System i

Database Commitment control

Version 5 Release 4

System i

Database
Commitment control

Version 5 Release 4

Note

Before using this information and the product it supports, read the information in “Notices,” on page 113.

Fourth Edition (February 2006)

This edition applies to version 5, release 4, modification 0 of IBM i5/OS (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Commitment control

Commitment control is a function that ensures data integrity. It defines and processes a group of changes to resources, such as database files or tables, as a transaction.

Commitment control ensures that either the entire group of individual changes occur on all systems that participate or that none of the changes occur. IBM® DB2 Universal Database™ for iSeries™ uses the commitment control function to commit and rollback database transactions that are running with an isolation level other than *NONE (No Commit).

You can use commitment control to design an application so that the system can restart the application if a job, an activation group within a job, or the system ends abnormally. With commitment control, you can have assurance that when the application starts again, no partial updates are in the database due to incomplete transactions from a prior failure.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 111.

What's new for V5R4

This topic highlights the changes made to this topic collection for V5R4.

What's new for Commitment control:

- Enhancements to XA functionality (see XA transaction support for commitment control for detailed information) :
  - Remote relational database (RDB) support for XA transactions with transaction scoped locks
  - Optional lock sharing for loosely coupled transaction branches
  - Choice of XA thread of control
  - Support for traditional database access within XA transactions using distributed data management (DDM)
- Soft commit

How to see what's new or changed

To help you see where technical changes have been made, this information uses:

- The » image to mark where new or changed information begins.
- The « image to mark where new or changed information ends.

To find other information about what's new or changed this release, see the Memo to users.

Printable PDF

Use this to view and print a PDF of this information.

To view or download the PDF version of this document, select Commitment control (about 1413 KB).

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).

2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Commitment control concepts

These commitment control concepts help you understand how commitment control works, how it interacts with your system, and how it interacts with other systems in your network.

How commitment control works

Commitment control ensures that either the entire group of individual changes occurs on all systems that participate or that none of the changes occur.

For example, when you transfer funds from a savings to a checking account, more than one change occurs as a group. To you, this transfer seems like a single change. However, more than one change occurs to the database because both savings and checking accounts are updated. To keep both accounts accurate, either all the changes or none of the changes must occur to the checking and savings account.

Commitment control allows you to complete the following tasks:
- Ensure that all changes within a transaction are completed for all resources that are affected.
- Ensure that all changes within a transaction are removed if processing is interrupted.
- Remove changes that are made during a transaction when the application determines that a transaction is in error.

You can also design an application so that commitment control can restart the application if a job, an activation group within a job, or the system ends abnormally. With commitment control, you can have assurance that when the application starts again, no partial updates are in the database due to incomplete transactions from a prior failure.

Transaction

A transaction is a group of individual changes to objects on the system that appears as a single atomic change to the user.

Note: iSeries Navigator uses the term *transaction*, whereas the character-based interface uses the term *Logical Unit of Work (LUW)*. The two terms are interchangeable. This topic, unless specifically referring to the character-based interface, uses the term transaction.

A transaction can be any of the following situations:
- Inquiries in which no database file changes occur.
- Simple transactions that change one database file.
- Complex transactions that changes one or more database files.
- Complex transactions that change one or more database files, but these changes represent only a part of a logical group of transactions.
- Simple or complex transactions that involve database files at more than one location. The database files can be in one of the following situations:
  - On a single remote system.

  - On the local system and one or more remote systems.
  - Assigned to more than one journal on the local system. Each journal can be thought of as a *local location*.
- Simple or complex transactions on the local system that involve objects other than database files.

How commit and rollback operations work

Commit and rollback operations affect changes that are made under commitment control.

The following programming languages and application programming interfaces (APIs) support commit and rollback operations.

Language or API	Commit	Rollback
CL	COMMIT command	ROLLBACK command
IBM Integrated Language Environment® (ILE) RPG	COMIT operation code	ROLBK operation code
ILE COBOL	COMMIT verb	ROLLBACK verb
ILE C	_Rcommit function	_Rrollbck function
PLI	PLICOMMIT subroutine	PLIROLLBACK subroutine
SQL	COMMIT statement	ROLLBACK statement
SQL Call Level Interface (CLI)	SQLTransact() function (It is used to commit and roll back a transaction)	
XA APIs	xa_commit() and db2xa_commit() APIs	xa_rollback() and db2xa_rollback() APIs

Related concepts

SQL call level interface

Database programming

Related information

WebSphere development studio: ILE C/C++ programmer's guide PDF

CL programming

Application programming interfaces

Commit operation

A commit operation makes permanent all changes made under commitment control since the previous commit or rollback operation. The system also releases all locks related to the transaction.

The system performs the following steps when it receives a request to commit:
- The system saves the commit identification, if one is provided, for use at recovery time.
- The system writes records to the file before performing the commit operation if both of the following conditions are true:
  - Records were added to a local or remote database file under commitment control.
  - SEQONLY(*YES) was specified when the file was opened so that blocked I/O feedback is used by the system and a partial block of records exists.

  Otherwise, the I/O feedback area and I/O buffers are not changed.
- The system makes a call to the commit and rollback exit program for each API commitment resource that is present in the commitment definition. If a location has more than one exit program registered, the system calls exit programs for that location in the order that they were registered.

- If any record changes were made to resources assigned to a journal, the system writes a C CM journal entry to every local journal associated with the commitment definition. Sequence of journal entries under commitment control shows the entries that are typically written while a commitment definition is active.
- The system makes permanent object-level changes that are pending.
- The system unlocks record and object locks that were acquired and kept for commitment control purposes. Those resources are made available to other users.
- The system changes information in the commitment definition to show that the current transaction has been ended.

The system must perform all of the previous steps correctly for the commit operation to be successful.

Related concepts

"Commitment definition" on page 5
The commitment definition contains information that pertains to the resources that are being changed under commitment control during a transaction.

"Sequence of journal entries under commitment control" on page 16
This table shows the sequence of entries that are typically written while a commitment definition is active. You can use the Journal entry information finder to get more information about the contents of the journal entries.

Rollback operation

A rollback operation removes all changes made since the previous commit or rollback operation. The system also releases all locks related to the transaction.

The system performs the following steps when it receives a request to roll back:

- The system clears records from the I/O buffer if both of the following conditions are true:
  - If records were added to a local or remote database file under commitment control.
  - If SEQONLY(*YES) was specified when the file was opened so that blocked I/O is used by the system and a partial block of records exists that has not yet been written to the database.

  Otherwise, the I/O feedback area and I/O buffers remain unchanged.
- The system makes a call to the commit or rollback exit program for each API commitment resource that is present in the commitment definition. If a location has more than one exit program registered, the system calls the exit programs for that location in reverse order from the order in which they were registered.
- If a record was deleted from a file, the system adds the record back to the file.
- The system removes any changes to records that have been made during this transaction, and places the original records (the before-images) back into the file.
- If any records were added to the file during this transaction, they remain in the file as deleted records.
- If any record changes were made to resources assigned to a journal during the transaction, the system adds a journal entry of C RB to the journal, indicating that a rollback operation occurred. The journal also contains images of the record changes that were rolled back. Before the rollback operation was requested, the before-images and after-images of changed records were placed in the journal. The system also writes C RB entry to the default journal if any committable resources are assigned to that journal.
- The system positions the open files under commitment control at one of the following positions:
  - The last record accessed in the previous transaction
  - At the open position if no commit operation has been performed for the file using this commitment definition

  This consideration is important if you are doing sequential processing.

- The system does not roll back noncommittable changes for database files. For example, opened files are not closed, and cleared files are not restored. The system does not reopen or reposition any files that were closed during this transaction.
- The system unlocks record locks that were acquired for commitment control purposes and makes those records available to other users.
- The commit identification currently saved by the system remains the same as the commit identification provided with the last commit operation for the same commitment definition.
- The system reverses or rolls back object-level committable changes made during this transaction.
- Object locks that were acquired for commitment control purposes are unlocked and those objects are made available to other users.
- The system establishes the previous commitment boundary as the current commitment boundary.
- The system changes information in the commitment definition to show that the current transaction has been ended.

The system must perform all of the previous steps correctly for the rollback operation to be successful.

Commitment definition

The commitment definition contains information that pertains to the resources that are being changed under commitment control during a transaction.

To create a commitment definition, use the Start Commitment Control (STRCMTCTL) command to start commitment control on your system. Also, DB2 Universal Database (UDB) for iSeries automatically creates a commitment definition when the isolation level is other than No Commit.

The system maintains the commitment control information in the commitment definition as the commitment resources change, until the commitment definition is ended. Each active transaction on the system is represented by a commitment definition. A subsequent transaction can reuse a commitment definition after each commit or rollback of an active transaction.

A commitment definition generally includes the following information:

- The parameters on the STRCMTCTL command.
- The current status of the commitment definition.
- Information about database files and other committable resources that contain changes that are made during the current transaction.

For commitment definitions with job-scoped locks, only the job that starts commitment control knows that commitment definition. No other job knows that commitment definition.

Programs can start and use multiple commitment definitions. Each commitment definition for a job identifies a separate transaction that has committable resources associated with it. These transactions can be committed or rolled back independently from transactions that are associated with other commitment definitions that are started for the job.

Related concepts

"Commit operation" on page 3
A commit operation makes permanent all changes made under commitment control since the previous commit or rollback operation. The system also releases all locks related to the transaction.

"Commitment control and independent disk pools" on page 20
Independent disk pools and independent disk pool groups can each have a separate i5/OS® database. You can use commitment control with these databases.

"Independent disk pool considerations for commitment definitions" on page 20
You must be aware of these considerations for commitment definitions when you use independent disk pools.

Scope for a commitment definition

The *scope* of a commitment definition determines which programs use that commitment definition, and how locks acquired during transactions are scoped.

The interface that starts the commitment definition determines the scope of the commitment definition. There are four possible scopes for a commitment definition, which fall under two general categories:

Commitment definitions with job-scoped locks

- Activation-group-level commitment definition
- Job-level commitment definition
- Explicitly-named commitment definition

Commitment definitions with transaction-scoped locks

- Transaction-scoped commitment definition

Commitment definitions with job-scoped locks can be used only by programs that run in the job that started the commitment definitions. In comparison, more than one job can use commitment definitions with transaction-scoped locks.

Applications typically use either activation-group-level or job-level commitment definitions. These commitment definitions are created either explicitly with the Start Commitment Control (STRCMTCTL) command, or implicitly by the system when an SQL application runs with an isolation level other than *NONE.

Activation-group-level commitment definition

The most common scope is to the activation group. The activation-group-level commitment definition is the default scope when the STRCMTCTL command explicitly starts the commitment definition, or when an SQL application that runs with an isolation level other than No Commit implicitly starts the commitment definition. Only programs that run within that activation group use that commitment definition. Many activation-group-level commitment definitions can be active for a job at one time. However, each activation-group-level commitment definition can be associated only with a single activation group. The programs that run within that activation group can associate their committable changes only with that activation-group-level commitment definition.

When iSeries Navigator, the Work with Commitment Definitions (WRKCMTDFN) command, the Display Job (DSPJOB) command, or the Work with Job (WRKJOB) command displays an activation-group-level commitment definition, these fields display the following information:

- The commitment definition field displays the name of the activation group. It shows the special value *DFTACTGRP to indicate the default activation group.
- The activation group field displays the activation group number.
- The job field displays the job that started the commitment definition.
- The thread field displays *NONE.

Job-level commitment definition

A commitment definition can be scoped to the job only by issuing STRCMTCTL CMTSCOPE (*JOB). Any program running in an activation group that does not have an activation-group-level commitment definition started uses the job-level commitment definition, if it has already been started by another program for the job. You can only start a single job-level commitment definition for a job.

When iSeries Navigator, the Work with Commitment Definitions (WRKCMTDFN) command, the Display Job (DSPJOB) command, or the Work with Job (WRKJOB) command displays a job-level commitment definition, these fields display the following information:

- The commitment definition field displays the special value *JOB.
- The activation group field displays a blank.
- The job field displays the job that started the commitment definition.
- The thread field displays *NONE.

For a given activation group, the programs that run within that activation group can use only a single commitment definition. Therefore, programs that run within an activation group can either use the job-level or the activation-group-level commitment definition, but not both at the same time. In a multi-threaded job that does not use SQL server mode, transactional work for a program is scoped to the appropriate commitment definition with respect to the activation group of the program, regardless of which thread performs it. If multiple threads use the same activation group, they must cooperate to perform the transactional work and ensure that commits and rollbacks occur at the correct time.

Even when the job-level commitment definition is active for the job, a program can still start the activation-group-level commitment definition if no program running within that activation group has performed any commitment control requests or operations for the job-level commitment definition. Otherwise, you must first end the job-level commitment definition before you can start the activation-group-level commitment definition. The following commitment control requests or operations for the job-level commitment definition can prevent the activation-group-level commitment definition from being started:

- Opening (full or shared) a database file under commitment control.
- Using the Add Commitment Resource (QTNADDCR) API to add an API commitment resource.
- Committing a transaction.
- Rolling back a transaction.
- Adding a remote resource under commitment control.
- Using the Change Commitment Options (QTNCHGCO) API to changing commitment options.
- Bringing the commitment definition to a rollback required state using the Rollback Required (QTNRBRQD) API.
- Sending a user journal entry that includes the current commit cycle identifier by using the Send Journal Entry (QJOSJRNE) API with the Include Commit Cycle Identifier parameter.

Likewise, if the programs within an activation group are currently using the activation-group-level commitment definition, the commitment definition must first be ended before programs running within that same activation group can use the job-level commitment definition.

When opening a database file, the open scope for the opened file can be either to the activation group or to the job with one restriction: if a program is opening a file under commitment control and the file is scoped to the job, then the program making the open request must use the job-level commitment definition.

Explicitly-named commitment definition

Explicitly-named commitment definitions are started by the system when it needs to perform its own commitment control transactions without affecting any transactions used by an application. An example of a function that starts these types of commitment definitions is the problem log. An application cannot start explicitly-named commitment definitions.

When iSeries Navigator, the Work with Commitment Definitions (WRKCMTDFN) command, the Display Job (DSPJOB) command, or the Work with Job (WRKJOB) command displays an explicitly-named commitment definition, these fields display the following information:

- The commitment definition field displays the name given to it by the system.
- The activation group field displays a blank.

- The job field displays the job that started the commitment definition.
- The thread field displays *NONE.

Transaction-scoped commitment definitions

Transaction-scoped commitment definitions are started with the XA APIs for Transaction Scoped Locks.

These APIs use commitment control protocols that are thread based or SQL connection based, and not activation group based. In other words, the APIs are used to associate the commitment definition with a particular thread or SQL connection while the transactional work is performed, and to commit or rollback the transactions. The system attaches these commitment definitions to the threads that perform the transactional work, with respect to the API protocols. They can be used by threads in different jobs.

When iSeries Navigator, the Work with Commitment Definitions (WRKCMTDFN) command, the Display Job (DSPJOB) command, or the Work with Job (WRKJOB) command displays a transaction-scoped commitment definition, these fields display the following information:

- The commitment definition field displays the special value *TNSOBJ.
- The activation group field displays a blank.
- The job field displays the job that started the commitment definition. Or, if the commitment definition is currently attached to a thread, the thread's job is displayed.
- The thread field displays the thread to which the commitment definition is attached (or *NONE if the commitment definition is not currently attached to any thread).

Related reference

XA APIs

Commitment definition names

The system gives names to all commitment definitions that are started for a job.

The following table shows various commitment definitions and their associated names for a particular job.

Activation group	Commit scope	Commitment definition name
Any	Job	*JOB
Default activation group	Activation group	*DFTACTGRP
User-named activation group	Activation group	Activation group name (for example, PAYROLL)
System-named activation group	Activation group	Activation group number (for example, 0000000145)
None	Explicitly named	QDIR001 (example of a system-defined commitment definition for system use only). System-defined commitment definition names begin with Q.
None	Transaction	*TNSOBJ

Only IBM Integrated Language Environment (ILE) compiled programs can start commitment control for activation groups other than the default activation group. Therefore, a job can use multiple commitment definitions only if the job is running one or more ILE compiled programs.

Original Program Model (OPM) programs run in the default activation group, and by default use the *DFTACTGRP commitment definition. In a mixed OPM and ILE environment, jobs must use the job-level commitment definition if all committable changes made by all programs are to be committed or rolled back together.

An opened database file scoped to an activation group can be associated with either an activation-group-level or job-level commitment definition. An opened database file scoped to the job can be associated only with the job-level commitment definition. Therefore, any program, OPM or ILE, which opens a database file under commitment control scoped to the job needs to use the job-level commitment definition.

Application programs do not use the commitment definition name to identify a particular commitment definition when making a commitment control request. Commitment definition names are primarily used in messages to identify a particular commitment definition for a job.

For activation-group-level commitment definitions, the system determines which commitment definition to use, based on which activation group the requesting program is running in. This is possible because the programs that run within an activation group at any point in time can only use a single commitment definition.

For transactions with transaction-scoped locks, the XA APIs and the transaction related attributes added to the CLI determine which commitment definition the invoking thread uses.

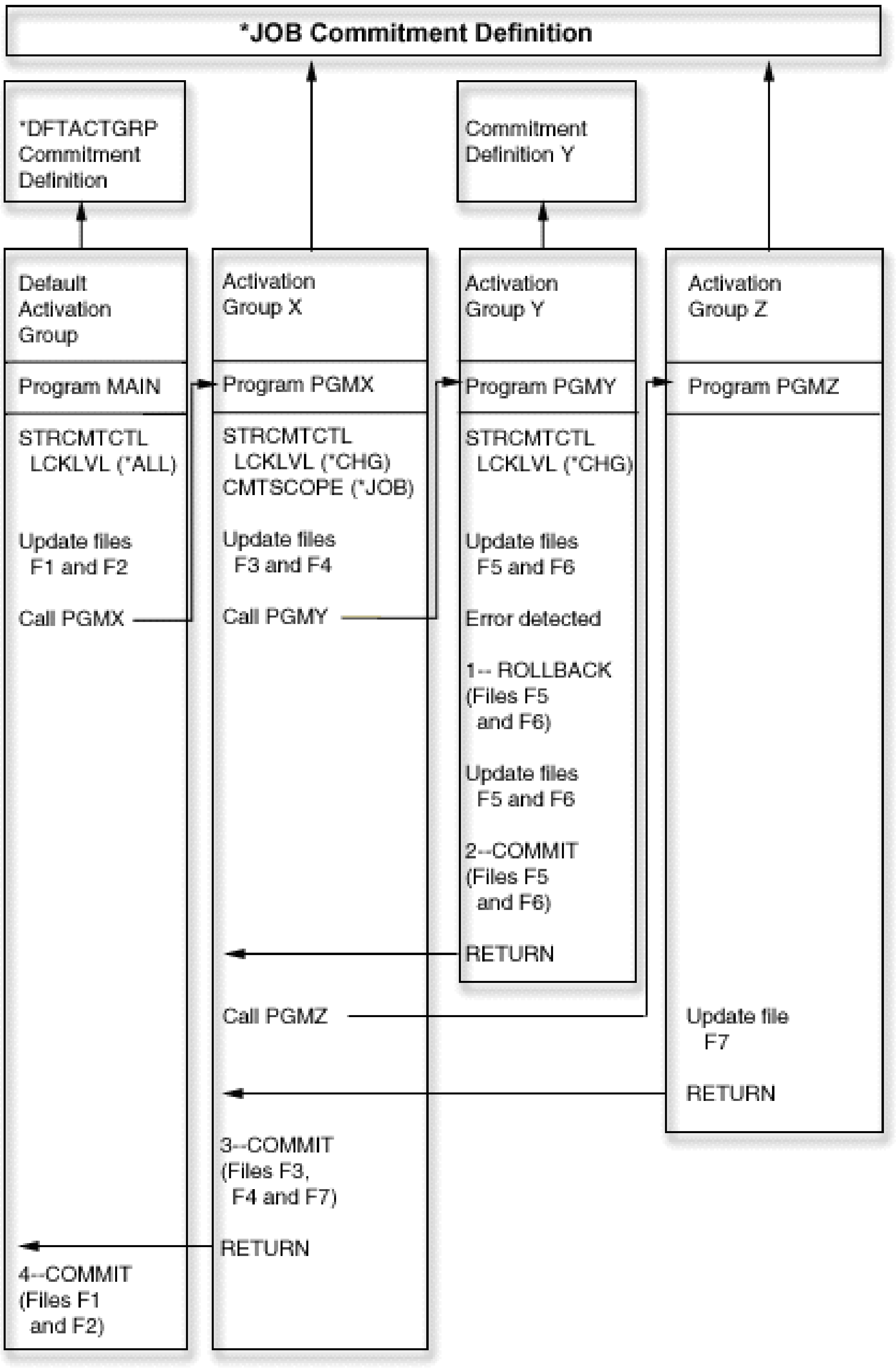
Related information

ILE concepts PDF

Example: Jobs and commitment definitions

This figure shows an example of a job that uses multiple commitment definitions.

