after time T1, although the recovery will succeed and DEPT will be at a point of consistency. Unfortunately, both DEPT and EMP are a set and as a set, they will be inconsistent. If both DEPT and EMP are recovered to log point R3 taken at time T3 with the QUIESCE utility, then the table space set will be at a point of consistency.

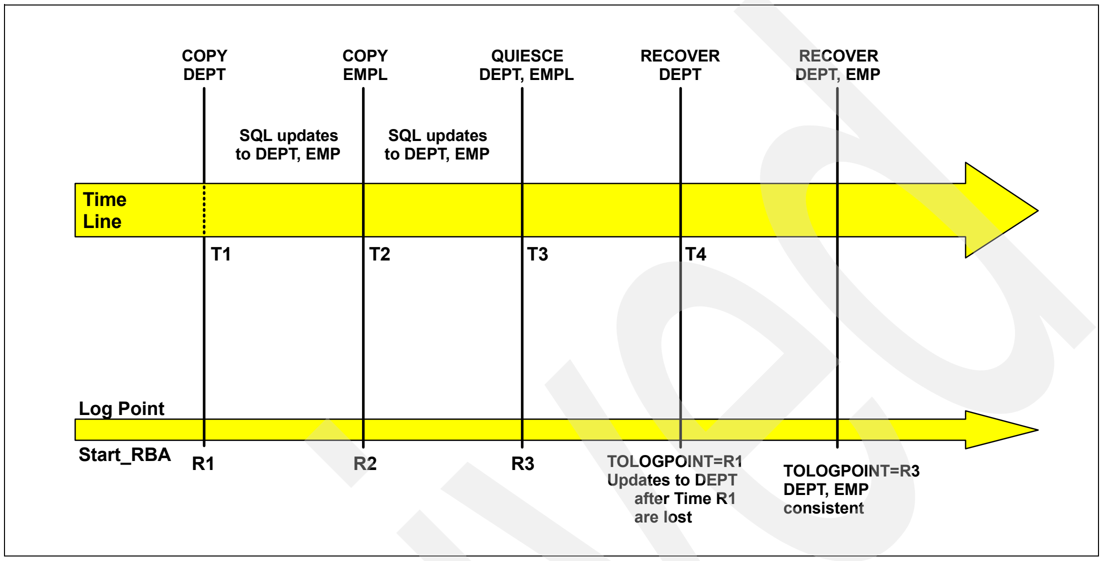


*Figure 6-14 Recovery of table spaces TOLOGPOINT*

## BSDS

The BSDS is the repository for DB2 to record control information. Sections of information that are relevant for recovery are displayed in Example 6-22.

The log point X'18FE0AB8C' was created with ARCHIVE LOG MODE(QUIESCE) command. All recoveries to this log point will guarantee data consistency. Obviously all updates beyond this log point are lost.

The “CHECKPOINT QUEUE” is created by DB2 based on the DSNZPARM value “CHKFREQ”.

*Example 6-22 Sample BSDS listing*

```
ARCHIVE LOG COMMAND HISTORY
                      17:27:59 OCTOBER 20, 2005
   DATE          TIME          RBA         MODE   WAIT   TIME
------------  ----------  ------------   -------  ----   -----
OCT 20, 2005  17:27:43.3  00018FE0AB8C   QUIESCE   NO      5 D
AUG 12, 2005  06:57:12.3  000001C239C0

CHECKPOINT QUEUE
                     17:27:59 OCTOBER 20, 2005
     TIME OF CHECKPOINT       17:27:43 OCTOBER 20, 2005
     BEGIN CHECKPOINT RBA            00018FE0CBBE
     END CHECKPOINT RBA              00018FE123DF
     TIME OF CHECKPOINT       17:00:05 OCTOBER 20, 2005
     BEGIN CHECKPOINT RBA            00018E5DAC57
     END CHECKPOINT RBA              00018E5E157D

ARCHIVE LOG COPY 1 DATA SETS
    START RBA/TIME         END RBA/TIME        DATE    LTIME DATA SET INFORMATIO
```

```
 --------------------  --------------------  --------  ----- -------------------
   0000934E4000          0000956A3FFF      2004.315  13:50 DSN=DB8AU.ARCHLOG1.A0000001
 2004.210  04:49:42.2  2004.251  14:53:26.5                PASSWORD=(NULL) VOL=SBOXEC
UNIT=3390
                                                            CATALOGUED
```

### DSN1LOGP utility

DSN1LOGP is a standalone utility that can be used to list the active or archive log to find a point of recovery for a table space (identified by DBID and OBID) and specific unit of recovery. Example 6-23 shows how to extract information from the recovery log when you have the BSDS available. The extraction starts at the log RBA of X'AF000' and ends at the log RBA of X'B3000'. The DSN1LOGP utility identifies the table or index space by the DBID of X'10A' (266 decimal) and the OBID of X'1F' (31 decimal). Refer to *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427, for details about the DSN1LOGP utility.

*Example 6-23 Extracting information from the recovery log with an available BSDS*

```
//STEP1 EXEC PGM=DSN1LOGP
//STEPLIB DD DSN=DB8A8.SDSNLOAD
//SYSPRINT DD SYSOUT=*
//SYSABEND DD SYSOUT=*
//BSDS DD DSN=DB8AU.BSDS01,DISP=SHR
//SYSIN DD *
RBASTART (AF000) RBAEND (B3000) DBID (10A) OBID(1F)
```

### REPORT utility

You can use the REPORT utility to plan for recovery. REPORT provides information necessary for recovering a page set. REPORT displays:

- Recovery information from the SYSIBM.SYSCOPY catalog table
- Log ranges of the table space from the SYSIBM.SYSLGRNX directory
- Archive log data sets from the bootstrap data set
- The names of all members of a table space set

Details about the REPORT utility and examples showing the results obtained when using the RECOVERY option are contained in Part 2 of *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427.