The figure indicates which file updates are committed or rolled back at each activation group level. The example assumes that all of the updates that are made to the database files by all of the programs are made under commitment control.


*JOB Commitment Definition
*DFTACTGRP Commitment Definition
Commitment Definition Y
Default Activation Group
Activation Group X
Activation Group Y
Activation Group Z
Program MAIN
Program PGMX
Program PGMY
Program PGMZ
STRCMTCTL LCKLVL (*ALL)
STRCMTCTL LCKLVL (*CHG) CMTSCOPE (*JOB)
STRCMTCTL LCKLVL (*CHG)
Update files F1 and F2
Update files F3 and F4
Update files F5 and F6
Call PGMX
Call PGMY
Error detected
1-- ROLLBACK (Files F5 and F6)
Update files F5 and F6
2--COMMIT (Files F5 and F6)
RETURN
Call PGMZ
Update file F7
RETURN
3--COMMIT (Files F3, F4 and F7)
RETURN
4--COMMIT (Files F1 and F2)

The following table shows how files are committed or rolled back if the scenario in the previous figure changes.

Additional examples of multiple commitment definitions in a job

	Effect on changes to these files:			
Change in scenario	**F1 and F2**	**F3 and F4**	**F5 and F6**	**F7**
PGMX performs a rollback operation instead of a commit operation (3= =COMMIT becomes ROLLBACK).	Still pending	Rolled back	Already committed	Rolled back
PGMZ performs a commit operation before returning to PGMX.	Still pending	Committed by PGMZ	Already committed	Committed
PGMZ attempts to start commitment control specifying `CMTSCOPE(*ACTGRP)` after updating file F7. The attempt fails because changes are pending using the job-level commitment definition.	Still pending	Still pending	Already committed	Still pending
PGMX does not start commitment control and does not open files F3 and F4 with COMMIT(*YES). PGMZ attempts to open file F7 with COMMIT(*YES).	Still pending	Not under commitment control	Already committed	File F7 cannot be opened because no *JOB commitment definition exists (PGMX did not create it).

How commitment control works with objects

When you place an object under commitment control, it becomes a committable resource. It is registered with the commitment definition. It participates in each commit operation and rollback operation that occurs for that commitment definition.

The following topics describe these attributes of a committable resource:
- Resource type
- Location
- Commit protocol
- Access intent

Types of committable resources

This table lists the different types of committable resources, including FILE, Data Definition Language (DDL), distributed data management (DDM), logical unit (LU) 6.2, Distributed Relational Database Architecture™ (DRDA®), API, and TCP.

The table shows the following items:
- The types of committable resources.

- How they are placed under commitment control.
- How they are removed from commitment control.
- Restrictions that apply to the resource type.

Resource type	**How to place it under commitment control**	**How to remove it from commitment control**	**What kinds of changes are committable**	**Restrictions**
FILE- local database files	Opening under commitment control[1]	Closing the file, if no changes are pending. If changes are pending when the file is closed, after performing the next commit or rollback operation.	Record-level changes	No more than 500 000 000 records can be locked for a single transaction[2].
DDL- object-level changes to local SQL tables and SQL collections.	Running SQL under commitment control	Performing a commit or rollback operation after the object-level change.	Object-level changes, such as: • Create SQL package • Create SQL table • Drop SQL table	Only object-level changes made using SQL are under commitment control.
DDM- remote distributed data management (DDM) file	Opening under commitment control. Commitment control support for DDM has more information about commitment control and distributed data management.	Closing the file, if no changes are pending. If changes are pending when the file is closed, after performing the next commit or rollback operation.	Record-level changes	
LU 6.2- protected conversation	Starting the conversation[3]	Ending the conversation		
DRDA- distributed relational database	Using SQL CONNECT statement	Ending the connection		
API- local API commitment resource	Add Commitment Resource (QTNADDCR) API	Remove Commitment Resource (QTNRMVCR) API	The user program determines this. Journal entries might be written by the user program using the Send Journal Entry (QJOSJRNE) API to assist with tracking these changes.	The application must provide an exit program to be called during commit, rollback, or resynchronization operations.
TCP-TCP/IP connection	Using SQL CONNECT statement to an RDB defined to use TCP/IP connections, or opening a DDM file defined with a TCP/IP location	Ending the SQL connection, or closing the DDM file if no changes are pending. If the DDM file is closed with changes pending, the connection is closed after performing the next commit or rollback operation.		

Resource type	How to place it under commitment control	How to remove it from commitment control	What kinds of changes are committable	Restrictions
Notes: [1] For details on how to place a database file under commitment control, see the appropriate language reference manual. Related information for commitment control links to language manuals that you can use. [2] You can use a QAQQINI file to reduce the limit of 500 000 000. See "Managing transaction size" on page 68 for instructions. [3] When a DDM connection is started, the DDM file specifies PTCCNV(*YES), and the DDM file is defined with an SNA remote location; an LU 6.2 resource is added with the DDM resource. When a DRDA connection is started, an LU 6.2 resource is added with the DRDA resource if both of the following conditions are true: • The program is using the distributed unit of work connection protocols. • The connection is to a rational database (RDB) that is defined with an SNA remote location. For more information about starting protected conversions, see APPC Programming .				

Related concepts

Commitment control support for DDM

"Updates to the notify object" on page 59
The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

Related reference

Add Commitment Resource (QTNADDCR) API

Remove Commitment Resource (QTNRMVCR) API

Send Journal Entry (QJOSJRNE) API

Local and remote committable resources

A committable resource can be either a local resource or a remote resource.

Local committable resource

A local committable resource is on the same system as the application. Each journal associated with resources under commitment control can be thought of as a local location. All the resources that are registered without a journal (optionally for both DDL resources and API resources) can be thought of as a separate local location.

If a committable resource is on an independent disk pool and the commitment definition is on a different disk pool, the resource is not considered local.

Remote committable resources

A remote committable resource is on a different system from the application. A remote location exists for each unique conversation to a remote system. A commitment definition might have one or more remote locations on one or more remote systems.

When you place a local resource under commitment control for the system disk pool, or any independent disk pools, you must use Distributed Relational Database Architecture (DRDA) to access resources under commitment control in any other independent disk pools.

The following table shows the types of committable resources and their locations.

Resource type	Location
API	Local
DDL	Local
DDM	Remote
DRDA	Local or remote
FILE	Local
LU62	Remote
TCP	Remote

Related concepts

"Commitment control and independent disk pools" on page 20
Independent disk pools and independent disk pool groups can each have a separate i5/OS database. You can use commitment control with these databases.

Access intent of a committable resource

The access intent determines how the resources participate together in a transaction.

When a resource is placed under commitment control, the resource manager indicates how the resource is accessed:

- Update
- Read-only
- Undetermined

The following table shows what access intents are possible for a particular type of resource and how the system determines the access intent for a resource when it is registered.

Resource type	Possible access intents	How the access intent is determined
FILE	Update, read-only	Based on how the file was opened
DDL	Update	Always update
API	Update	Always update
DDM	Update, read-only	Based on how the file was opened
LU62	Undetermined	Always undetermined
DRDA	Update, read-only, undetermined	For DRDA Level 1, the access intent is update if no other remote resources are registered. Otherwise, the access intent is read-only. For DRDA Level 2, the access intent is always undetermined.
TCP	Undetermined	Always undetermined

The access intent of resources that are already registered determines whether a new resource can be registered. The following rules apply:

- A one-phase resource whose access intent is update cannot be registered when any of the following conditions is true:
  - Resources whose access intent is update are already registered at other locations.
  - Resources whose access intent is undetermined are already registered at other locations.

– Resources whose access intent is undetermined are already registered at the same location and the resources have been changed during the current transaction.

• A two-phase resource whose access intent is update cannot be registered when a one-phase resource whose access intent is update is already registered.

The commit protocol of a committable resource

Commit protocol is the capability a resource has to participate in one-phase or two-phase commit processing. Local resources, except API committable resources, are always two-phase resources.

If a committable resource resides on an independent disk pool and the commitment definition resides on a different disk pool, the resource is not considered as a local resource or a two-phase resource.

A two-phase resource is also called a *protected resource*. Remote resources and API committable resources must be registered as one-phase resources or two-phase resources when they are placed under commitment control. The following table shows what types of committable resources can coexist in a commitment definition with a one-phase resource.

Resource type	Can coexist with
One-phase API resource	Other local resources. No remote resources.
One-phase remote resource	Other one-phase resources at the same location. No local resources.

Related concepts

"Commitment control and independent disk pools" on page 20
Independent disk pools and independent disk pool groups can each have a separate i5/OS database. You can use commitment control with these databases.

Journaled files and commitment control

You must journal (log) a database file (resource type FILE or DDM) before it can be opened for output under commitment control or referenced by an SQL application that uses an isolation level other than No Commit. A file does not need to be journaled in order to open it for input only under commitment control.

An error occurs if any of the following conditions is true:

• An attempt is made to open a database file for output under commitment control, but the file is not currently journaled.
• No commitment definition is started that can be used by the file being opened under commitment control.

If only the after-images are being journaled for a database file when that file is opened under commitment control, the system automatically starts journaling both the before-images and after-images. The before-images are written only for changes to the file that occur under commitment control. If other changes that are not under commitment control occur to the file at the same time, only after-images are written for those changes.

The system automatically writes record-level committable changes and object-level committable changes to a journal. For record-level changes, the system then uses the journal entries, if necessary, for recovery purposes; the system does not use entries from object-level committable changes for recovery purposes. Furthermore, the system does not automatically write journal entries for API commitment resources. However, the exit program for the API resource can use the Send Journal Entry (QJOSJRNE) API to write journal entries to provide an audit trail or to assist with recovery. The content of these entries is controlled by the user exit program.

The system uses a technique other than a journal to perform recovery for object-level commitment resources. Recovery for API commitment resources is accomplished by calling the commit and rollback exit program associated with each particular API commitment resource. The exit program has the responsibility for performing the actual recovery that is necessary for the situation.

Related concepts

Journal management

Sequence of journal entries under commitment control

This table shows the sequence of entries that are typically written while a commitment definition is active. You can use the Journal entry information finder to get more information about the contents of the journal entries.

Commitment control entries are written to a local journal if at least one of the following conditions is true:

- The journal is specified as the default journal on the Start Commitment Control (STRCMTCTL) command.
- At least one file journaled to the journal is opened under commitment control.
- At least one API commitment resource associated with the journal is registered under commitment control.

Entry type	Description	Where it is written	When it is written
C BC	Begin commitment control	To the default journal, if one is specified on the STRCMTCTL command.	When the STRCMTCTL command is used.
		To the journal.	When the first file journaled to a journal is opened or when an API resource is registered for a journal.
C SC	Start commit cycle	To the journal.	When the first record change occurs for the transaction for a file journaled to this journal[1].
		To the journal for an API resource.	When the QJOSJRNE API is first used with the *Include Commit Cycle Identifier* key.
Journal codes D and F	DDL object-level entries	To the journal associated with the object being updated. Only journal entries that contain a commit cycle identifier represent a DDL object-level change that is part of the transaction.	When updates occur.
Journal code R	Record-level entries	To the journal associated with the file being updated.	When the updates occur.
Journal code U	User-created entries	To the journal associated with an API resource.	If the application program uses the QJOSJRNE API is first used with the *Include Commit Cycle Identifier* key.
C CM	Commit	To the journal.	When the commit has completed successfully.
		To the default journal.	If any committable resources are associated with the journal.

Entry type	Description	Where it is written	When it is written
C RB	Rollback	To the journal.	After the rollback operation has completed.
		To the default journal.	If any committable resources are associated with the journal.
C LW	End transaction	To the default journal, if one is specified on the STRCMTCTL command. The system writes an LW header record and one or more detail records. These entries are written only if OMTJRNE(*NONE) is specified on the STRCMTCTL command or if a system error occurs.	When the commit or rollback operation has completed.
C EC	End commitment control	To the journal.	When the End Commitment Control (ENDCMTCTL) command is completed.
		To a local journal that is not the default journal.	When a commit boundary is established, following the point when all committable resources associated with that journal have been removed from commitment control.
C SB	Start of savepoint or nested commit cycle.	To the journal.	When the application creates an SQL SAVEPOINT, or when the system creates an internal nested commit cycle to handle a series of database functions as a single operation[2].
C SQ	Release of savepoint or commit of nested commit cycle.	To the journal.	When the application releases an SQL SAVEPOINT, or when the system commits an internal nested commit cycle[2].
C SU	Rollback of savepoint or nested commit cycle.	To the journal.	When the application rolls back an SQL SAVEPOINT, or when the system rolls back an internal nested commit cycle[2].

Entry type	Description	Where it is written	When it is written
Notes: [1] You can specify that the fixed-length portion of the journal entry includes transaction information by specifying the Logical Unit of Work (*LUW) value for the Fixed-Length Data (FIXLENDTA) parameter of the Create Journal (CRTJRN) or Change Journal (CHGJRN) command. By specifying the FIXLENDTA (*LUW) parameter, the fixed-length portion of each C SC journal entry will contain the Logical Unit of Work ID (LUWID) of the current transaction. Likewise for XA transactions, if you specify the FIXLENDTA(*XID) parameter, the fixed-length portion of each C SC journal entry will contain the XID of the current transaction. The LUWID or XID can help you find all the commit cycles for a particular transaction if multiple journals or systems are involved in the transaction. [2] These entries are sent only if you set the QTN_JRNSAVPT_MYLIB_MYJRN environment variable to *YES where MYJRN is the journal you are using and MYLIB is the library the journal is stored in. Special value *ALL is supported for the MYLIB and MYJRN values. You can set these variables system-wide or for a specific job. To have the entries sent for journal MYLIB/MYJRN for just one job, use this command in that job: • ADDENVVAR ENVVAR(QTN_JRNSAVPT_MYLIB_MYJRN) VALUE(*YES) To have entries sent for all journals for all jobs, use this command: • ADDENVVAR ENVVAR('QTN_JRNSAVPT_*ALL_*ALL') VALUE(*YES) LEVEL(*SYS) You need to set the environment variable before you start commitment control.			

Related concepts

"Commit operation" on page 3
A commit operation makes permanent all changes made under commitment control since the previous commit or rollback operation. The system also releases all locks related to the transaction.

Journal entry information finder

Related reference

End Commitment Control (ENDCMTCTL) command

Commit cycle identifier

A *commit cycle* is the time from one commitment boundary to the next. The system assigns a *commit cycle identifier* to associate all of the journal entries for a particular commit cycle together. Each journal that participates in a transaction has its own commit cycle and its own commit cycle identifier.

The commit cycle identifier is the journal sequence number of the C SC journal entry written for the commit cycle. The commit cycle identifier is placed in each journal entry written during the commit cycle. If more than one journal is used during the commit cycle, the commit cycle identifier for each journal is different.

You can specify that the fixed-length portion of the journal entry includes transaction information by specifying the Logical Unit of Work (*LUW) value for the Fixed-Length Data (FIXLENDTA) parameter of the Create Journal (CRTJRN) or Change Journal (CHGJRN) command. By specifying the FIXLENDTA (*LUW) parameter, the fixed-length portion of each C SC journal entry will contain the Logical Unit of Work ID (LUWID) of the current transaction. Likewise for XA transactions, if you specify the FIXLENDTA (*XID) parameter, the fixed-length portion of each C SC journal entry will contain the XID of the current transaction. The LUWID or XID can help you find all the commit cycles for a particular transaction if multiple journals or systems are involved in the transaction.

You can use the Send Journal Entry (QJOSJRNE) API to write journal entries for API resources. You have the option of including the commit cycle identifier on those journal entries.

You can use the commit cycle identifier to apply or remove journaled changes to a commitment boundary using the Apply Journaled Changes (APYJRNCHG) command or the Remove Journaled Changes (RMVJRNCHG) command. These limitations apply:

- Most object-level changes made under commitment control are written to the journal but are not applied or removed using the APYJRNCHG and RMVJRNCHG commands.
- The QJOSJRNE API writes user-created journal entries with a journal code of U. These entries cannot be applied or removed using the APYJRNCHG and RMVJRNCHG commands. They must be applied or removed with a user-written program.

Record locking

When a job holds a record lock and another job attempts to retrieve that record for update, the requesting job waits and is removed from active processing.

The requesting job will be active till one of the following events occurs:
- The record lock is released.
- The specified wait time ends.

More than one job can request a record to be locked by another job. When the record lock is released, the first job to request the record receives that record. When waiting for a locked record, specify the wait time in the WAITRCD parameter on the following create, change, or override commands:
- Create Physical File (CRTPF)
- Create Logical File (CRTLF)
- Create Source Physical File (CRTSRCPF)
- Change Physical File (CHGPF)
- Change Logical File (CHGLF)
- Change Source Physical File (CHGSRCPF)
- Override Database File (OVRDBF)

When you specify wait time, consider the following information:
- If you do not specify a value, the program waits the default wait time for the process.
- For commitment definitions with transaction-scoped locks only, the job default wait time can be overridden by a transaction lock-wait time that can be specified on:
  - The xa_open API.
  - A JDBC or JTA interface. Distributed transactions lists these APIs.
- If the record cannot be allocated within the specified time, a notify message is sent to the high-level language program.
- If the wait time for a record is exceeded, the message sent to the job log gives the name of the job holding the locked record that caused the requesting job to wait. If you experience record lock exceptions, you can use the job log to help determine which programs to alter so they will not hold locks for long durations.