## Recover table spaces

The Recover utility is executed at table space or partition level. It cannot recover individual tables.

Once you have identified the recovery point and table space set to recover, use the RECOVER utility to recover the objects. Example 6-24 has a sample JCL to recover all table spaces in database DSN8D81L to a log point that was created by the QUIESCE utility. The RECOVER of table spaces will place the index spaces in REBUILD-pending status (RBDP). The subsequent REBUILD utility will the rebuild the indexes.

*Example 6-24 Recover to a log point created by QUIESCE utility*

```
//*   RECOVER  UTILITY
//*
//RECOVER  EXEC DSNUPROC,SYSTEM=DB8A,UID=PAOLO,
//         LIB='DB8A8.SDSNLOAD'
//DSNUPROC.SYSUT1 DD DSN=DB8ALDS.SYSUT1,
//             DISP=(MOD,DELETE,CATLG),
//             SPACE=(TRK,(10,5),RLSE),
//             UNIT=SYSDA
//DSNUPROC.SYSIN  DD  *
   LISTDEF  PAOLO
            INCLUDE table space DSN8D81L.*
   RECOVER LIST PAOLO PARALLEL
           TOLOGPOINT X'0001900E13D7'
   REBUILD INDEX LIST PAOLO
```

DB2 inserts a row for each table space that was recovered to the log point. The attribute PIT_RBA contains the RBA value of the recovery point as in Example 6-25 with ICTYPE=P.

*Example 6-25 SYSCOPY listing after recovery to log point*

```
SELECT DBNAME,TSNAME,ICTYPE,HEX(START_RBA) AS START_RBA, HEX(PIT_RBA) AS PIT_RBA
FROM SYSIBM.SYSCOPY WHERE TSNAME = 'DSN8S81L' AND DBNAME = 'DSN8D81L' ;

DBNAME   TSNAME   ICTYPE START_RBA    PIT_RBA
*        *        *      *            *
-------- -------- ------ ------------ ------------
DSN8D81L DSN8S81L P      000190100701 0001900E13D7
DSN8D81L DSN8S81L Q      0001900E13D7 000000000000
DSN8D81L DSN8S81L F      0001900D23D6 000000000000
```

When you recover table spaces to a prior point of consistency, you need to consider how partitioned table spaces, segmented table spaces, LOB tables spaces, and table space sets can restrict recovery.

### *Recovering partitioned table spaces*

You cannot recover a table space to a point in time prior to rotating partitions. After you rotate a partition, you cannot recover the contents of that partition to a point in time prior to the ROTATE.

If you recover to a point in time prior to the addition of a partition, DB2 cannot roll back the definition of the partition. In such a recovery, DB2 clears all data from the partition, and the partition remains part of the database.

If you recover a table space partition to a point in time before the table space partitions were rebalanced, you must include all partitions that are affected by that rebalance in your recovery list.

Refer to 3.13, "Partition Management", *DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More*, SG24-6079, for adding and rotating partitions in DB2 V8.

### *Recovering segmented table spaces*

When data is restored to a prior point in time on a segmented table space, information in the current DBD for the table space might not match the restored table space. If you use the DB2 RECOVER utility, the database descriptor (DBD) is updated dynamically to match the restored table space on the next non-index access of the table. The table space must be in write access mode.

If you use a method outside of DB2's control, such as DSN1COPY, to restore a table space to a prior point in time, run the REPAIR utility with the LEVELID option to force DB2 to accept the down-level data. Then, run the REORG utility on the table space to correct the DBD.

### *Recovering LOB table spaces*

When you recover tables with LOB columns, recover the entire set of objects, including the base table space, the LOB table spaces, and index spaces for the auxiliary indexes. If you use the RECOVER utility to recover a LOB table space to a prior point of consistency, RECOVER might place the table space in a pending status.

### *Compressed table spaces*

The REORG and LOAD utility create the compression and decompression dictionary when the utilities are run without the KEEPDICTIONARY option. These dictionary pages are stored in the first 16 pages of the table space. The dictionary is loaded in the virtual buffer pool and the data manager address when the table space is opened.

All log records for the table space are also compressed using the same dictionary. If a REORG is run without the KEEPDICTIONARY option, then a new dictionary is created by REORG and activated after the switch phase. All log records prior to the switch phase are invalid.

In Figure 6-15, REORG was started for table space TS1 at time T2 with SHRLEVEL CHANGE and without the KEEPDICTIONARY option. REORG uses the shadow data set for its reorganization and it also builds the new dictionary. Between time T1 and T4, all update logs for TS1 are compressed with “old” dictionary D1. When REORG completes the switch phase, the new compression dictionary D2 is activated. A new inline image copy IM2 is also created by REORG. All subsequent update logs to TS1 are compressed using dictionary D2.

At time T5, a recovery of TS1 to point in time T3 is initiated. Recovery of table space TS1 to point in time T3 will restore image copy IM1 and apply all logs to T3. The decompression of the logs is possible since the dictionary D1 was restored from image copy IM1. All logs beyond time T4 are invalid.