Programs keep record locks over long durations for one of the following reasons:
- The record remains locked while the workstation user is considering a change.
- The record lock is part of a long commitment transaction. Consider making smaller transactions so a commit operation can be performed more frequently.
- An undesired lock has occurred. For example, assume that a file is defined as an update file with unique keys, and that the program updates and adds additional records to the file. If the workstation user wants to add a record to the file, the program might attempt to access the record to determine whether the key already exists. If it does, the program informs the workstation user that the request made is not valid. When the record is retrieved from the file, it is locked until it is implicitly released by another read operation to the same file, or until it is explicitly released.

Note: For more information about how to use each high-level language interface to release record locks, see the appropriate high-level language reference manual. Related information for commitment control has links to high-level language manuals that you can use with commitment control.

The duration of the lock is much longer if LCKLVL(*ALL) is specified because the record that was retrieved from the file is locked until the next commit or rollback operation. It is not implicitly released by another read operation and cannot be explicitly released.

Another function that can put a lock on a file is the save-while-active function.

Related concepts

JDBC Distributed transactions

Saving your system while it is active

Related reference

"Related information for commitment control" on page 110
Listed here are the product manuals and IBM Redbooks™ (in PDF format), Web sites, and information center topics that relate to the commitment control topic. You can view or print any of the PDFs.

Commitment control and independent disk pools

Independent disk pools and independent disk pool groups can each have a separate i5/OS database. You can use commitment control with these databases.

However, because each independent disk pool or independent disk pool group has a separate SQL database, you should be aware of some considerations.

Related concepts

"Commitment definition" on page 5
The commitment definition contains information that pertains to the resources that are being changed under commitment control during a transaction.

"Local and remote committable resources" on page 13
A committable resource can be either a local resource or a remote resource.

"The commit protocol of a committable resource" on page 15
Commit protocol is the capability a resource has to participate in one-phase or two-phase commit processing. Local resources, except API committable resources, are always two-phase resources.

Independent disk pool considerations for commitment definitions

You must be aware of these considerations for commitment definitions when you use independent disk pools.

QRECOVERY library considerations

When you start commitment control, the commitment definition is created in the QRECOVERY library. Each independent disk pool or independent disk pool group has its own version of a QRECOVERY library. On an independent disk pool, the name of the QRECOVERY library is QRCYxxxxx, where xxxxx is the number of the independent disk pool. For example, the name of the QRECOVERY library for independent disk pool 39 is QRCY00039. Furthermore, if the independent disk pool is part of a disk pool group, only the primary disk pool has a QRCYxxxxx library.

When you start commitment control, the commitment definition is created in the QRECOVERY library of the independent disk pool that is associated with that job, making commitment control active on the independent disk pool.

Set ASP Group considerations

Using the Set ASP Group (SETASPGRP) command while commitment control is active on an independent disk pool has the following effects:

- If you switch from an independent disk pool and resources are registered with commitment control on the disk pool, the SETASPGRP command fails with message CPDB8EC, reason code 2, `The thread has an uncommitted transaction.` This message is followed by message CPFB8E9.
- If you switch from an independent disk pool and no resources are registered with commitment control, the commitment definitions are moved to the independent disk pool to which you are switching.
- If you switch from the system disk pool (ASP group *NONE), commitment control is not affected. The commitment definitions stay on the system disk pool.
- If you use a notify object, the notify object must reside on the same independent disk pool or independent disk pool group as the commitment definition.
- If you move the commitment definition to another independent disk pool or independent disk pool group, the notify object must also reside on that other independent disk pool or independent disk pool group. The notify object on the other independent disk pool or independent disk pool group is updated if the commitment definition ends abnormally. If the notify object is not found on the other independent disk pool or independent disk pool group, the update fails with message CPF8358.

Default journal considerations

You should be aware of the following default journal considerations:

- If you use the default journal, the journal must reside on the same independent disk pool or independent disk pool group as the commitment definition.
- If the default journal is not found on the other independent disk pool or independent disk pool group when commitment control starts, the commitment control start fails with message CPF9873.
- If you move the commitment definition to another independent disk pool or independent disk pool group, the default journal must also reside on that other independent disk pool or independent disk pool group. If the journal is not found on the other independent disk pool or independent disk pool group, the commitment definition is moved, but no default journal is used from this point on.

Initial program load (IPL) and vary off considerations

You should be aware of the following IPL and vary off considerations:

- Recovery of commitment definitions residing on an independent disk pool is performed during the vary on processing of the independent disk pool and is similar to IPL recovery.
- Commitment definitions in an independent disk pool are not recovered during the system IPL.
- The vary off of an independent disk pool has the following effects on commitment definitions:
  - Jobs associated with the independent disk pool end.
  - No new commitment definitions are allowed to be created on the independent disk pool.
  - Commitment definitions residing on the independent disk pool become unusable.
  - Commitment definitions residing on the independent disk pool, but not attached to a job, release transaction scoped locks.

Remote database considerations

You should be aware of the following remote database considerations:

- You cannot use an LU 6.2 SNA connection (protected conversations or Distributed Unit of Work (DUW)) to connect to a remote database from an independent disk pool database. You can use unprotected SNA conversations to connect from an independent disk pool database to a remote database.

- When commitment control is active for a job or thread, access to data outside the independent disk pool or disk pool group to which the commitment definition belongs is only possible remotely, as if it were data that resides on another system. When you issue an SQL CONNECT statement to connect to the relational database (RDB) on the independent disk pool, the system makes the connection a remote connection.
- The system disk pool and basic disk pools do not require a remote connection for read-only access to data that resides on an independent disk pool. Likewise, an independent disk pool does not require a remote connection for read-only access to data that resides on the system disk pool or a basic disk pool.

Related concepts

"Commitment definition" on page 5
The commitment definition contains information that pertains to the resources that are being changed under commitment control during a transaction.

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Considerations for XA transactions

In the XA environment, each database is considered a separate resource manager. When a transaction manager wants to access two databases under the same transaction, it must use the XA protocols to perform two-phase commit with the two resource managers.

Because each independent disk pool is a separate SQL database, in the XA environment each independent disk pool is also considered a separate resource manager. For an application server to perform a transaction that targets two different independent disk pools, the transaction manager must also use a two-phase commit protocol.

Related concepts

"XA transaction support for commitment control" on page 42
DB2 Universal Database (UDB) for iSeries can participate in X/Open global transactions.

Independent disk pool examples

Considerations and restrictions for commitment control

You need to be aware of these considerations and restrictions for commitment control.

Database file considerations

- If you specify that a shared file be opened under commitment control, all subsequent uses of that file must be opened under commitment control.
- If SEQONLY(*YES) is specified for the file opened for read-only with LCKLVL(*ALL) (either implicitly or by a high-level language program, or explicitly by the Override with Database File (OVRDBF) command), then SEQONLY(*YES) is ignored and SEQONLY(*NO) is used.
- Record-level changes made under commitment control are recorded in a journal. These changes can be applied to or removed from the database with the Apply Journaled Changes (APYJRNCHG) command or the Remove Journaled Changes (RMVJRNCHG) command.
- Both before-images and after-images of the files are journaled under commitment control. If you specify only to journal the after-images of the files, the system also automatically journals the before-image of the file changes that occurred under commitment control. However, because the before-images are not captured for all changes made to the files, you cannot use the RMVJRNCHG command for these files.

Considerations for object-level and record-level changes

- Object-level and record-level changes made under commitment control using SQL use the commitment definition that is currently active for the activation group that the requesting program is running in. If neither the job-level nor the activation-group-level commitment definition is active, SQL will start an activation-group-level commitment definition.

One-phase and two-phase commit considerations

- While a one-phase remote conversation or connection is established, remote conversations or connections to other locations are not allowed. If a commitment boundary is established and all resources are removed, the location can be changed.
- If you are using two-phase commit, you do not need to use the Submit Remote Command (SBMRMTCMD) command to start commitment control or perform any other commitment control operations at the remote locations. The system performs these functions for you.
- For a one-phase remote location, the COMMIT and ROLLBACK CL commands will fail if SQL is in the call stack and the remote relational database is not on a system. If SQL is not on the call stack, the COMMIT and ROLLBACK commands will not fail.
- For a one-phase remote location, commitment control must be started on the source system before making committable changes to remote resources. The system automatically starts commitment control for distributed database SQL on the source system at connection time if the SQL program is running with the commitment control option other than *NONE. When the first remote resource is placed under commitment control, the system starts commitment control on the target system.

Save consideration

A save operation is prevented if the job performing the save has one or more active commitment definitions with any of the following types of committable changes:

- A record change to a file that resides in the library being saved. For logical files, all the related physical files are checked.
- Any object-level changes within a library that is being saved.
- Any API resource that was added using the Add Commitment Resource (QTNADDCR) API and with the Allow normal save processing field set to the default value of N.

This prevents the save operations from saving to the save media changes that are due to a partial transaction.

Note: If you use the new save with partial transactions feature, the object can be saved without ending a commitment definition.

Object locks and record locks prevent pending changes from commitment definitions in other jobs from being saved to the save media. This is true only for API commitment resources if locks are acquired when changes are made to the object or objects associated with the API commitment resource.

Miscellaneous considerations and restrictions

- Before upgrading your system to a new release, all pending resynchronizations must either be completed or canceled.
- The COMMIT and ROLLBACK values are shown on the WRKACTJOB Function field during a commit or rollback. If the Function remains COMMIT or ROLLBACK for a long time, one of the following events might have occurred:
  - A resource failure during the commit or rollback requires resynchronization. Control will not return to the application until the resynchronization completes or is canceled.
  - This system voted read-only during the commit. Control will not return to the application until the system that initiated the commit sends data to this system.

– This system voted OK to leave out during the commit. Control will not return to the application until the system that initiated the commit sends data to this system.

Related concepts

Ensuring two-phase commit integrity

"Commit lock level" on page 50
The value you specify for the LCKLVL parameter on the Start Commitment Control (STRCMTCTL) command becomes the default level of record locking for database files that are opened and placed under commitment control for the commitment definition.

Related reference

Override with Database File (OVRDBF) command

Apply Journaled Changes (APYJRNCHG) command

Remove Journaled Changes (RMVJRNCHG) command

SQL programming

Submit Remote Command (SBMRMTCMD) command

Commit (COMMIT) command

Rollback (ROLLBACK) command

Add Commitment Resource (QTNADDCR) API

Commitment control for batch applications

Batch applications might or might not need commitment control. In some cases, a batch application can perform a single function of reading an input file and updating a master file. However, you can use commitment control for this type of application if it is important to start it again after an abnormal end.

The input file is an update file with a code in the records to indicate that a record was processed. This file and any files updated are placed under commitment control. When the code is present in the input file, it represents a completed transaction. The program reads through the input file and bypasses any records with the completed code. This allows the same program logic to be used for normal and starting-again conditions.

If the batch application contains input records dependent on one another and contains switches or totals, a notify object can be used to provide information about starting again. The values held in the notify object are used to start processing again from the last committed transaction within the input file.

If input records are dependent on one another, they can be processed as a transaction. A batch job can lock a maximum of 500 000 000 records. You can reduce this limit by using a Query Options File (QAQQINI). Use the QRYOPTLIB parameter of the Change Query Attributes (CHGQRYA) command to specify a Query Options File for a job to use. Use the COMMITMENT_CONTROL_LOCK_LEVEL value in the Query Options File as the lock limit for the job.

Any commit cycle that exceeds 2000 locks probably slows down system performance noticeably. Otherwise, the same locking considerations exist as for interactive applications, but the length of time records are locked in a batch application might be less important than in interactive applications.

Related concepts

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

"Managing transaction size" on page 68
Another way to minimize record locks is to manage the size of the transaction.

Related reference

Change Query Attributes (CHGQRYA) command

Two-phase commitment control

Two-phase commitment control ensures that committable resources on multiple systems remain synchronized.

i5/OS operating system supports two-phase commit in accordance with the SNA LU 6.2 architecture. For more detailed information about the internal protocols used by the system for two-phase commit, see the *SNA Transaction Programmer's Reference for LU Type 6.2, GC30-3084-05*. All supported releases of i5/OS operating system support the Presumed Nothing protocols of SNA LU 6.2 and the Presumed Abort protocols of SNA LU 6.2.

Two-phase commit is also supported using TCP/IP as a Distributed Unit of Work (DUW) *Distributed Relational Database Architecture (DRDA)* protocol. To use TCP/IP DUW connections, all of the systems (both the application requester and the application server) must be at V5R1M0 or newer. For more information about DRDA, see the Open Group Technical Standard, *DRDA V2 Vol. 1: Distributed Relational Database Architecture* at the Open Group Web site.

Under two-phase commit, the system performs the commit operation in two waves:

- During the *prepare wave*, a resource manager issues a commit request to its transaction manager. The transaction manager informs any other resources it manages and the other transaction managers that the transaction is ready to be committed. All the resource managers must respond that they are ready to commit. This is called the *vote*.
- During the *committed wave*, the transaction manager that initiates the commit request decides what to do, based on the outcome of the prepare wave. If the prepare wave completes successfully and all participants vote ready, the transaction manager instructs all the resources it manages and the other transaction managers to commit the transaction. If the prepare wave does not complete successfully, all the transaction managers and resource managers are instructed to roll back the transaction.

Commit and rollback operations with remote resources

When remote resources are under commitment control, the initiator sends a commit request to all remote agents. The request is sent throughout the transaction program network. Each agent responds with the results of the commit operation.

If errors occur during the prepare wave, the initiator sends a rollback request to all agents. If errors occur during the committed wave, the system attempts to bring as many locations as possible to committed status. These attempts might result in a heuristic mixed state. See States of the transaction for two-phase commitment control for more information about the possible states.

Any errors are sent back to the initiator where they are signaled to the user. If a default journal was specified on the Start Commitment Control (STRCMTCTL) command, C LW entries are written. If errors occur, these entries are written, even if OMTJRNE(*LUWID) was specified. You can use these entries, along with the error messages and the status information for the commitment definition, to attempt to synchronize the committable resources manually.

When remote resources are under commitment control, the initiator sends a rollback request to all remote agents. The request is sent throughout the transaction program network. Each agent responds with the results of the rollback operation.

Related concepts

The Open Group Web site

Related reference

Start Commitment Control (STRCMTCTL) command

Roles in commit processing

If a commit of a transaction involves more than one resource manager, each resource manager plays a role in the transaction. A resource manager is responsible for committing or rolling back changes made during the transaction.

The resource managers by resource type are as follows:

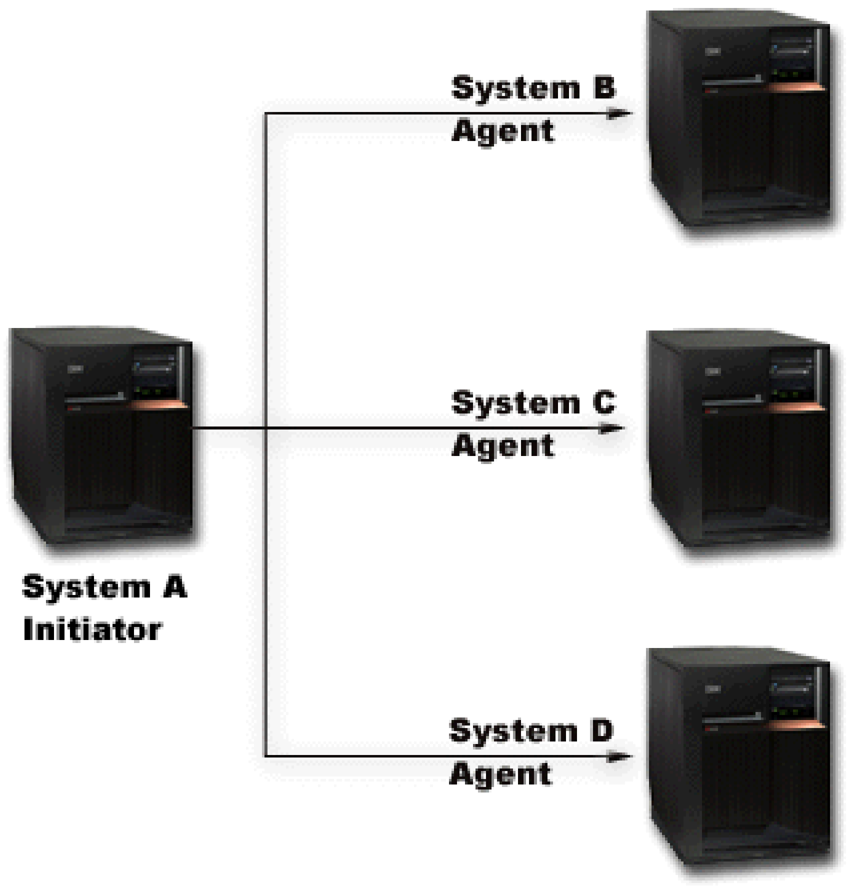
FILE	Database manager
DDM	Database manager
DDL	Database manager
DRDA	Communications transaction program
LU62	Communications transaction program
API	API exit program

The following figures shows the basic roles in a transaction. The structure shown in the figures is called a *transaction program network*. The structure can be in a single-level tree and a multilevel tree.

Roles in two-phase commit processing: Single-level tree

When an application on System A issues a commit request, the resource manager on System A becomes the *initiator*. For Distributed Relational Database Architecture (DRDA) distributed unit of work over TCP/IP, the initiator is called the *coordinator*.

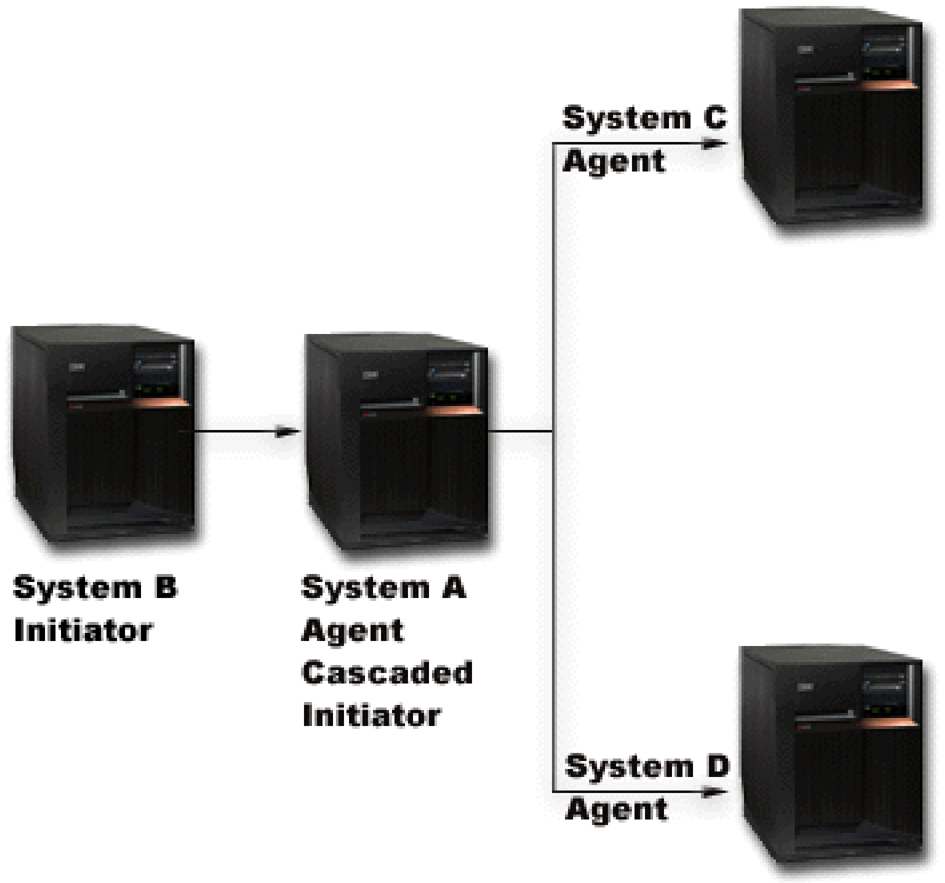
The resource managers for the other three systems (B, C, and D) become *agents* for this transaction. For DRDA distributed unit of work over TCP/IP, agents are sometimes called *participants*.