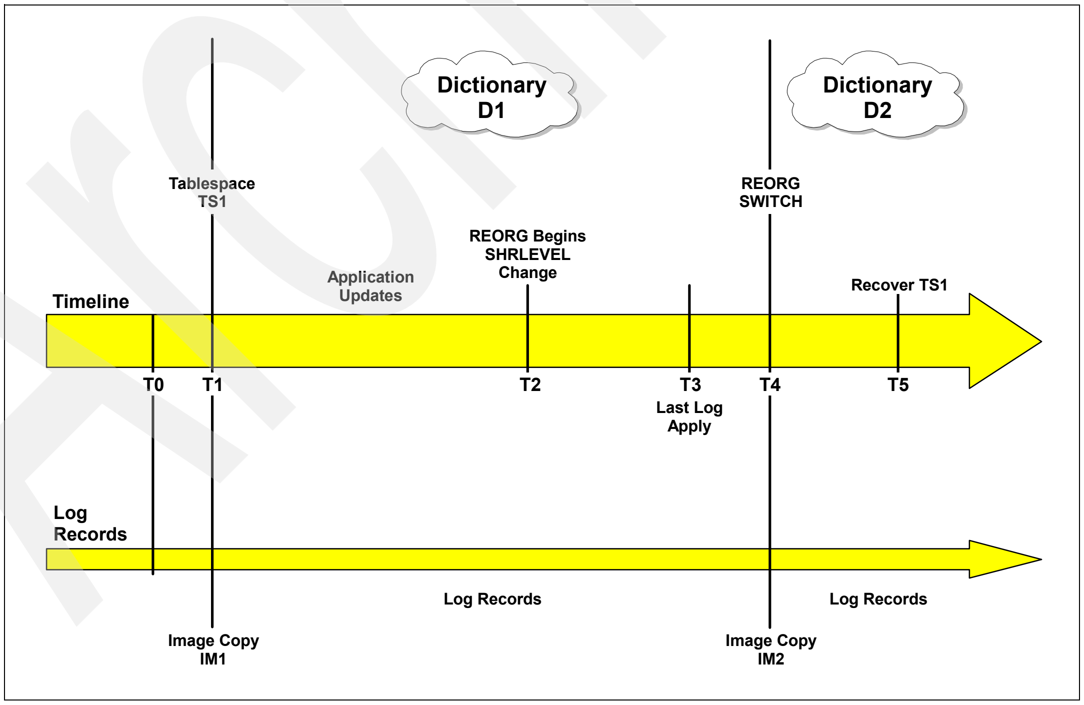


*Figure 6-15 Compressed table space and log records*

### *Recovering tables that contain identify columns*

The column attribute AS IDENTITY was introduced in the DB2 V6 refresh through APAR PQ30652 and was delivered as part of DB2 V7. DB2 V8 enhances identity columns by extending the ALTER COLUMN clause of the ALTER TABLE SQL statement to include the identity column specification. In addition, there is a close tie between the enhancements to identity columns and sequences. Refer to 4.12, "Identity column enhancements", *DB2 UDB for z/OS Version 8: Everything You Ever Wanted thing to Know, ... and More*, SG24-6079.

When recovering a table that has an identity column to a point in time, you can create a gap in the identify column values. When you insert a row after this recovery, DB2 produces an identity value for the row as if all previously added rows still existed.

Consider the following example.

- Create EMPL table.

```
CREATE TABLE EMPL
    (EMPNO       INTEGER GENERATED ALWAYS AS IDENTITY,
     EMPNAME     CHAR(20),
     CITY        CHAR(20) NOT NULL DEFAULT 'KANSAS CITY',
     SALARY      DECIMAL(9,2))
   IN DSNDB04.RAMATEST ;
```

- Insert three rows into the table.

```
INSERT INTO EMPL(EMPNAME, SALARY) VALUES ('Suresh Sane',10000.00) ;
INSERT INTO EMPL(EMPNAME, SALARY) VALUES ('Rama Naidoo',10000.00) ;
INSERT INTO EMPL(EMPNAME, SALARY) VALUES ('Paolo Bruni',10000.00) ;
```

- A SELECT * from EMPL will list three rows.

```
EMPNO  EMPNAME              CITY                       SALARY
---+---------+---------+---------+---------+---------+---------+
    1  SURESH SANE          KANSAS CITY              10000.00
    2  RAMA NAIDOO          KANSAS CITY              10000.00
    3  PAOLO BRUNI          KANSAS CITY              10000.00
```

- Next take an image copy with SHRLEVEL REFERENCE.
- Insert three more rows into EMPL table to get EMPNO 4, 5, and 6.

```
EMPNO  EMPNAME              CITY                       SALARY
---+---------+---------+---------+---------+---------+---------+----
    1  SURESH SANE          KANSAS CITY              10000.00
    2  RAMA NAIDOO          KANSAS CITY              10000.00
    3  PAOLO BRUNI          KANSAS CITY              10000.00
    4  NEVILLE HARLOCK      KANSAS CITY              10000.00
    5  TONY GEORGE          MELBOURNE                20000.00
    6  ELENA IDEL           SYDNEY                   20000.00
```

- Recover the table space to last image copy (TOLASTCOPY).
- Add the same three rows as above and list the rows.

```
EMPNO  EMPNAME              CITY                       SALARY
---+---------+---------+---------+---------+---------+---------+----
    1  SURESH SANE          KANSAS CITY              10000.00
    2  RAMA NAIDOO          KANSAS CITY              10000.00
    3  PAOLO BRUNI          KANSAS CITY              10000.00
    7  NEVILLE HARLOCK      KANSAS CITY              10000.00
    8  TONY GEORGE          MELBOURNE                20000.00
    9  ELENA IDEL           SYDNEY                   20000.00
```

- Note that there is a gap in EMPNO (4, 5, and 6 missing). DB2 created the next sequence number for insert at value 7, as if rows 4, 5, and 6 are still in existence.

- To prevent a gap in identity column values, starting with DB2 V8, you can use the following ALTER TABLE statement to modify the attributes of the identity column before you insert rows after the recovery:

```
ALTER TABLE EMPL
        ALTER COLUMN EMPNO
        RESTART WITH 4    ;
  COMMIT ;
```

- An insert of three more rows starts the EMPNO at value “4”.

```
---+---------+---------+---------+---------+---------+---------+-
EMPNO  EMPNAME              CITY                       SALARY
---+---------+---------+---------+---------+---------+---------+-
    1  NEVILLE HARLOCK      KANSAS CITY              10000.00
    2  TONY GEORGE          MELBOURNE                20000.00
    3  ELENA IDEL           SYDNEY                   20000.00
    4  SURESH SANE          KANSAS CITY              10000.00
    5  PAOLO BRUNI          SAN JOSE                 20000.00
    6  QUYNH NGUYEN         SYDNEY                   20000.00
```

**Tip:** To determine the last value in an identity column, issue the MAX column function for ascending sequences of identity column values or the MIN column function for descending sequences of identity column values. This method works only if the identity column does not use CYCLE.

## Active and archive logs

The number of active logs and the size of active logs can influence the elapsed time of recovery. The active logs are always online and available to DB2. Archive logs which are written to tape will require a tape mount. Generally reading log records from a tape media is slower than a disk media. Archive logs that are written to disk may be migrated by HSM or other products to conserve disk space. These migrated datasets need to be recalled prior to accessing the log data.

The order of recovery from logs is from active logs followed by archive logs. If the active logs are structured to hold a significant amount of log data (24, 48, or even 72 hours), then all recovery and log applies will be from the active logs. DB2 V8 has increased the MAX number of active logs for a DB2 subsystem from 31 to 93.

The active logs (also the archive logs) are a common resource for the whole DB2 subsystem. This implies that it is difficult to isolate all log records for a particular set of table spaces. The checkpoint taken by DB2 and recorded in BSDS based on the DSNZPARM parameter CHKFREQ may not be suitable to recover a set of table spaces to a log point. Similarly, the QUIESCE point recorded by the ARCHIVE LOG MODE(QUIESCE) is for the whole DB2 subsystem. Recovering a set of table spaces to the archive log quiesce point will retain data integrity.

## Recovering indexes

As in Example 6-24 on page 229, the recovery of table spaces to a log point places the associated indexes in RDBP. The indexes must be recovered to the same consistency point.

- If the COPY YES is set for indexes and an image copy exist for the index, then use the RECOVER utility.
- If indexes do not have an image copy, then use REBUILD INDEX to recreate the indexes after the corresponding table space is recovered.

### *Rebuilding indexes on altered tables*

When an index is altered with the following statements:

- ALTER INDEX PADDED
- ALTER INDEX NOT PADDED
- ALTER TABLE SET DATA TYPE on an indexed column for numeric data type changes
- ALTER TABLE ADD COLUMN and ALTER INDEX ADD COLUMN that are not issued in the same commit scope

the index is placed in RDBP pending status. Indexes in REBUILD-pending status cannot be recovered using the RECOVER utility. The indexes must be rebuilt with the REBUILD utility to align them to the same point of consistency as the recovered table spaces.

### *Recovering indexes on tables in partitioned table spaces*

The data partioned secondary index can be image-copied and recovered either entirely or by partition. The following applies:

- If the COPY is at partition level, then RECOVER only the index partition.
- If the COPY is at partition level and a RECOVER of the entire index is attempted, an error occurs.
- If the COPY is at index level, then the RECOVER of the individual partition or the entire index is possible.

You cannot recover an index space to a point in time prior to rotating partitions. After you rotate a partition, you cannot recover the contents of that partition to a point in time before the rotation.

If you recover to a point in time prior to the addition of a partition, DB2 cannot roll back the addition of that partition. In this type of a recovery, DB2 clears all data from the partition, and it remains part of the database.

## 6.10.3 Restore data to previous point in time

Running RECOVER utility with TOCOPY, TOLASTCOPY, TOLASTFULLCOPY, and TOLOGPOINT recovers the table space to a specific point in time.

RECOVER with TOCOPY requires the data set name of the image copy. The data set must exist in SYSCOPY. If the data set is cataloged, then DB2 can identify and access the data set for recovery from the ICF catalog. If the data set is not cataloged, then the volume serial can be identified by using the TOVOLUME volser. RECOVER with TOLASTCOPY or TOLASTFULLCOPY will scan catalog table SYSCOPY and select the data set for the latest image copy. In all cases, if the image copy data set is not usable or not found, then RECOVER will fall back to the previous image copy and apply log records up to the point at which the specified image copy was taken.

```
RECOVER TABLESPACE DSN8D81L.DSN8S81L DSNUM ALL
TOCOPY DB8ALDS.DSN8D81L.DSN8S81L.D2005293
TOVOLUME CATALOG
```

RECOVER with TOLOGPOINT restores the most recent full image copy and the most recent set of incremental copies that occur before the specified log point. The log and logged changes are applied up to, and including, the record that contains the log point. If no full image copy exists before the chosen log point, recovery is attempted entirely from the log. The log is applied from the log point at which the page set was created or the last LOAD LOG

YES or REORG TABLESPACE utility was run to a log point that was specified. The log range records must exist in SYSLGRNX for the log apply to succeed.

TOCOPY and TOLOGPOINT are viable alternatives in many situations in which recovery to the current point in time is impossible or is not desirable. To make these options work best, take periodic quiesce points at points of consistency that are appropriate to your applications. Refer to 6.10.1, “Copying the data” on page 222, for discussion about copying the data to a point of consistency.