Roles in two-phase commit processing: Multi-level tree

If the application is using APPC communications to perform the two-phase commit, the relationship between systems can change from one transaction to the next. The following figure shows the same systems when an application on System B issues the commit request. This configuration is a multi-level tree.

The roles in this figure do not apply to DRDA distributed unit of work over TCP/IP because multi-level transactions trees are not supported.



The transaction program network has another level because System B is not communicating directly with System C and System D. The resource manager in System A now has the roles of agent and cascaded initiator.

To improve performance of LU 6.2 two-phase transactions, the initiator might assign the role of last agent to one of the agents. The last agent does not participate in the prepare wave. In the committed wave, the last agent commits first. If the last agent does not commit successfully, the initiator instructs the other agents to roll back.

For DRDA distributed unit of work over TCP/IP, the coordinator might assign the role of resync server to a participant. The resync server is responsible to resynchronize the other participants in the event in which there is a communications failure with the coordinator, or the coordinator has a systems failure.

Related concepts

"Commitment definition for two-phase commit: Allow vote read-only" on page 31
Normally, a transaction manager participates in both phases of commit processing. To improve the performance of commit processing, you can set up some or all locations in a transaction to allow the transaction manager to vote read-only.

States of the transaction for two-phase commitment control

A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

The system uses the state to decide whether to commit or roll back if a transaction is interrupted by a communication or system failure. If multiple locations are participating in a transaction, the states of the transactions at each location might be compared to determine the correct action (commit or rollback). This process of communicating between locations to determine the correct action is called *resynchronization*.

The following table shows these items:
- The basic states that might occur during a transaction.
- Additional states that might occur.
- Whether a state requires resynchronization if the transaction is interrupted by a communications or system failure. The possible values are as follows:

  Not needed
  Each location can make the correct decision independently.

  May be necessary
  Each location can make the correct decision, but the initiator might need to be informed of the decision.

  Required
  The state of each location must be determined before the correct decision can be made.
- Action taken by a communications or system failure.

State name	Description	Resynchronization if the transaction is interrupted	Action taken by a communications or system failure
Basic states during two-phase commit processing:			
Reset (**RST**)	From the commitment boundary until a program issues a request to commit or roll back.	Not needed.	Pending changes are rolled back.
Prepare in Progress (**PIP**)	The initiator has started the prepare wave. All locations have not yet voted.	May be necessary.	Pending changes are rolled back.
Prepared (**PRP**)	This location and all locations further down in the transaction program network have voted to commit. This location has not yet received notification from the initiator to commit.	Required.	In doubt. It depends on the results of the resynchronization process.
Commit in Progress (**CIP**)	All locations have voted to commit. The initiator has started the committed wave.	Required.	Pending changes are committed. Resynchronization is performed to ensure that all locations have committed. If a heuristic rollback is reported by another location, an error is reported.
Committed (**CMT**)	All agents have committed and returned a reply to this node.	May be necessary.	None.
Additional states during two-phase commit processing:			

State name	Description	Resynchronization if the transaction is interrupted	Action taken by a communications or system failure
Last Agent Pending (**LAP**)	If a last agent is selected, this state occurs at the initiator between the PIP state and the CIP state. The initiator has instructed the last agent to commit and has not yet received a response.	Required.	In doubt. It depends on the results of the resynchronization process.
Vote-Read-Only (**VRO**)	This agent responded to the prepare wave by indicating that it has no pending changes. If the vote-read-only state is permitted, this agent is not included in the committed wave.	May be necessary.	None.
Rollback Required (**RBR**)	One of the following events occurred: • An agent issued a rollback request before the commit operation. • A transaction failure has occurred. • The QTNRBRQD API was used to place the transaction in a rollback required state. The transaction program is not allowed to perform any additional changes under commitment control.	May be necessary.	Pending changes are rolled back.
Conditions that occur because of operator actions or errors:			
Forced Rollback	This location and all locations further down the transaction program network, except the last agent, have been rolled back through operator intervention.	May be necessary.	Pending changes have already been rolled back.
Forced Commit	This location and all locations further down the transaction program network, except the last agent, have committed through operator intervention.	May be necessary.	Pending changes have already been committed.

State name	Description	Resynchronization if the transaction is interrupted	Action taken by a communications or system failure
Heuristic Mixed (**HRM**)	Some resource managers have committed. Some have rolled back. Operator intervention was used or a system error occurred. Heuristic mixed does not appear as a status on the commitment definition displays. Notification messages are sent to the operator.	May be necessary.	The operator must perform a restore operation at all participating locations to bring the database to a consistent state.

Related concepts

"Commitment definition for two-phase commit: Allow vote read-only"
Normally, a transaction manager participates in both phases of commit processing. To improve the performance of commit processing, you can set up some or all locations in a transaction to allow the transaction manager to vote read-only.

"Commitment definition for two-phase commit: Not wait for outcome" on page 33
When a communication or system failure occurs during a commit operation so that resynchronization is required, the default is to wait until the resynchronization is finished before the commit operation completes.

"Starting commitment control" on page 47
To start commitment control, use the Start Commitment Control (STRCMTCTL) Command.

"Commitment control recovery during initial program load after abnormal end" on page 61
When you perform an initial program load (IPL) after your system ends abnormally, the system attempts to recover all the commitment definitions that were active when the system ended.

"Commitment control errors" on page 99
When you use commitment control, it is important to understand which conditions cause errors and which do not.

Commitment definitions for two-phase commitment control

To change the commitment options for your transaction after you start commitment control, use the Change Commitment Options (QTNCHGCO) API.

Depending on your environment and your applications, changing the commitment options can improve your system's performance.

Note: If you are using a DRDA distributed unit of work over TCP/IP connection, the only option that applies is Allow vote read-only.

Related reference

Change Commitment Options (QTNCHGCO) API

Commitment definition for two-phase commit: Allow vote read-only:

Normally, a transaction manager participates in both phases of commit processing. To improve the performance of commit processing, you can set up some or all locations in a transaction to allow the transaction manager to vote read-only.

If the location has no committable changes during a transaction, the transaction manager votes read-only during the prepare wave. The location does not participate in the committed wave. This improves overall performance because the communication flows that normally occur during the committed wave are eliminated during transactions in which no updates are made at one or more remote locations.

After you start commitment control, you can use the Change Commitment Options (QTNCHGCO) API to change the `Vote read-only permitted` option to Y. You might want to do this if the following conditions are true:

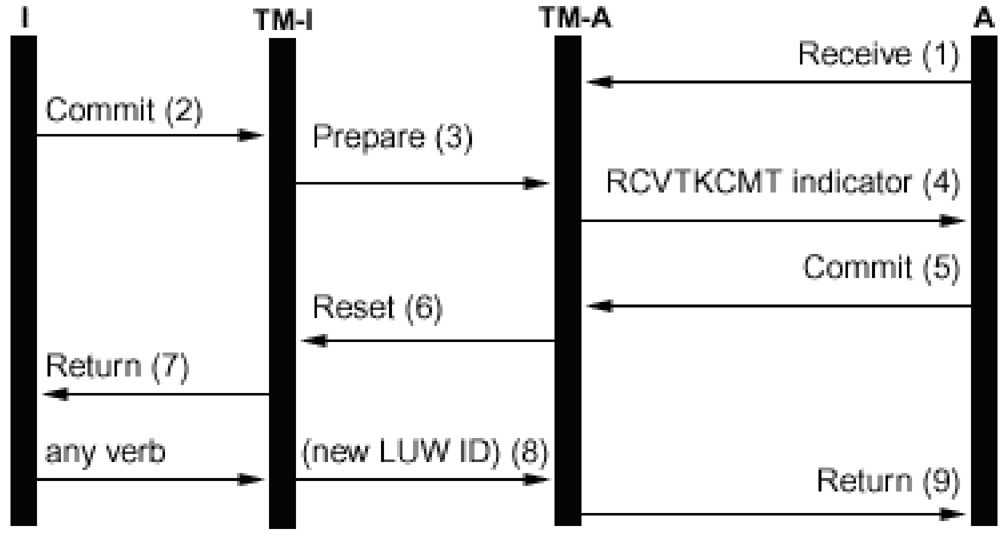
- One or more remote systems often do not have any committable changes for a transaction.
- A transaction does not depend on where the file cursor (next record) was set by the previous transaction. When a location votes read-only, the application is never notified if the transaction is rolled back. The location has committed any read operations to the database files and, thus, moved the cursor position. The position of the file cursor is typically important only if you do sequential processing.

If your commitment definition is set up to allow vote read-only, the application waits for the next message flow from another location.

The `Vote read-only permitted` option is intended for applications that are client/server in nature. If the purpose of program A is only to satisfy requests from program I, not to do any independent work, it is appropriate to allow the `Vote read-only` option for program A.

Flow of commit processing without last agent optimization when agent votes read-only

The following figure shows the flow of messages among the application programs and the transaction managers when an application program issues a commit instruction without last agent optimization when the agent votes read-only. Neither the initiator application program nor the agent application programs is aware of the two-phase commit processing. The numbers in parentheses () in the figure correspond to the numbered items in the description that follows.



Legend

I	=	Initiator (Application that initiates commit request)
TM-I	=	Transaction manager for initiator
A	=	Agent (Application that receives commit request)
TM-A	=	Transaction manager for agent

The following list is a description of the events for normal processing without last agent optimization when the agent votes read only. This describes a basic flow. The sequence of events can become much more complex when the transaction program network has multiple levels or when errors occur.

1. Application program A does a receive request to indicate that it is ready to receive a request from program I.
2. The initiator application (I) issues a commit instruction.
3. The transaction manager for the initiator (TM-I) takes the role of initiator for this transaction. It starts the prepare wave by sending a prepare message to all the other locations that are participating in the transaction.
4. The transaction managers for every other location take the role of agent (TM-A). The application program A is notified by TM-A that a request to commit has been received. For ICF files, the notification is in the form of the Receive Take Commit (RCVTKCMT) ICF indicator being set on.
5. The application program A responds by issuing a commit instruction (or a rollback instruction). This is the application program's vote.
6. If application program A has used the Change Commitment Options API (QTNCHGCO) to set the Vote read-only permitted commitment option to Y and no changes have been made at the agent during the transaction, the agent (TM-A) responds to the initiator (TM-I) with a reset message. There is no committed wave for the agent.
7. A return is sent to the application program (A) to indicate that the transaction is complete at agent TM-A.
8. The next time the initiator (TM-I) issues any messages to the agent (TM-A), either a data flow or a commitment instruction, TM-I causes its current transaction ID to be sent with the message. The reason for this is that a new transaction ID might have been generated at TM-I if a communications failure had occurred between TM-I and another system during the commit operation.
9. A return is sent to the application program (A) to indicate that the transaction is complete at agent TM-A. The return is delayed until after the next message is received because a new transaction ID must be received from TM-I before the next transaction can be started by application A.

Related concepts

"Roles in commit processing" on page 26
If a commit of a transaction involves more than one resource manager, each resource manager plays a role in the transaction. A resource manager is responsible for committing or rolling back changes made during the transaction.

"States of the transaction for two-phase commitment control" on page 28
A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

"Optimizing performance for commitment control" on page 64
Using commitment control requires resources that can affect system performance. Several factors affect system performance regarding commitment control.

Related reference

Change Commitment Options (QTNCHGCO) API

Commitment definition for two-phase commit: Not wait for outcome:

When a communication or system failure occurs during a commit operation so that resynchronization is required, the default is to wait until the resynchronization is finished before the commit operation completes.

Note: The Not wait for outcome option does not apply if you are using a Distributed Relational Database Architecture (DRDA) distributed unit of work over TCP/IP connection. DRDA distributed unit of work over TCP/IP connections never waits for outcome.

Consider changing this behavior if the following conditions are true:

- The applications that participate are independent of each other.

- Your program logic does not need the results of previous transactions to ensure that your database files remain synchronized.

After you start commitment control, you can use the Change Commitment Options (QTNCHGCO) API to specify that the commitment definition does not wait for the outcome of resynchronization. If you specify N (No) for the `Wait for outcome` option, the system uses a database server job (QDBSRVnn) to handle resynchronization asynchronously.

Note: These database server jobs are started during the initial program load (IPL) process. If you change the options for commitment control, this has no effect on the number of jobs that the system starts.

This topic only refers to two values for the resolved `Wait for outcome` option, Y (Yes) and N (No). There are actually two more values that you can specify, L (Yes or Inherit from Initiator) and U (No or Inherit from Initiator). When you use these values, the actual value used during each commit operation is resolved to Yes or No by the system. The QTNCHGCO (Change Commitment Options) API topic has more details about these values.

Note: The initiator's value can only be inherited by an agent if both the initiator and the agent support presumed abort.

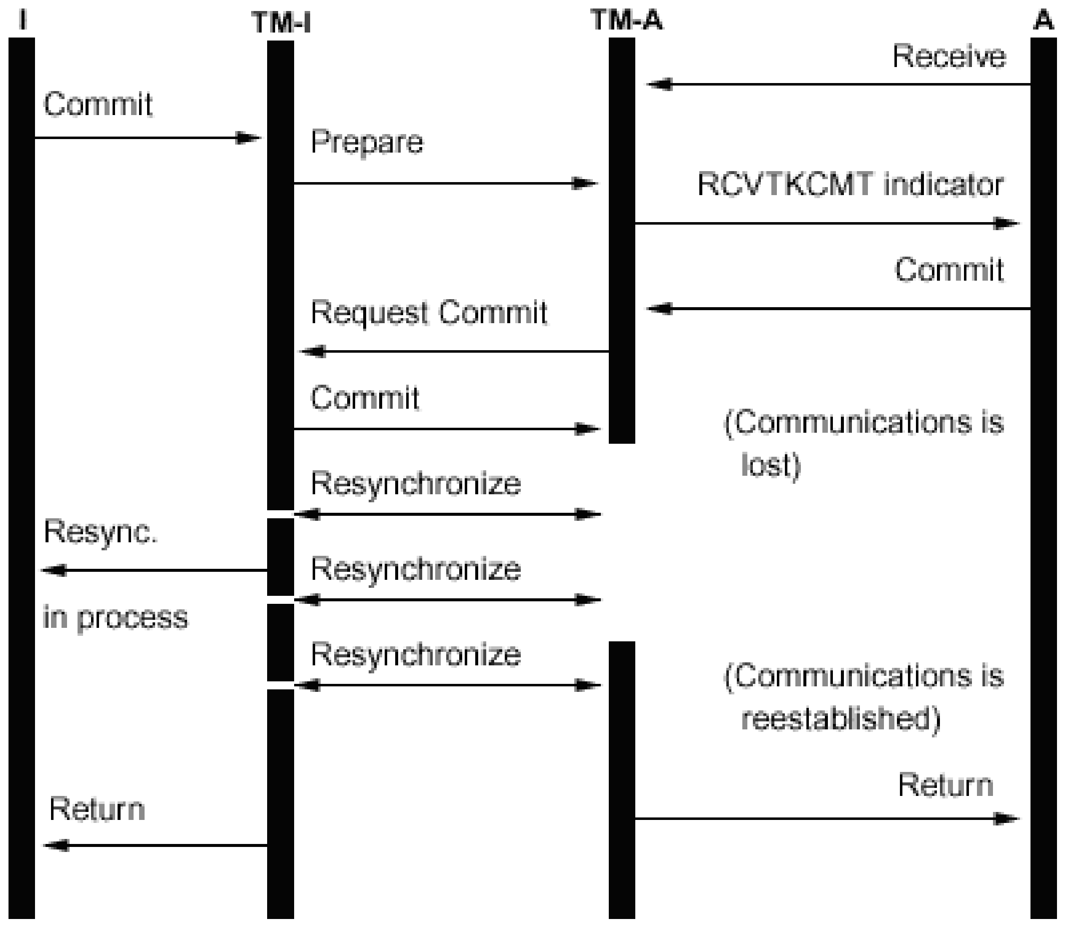
The `wait for outcome` (WFO) option does not affect normal, error-free commit processing. If an error occurs, the WFO option determines whether the application waits for resynchronization or not, with the following conditions:

- If the resolved WFO option is Y (Yes), the application waits for the result of the resynchronization.
- If the resolved WFO option is N (No) and a communication failure occurs during the prepare wave or rollback of a location that supports presumed abort protocols, no resynchronization is performed and the commitment definition is rolled back.
- If the commitment definition is in doubt (transaction state is prepared or Last Agent Pending), the application will wait for the result of the resynchronization regardless of the resolved WFO value.
- If the resolved WFO option is N and neither one of conditions two or three is true, the system attempts to resynchronize once. If it is not successful, the system signals STATUS message CPF83E6 to the application to indicate that resynchronization is in progress.

  Because CPF83E6 is a STATUS message, it only has an effect if the application is monitoring for it. Normally, your application can treat this message as an informational message. The systems that are participating in the transaction attempt to resynchronize the transaction until the failure is repaired. These subsequent resynchronization attempts are performed in the database server jobs. If a subsequent resynchronization attempt that is performed in a database server job fails, the message CPI83D0 is sent to QSYSOPR.