**Tip:** Use the TOLOGPOINT keyword instead of the TORBA keyword. Although DB2 still supports the TORBA option, the TOLOGPOINT option supports both data sharing and non-data sharing environments and is used for both of these environments.

### Ensuring consistency

RECOVER TOLOGPOINT and RECOVER TOCOPY can be used on a single:

- Partition of a partitioned table space
- Partition of a partitioning index space
- Data set of a simple table space

All page sets must be restored to the same level; otherwise, the data is inconsistent. The corresponding indexes must all be recovered to the same log point or REBUILD from the table space. If the log point is a quiesce point or a common SHRLEVEL REFERENCE copy point, then data integrity is ensured for the table space set. Refer to 6.10, “Preparing to recover to a point of consistency” on page 219, for identifying a point of consistency.

Point in time recovery can cause table spaces to be placed in CHECK-pending status if they have table check constraints or referential constraints defined on them. When recovering tables that are involved in a referential constraint, you should recover all the table spaces that hold all the tables that are part of the referential structure associated with the constraint. This is the *table space set.* To avoid setting CHECK-pending status, you must perform both of the following tasks:

- Recover the table space set to a quiesce point. If you do not recover each table space of the table space set to the same quiesce point, and if any of the table spaces are part of a RI structure:
  - All dependent table spaces that are recovered are placed in CHECK-pending status with the scope of the whole table space.
  - All table spaces that are dependent on the table spaces that are recovered are placed in CHECK-pending status with the scope of the specific dependent tables.
- Establish a quiesce point or take an image copy after you add check constraints or referential constraints to a table.

If you recover each table space of a table space set to the same quiesce point, but referential constraints were defined after the quiesce point, the CHECK-pending status is set for the table space containing the table with the referential constraint.

## 6.10.4 New utilities in DB2 V8 for online backup and point in time recovery

For detailed information about disaster recovery in general and point in time recovery, refer to *Disaster Recovery with DB2 UDB for z/OS,* SG24-6370. In this section, we provide only a quick overview.

DB2 V8 provides an easier and less disruptive way for fast volume-level backup and recovery. This utility greatly simplifies backing up systems, such as SAP, in which the high number of database objects in use, as well as recovery requirements, makes volume-based backups the most efficient option.

The total solution provided by this utility is dependent on the DFSMShsm™ in z/OS V1.5 and a disk system that provides hardware-assisted volume-level copy. In order for the backup to be registered, the disk system has to write to the DFSMShsm API. IBM ESS disk systems using FlashCopy take full advantage of this solution. Even so, it is possible to take advantage of some of its features with other disk models and fast copy solutions.

One of the challenges of the current online volume backup solutions is the need for coordination between DB2, using the SET LOG SUSPEND command, and the mechanism for triggering the FlashCopy. In DB2 V7, the physical copy is not registered in DB2, hence, it is out of DB2's control for later use as a recovery point or as a registered copy to be used in a point in time recovery. The procedure for obtaining a system-level copy using FlashCopy must ensure that all DB2 subsystem volumes are included and data consistency can be enforced. The procedure for recovering a subsystem using the flashcopied backup must ensure that:

- All volumes are correctly restored.
- There is a process to identify which DB2 objects (pagesets) require recovery.
- There is a process for generating the recovery jobs.

In DB2 V8, the utilities BACKUP and RESTORE have been developed integrating DB2 and the fast volume copy capability. Now system-level backups using the fast volume-level copy are managed by DB2 and DFSMShsm, which work together to support a system-level point in time recovery. Thus, suspending the DB2 log will no longer be necessary.

We now provide a brief description of the new DB2 utilities with special considerations for data sharing:

- BACKUP SYSTEM
- RESTORE SYSTEM

### BACKUP SYSTEM

This utility obtains a complete copy of a DB2 system. There are two kinds: DATAONLY and FULL. DATAONLY is used to obtain a FlashCopy of the data objects, where FULL is used to flashcopy data objects, DB2 logs, and BSDSs. FULL is designed to restore the whole DB2 environment to the backup copy.

This utility requires z/OS V1.5 support for HSM COPYPOOLS. A COPYPOOL is a new SMS construct representing a set of SMS storage groups that are copied together in a single DFSMShsm invocation. A COPYPOOL is limited to 256 SMS storage groups and can be defined with a version attribute in order to keep up to 15 backup versions.

Each DB2 system or data sharing group has two SMS COPYPOOLs:

- DATA COPYPOOL (DSN$location_name$DB)
- LOG COPYPOOL (DSN$location_name$LG)

LOG COPYPOOL is not needed for BACKUP SYSTEM DATAONLY.

Each SMS storage group, including a COPYPOOL, must have a dedicated new type of storage group: *copypool backup storage group*, which is used to hold volume copies of disk defined in the COPYPOOL.

When the BACKUP SYSTEM utility is used, DB2:

- Suspends 32 KB writes for objects created before NFM.
- Suspends data set creation, deletion, rename, and extensions operations.

- Prevents data set from being pseudo-closed.
- Records the Recover Based Log Point (RBLP) in DBD01 with a logscan start point.
- Invokes DFSMShsm to take a FlashCopy of ‘DB’ COPYPOOL.
- Uses DSS COPY to copy the volumes in the COPYPOOL.
- A volume copy of the COPYPOOL is registered by DB2 in BSDSs and by HSM with an associated RBLP inserted in DBD01.
- For BACKUP SYSTEM FULL, DB2 invokes DFSMShsm to take a FlashCopy of the “LG” COPYPOOL.
- The Copy is registered in the BSDS of the submitting member.
- Resumes the quiesced activities.