Wait for outcome value: Yes

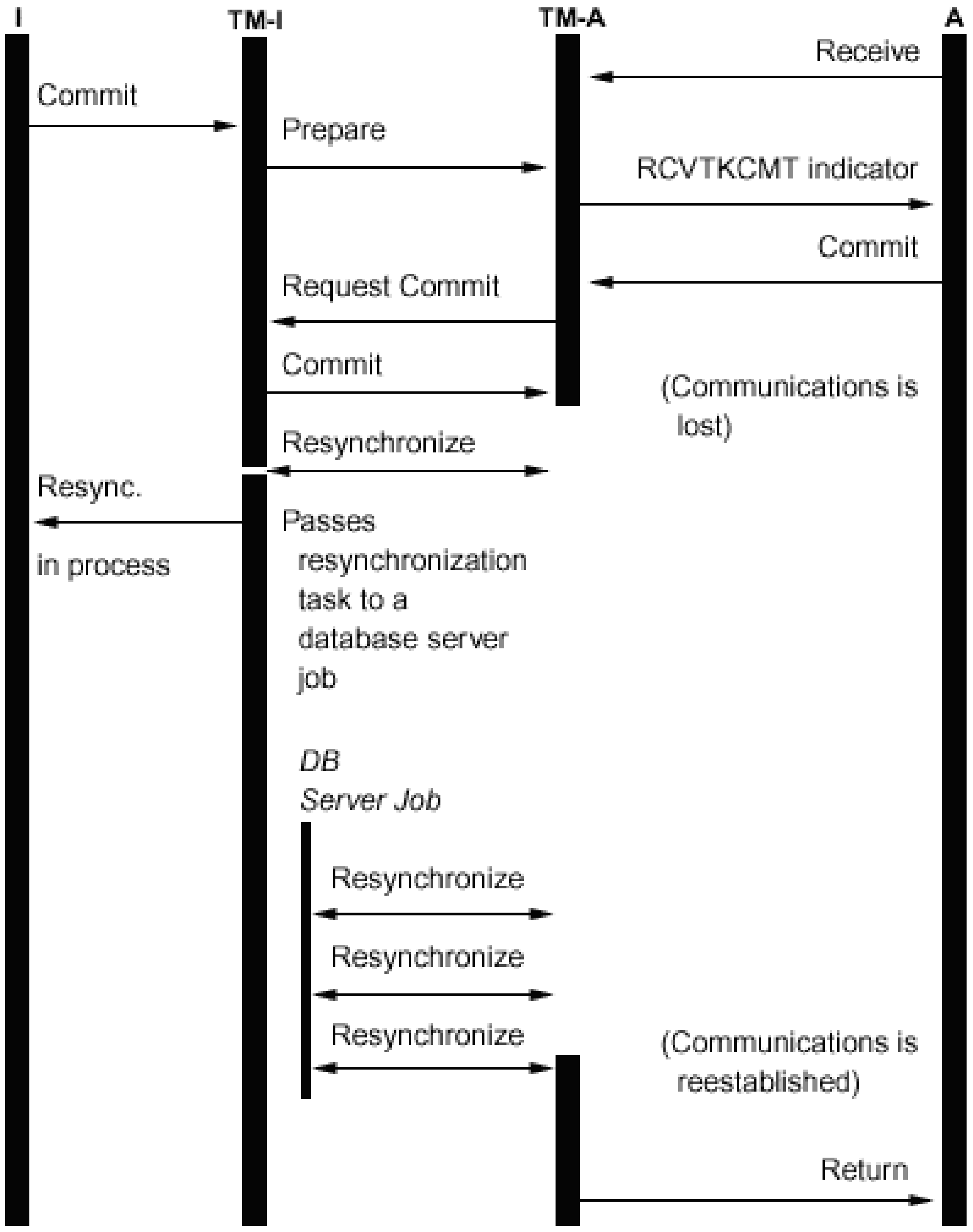
In the following figure, the commitment definition for the initiator (I) uses the default value of Y (Yes) for the `Wait for outcome` option. When communications between TM-I and TM-A is lost, both application A and application I wait until the transaction is resynchronized.



Wait for outcome value: No

In the following figure, the commitment definition for the initiator has the resolved WFO set to N (No). TM-A meets condition 3 in the preceding list, while TM-I meets condition 4. Control is returned to application I after one attempt to resynchronize with TM-A. A database server job attempts to resynchronize. Application I never receives the return indicator when the commit request has completed successfully. Control is not returned to the agent application (A) until after communications is reestablished. This depends on the timing of the failure. In this case, the communications failure occurs before the commit message is received from the initiator, leaving TM-A in doubt as to whether to commit or rollback. When the transaction manager is in doubt, it retains control until the resynchronization is completed, regardless of the resolved WFO value at that system.

If you want the applications at all systems to continue before resynchronization completes, you must either change the resolved WFO option to N (No) on all systems, or set the initiator to N and the rest of the systems to U (No or Inherit from Initiator). But remember that the resolved WFO option is ignored when the transaction manager is in doubt as to whether to commit or rollback, and always waits until resynchronization completes before returning control.



When a connection is made to a remote relational database, and no protected conversations have already been started, the system implicitly changes the `Wait for outcome` value to N. The reason for this is that the performance of commit operations is improved when the `Wait for outcome` value is N and the remote system supports presumed abort. This implicit change of the `Wait for outcome` value is only performed for DRDA and DDM applications. APPC applications use the default `Wait for outcome` value of Y unless they call the QTNCHGCO API to change it.

Related concepts

"States of the transaction for two-phase commitment control" on page 28
A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

"Commitment control errors" on page 99
When you use commitment control, it is important to understand which conditions cause errors and which do not.

Related reference

Change Commitment Options (QTNCHGCO) API

Commitment definition for two-phase commit: Indicate OK to leave out:

Normally, the transaction manager at every location in the transaction program network participates in every commit or rollback operation. To improve performance, you can set up some or all locations in a transaction to allow the transaction manager to indicate OK to leave out.

Note: The Indicate OK to leave out option does not apply if you are using a DRDA distributed unit of work over TCP/IP connection.

If no communications flows are sent to the location during a transaction, the location is left out when a commit or rollback operation is performed. This improves overall performance because the communications flows that normally occur during the commit or rollback are eliminated during transactions in which no data is sent to one or more remote locations.

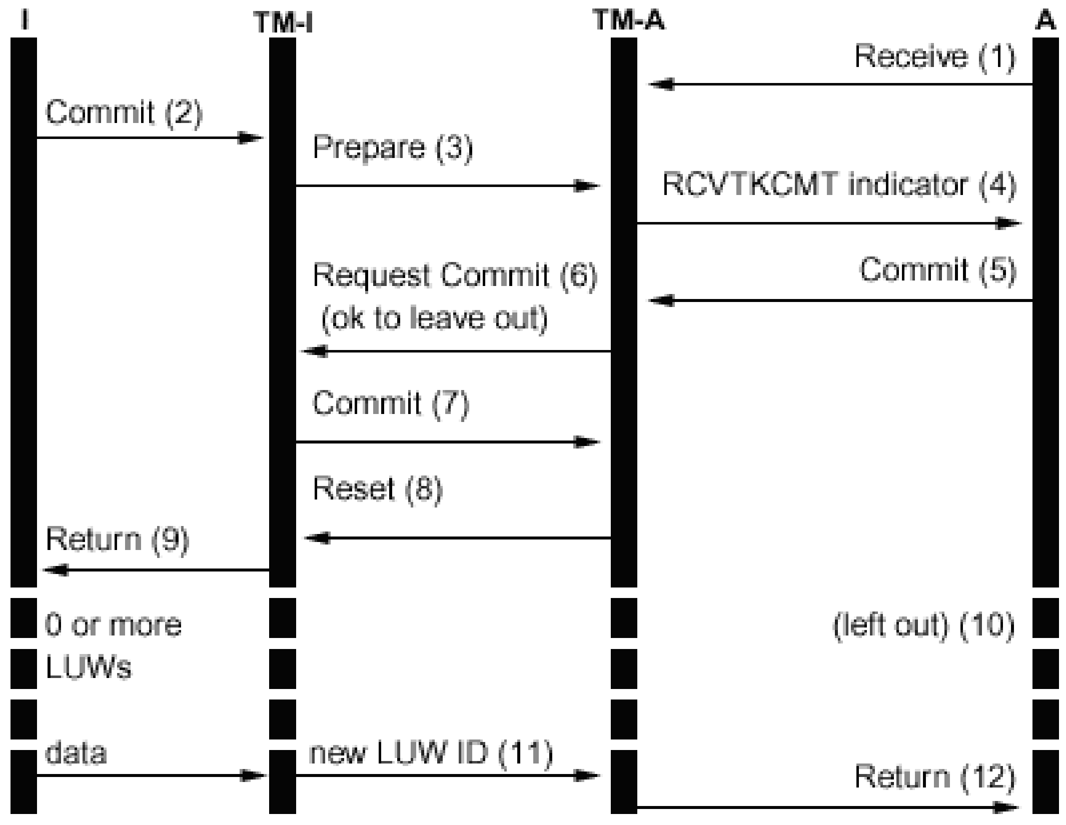
After you start commitment control, you can use the Change Commitment Options (QTNCHGCO) API to change the OK to leave out option to Y (Yes). You might want to do this if one or more remote systems often are not involved in a transaction.

If your commitment definition is set up to indicate OK to leave out, the application waits for the next message flow from another location.

The OK to leave out option is intended for applications that are client/server in nature. If the only purpose of program A is to satisfy requests from program I and not to do any independent work, then it is appropriate to allow the OK to leave out option for program A.

Flow of commit processing without last agent optimization when agent votes OK to leave out

The following figure shows the flow of messages among the application programs and the transaction managers when an application program issues a commit instruction without last agent optimization when the agent indicates OK to leave out. Both the initiator application program and the agent application programs are unaware of the two-phase commit processing. The numbers in parentheses () in the figure correspond to the numbered items in the description that follows.



Legend

I	=	Initiator (Application that initiates commit request)
TM-I	=	Transaction manager for initiator
A	=	Agent (Application that receives commit request)
TM-A	=	Transaction manager for agent

Here is a description of the events for normal processing without last agent optimization when the agent votes OK to leave out. This describes a basic flow. The sequence of events can become much more complex when the transaction program network has multiple levels or when errors occur.

1. Application program A does a receive request to indicate that it is ready to receive a request from program I.
2. The initiator application (I) issues a commit instruction.
3. The transaction manager for the initiator (TM-I) takes the role of initiator for this transaction. It starts the prepare wave by sending a prepare message to all the other locations that are participating in the transaction.
4. The transaction managers for every other location take the role of agent (TM-A). The application program A is notified by TM-A that a request to commit has been received. For ICF files, the notification is in the form of the Receive Take Commit (RCVTKCMT) ICF indicator being set on.
5. The application program A responds by issuing a commit instruction (or a rollback instruction). This is the application program's vote.
6. If application program A has used the Change Commitment Options API (QTNCHGCO) to set the OK to leave out commitment option to Y, an indicator is sent when the agent (TM-A) responds to the initiator (TM-I) with a request commit message.

   Note: Any change to the OK to leave out commitment option does not take effect until the next successful commit operation.

7. When the initiator (TM-I) receives all the votes, the TM-I sends a commit message. This starts the committed wave.
8. Each agent (TM-A) commits and responds with a reset message.
9. A return is sent to the application program (I) to indicate that the transaction is complete at the initiator.
10. Any number of transactions might occur on TM-I, none of which requires changes to TM-A or data from TM-A. TM-A is not included in these transactions.
11. The next time the initiator (TM-I) issues a message to the agent (A), a new transaction ID is sent with the message. If the initiator performs any commit or rollback operations before sending a message to the agent, no messages are sent to the agent during those operations (the agent is 'left out' of those commits or rollbacks). Because one or more transactions might have been committed or rolled back at the initiator while the agent was left out, the initiator must communicate its current transaction ID when the next message is sent to the agent.
12. A return is sent to the application program (A) to indicate that the original commit is complete and that it is participating in the current transaction.

Related concepts

"Optimizing performance for commitment control" on page 64
Using commitment control requires resources that can affect system performance. Several factors affect system performance regarding commitment control.

Commitment definition for two-phase commit: Not select a last agent:

By default, the transaction manager for the initiator is free to select any agent as a last agent during a commit operation.

Note: The Not select last agent option does not apply if you are using a DRDA distributed unit of work over TCP/IP connection.

In case of a multi-level tree, any agent selected as a last agent by its initiator is also free to select a last agent of its own. Performance is improved when a last agent is selected during the commit operation because two communications flows are eliminated between an initiator and its last agent (the prepare phase is eliminated for these systems).

However, when the initiator sends the request commit to its last agent, it must wait until it has received the last agent's vote before it can continue. This is regardless of the Wait for outcome value for the commitment definition. During normal, error-free commit processing, this is not an issue. But, if an error occurs during this window, the initiator cannot continue until resynchronization completes. If the initiator application is handling requests from a user at a terminal, this can be a usability consideration.

You must consider whether the improved performance during normal commit operations is more important than the impact on usability when such an error occurs. Note that if the error occurs before the request commit is sent to the last agent, the LUW will immediately roll back and the initiator will not wait. Therefore, the window when an error can cause the initiator to wait is quite small, so such an error is rare.

If you decide that the usability impact is not worth the improved performance, you can change your commitment definitions to not select a last agent. After you start commitment control, you can use the Change Commitment Options (QTNCHGCO) API to change the `Last agent permitted` option to N.

Related reference

Change Commitment Options (QTNCHGCO) API

Vote reliable effect on flow of commit processing:

Vote reliable is an optimization that improves performance by returning earlier to the initiator application after a commit operation and eliminating one message during a commit operation.

There is no explicit vote reliable optimization for Distributed Relational Database Architecture (DRDA) distributed unit of work over TCP/IP. However, i5/OS operating system never requests a reset (forget) confirmation for TCP/IP connections. Therefore, a reset (forget) is always implied for TCP/IP connections.

After you start commitment control, you can use the Change Commitment Options (QTNCHGCO) API to change the `Accept vote reliable` option to Y.

Vote reliable can be thought of as a promise by an agent to its initiator that no heuristic decisions will be made at the agent if communications failure occurs while the agent is in doubt. An agent that is using the vote reliable optimization sends an indicator to the initiator during the prepare wave of the commit. If the initiator is also using the vote reliable optimization, it then sends an indicator to the agent that no reset is required in response to the commit message. This eliminates the reset message, and allows the transaction manager to return to the application at the initiator as soon as the commit message is sent.

Consider using the vote reliable optimization if the following conditions are true:

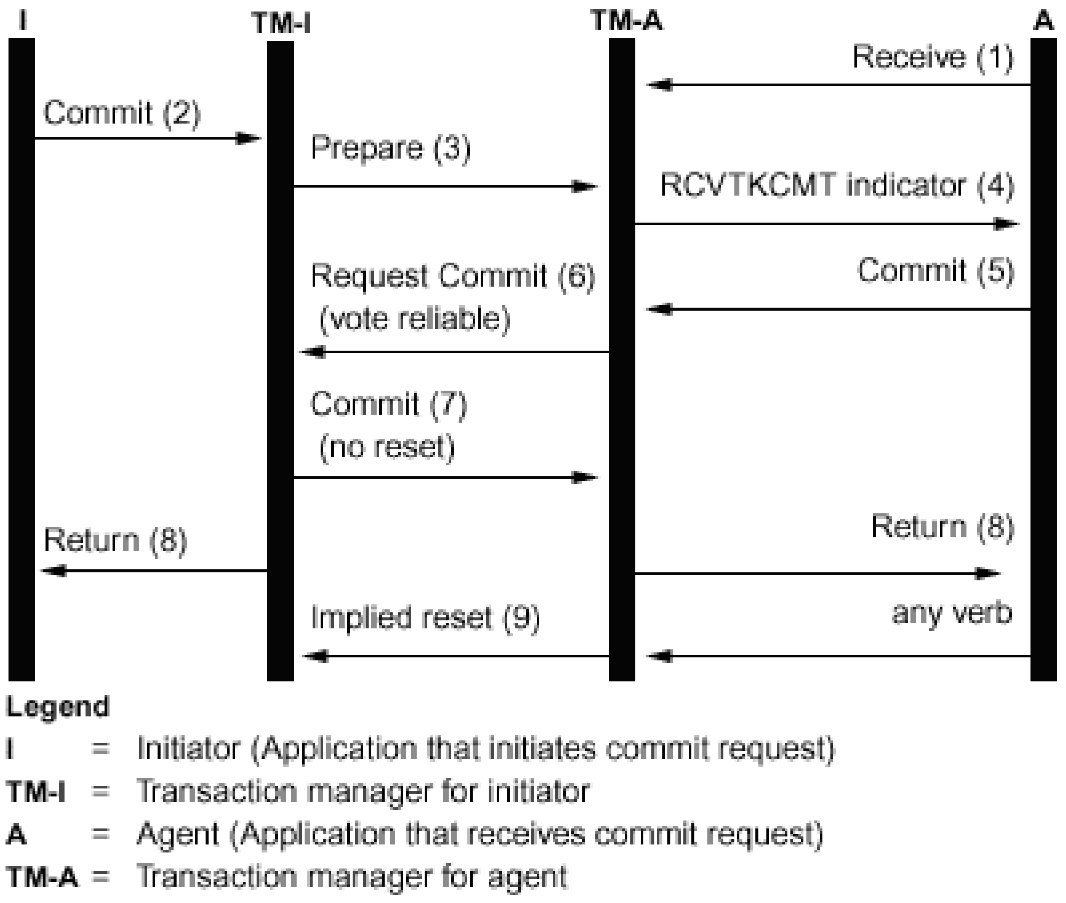
- It is unlikely that a heuristic decision is made at an in doubt agent in the event of a systems or communications failure unless the failure cannot be repaired.
- Your program logic does not need the results of previous transactions to ensure that your database files remain synchronized.

The vote reliable optimization is used by i5/OS operating system only if all the following conditions are true:

- The initiator and agent locations support the presumed abort level of commitment control.
- The initiator location accepts the vote reliable indication from the agent. On i5/OS initiators, this depends on the value of two commitment options:
  - The value of the Wait for outcome commitment option must be No (Yes is the default).
  - The value of the Accept vote reliable commitment option must be Yes (Yes is the default).
- The agent location votes reliable during the prepare wave. i5/OS agents always vote reliable. This is because heuristic decisions can be made only through a manual procedure that warns of the possible negative side-effects of making a heuristic decision.

Flow of commit processing with vote reliable optimization

The following figure shows the flow of messages among the application programs and the transaction managers when the vote reliable optimization is used. Both the initiator application program and the agent application programs are unaware of the two-phase commit processing. The numbers in parentheses () in the figure correspond to the numbered items in the description that follows.


I
TM-I
TM-A
A
Receive (1)
Commit (2)
Prepare (3)
RCVTKCMT indicator (4)
Commit (5)
Request Commit (6) (vote reliable)
Commit (7) (no reset)
Return (8)
Return (8)
Implied reset (9)
any verb
Legend
I = Initiator (Application that initiates commit request)
TM-I = Transaction manager for initiator
A = Agent (Application that receives commit request)
TM-A = Transaction manager for agent

The following list describes the events for normal processing without last agent optimization when the agent votes reliable. This describes a basic flow. The sequence of events can become much more complex when the transaction program network has multiple levels or when errors occur.