## RESTORE SYSTEM

This utility is needed to recover the system to an arbitrary point in time.

It may use, as input, copies from BACKUP SYSTEM FULL or DATAONLY. The recovery does not require the restore of the log backup copies; it uses the DATAONLY option of this utility and, if needed, applies the log records from the stopped system.

The RESTORE of a SYSTEM FULL is useful for cloning flashcopies of the whole subsystem, data, and logs.

Recovery executes with two phases:

- RESTORE phase: Recover the data volumes from the latest backup version prior to the arbitrary point in time.
- LOG APPLY phase: Apply log records to recover the database to that arbitrary point in time.

When the recovery point in time, and therefore a target LRSN, has been determined, the first thing to do is to deallocate the data sharing group structures in the coupling facility. Afterward, the truncation target LRSN must be established on all active members using CRESTART CREATE SYSPITR=end-lrsn.

- All members must be restarted.
- From one of the members, issue RESTORE SYSTEM.

When a single DB2 or a data sharing member is conditionally restarted using SYSPITR, the system enters into a System Recover Pending mode. In this status, DB2 automatically uses DEFER ALL, FORWARD = NO, and ACCESS(MAINT) when restarting. DB2 establishes consistency during restart by using the log and recreating the data sharing group coupling facility structures.

In order to reset the System Recover Pending status, you must first submit the RESTORE SYSTEM utility, then restart DB2.

When it is issued, DB2 asks HSM for the COPYPOOL version that was taken by BACKUP SYSTEM prior to the specified point in time recovery point. Subsequently, DB2 performs the log apply function.

During log apply phase, DB2:

- Reads the DBD01 header page to retrieve RBLP and the log scan starting point.
- Applies log recovering objects in parallel and using Fast log apply (FLA).

- Detects creates, drops, extends, and LOG NO events. Objects are marked in RECP or RBDP status.

RESTORE SYSTEM can handle fast volume copies obtained with SET LOG SUSPEND and without BACKUP SYSTEM starting in z/OS V1.3, but you need to manually restore the backups. In this case, the LOGONLY option must be specified.

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 240. Note that some of the documents referenced here may be available in softcopy only.

- *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*, SG24-7134
- *DB2 UDB for z/OS Version 8 Performance Topics*, SG24-6465
- *Disk storage access with DB2 for z/OS*, REDP-4187
- *DB2 UDB for z/OS Version 8: Everything You Ever Wanted to Know, ... and More,* SG24-6079
- *Disk storage access with DB2 for z/OS,* REDP-4187-00
- *DB2 for z/OS and OS/390 Version 7 Performance Topics,* SG24-6129
- *DB2 for z/OS Application Programming Topics*, SG24-6300
- *DB2 for z/OS Stored Procedures: Through the CALL and Beyond,* SG24-7083
- *WebSphere Information Integrator Q Replication: Fast Track Implementation Scenarios*, SG24-6487
- *DB2 UDB for OS/390 and Continuous Availability*, SG24-5486
- *Disaster Recovery with DB2 UDB for z/OS,* SG24-6370
- *IBM System z9 109 Configuration Setup*, SG24-7203
- *IBM System z9 109 Technical Guide*, SG24-7124

## Other publications

These publications are also relevant as further information sources:

- *The Official Introduction to DB2 UDB for z/OS* by Susan Graziano Sloan, IBM Press, Prentice Hall
- *Transforming enterprise information integrity*, white paper G510-3831-00 by Susanne Ruschka-Taylor, available at:

  `http://www.ibm.com/services/us/bcs/pdf/g510-3831-transforming-enterprise-information-integrity.pdf`
- *DB2 UDB for z/OS Version 8 Administration Guide,* SC18-7413-03
- *DB2 UDB for z/OS Version 8 Application Programming and SQL Guide,* SC18-7415-03
- *DB2 UDB for z/OS Version 8 Application Programming Guide and Reference for Java,* SC18-7414-02
- *DB2 UDB for z/OS Version 8 Command Reference,* SC18-7416-03
- *DB2 UDB for z/OS Version 8 Data Sharing: Planning and Administration,* SC18-7417-03

- *DB2 UDB for z/OS Version 8 Installation Guide,* GC18-7418-04
- *DB2 UDB for z/OS Version 8 Codes,* GC18-9603-01
- *DB2 UDB for z/OS Version 8 Messages,* GC18-9602-01
- *DB2 UDB for z/OS Version 8 SQL Reference,* SC18-7426-03
- *DB2 UDB for z/OS Version 8 Utility Guide and Reference,* SC18-7427-03
- *z/OS V1R6 Language Environment Debugging Guide*, GA22-7560-05
- *z/OS V1R6 Language Environment Customization Guide,* SA22-7564-06
- *DB2 UDB for z/OS Version 8 Diagnostics Guide and Reference*, LY37-3201-02

  This is licensed material of IBM.

## Online resources

These Web sites and URLs are also relevant as further information sources:

- DB2 UDB for z/OS Version 8

  `http://www.ibm.com/software/data/db2/zos/db2zosv8.html`
- The DB2 Information Management Software Information Center

  `http://publib.boulder.ibm.com/infocenter/dzichelp/index.jsp`

## How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

**ibm.com**/redbooks

## Help from IBM

IBM Support and downloads

**ibm.com**/support

IBM Global Services

**ibm.com**/services

# Abbreviations and acronyms

| | |
|---|---|
| **AC** | autonomic computing |
| **ACS** | automatic class selection |
| **AIX®** | Advanced Interactive eXecutive from IBM |
| **APAR** | authorized program analysis report |
| **API** | application programming interface |
| **AR** | application requester |
| **ARM** | automatic restart manager |
| **AS** | application server |
| **ASCII** | American National Standard Code for Information Interchange |
| **B2B** | business-to-business |
| **BCDS** | DFSMShsm backup control data set |
| **BCRS** | business continuity recovery services |
| **BI** | Business Intelligence |
| **BLOB** | binary large objects |
| **BPA** | buffer pool analysis |
| **BSDS** | boot strap data set |
| **CBU** | Capacity BackUp |
| **CCA** | channel connection address |
| **CCA** | client configuration assistant |
| **CCP** | collect CPU parallel |
| **CCSID** | coded character set identifier |
| **CCW** | channel control word |
| **CD** | compact disk |
| **CDW** | central data warehouse |
| **CEC** | central electronics complex |
| **CF** | coupling facility |
| **CFCC** | coupling facility control code |
| **CFRM** | coupling facility resource management |
| **CI** | control interval |
| **CICS** | Customer Information Control System |
| **CIDF** | control interval definition field |
| **CLI** | call level interface |
| **CLOB** | character large object |
| **CLP** | command line processor |
| **CMOS** | complementary metal oxide semiconductor |
| **CP** | central processor |
| **CPU** | central processing unit |
| **CRCR** | conditional restart control record |
| **CRD** | collect report data |
| **CRUD** | create, retrieve, update or delete |
| **CSA** | common storage area |
| **CSF** | Integrated Cryptographic Service Facility |
| **CTE** | common table expression |
| **CTT** | created temporary table |
| **CUoD** | Capacity Upgrade on Demand |
| **DAC** | discretionary access control |
| **DASD** | direct access storage device |
| **DB** | database |
| **DB2** | Database 2™ |
| **DB2 PE** | DB2 Performance Expert |
| **DBA** | database administrator |
| **DBAT** | database access thread |
| **DBCLOB** | double-byte character large object |
| **DBCS** | double-byte character set |
| **DBD** | database descriptor |
| **DBID** | database identifier |
| **DBM1** | database master address space |
| **DBRM** | database request module |
| **DCL** | data control language |
| **DDCS** | distributed database connection services |
| **DDF** | distributed data facility |
| **DDL** | data definition language |
| **DDL** | data definition language |
| **DES** | Data Encryption Standard |
| **DLL** | dynamic load library manipulation language |
| **DML** | data manipulation language |
| **DDM** | disk drive module |
| **DNS** | domain name server |
| **DPSI** | data partitioning secondary index |
| **DRDA** | Distributed Relational Data Architecture |
| **DSC** | dynamic statement cache, local or global |

| | |
|---|---|
| **DSNZPARMs** | DB2's system configuration parameters |
| **DSS** | decision support systems |
| **DTT** | declared temporary tables |
| **DWDM** | dense wavelength division multiplexer |
| **DWT** | deferred write threshold |
| **EA** | extended addressability |
| **EAI** | enterprise application integration |
| **EAS** | Enterprise Application Solution |
| **EBCDIC** | extended binary coded decimal interchange code |
| **ECS** | enhanced catalog sharing |
| **ECSA** | extended common storage area |
| **EDM** | environmental descriptor manager |
| **EJB™** | Enterprise JavaBean |
| **ELB** | extended long busy |
| **ENFM** | enable-new-function mode |
| **ERP** | enterprise resource planning |
| **ERP** | error recovery procedure |
| **ESA** | Enterprise Systems Architecture |
| **ESP** | Enterprise Solution Package |
| **ESS** | Enterprise Storage Server® |
| **ETR** | external throughput rate, an elapsed time measure, focuses on system capacity |
| **EWLC** | Entry Workload License Charges |
| **EWLM** | Enterprise Workload Manager |
| **FIFO** | first in first out |
| **FLA** | fast log apply |
| **FTD** | functional track directory |
| **FTP** | File Transfer Program |
| **GB** | gigabyte (1,073,741,824 bytes) |
| **GBP** | group buffer pool |
| **GDPS®** | Geographically Dispersed Parallel Sysplex™ |
| **GLBA** | Gramm-Leach-Bliley Act of 1999 |
| **GRS** | global resource serialization |
| **GUI** | graphical user interface |
| **HALDB** | High Availability Large Databases |
| **HPJ** | high performance Java |
| **HTTP** | Hypertext Transfer Protocol |
| **HW** | hardware |
| **I/O** | input/output |
| **IBM** | International Business Machines Corporation |
| **ICF** | internal coupling facility |
| **ICF** | integrated catalog facility |
| **ICMF** | integrated coupling migration facility |
| **ICSF** | Integrated Cryptographic Service Facility |
| **IDAW** | indirect address word |
| **IDE** | integrated development environments |
| **IFCID** | instrumentation facility component identifier |
| **IFI** | Instrumentation Facility Interface |
| **IFL** | Integrated Facility for Linux |
| **IGS** | IBM Global Services |
| **IMS** | Information Management System |
| **IORP** | I/O Request Priority |
| **IPLA** | IBM Program Licence Agreement |
| **IRD** | Intelligent Resource Director |
| **IRLM** | internal resource lock manager |
| **IRWW** | IBM Relational Warehouse Workload |
| **ISPF** | interactive system productivity facility |
| **ISMF** | Interactive Storage Management Facility) |
| **ISV** | independent software vendor |
| **IT** | information technology |
| **ITR** | internal throughput rate, a processor time measure, focuses on processor capacity |
| **ITSO** | International Technical Support Organization |
| **IVP** | installation verification process |
| **J2EE™** | Java 2 Enterprise Edition |
| **JDBC** | Java Database Connectivity |
| **JFS** | journaled file systems |
| **JNDI** | Java Naming and Directory Interface |
| **JTA** | Java Transaction API |
| **JTS** | Java Transaction Service |
| **JVM™** | Java Virtual Machine |
| **KB** | kilobyte (1,024 bytes) |
| **LCU** | Logical Control Unit |
| **LDAP** | Lightweight Directory Access Protocol |
| **LOB** | large object |
| **LPAR** | logical partition |
| **LPL** | logical page list |