1. Application program A does a receive request to indicate that it is ready to receive a request from program I.
2. The initiator application (I) issues a commit instruction.
3. The transaction manager for the initiator (TM-I) takes the role of initiator for this transaction. It starts the prepare wave by sending a prepare message to all the other locations that are participating in the transaction.
4. The transaction managers for every other location take the role of agent (TM-A). The application program A is notified by TM-A that a request to commit has been received. For ICF files, the notification is in the form of the Receive Take Commit (RCVTKCMT) ICF indicator being set on.
5. The application program A responds by issuing a commit instruction (or a rollback instruction). This is the application program's vote.
6. The agent (TM-A) responds to the initiator (TM-I) with a request commit message. i5/OS systems send a vote reliable indicator with the request commit.
7. When the initiator (TM-I) receives all the votes, the TM-I sends a commit message. If the Wait for outcome commitment option is N (No) and the Accept vote reliable commitment option is Y (Yes), a no reset indicator is sent with the commit message. This tells the agent that no reset message is required in response to the commit.
8. The transaction is complete. A return is sent to the application programs (I and A). This return indicates that the commit operation was successful. If a heuristic damage occurs at system A due to a

heuristic decision being made before the committed message is received, application I is not informed. Instead, a message is sent to the QSYSOPR message queue. However, application A receives the heuristic damage indication.

9. The next time the agent (TM-A) sends any message to the initiator (TM-I), either a data flow or a commitment instruction, an implied reset indicator is sent with the message to inform TM-I that TM-A completed the commit successfully. The reason for this is that TM-I must retain information about the completed transaction until it has confirmed that TM-A successfully received the commit message in step 7 on page 41

   Related reference

   Change Commitment Options (QTNCHGCO) API

XA transaction support for commitment control

DB2 Universal Database (UDB) for iSeries can participate in X/Open global transactions.

The Open Group has defined an industry-standard model for transactional work that allows changes made against unrelated resources to be part of a single global transaction. An example of this is changes to databases that are provided by two separate vendors. This model is called the *X/Open Distributed Transaction Processing* model.

The following publications describe the X/Open Distributed Transaction Processing model in detail:

- *X/Open Guide*, February 1996, Distributed Transaction Processing: Reference Model, Version 3 (ISBN:1-85912-170-5, G504), The Open Group.
- *X/Open CAE Specification*, December 1991, Distributed Transaction Processing: The XA Specification (ISBN:1-872630-24-3, C193 or XO/CAE/91/300), The Open Group.
- *X/Open CAE Specification*, April 1995, Distributed Transaction Processing: The TX (Transaction Demarcation) Specification (ISBN:1-85912-094-6, C504), The Open Group.

Be familiar with the information in these books, particularly the XA Specification, before attempting to use the XA transaction support provided by DB2® UDB for iSeries. You can find these books at the Open Group Web site.

There are five components to the DTP model:

Application Program (AP)
: It implements the required function of the user by specifying a sequence of operations that involves resources such as databases. It defines the start and end of global transactions, accesses resources within transaction boundaries, and normally makes the decision whether to commit or roll back each transaction.

Transaction Manager (TM)
: It manages global transactions and coordinates the decision to start them and commit them, or roll them back in order to ensure atomic transaction completion. The TM also coordinates recovery activities with the RMs after a component fails.

Resource Manager (RM)
: It manages a defined part of the computer's shared resources, such as a database management system. The AP uses interfaces defined by each RM to perform transactional work. The TM uses interfaces provided by the RM to carry out transaction completion.

Communications Resource Manager (CRM)
: It allows an instance of the model to access another instance either inside or outside the current TM domain. CRMs are outside the scope of DB2 UDB for iSeries and are not discussed here.

Communication Protocol
: The protocols used by CRMs to communicate with each other. This is outside the scope of DB2 UDB for iSeries and is not discussed here.

The XA Specification is the part of the DTP model that describes a set of interfaces that is used by the TM and RM components of the DTP model. DB2 UDB for iSeries implements these interfaces as a set of UNIX® platform-style APIs and exit programs. See XA APIs for detailed documentation of these APIs and for more information about how to use DB2 UDB for iSeries as an RM.

iSeries Navigator and XA transactions

iSeries Navigator supports the management of XA transactions as *Global Transactions*.

A Global Transaction might contain changes both outside and within DB2 UDB for iSeries. A global transaction is coordinated by an external Transaction Manager using the Open Group XA architecture, or another similar architecture. An application commits or rolls back a global transaction using interfaces provided by the Transaction Manager. The Transaction Manager uses commit protocols defined by the XA architecture, or another architecture, to complete the transaction. DB2 UDB for iSeries acts as an XA Resource Manager when participating in a global transaction. There are two types of global transactions:

- **Transaction-scoped locks:** Locks acquired on behalf of the transaction are scoped to the transaction. The transaction can move from one job or thread to another.
- **Job-scoped locks:** Locks acquired on behalf of the transaction are scoped to the job. The transaction cannot move from the job that started it.

Considerations for XA transactions

| The XA APIs for transaction-scoped locks are recommended for new users of the XA transaction support.
| The XA APIs for job-scoped locks will continue to be supported, but no longer have any advantages over
| the XA APIs for transaction-scoped locks. The XA APIs for transaction-scoped locks have fewer
| restrictions and better performance in the following situations:

- If multiple SQL connections are ever used to work on a single XA transaction branch.
- If a single SQL connection is used to work on multiple, concurrent XA transaction branches.

In these situations, a separate job must be started to run XA transaction branches when you use the XA APIs for Job Scoped Locks.

Understand the following considerations and restrictions before using DB2 UDB for iSeries as a RM. The term *thread* refers to either a job that is not thread capable, or a single thread within a thread capable job.

The following considerations apply to both transactions with transaction-scoped locks and transactions with job-scoped locks unless noted otherwise.

DB2 UDB for iSeries considerations

| • XA transactions against a local database must be performed in jobs that are running in SQL server
| mode, which means that applications are limited to SQL interfaces when making changes to DB2 UDB
| for iSeries during an XA transaction. For such transactions, if the xa_open() or db2xa_open() API is
| used in a job that is not already running in SQL server mode, SQL server mode is implicitly started.

- XA transactions against a remote database are required to use SQL server mode when you use the XA APIs for job-scoped locks. However, server mode is optional for XA transactions against a remote database when you use the XA APIs for transaction-scoped locks. Furthermore, changes to DDM files using traditional i5/OS database access methods are allowed within XA transactions against a remote database when SQL server mode is not used.
- Any errors that are detected by DB2 UDB for iSeries during the XA API invocations are reported through return codes by the XA specification. Diagnostic messages are left in the job log when the meaning of the error can not be clear from the return code alone.

Embedded SQL considerations

- In order to use a Structured Query Language (SQL) connection for XA transactions, you must use the xa_open() or db2xa_open() application programming interface (API) before the SQL connection is made. The relational database that will be connected to must be passed to the xa_open() or db2xa_open() API by the xainfo parameter. The user profile and password to be used in the job that the connection is routed to might be passed to the xa_open() or db2xa_open() API. If it is not passed, the profile uses the one that was specified or used as the default during the connection attempt.

  Note: The following consideration applies only to transactions with job-scoped locks.
- If embedded SQL is used to perform XA transactions, the work performed for each connection is routed to a different job, even if the connections are made in the same thread. This is different than SQL server mode without XA, where work performed for all connections in a single thread is routed to the same job. This is because the XA specification requires a separate prepare, commit or rollback call for each resource manager instance.

  Note: The following consideration applies only to transactions with job-scoped locks.
- If embedded SQL is used to perform XA transactions, only one connection per relational database can be made per thread. Whenever the thread is not actively associated with a transaction branch, work requested over one of the thread's connections will cause the RM to use the TM's ax_reg() exit program to determine whether the work is to start, resume or join a transaction branch.

  If the work is to start a transaction branch, it is performed over that thread's connection to the corresponding relational database.

  If the work is to join a transaction branch, it is rerouted over the connection to the corresponding relational database that was made in the thread that started the transaction branch. Note that the system does not enforce that the user profile for that connection is the same as the one for the connection of the joining thread. The TM is responsible to ensure that this is not a security concern. Typical TMs use the same user profile for all connections. This user profile is authorized to all data that is managed by the TM. Further security of access to this data is managed by the TM or AP instead of using the standard i5/OS security techniques.

  Note: The following consideration applies only to transactions with job-scoped locks.
- If the work is to resume a transaction branch, the connection that is used depends on whether the suspended transaction branch association was established by starting or joining the transaction branch.

  Subsequent work is performed over the same connection until the db2xa_end() API is used to suspend or end the thread's association with that transaction branch.

CLI considerations

- If the CLI is used to perform XA transactions, more than one connection might be made in the same thread after the db2xa_open() API is used. The connections can be used in other threads to perform XA transactions, as long as those other threads first use the db2xa_open() API with the same xainfo parameter value.

  Note: The following consideration applies only to transactions with job-scoped locks.
- If the CLI is used to perform XA transactions, the connection that is used to start a transaction branch must be used for all work on that transaction branch. If another thread is to join the transaction branch, the connection handle for the connection used to start the transaction branch must be passed to the joining thread so that it can perform work over that same connection. Likewise, if a thread is to resume the transaction branch, the same connection must be used.

  Because CLI connection handles cannot be used in a different job, the join function is limited to threads running in the same job that started the transaction branch when the CLI is used.

Remote relational database considerations

Note: These considerations for a remote relational database apply only to transactions with job-scoped locks.

- XA connections to a remote relational database are supported only if the relational database resides on a system that supports Distributed Unit of Work (DUW) DRDA connections. This includes System i™ products that run Distributed Relational Database Architecture (DRDA) over SNA LU 6.2 conversations, or that use V5R1 or later when running DRDA using TCP/IP connections. This also includes other platforms that support DRDA over SNA LU 6.2 or that support the XA protocol using DRDA over TCP/IP.
- Before using the XA join function, the db2xa_open() API must be used in the joining thread. The same relational database name and RMID must be specified on the db2xa_open() API in both the thread that started the transaction branch and the joining thread. If the transaction branch is active when a join is attempted, the joining thread is blocked. The joining thread remains blocked until the active thread suspends or ends its association with the transaction branch.

Recovery consideration

- The manual heuristic commit and rollback support that is provided for all commitment definitions can be used if it becomes necessary to force a transaction branch to commit or roll back while it is in a prepared state.

Transaction branch considerations

- Information about XA transaction branches is shown as part of the commitment control information displayed by iSeries Navigator and the Work with Job (WRKJOB), Display Job (DSPJOB), and Work with Commitment Definition (WRKCMTDFN) commands. The TM name, transaction branch state, transaction identifier and branch qualifier are all shown. The commitment definitions related to all currently active XA transactions can be displayed by using the command `WRKCMTDFN JOB(*ALL) STATUS(*XOPEN)` or by displaying the **Global Transactions** in iSeries Navigator.

  Note: The following item applies only to transactions with job-scoped locks.
- If an association between a thread and an existing transaction branch is suspended or ended using the db2xa_end() API, the thread might start a new transaction branch. If the connection used to start the new transaction branch was used earlier to start a different transaction branch and the thread's association with that transaction branch has been ended or suspended by the db2xa_end() API, a new SQL server job might be started. A new SQL server job is needed only if the first transaction branch has not yet been completed by the db2xa_commit() or db2xa_rollback() API. In this case, another completion message SQL7908 is sent to the job log identifying the new SQL server job, just as the connection's original SQL server job was identified when the connection was established. All SQL requests for the new transaction branch are routed to the new SQL server job. When the transaction branch is completed by the db2xa_commit() or db2xa_rollback() API, the new SQL server job is recycled and returned to the prestart job pool.
- A transaction branch is marked `Rollback Only` in the following situations only for the XA transactions for job-scoped locks:
  - A thread ends when it is still associated with the transaction branch. This includes a thread ending as the result of process termination.
  - The system fails.
- With XA transactions for transaction-scoped locks, a transaction branch is rolled back by the system if any threads are still associated with it when any of the following situations occur:
  - The connection that is related to the transaction branch is ended.
  - The job that started the transaction branch is ended.
  - The system fails.

  Note: The following consideration applies only to transactions with job-scoped locks.

- There is one situation where a transaction branch will be rolled back by the system, regardless of whether there are still associated threads. This occurs when the SQL server job that the connection's work is being routed to is ended. This can only happen when the End Job (ENDJOB) CL command is used against that job.
- A transaction branch is not affected if no threads have an active association with it when any of the following situations occur. The TM can commit or roll back the transaction branch from any thread that has used the xa_open() or db2xa_open() API with the same xainfo parameter value that was specified in the thread that started the transaction branch.
  - The connection that is related to the transaction branch is ended.
  - A thread or job that performed work for the transaction branch uses the xa_close() or db2xa_close() API.
  - The system fails. In this case, the transaction branch is not affected only if it is in prepared state. If it is in idle state, the system rolls it back.
- When the transaction identifier (XID) of two XA transaction branches have the same global transaction identifier (GTRID), but different branch qualifiers (BQUALs), they are said to be *loosely coupled*. By default, loosely coupled transaction branches do not share locks. However, when using the XA APIs for transaction-scoped locks, there is an option that allows loosely coupled transactions to share locks.

Related concepts

"Considerations for XA transactions" on page 22
In the XA environment, each database is considered a separate resource manager. When a transaction manager wants to access two databases under the same transaction, it must use the XA protocols to perform two-phase commit with the two resource managers.

The Open Group Web site

"SQL server mode and thread-scoped transactions for commitment control"
Commitment definitions with job-scoped locks are normally scoped to an activation group.

Related tasks

"When to force commit and rollback operations and when to cancel resynchronization" on page 107
The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

SQL server mode and thread-scoped transactions for commitment control

Commitment definitions with job-scoped locks are normally scoped to an activation group.

If a job is multithreaded, all threads in the job have access to the commitment definition and changes made for a particular transaction can be spread across multiple threads. That is, all threads whose programs run in the same activation group participate in a single transaction.

There are cases where it is desirable for transactional work to be scoped to the thread, rather than an activation group. In other words, each thread has its own commitment definition and transactional work for each commitment definition is independent of work performed in other threads.

This is supported by DB2 Universal Database (UDB) for iSeries by using the Change Job (QWTCHGJB) API to change the job to run in SQL server mode. When an SQL connection is requested in SQL server mode, it is routed to a separate job. All subsequent SQL operations that are performed for that connection are also routed to that job. When the connection is made, completion message SQL7908 is sent to the SQL server mode job's job log indicating which job the SQL requests are being routed to. The commitment definition is owned by the job that is indicated in this message. If errors occur, it might be necessary to look at the job logs for both jobs to understand the source of the problem because no real work is being done in the job performing the SQL statements.

When running in SQL server mode, only SQL interfaces can be used to perform work under commitment control. Embedded SQL or Call Level Interface (CLI) can be used. All connections made through embedded SQL in a single thread are routed to the same back-end job. This allows a single commit request to commit the work for all the connections, just as it can be in a job that is not running in SQL server mode. Each connection made through the CLI is routed to a separate job. The CLI requires work that is performed for each connection to be committed or rolled back independently.

You cannot perform the following operations under commitment control when running in SQL server mode:

- Record changes that are made with interfaces that are not SQL interfaces
- Changes to DDM files
- Changes to API commitment resources

You cannot start commitment control directly in a job running in SQL server mode.

Related concepts

"XA transaction support for commitment control" on page 42
DB2 Universal Database (UDB) for iSeries can participate in X/Open global transactions.

Running DB2 UDB CLI in server mode

Starting DB2 CLI in SQL server mode

Restrictions for running DB2 UDB CLI in server mode

Related reference

Change Job (QWTCHGJB) API

Starting commitment control

To start commitment control, use the Start Commitment Control (STRCMTCTL) Command.

Note: Commitment control does not need to be started by SQL applications. SQL implicitly starts commitment control at connect time when the SQL isolation level is not *NONE.

When you use the STRCMTCTL command, you can specify these parameters.

Commit lock-level
Specify the lock-level with the LCKLVL parameter on the STRCMTCTL command. The level you specify becomes the default level of record locking for database files that are opened and placed under commitment control for the commitment definition.

Commit notify object
Use the NTFY parameter to specify the notify object. A notify object is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Commit scope parameter
Use the CMTSCOPE parameter to specify commit scope. When commitment control is started, the system creates a commitment definition. The commit scope parameter identifies the scope for the commitment definition. The default is to scope the commitment definition to the activation group of the program making the start commitment control request. The alternative scope is to the job.

Default journal parameter
You can specify a default journal when you start commitment control. You might use a default journal for these reasons:

- You want to capture transaction journal entries. These entries can assist you in analyzing the history of what resources are associated with a transaction. They are not used for applying and removing journaled changes. The omit journal entries (OMTJRNE) parameter determines whether the system writes transaction entries.
- You want to improve performance for jobs that close files and open them again within a routing step. If you close all the files assigned to a journal that is not the default journal, all the system information about the journal is removed from the routing step. If a file that is assigned to that journal is opened later, all the information about the journal must be created again. The system keeps information about the default journal with the commitment definition, whether any resources that are assigned to the journal are active.

Commit text parameter

Use the TEXT parameter to identify the specific text to be associated with a commitment definition when displaying information about the commitment definitions started for a job. If no text is specified, the system provides a default text description.

Omit journal entries parameter

If you specify a default journal to improve performance, you can use the OMTJRNE parameter to prevent the system from writing transaction journal entries. Having the system write transaction entries significantly increases the size of your journal receiver and degrades performance during commit and rollback operations.

Transaction entries can be useful when you are setting up and testing either your commitment control environment or a new application.

Transaction entries are written to the default journal regardless of the value of the OMTJRNE parameter under these conditions:

- A system error occurs during a commit or rollback operation.
- A manual change is made to a resource that participated in a transaction, and the change caused a heuristic mixed condition. See States of the transaction for two-phase commitment control for a description of the heuristic mixed condition. This type of manual change is called a heuristic decision.

You can use the information about what resources participated in the transaction to determine what action to take in these situations.

You can use the Journal entry information finder to show the layouts for the entry-specific data for transaction (commitment control) journal entries.

Related concepts

"States of the transaction for two-phase commitment control" on page 28
A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

Journal entry information finder

Related tasks

"When to force commit and rollback operations and when to cancel resynchronization" on page 107
The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

Related reference

Start Commitment Control (STRCMTCTL) command

Commit notify object

A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

The information used to identify the last successful transaction for a commitment definition is given by the *commit identification* that associates a commit operation with a specific set of committable resource changes.

The commit identification of the last successful transaction for a commitment definition is placed in the notify object only if the commitment definition does not end normally. This information can be used to help determine where processing for an application ended so that the application can be started again.

For independent disk pools, the notify object must reside on the same independent disk pool or independent disk pool group as the commitment definition. If you move the commitment definition to another independent disk pool or independent disk pool group, the notify object must also reside on that other independent disk pool or independent disk pool group. The notify object on the other independent disk pool or independent disk pool group is updated if the commitment definition ends abnormally. If the notify object is not found on the other independent disk pool or independent disk pool group, the update fails with message CPF8358.

If journaled resources participate in the current transaction and a commit operation is performed with a commit identification, the commit identification is placed in the commit journal entry (journal code and entry type of C CM) that identifies that particular transaction as being committed. A commit journal entry containing the commit identification is sent to each journal associated with resources that participated in the transaction.

The following table shows how you specify the commit identification and its maximum size. If the commit identification exceeds its maximum size, it is truncated when it is written to the notify object.

Language	Operation	Maximum characters in commit identification
CL	COMMIT command	3000 [1]
Integrated Language Environment (ILE) RPG	COMIT operation code	4000 [1]
PLI	PLICOMMIT subroutine	4000 [1]
ILE C	_Rcommit function	4000 [1]
ILE COBOL	COMMIT verb	Not supported
SQL	COMMIT statement	Not supported
Note: [1]If the notify object is a data area, the maximum size is 2000 characters.		

When a notify object is updated with the commit identification, it is updated as follows:

Database file
: If a database file is used as the notify object, the commit identification is added to the end of the file. Any existing records will be left in the file. Because several users or jobs can be changing records at the same time, each commit identification in the file contains unique information to associate the data with the job and commitment definition that failed. The file that serves can be journaled

Data area
: If a data area is used as the notify object, the entire content of the data area is replaced when the commit identification is placed in the data area. If more than one user or job is using the same program, only the commit identification from the last commitment definition that did not end normally will be in the data area. Consequently, a single data area notify object might not produce the correct information for starting the application programs again. To solve this problem, use a separate data area for each commitment definition for each workstation user or job.

Message queue

If a message queue is used as a notify object, message CPI8399 is sent to the message queue. The commit identification is placed in the second-level text for message CPI8399. As with using a database file for the notify object, the contents of each commit identification uniquely identify a particular commitment definition for a job so that an application program can be started again.

Related concepts

"Commitment control for batch applications" on page 24
Batch applications might or might not need commitment control. In some cases, a batch application can perform a single function of reading an input file and updating a master file. However, you can use commitment control for this type of application if it is important to start it again after an abnormal end.

"Example: Using a notify object to start an application" on page 86
When a program is started after an abnormal end, it can look for an entry in the notify object. If the entry exists, the program can start a transaction again. After the transaction has been started again, the notify object is cleared by the program to prevent it from starting the same transaction yet another time.

Commit lock level

The value you specify for the LCKLVL parameter on the Start Commitment Control (STRCMTCTL) command becomes the default level of record locking for database files that are opened and placed under commitment control for the commitment definition.

The default level of record locking cannot be overridden when opening local database files. However, database files accessed by SQL use the current SQL isolation level in effect at the time of the first SQL statement issued against it.

The lock level must be specified with respect to your needs, the wait periods allowed, and the release procedures used most often.

The following descriptions apply only to files that are opened under commitment control:

***CHG Lock Level**

Use this value if you want to protect changed records from changes by other jobs running at the same time. For files that are opened under commitment control, the lock is held for the duration of the transaction. For files not opened under commitment control, the lock on the record is held only from the time the record is read until the update operation is complete.

***CS Lock Level**

Use this value to protect both changed and retrieved records from changes by other jobs running at the same time. Retrieved records that are not changed are protected only until they are released, or a different record is retrieved.

The *CS lock level ensures that other jobs are not able to read a record for update that this job has read. In addition, the program cannot read records for update that have been locked with a record lock type of *UPDATE in another job until that job accesses a different record.

***ALL Lock Level**

Use this value to protect changed records and retrieved records that are under commitment control from changes by other jobs running under commitment control at the same time. Records that are retrieved or changed are protected until the next commit or rollback operation.

The *ALL lock level ensures that other jobs are not able to access a record for update that this job has read. This is different from normal locking protocol. When the lock level is specified as *ALL, even a record that is not read for update cannot be accessed if it is locked with a record lock type of *UPDATE in another job.

The following table shows the duration of record locks for files under and not under commitment control.

Request	LCKLVL parameter	Duration of lock	Lock type
Read-only	No commitment control	No lock	None
	*CHG	No lock	None
	*CS	From read to next read, commit, or rollback	*READ
	*ALL	From read to commit or rollback	*READ
Read for update then update or delete[1]	No commitment control	From read to update or delete	*UPDATE
	*CHG	From read to update or delete	*UPDATE
		Then from update or delete to next commit or rollback[2]	*UPDATE
	*CS	From read to update or delete	*UPDATE
		Then from update or delete to next commit or rollback[2]	*UPDATE
	*ALL	From read to update or delete	*UPDATE
		Then from update or delete to next commit or rollback[2]	
Read for update then release[1]	No commitment control	From read to release	*UPDATE
	*CHG	From read to release	*UPDATE
	*CS	From read to release, commit, or rollback	*UPDATE
		Then from release to next read, commit, or rollback	*UPDATE
	*ALL	From read to release, commit, or rollback	*UPDATE
		Then from release to next commit or rollback	
Add	No commitment control	No lock	None
	*CHG	From add to commit or rollback	*UPDATE
	*CS	From add to commit or rollback	*UPDATE
	*ALL	From add to commit or rollback	*UPDATE
Write direct	No commitment control	For duration of write direct	*UPDATE
	*CHG	From write direct to commit or rollback	*UPDATE
	*CS	From write direct to commit or rollback	*UPDATE
	*ALL	From write direct to commit or rollback	*UPDATE

Request	LCKLVL parameter	Duration of lock	Lock type
Notes: [1]If a commit or rollback operation is performed after a read-for-update operation but before the record is updated, deleted, or released, the record is unlocked during the commit or rollback operation. The protection on the record is lost as soon as the commit or rollback completes. [2]If a record is deleted but the commit or rollback has not yet been issued for the transaction, the deleted record does not remain locked. If the same or a different job attempts to read the deleted record by key, the job receives a record not found indication. However, if a unique keyed access path exists over the file, another job is prevented from inserting or updating a record with the same unique key value as that of the deleted record until the transaction is committed.			

A record lock type of *READ is obtained on records that are not read for update when the lock level is *CS or *ALL. This type of lock prevents other jobs from reading the records for update but does not prevent the records from being accessed from a read-only operation.

A record lock type of *UPDATE is obtained on records that are updated, deleted, added, or read for update. This type of lock prevents other jobs from reading the records for update, and prevents jobs running under commitment control with a record lock level of *CS or *ALL from accessing the records for even a read-only operation.

Programs that are not using commitment control can read records locked by another job, but cannot read records for update, regardless of the value specified for the LCKLVL parameter.

The lock level, specified for a commitment definition when commitment control is started for an activation group or for the job, applies only to opens associated with that particular commitment definition.

Note: The *CS and *ALL lock-level values protect you from retrieving a record that currently has a pending change from a different job. However, the *CS and *ALL lock-level values *do not* protect you from retrieving a record using a program running in one activation group that currently has a pending change from a program running in a different activation group within the same job.

Within the same job, a program can change a record that has already been changed within the current transaction as long as the record is accessed again using the same commitment definition. When using the job-level commitment definition, the access to the changed record can be made from a program running within any activation group that is using the job-level commitment definition.

Related concepts

"Considerations and restrictions for commitment control" on page 22
You need to be aware of these considerations and restrictions for commitment control.

Related reference

Start Commitment Control (STRCMTCTL) command

Ending commitment control

The End Commitment Control (ENDCMTCTL) command ends commitment control for either the job-level or activation-group-level commitment definition.

Issuing the ENDCMTCTL command indicates to the system that the commitment definition in use by the program making the request is to be ended. The ENDCMTCTL command ends only one commitment definition for the job and all other commitment definitions for the job remain unchanged.

If the activation-group-level commitment definition is ended, then programs running within that activation group can no longer make changes under commitment control, unless the job-level commitment definition is already started for the job. If the job-level commitment definition is active, then it is immediately made available for use by the programs running within the activation group that just ended commitment control.

If the job-level commitment definition is ended, then any program running within the job that was using the job-level commitment definition can no longer make changes under commitment control without first starting commitment control again with the STRCMTCTL command.

Before issuing the ENDCMTCTL command, the following conditions must be satisfied for the commitment definition to be ended:

- All files opened under commitment control for the commitment definition to be ended must first be closed. When ending the job-level commitment definition, this includes all files opened under commitment control by any program running in any activation group that is using the job-level commitment definition.
- All API commitment resources for the commitment definition to be ended must first be removed using the QTNRMVCR API. When ending the job-level commitment definition, this includes all API commitment resources added by any program running in any activation group that is using the job-level commitment definition.
- A remote database associated with the commitment definition to be ended must be disconnected.
- All protected conversations associated with the commitment definition must be ended normally using the correct synchronization level.

If commitment control is being ended in an interactive job and one or more committable resources associated with the commitment definition have pending changes, inquiry message CPA8350 is sent to the user asking whether to commit the pending changes, roll back the pending changes, or cancel the ENDCMTCTL request.

If commitment control is being ended in a batch job, and one or more closed files associated with the commitment definition to be ended still have pending changes, the changes are rolled back and a message is sent:

- CPF8356 if only local resources are registered
- CPF835C if only remote resources are registered
- CPF83E4 if both local and remote resources are registered

If a notify object is defined for the commitment definition being ended, it might be updated.

When an activation group that has an API registered as the last agent is ending, the exit program for the API is called to receive the commit or rollback decision. In this case, even though the activation group is ending normally, a rollback request can still be returned from the API exit program. Thus, the implicit commit operation might not be performed.

After the commitment definition has successfully ended, all the necessary recovery, if any, has been performed. No additional recovery is performed for the commitment resources associated with the commitment definition just ended.

After the commitment definition is ended, the job-level or activation-group-level commitment definition can then be started again for the programs running within the activation group. The job-level commitment definition can be started only if it is not already started for the job.

Although commitment definitions can be started and ended many times by the programs that run within an activation group, the amount of system resources required for the repeated start and end operations can cause a decrease in job performance and overall system performance. Therefore, it is recommended that a commitment definition be left active if a program to be called later will use it.

Related concepts

"Updates to the notify object" on page 59
The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

Related reference

End Commitment Control (ENDCMTCTL) command

System-initiated end of commitment control

The system can end commitment control, or perform an implicit commit or rollback operation. Sometimes the system-initiated end of commitment control is normal. Other times, commitment control ends with an abnormal system or job end.

Commitment control during activation group end

The system automatically ends an activation-group-level commitment definition when an activation group ends.

If pending changes exist for an activation-group-level commitment definition and the activation group is ending normally, the system performs an implicit commit operation for the commitment definition before it is ended. Otherwise, an implicit rollback operation is performed for the activation-group-level commitment definition before being ended if the activation group is ending abnormally, or if errors were encountered by the system when closing any files opened under commitment control scoped to the activation group.

Note: An implicit commit or rollback operation is never performed during activation-group end processing for the *JOB or *DFTACTGRP commitment definitions. This is because the *JOB and *DFTACTGRP commitment definitions are never ended due to an activation group ending. Instead, these commitment definitions are either explicitly ended with an ENDCMTCTL command or ended by the system when the job ends.

The system automatically closes any files scoped to the activation group when the activation group ends. This includes any database files scoped to the activation group opened under commitment control. The close for any such file occurs before any implicit commit operation that might be performed for the activation-group-level commitment definition. Therefore, any records that reside in an I/O buffer are first forced to the database before any implicit commit operation is performed.

As part of the implicit commit or rollback operation that might be performed, a call is made to the API commit and rollback exit program for each API commitment resource associated with the activation-group-level commitment definition. The exit program must complete its processing within 5 minutes. After the API commit and rollback exit program is called, the system automatically removes the API commitment resource.

If an implicit rollback operation is performed for a commitment definition that is being ended due to an activation group being ended, then the notify object, if one is defined for the commitment definition, might be updated.

Related concepts

"Updates to the notify object" on page 59
The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

Implicit commit and rollback operations

In some instances, a commit or rollback operation is initiated by the system for a commitment definition. These types of commit and rollback operations are known as *implicit commit and rollback requests*.

Typically, a commit or rollback operation is initiated from an application program using one of the available programming languages that supports commitment control. These types of commit and rollback operations are known as *explicit commit and rollback requests*.

The following two tables show what the system does when certain events occur related to a commitment definition that has pending changes. A commitment definition has pending changes if any of the following conditions is true:

- Any committable resource has been updated.
- A database file opened under commitment control has been read because reading a file changes the file position.
- The commitment definition has an API resource. Because changes to API resources are done by a user program, the system must assume that all API resources have pending changes.

The C CM (commit operation) journal entry and C RB (rollback operation) journal entry indicate whether the operation was explicit or implicit.

The following table shows the actions the system takes when a job ends, either normally or abnormally, based on the following situations:

- The state of the transaction.
- The action-if-end job value for the commitment definition.
- Whether an API resource is the last agent.

State	Last agent API	Action if Endjob[1] option	Commit or rollback operation
RST	N/A	N/A	If the commitment definition is not associated with an X/Open global transaction, an implicit rollback is performed. If the commitment definition is associated with an X/Open global transaction, the following events occur: • If the transaction branch state is not Active (S1), no action is performed and the transaction branch is left in the same state. • If the transaction branch state is Active (S1), an implicit rollback is performed.
PIP	N/A	N/A	If the commitment definition is not associated with an X/Open global transaction, an implicit rollback is performed. If the commitment definition is associated with an X/Open global transaction, the transaction branch is in the Idle (S2) state, and it is left in the Idle (S2) state.

State	Last agent API	Action if Endjob[1] option	Commit or rollback operation
PRP	N/A	WAIT	If the commitment definition is not associated with an X/Open[2] global transaction, the following occurs: • Resynchronization is started to receive the decision from the initiator of the commit operation. • The returned decision to commit or rollback is performed. It is considered an explicit operation.
PRP	N/A	C	If the commitment definition is not associated with an X/Open[2] global transaction, an implicit commit operation is performed.
		R	If the commitment definition is not associated with an X/Open global transaction, an implicit rollback operation is performed. If the commitment definition is associated with an X/Open global transaction, the following occurs: • If the job that started the transaction ends, the transaction is left in a prepared state until the XA TM either commits it or rolls it back. The XA transaction branch state will be left at Prepared (S3) in this case. • If the SQL server job that the transaction's work is being routed to is ended, a forced rollback is implicitly performed. The XA transaction branch state will be changed to Heuristically Completed (S5) in this case.
CIP	N/A	N/A	An explicit commit operation is performed.
LAP	NO	WAIT	1. Resynchronization to the last agent is used to retrieve the decision to commit or to roll back.
			2. The returned decision to commit or to roll back is performed. It is considered an explicit operation.
LAP	YES	WAIT	1. The last agent API is called to retrieve the commit or rollback decision.
			2. The commit or rollback operation is performed. It is considered an explicit operation.
LAP	N/A	C	An implicit commit operation is performed.
		R	An implicit rollback operation is performed.
CMT	N/A	N/A	A commit operation has already completed for this commitment definition and any downstream locations. The commit operation is complete.

State	Last agent API	Action if Endjob[1] option	Commit or rollback operation
VRO	N/A	N/A	The local and remote agents voted to read-only. All downstream agents must also have voted to read-only. No action is required.
RBR	N/A	N/A	A rollback operation is required. An explicit rollback operation is performed.

Note:

[1] You can change the `Action if Endjob` option with the Change Commitment Options (QTNCHGCO) API.

[2]If the commitment definition is associated with an X/Open global transaction, the following events occur:
- If the job that started the transaction ends, the transaction is left in a prepared state until the XA TM either commits it or rolls it back. The XA transaction branch state will be left at Prepared (S3) in this case.
- For transaction-scoped locks only, if the SQL server job that the transaction's work is being routed to is ended, a forced rollback is implicitly performed. The XA transaction branch state will be changed to Heuristically Completed (S5) in this case.

The following table shows the actions the system takes when an activation group ends and applies only to transactions with job-scoped locks. The system actions are based on the following items:
- The state of the transaction. (It is always reset (RST) when an activation group ends.)
- How the activation group ends-normally or abnormally.
- Whether an API resource is the last agent.

Note: If an API resource is registered as the last agent, this gives control of the commit or rollback decision to the last agent. The result of the decision is considered an explicit operation

State	Last agent API	Type of end	Commit or rollback operation
RST	No	Normal	An implicit commit operation is performed. If protected conversations exist, the commitment definition will become the root initiator of the commit operation.
RST	No	Abnormal	An implicit rollback is performed.
RST	Yes	Normal	The API exit program is called. The commit or rollback operation is determined by the API.
RST	Yes	Abnormal	The API exit program is called. The commit or rollback operation is determined by the API.

Related concepts

"Commitment control during abnormal system or job end" on page 58
The system ends all commitment definitions for a job when the job ends abnormally. These commitment definitions are ended during the job end processing. This topic applies only to commitment definitions with job-scoped locks.

Related reference

Change Commitment Options (QTNCHGCO) API

Commitment control during normal routing step end

The system ends all commitment definitions for a job when a routing step is normally ended.

Note: The following information applies only to commitment definitions with job-scoped locks.

A routing step ends normally by one of the following situations:

- A normal end for a batch job.
- A normal sign-off for an interactive job.
- The Reroute Job (RRTJOB), Transfer Job (TFRJOB), or Transfer Batch Job (TFRBCHJOB) command ends the current routing step and starts a new routing step.

Any other end of a routing step is considered abnormal and is recognized by a nonzero completion code in job completion message CPF1164 in the job log.

Before ending a commitment definition during routing step end, the system performs an implicit rollback operation if the commitment definition has pending changes. This includes calling the API commit and rollback exit program for each API commitment resource associated with the commitment definition. The exit program must complete its processing within 5 minutes. After the API commit and rollback exit program is called, the system automatically removes the API commitment resource.

If a notify object is defined for the commitment definition, it can be updated.

Related concepts

"Updates to the notify object" on page 59
The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

Commitment control during abnormal system or job end

The system ends all commitment definitions for a job when the job ends abnormally. These commitment definitions are ended during the job end processing. This topic applies only to commitment definitions with job-scoped locks.

If the system ends abnormally, the system ends all commitment definitions that were started and being used by all active jobs at the time of the abnormal system end. These commitment definitions are ended as part of the database recovery processing that is performed during the next IPL after the abnormal system end.

Attention: The recovery for commitment definitions refers to an abnormal end for the system or a job due to a power failure, a hardware failure, or a failure in the operating system or licensed internal code. You must not use the End Job Abnormal (ENDJOBABN) command to force a job to end abnormally. The abnormal end can result in pending changes for active transactions for the job you are ending to be partially committed or rolled back. The next IPL might attempt recovery for any partial transactions for the job ended with the ENDJOBABN command.

The outcome of commitment control recovery that the system performs during an IPL for a job that you end with the ENDJOBABN command is uncertain. It is because all locks for commitment resources are released when the job is ended abnormally. Any pending changes due to partial transactions are made available to other jobs. These pending changes can then cause other application programs to make additional erroneous changes to the database. Likewise, any ensuing IPL recovery that is performed later can adversely affect the changes made by applications after the job was ended abnormally. For example, an SQL table might be dropped during IPL recovery as the rollback action for a pending create table. However, other applications might have already inserted several rows into the table after the job was ended abnormally.

The system performs as follows for commitment definitions being ended during an abnormal job end or during the next IPL after an abnormal system end:

- Before ending a commitment definition, the system performs an implicit rollback operation if the commitment definition has pending changes, unless processing for the commitment definition was interrupted in the middle of a commit operation. If ended in the middle of a commit operation, the transaction might be rolled back, resynchronized, or committed, depending on its state. The processing to perform the implicit rollback operation or to complete the commit operation includes calling the API commit and rollback exit program for each API commitment resource associated with the commitment definition. After the API commit and rollback exit program is called, the system automatically removes the API commitment resource.