| | |
|---|---|
| **LRECL** | logical record length |
| **LRSN** | log record sequence number |
| **LRU** | least recently used |
| **LSS** | logical subsystem |
| **LUW** | logical unit of work |
| **LVM** | logical volume manager |
| **MAC** | mandatory access control |
| **MB** | megabyte (1,048,576 bytes) |
| **MBps** | megabytes per second |
| **MIDAW** | modified indirect address word |
| **MLS** | multi-level security |
| **MQT** | materialized query table |
| **MTBF** | mean time between failures |
| **MVS** | Multiple Virtual Storage |
| **NALC** | New Application License Charge |
| **NFM** | new-function mode |
| **NFS** | Network File System |
| **NPI** | non-partitioning index |
| **NPSI** | nonpartitioned secondary index |
| **NVS** | non volatile storage |
| **ODB** | object descriptor in DBD |
| **ODBC** | Open Database Connectivity |
| **ODS** | Operational Data Store |
| **OLE** | Object Link Embedded |
| **OLTP** | online transaction processing |
| **OP** | Online performance |
| **OS/390** | Operating System/390® |
| **OSC** | optimizer service center |
| **PAV** | parallel access volume |
| **PCICA** | Peripheral Component Interface Cryptographic Accelerator |
| **PCICC** | PCI Cryptographic Coprocessor |
| **PDS** | partitioned data set |
| **PIB** | parallel index build |
| **PPRC** | Peer-to-Peer Remote Copy |
| **PR/SM** | Processor Resource/System Manager |
| **PSID** | pageset identifier |
| **PSP** | preventive service planning |
| **PTF** | program temporary fix |
| **PUNC** | possibly uncommitted |
| **PWH** | Performance Warehouse |
| **QA** | Quality Assurance |
| **QMF** | Query Management Facility |
| **QoS** | Quality of Service |
| **QPP** | Quality Partnership Program |
| **RACF** | Resource Access Control Facility |
| **RAS** | reliability, availability and serviceability |
| **RBA** | relative byte address |
| **RBLP** | recovery base log point |
| **RDBMS** | relational database management system |
| **RDS** | relational data system |
| **RECFM** | record format |
| **RI** | Referential Integrity |
| **RID** | record identifier |
| **RMM** | removable (tape) media manager |
| **ROI** | return on investment |
| **RPO** | recovery point objective |
| **RR** | repeatable read |
| **RRS** | resource recovery services |
| **RRSAF** | resource recovery services attach facility |
| **RS** | read stability |
| **RTO** | recovery time objective |
| **SAN** | storage area networks |
| **SBCS** | store single byte character set |
| **SCUBA** | self contained underwater breathing apparatus |
| **SDM** | System Data Mover |
| **SDP** | Software Development Platform |
| **SLA** | service-level agreement |
| **SMIT** | System Management Interface Tool |
| **SOA** | service-oriented architecture |
| **SOAP** | Simple Object Access Protocol |
| **SPL** | selective partition locking |
| **SQL** | Structured Query Language |
| **SQLJ** | Structured Query Language for Java |
| **SRM** | Service Request Manager |
| **SSL** | Secure Sockets Layer |
| **SU** | Service Unit |
| **TCO** | total cost of ownership |
| **TPF** | Transaction Processing Facility |
| **UA** | Unit Addresses |
| **UCB** | Unit Control Block |
| **UDB** | Universal Database |
| **UDF** | user-defined functions |

| | |
|---|---|
| **UDT** | user-defined data types |
| **UOW** | unit of work |
| **UR** | unit of recovery |
| **USS** | UNIX System Services |
| **vCF** | virtual coupling facility |
| **VIPA** | Virtual IP Addressing |
| **VLDB** | very large database |
| **VM** | virtual machine |
| **VVDS** | VSAM volume data set |
| **WEPR** | write error page range |

# Index

IBM

Redbooks

# Data Integrity with DB2 for z/OS

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

# Data Integrity with DB2 for z/OS

**Assert information integrity by exploiting DB2 functions**

**Understand constraints, referential integrity, and triggers**

**Review recovery-related functions**

DB2 provides functions to guarantee integrity at the system level and at the application level.
From the system point of view, DB2's integration with zSeries and disk storage architecture is the cornerstone for data integrity. Logging functionality and COPY and RECOVER utilities are the building blocks for bringing the table space back to a current or consistent status in case of hardware or software failures or when application events need to be rerun.
From the application point of view, DB2 supports locking and commit at the transaction level, and general data integrity (at entity and semantic level), and a set of referential constraint rules for each parent/dependent table relationship. The tables linked by referential integrity are recognized during the execution of the QUIESCE utility. Other logical relations across tables, necessary to support business rules, are implemented via constraints, triggers, user defined functions, and user defined tables. Informational constraints also exist, they are not enforced by the database manager, they are used to improve query performance.
In this IBM Redbook, we briefly describe the integration of DB2 for z/OS with System z architecture, we then explore the data integrity options and utilize the standard recovery functions for application-related issues.

SG24-7111-00 ISBN 0738495547