  Attention: Ending the job while a transaction is in doubt (transaction state is LAP or PRP) can cause inconsistencies in the database (changes might be committed on one or more systems and rolled back on other systems).

  – If the *Action if Endjob* commitment option is COMMIT, changes on this system are committed if the job is ended, without regard to whether changes on the other systems participating in the transaction are committed or rolled back.

  – If the *Action if Endjob* commitment option is ROLLBACK, changes on this system are rolled back if the job is ended, without regard to whether changes on the other systems participating in the transaction are committed or rolled back.

  – If the *Action if Endjob* commitment option is WAIT, the job will not end until resynchronization completes to the system that owns the commit or rollback decision. To make the job end before resynchronization is complete, a heuristic decision must be made and resynchronization must be canceled.

  Ending the job or system abnormally during a long-running rollback is not recommended. This causes another rollback to occur as the job ends (or during the next IPL if the system is ended). The subsequent rollback will repeat the work performed by the original rollback and take significantly longer to run.

- If a notify object is defined for the commitment definition, it might be updated.

If a process ends before commitment control is ended and protected conversations are still active, the commitment definition might be required to commit or roll back. The action is based on the State option and the Action if end job option for the commitment definition.

Related concepts

"Implicit commit and rollback operations" on page 55
In some instances, a commit or rollback operation is initiated by the system for a commitment definition. These types of commit and rollback operations are known as *implicit commit and rollback requests*.

"Updates to the notify object"
The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

Updates to the notify object

The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition.

For purposes of the notify object, these are considered uncommitted changes:

- An update to a record that is made under commitment control.
- A record that is deleted under commitment control.
- An object level change that is made to a local DDL object under commitment control.
- A read operation performed for a database file that was opened under commitment control. This is because file position is brought back to the last commitment boundary when a rollback operation is

performed. If you perform a read operation under commitment control, the file position is changed and therefore, an uncommitted change then exists for the commitment definition.
- A commitment definition with one of the following resources that are added is always considered to have uncommitted changes:
  - An API commitment resource
  - A remote Distributed Relational Database Architecture (DRDA *) resource
  - A Distributed Database Management Architecture (DDM) resource
  - An LU 6.2 resource

  This is because the system does not know when a real change is made to the object or objects that are associated with these types of resources. Types of committable resources has more information about how you add and work with these types of resources.

The system makes updates to the notify object, based on the following ways that a commitment definition can end:
- If a job ends normally and no uncommitted changes exist, the system does not place the commit identification of the last successful commit operation in the notify object.
- If an implicit commit operation is performed for an activation-group-level commitment definition when the activation group is ended, the system does not place the commit identification of the last successful commit operation in the notify object.

  Note: Implicit commit operations are never performed for the *DFTACTGRP or *JOB commitment definition
- If the system, job, or an activation group ends abnormally before the first successful commit operation for a commitment definition, the system does not update the notify object because there is no last commit identification. To differentiate between this condition and a normal program completion, your program must update the notify object with a specific entry before completing the first successful commit operation for the commitment definition.
- If an abnormal job end or an abnormal system end occurs after at least one successful commit operation, the system places the commit identification of that commit operation in the notify object. If the last successful commit operation did not specify a commit identification, then the notify object is not updated. For an abnormal job end, this notify object processing is performed for each commitment definition that was active for the job. For an abnormal system end, this notify object processing is performed for each commitment definition that was active for all jobs on the system.
- The system updates the notify object with the commit identification of the last successful commit operation for that commitment definition if all of the following events occur:
  - A nondefault activation group ends.
  - An implicit rollback operation is performed for the activation-group-level commitment definition.
  - At least one successful commit operation has been performed for that commitment definition.

  If the last successful commit operation did not specify a commit identification, then the notify object is not updated. An implicit rollback operation is performed for an activation-group-level commitment definition if the activation group is ending abnormally or errors occurred when closing the files that were opened under commitment control and that were scoped to that activation group. For more information about scoping database files to activation groups and how activation groups can be ended, see the reference book for the ILE language that you are using.
- If uncommitted changes exist when a job ends normally and at least one successful commit operation has been performed, the commit identification of the last successful commit operation is placed in the notify object and the uncommitted changes are rolled back. If the last successful commit operation did not specify a commit identification, then the notify object is not updated. This notify object processing is performed for each commitment definition that was active for the job when the job ended.
- If uncommitted changes exist when the ENDCMTCTL command is run, the notify object is updated only if the last successful commit operation specified a commit identification:

- For a batch job, the uncommitted changes are rolled back and the commit identification of the last successful commit operation is placed in the notify object.
- For an interactive job, if the response to inquiry message CPA8350 is to rollback the changes, the uncommitted changes are rolled back and the commit identification of the last successful commit operation is placed in the notify object.
- For an interactive job, if the response to inquiry message CPA8350 is to commit the changes, the system prompts for a commit identification to use and the changes are committed. The commit identification that is entered on the prompt display is placed in the notify object.
- For an interactive job, if the response to inquiry message CPA8350 is to cancel the ENDCMTCTL request, the pending changes remain and the notify object is not updated.

Related concepts

"Ending commitment control" on page 52
The End Commitment Control (ENDCMTCTL) command ends commitment control for either the job-level or activation-group-level commitment definition.

"Commitment control during activation group end" on page 54
The system automatically ends an activation-group-level commitment definition when an activation group ends.

"Commitment control during normal routing step end" on page 58
The system ends all commitment definitions for a job when a routing step is normally ended.

"Commitment control during abnormal system or job end" on page 58
The system ends all commitment definitions for a job when the job ends abnormally. These commitment definitions are ended during the job end processing. This topic applies only to commitment definitions with job-scoped locks.

"Types of committable resources" on page 11
This table lists the different types of committable resources, including FILE, Data Definition Language (DDL), distributed data management (DDM), logical unit (LU) 6.2, Distributed Relational Database Architecture (DRDA), API, and TCP.

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Commitment control recovery during initial program load after abnormal end

When you perform an initial program load (IPL) after your system ends abnormally, the system attempts to recover all the commitment definitions that were active when the system ended.

Likewise, when you vary on an independent disk pool, the system attempts to recover all the commitment definitions related to that independent disk pool that were active when it was varied off or ended abnormally.

The recovery is performed by database server jobs that are started by the system during IPL. Database server jobs are started by the system to handle work that cannot or must not be performed by other jobs.

The database server jobs are named QDBSRVnn, where `nn` is a two-digit number. The number of database server jobs depends on the size of your system. Likewise, the name of the database server job for an independent disk pool or independent disk pool group is QDBSxxxVnn, where `xxx` is the independent disk pol number and `nn` is a two-digit number. For example, QDBS035V02 can be the name of the database server job for independent disk pool 35.

States of the transaction for two-phase commitment control shows the actions that the system takes, depending on the state of the transaction when the failure occurred. For two states, PRP and LAP, the system action is in doubt.

Notes:

- The following applies only to commitment definitions with job-scoped locks.
- The transaction manager recovers commitment definitions associated with XA transactions (whether their locks are job-scoped or transaction-scoped) using XA APIs, not the resynchronization process described in this topic.

The system cannot determine what to do until it performs resynchronization with the other locations that participated in the transaction. This resynchronization is performed after the IPL or vary on operation completes.

The system uses the database server jobs to perform this resynchronization. The commitment definitions that need to be recovered are associated with the database server jobs. During the IPL, the system acquires all record locks and other object locks that were held by the commitment definition before the system ended. These locks are necessary to protect the local commitment resources until resynchronization is complete and the resources can be committed or rolled back.

Messages are sent to the job logs of the database server jobs to indicate the status of resynchronization with the remote locations. If the transaction is in doubt, resynchronization must be completed with the location that owns the decision for the transaction before local resources can be committed or rolled back.

When the decision for a transaction is made, the following messages might be sent to the job log for the database server job.

CPI8351
&1 pending changes being rolled back

CPC8355
Post-IPL recovery of commitment definition &8 for job &19/&18/&17 completed.

CPD835F
IPL recovery of commitment definition &8 for job &19/&18/&17 failed.

Other messages related to the recovery can also be sent. These messages are sent to the history (QHST) log. If errors occur, messages are also sent to the QSYSOPR message queue.

You can determine the progress of the recovery by using iSeries Navigator, by displaying the job log for the database server job, or by using the Work with Commitment Definitions (WRKCMTDFN) command. Although iSeries Navigator and the Work with Commitment Definitions display allow you to force the system to commit or roll back, you must use this only as a last resort. If you anticipate that all of the locations that participated in the transaction will eventually be returned to operation, you must allow the systems to resynchronize themselves. This ensures the integrity of your databases.

Related concepts

"States of the transaction for two-phase commitment control" on page 28
A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

Managing transactions and commitment control

By using these instructions, you can manage a system with commitment control.

Displaying commitment control information

iSeries Navigator can display information about all transactions (logical units of work) on the system. iSeries Navigator can also display information about the job, if any, associated with a transaction.

Note: These display operations do not display the isolation level for SQL applications.

To display information, proceed as follows:

1. From iSeries Navigator, expand the system you want to use.
2. Expand **Databases**.
3. Expand the system you want to work with.
4. Expand **Transactions**.

   Note: To view a transaction that is associated with an X/Open global transaction, expand **Global Transactions**. To view a DB2 Universal Database (UDB) managed transaction, expand **Database Transactions**.
5. Expand **Global Transactions** or **Database Transactions**.

This display shows the following information:

- Unit of Work ID
- Unit of Work State
- Job
- User
- Number
- Resynchronization in Progress
- Commitment Definition

Online help provides information about all the status displays and the fields on each display.

Related tasks

"Detecting deadlocks" on page 106
A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

"Recovering transactions after communications failure" on page 107
These instructions help you handle transactions performing work on a remote system after communication with that system fails.

"When to force commit and rollback operations and when to cancel resynchronization" on page 107
The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

Displaying locked objects for a transaction

You can display locked objects for global transactions with transaction-scoped locks only.

To display locked objects for a transaction, follow these steps:

1. From iSeries Navigator, expand the system you want to use.
2. Expand **Databases**.
3. Expand the system you want to work with.
4. Expand **Transactions**.
5. Expand **Global Transactions**.
6. Right-click the transaction that you want to work with and select **Locked Objects**.

Related tasks

"Detecting deadlocks" on page 106
A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

Displaying jobs associated with a transaction

To display jobs associated with a transaction, follow these steps.

1. In the iSeries Navigator window, expand the system you want to use.
2. Expand **Databases**.
3. Expand the system you want to work with.
4. Expand **Transactions**.
5. Expand **Global Transactions.** or **Database Transactions**.
6. Right-click the transaction that you want to work with and select **Jobs**.

For database transactions and global transactions with job-scoped locks, a list of the jobs associated with the transaction is displayed.

For global transactions with transaction-scoped locks, a list of jobs with this transaction object attached or waiting for this transaction object to be attached is displayed

Related tasks

"Detecting deadlocks" on page 106
A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

Displaying resource status of a transaction

To display the resource status of a transaction, follow these steps.

1. In the iSeries Navigator window, expand the system you want to use.
2. Expand **Databases**.
3. Expand the system you want to work with.
4. Expand **Transactions**.
5. Expand **Global Transactions** or **Database Transactions**.
6. Right-click the transaction that you want to work with and select **Resource status**.

Displaying transaction properties

To display transaction properties, follow these steps.

1. In the iSeries Navigator window, expand the system you want to use.
2. Expand **Databases**.
3. Expand the system you want to work with.
4. Expand **Transactions**.
5. Expand **Global Transactions** or **Database Transactions**.
6. Right-click the transaction that you want to work with and select **Properties**.

Optimizing performance for commitment control

Using commitment control requires resources that can affect system performance. Several factors affect system performance regarding commitment control.

Factor that does not affect performance

Open a file
If you open a file without specifying the commit open option, no additional system resource is used even if a commitment definition has been started. For more information about specifying the commit open option, see the appropriate high-level language reference manual.

Factors that degrade performance

Journal
: Journaling a file requires system resources. However, in most cases journaling performs better with commitment control than without commitment control. If you specify only after-images, commitment control changes this to both before-images and after-images while commitment control is in effect. Typically this is a space, not a performance, consideration.

Commit operation
: If any changes were made to journaled resources during the transaction, each commit of a transaction adds two entries to each journal related to those resources. The number of entries added can increase significantly for a large volume of small transactions. You might want to place the journal receivers in a separate disk pool from the journals.

Rollback operation
: Because commitment control must roll back the pending changes recorded in the database, additional system resources are required whenever a rollback occurs. Also, if record changes are pending, a rollback operation causes additional entries to be added to the journal.

Start Commitment Control (STRCMTCTL) and End Commitment Control (ENDCMTCTL) commands
: Additional overhead is incurred by the system each time a commitment definition is started using the STRCMTCTL command and is ended using the ENDCMTCTL command. Avoid using the STRCMTCTL and ENDCMTCTL commands for each transaction. Use them only when necessary. You can establish a commitment definition at the beginning of an interactive job and use it for the duration of the job.

Use more than one journal for commitment control transactions
: With two-phase commit, files that are opened under commitment control can be journaled to more than one journal. However, using more than one journal takes additional system resources to manage the commitment definition. Using more than one journal can also make recovery more complicated.

Record locking
: Record locking can affect other applications. The number of records locked within a particular job increases the overall system resources used for the job. Applications needing to access the same record must wait for the transaction to end.

Request SEQONLY(*YES)
: If you request the SEQONLY(*YES) option (by using the OVRDBF command or the application program implicitly attempts to use SEQONLY(*YES)) and the file is opened for input only under commitment control with LCKLVL(*ALL), the option is changed to SEQONLY(*NO). This option can affect the performance of input files because records will not be blocked.

Request a record-level change for a database file when save-while-active processing is active
: A request to make a record-level change under commitment control for a database file might be delayed if the commitment definition is at a commitment boundary and a save-while-active operation is running in a different job. This can happen when a file is journaled to the same journal as some of the objects on the save request.

  Note: The Status column on the Work with Active Jobs (WRKACTJOB command) display shows CMTW (commit wait) when a job is being held due to save-while-active checkpoint processing.

Commit or roll back changes when save-while-active processing is active
: A commit or rollback operation might be delayed at a commitment boundary when a save-while-active operation is running in a different job. This can happen when an API commitment resource was previously added to the commitment definition, unless the API resource was added using the QTNADDCR API and the *Allow normal save processing* field has a value of Y.

Because the job is held during the commit or rollback request, and because a commit or rollback request can be performed only for a single commitment definition at a time, jobs with more than one commitment definition with API commitment resources added can prevent a save-while-active operation from completing.

Note: If you use the new save with partial transactions feature the object can be saved without ending a commitment definition.

Request an object-level change when save-while-active processing is active
A request to make an object-level change under commitment control might be delayed if the commitment definition is at a commitment boundary and a save-while-active operation is running in a different job. This can happen when an object-level change is made while the save-while-active operation is running against the library the object is in. For example, the create SQL table operation under commitment control for table MYTBL in library MYSQLLIB might be delayed when a save-while-active operation is running against library MYSQLLIB.

Note: If the wait time exceeds 60 seconds, inquiry message CPA8351 is sent to ask the user whether to continue waiting or cancel the operation.

Add an API resource using the QTNADDCR API
A request to add an API commitment resource using the QTNADDCR API might be delayed if all commitment definitions for the job are at a commitment boundary and a save-while-active operation is running in a different job.

Notes:

1. If the wait time exceeds 60 seconds, inquiry message CPA8351 is sent to ask the user whether to continue waiting or cancel the operation.
2. This does not apply to API resources that were added using the QTNADDCR API if the *Allow normal save processing* field has a value of `Y`.

Factors that improve performance

Use a default journal
Using a default journal can help performance if you close and reopen all files under commitment control while the commitment definition is active. However, using a default journal with OMTJRNE(*NONE) degrades the performance of commit and rollback operations.

Select a last agent
Performance is enhanced when a last agent is selected because fewer interactions between the system and the last agent are required during a commit operation. However, if a communications failure occurs during a commit operation, the commit operation will not complete until resynchronization completes, regardless of the value of the wait for outcome option. Such a failure is rare but this option allows the application writer to consider the negative impact of causing the user to wait indefinitely for the resynchronization to complete when a failure does occur. Weigh this against the performance enhancement that is provided by last agent optimization during successful commit operations. This consideration is generally more significant for interactive jobs than for batch jobs.

The default is that a last agent is permitted to be selected by the system but the user can modify this value using the QTNCHGCO API.

Not use the wait for outcome option
When remote resources are under commitment control, performance is improved when the Wait for Outcome option is set to `N` (No) and all remote systems support presumed abort. The Wait for Outcome option is set to N by the system for DRDA and DDM application when the first connection is made to a remote system. APPC applications must explicitly set the Wait for Outcome option, or the default value of `Y` will be used.

Select the OK to Leave Out option
Performance is improved when the OK to Leave Out option is selected.

Select the Vote Read Only option
Performance is improved when the Vote Read Only option is selected.

Related concepts

Journal management

"Commitment definition for two-phase commit: Indicate OK to leave out" on page 37
Normally, the transaction manager at every location in the transaction program network participates in every commit or rollback operation. To improve performance, you can set up some or all locations in a transaction to allow the transaction manager to indicate OK to leave out.

"Commitment definition for two-phase commit: Allow vote read-only" on page 31
Normally, a transaction manager participates in both phases of commit processing. To improve the performance of commit processing, you can set up some or all locations in a transaction to allow the transaction manager to vote read-only.

Minimizing locks

A typical way to minimize record locks is to release the record lock. (This technique does not work if LCKLVL(*ALL) has been specified.)

Here is an example of minimizing record locks by releasing the record lock. A single file maintenance application typically follows these steps:

1. Display a prompt for a record identification to be changed.
2. Retrieve the requested record.
3. Display the record.
4. Allow the workstation user to make the change.
5. Update the record.

In most cases, the record is locked from the access of the requested record through the update. The record wait time might be exceeded for another job that is waiting for the record. To avoid locking a record while the workstation user is considering a change, release the record after it is retrieved from the database (before the record display appears). You then need to access the record again before updating. If the record was changed between the time it was released and the time it was accessed again, you must inform the workstation user. The program can determine if the record was changed by saving one or more fields of the original record and comparing them to the fields in the same record after it is retrieved as follows:

- Use an update count field in the record and add 1 to the field just before an update. The program saves the original value and compares it to the value in the field when the record is retrieved again. If a change has occurred, the workstation user is informed and the record appears again. The update count field is changed only if an update occurs. The record is released while the workstation user is considering a change. If you use this technique, you must use it in every program that updates the file.
- Save the contents of the entire data record and compare it to the record the next time it is retrieved.

In both of the previous cases, the sequence of operations prevents the simple use of externally described data in RPG where the same field names are used in the master record and in the display file. Using the same field names (in RPG) does not work because the workstation user's changes are overlaid when the record is retrieved again.You can solve this problem by moving the record data to a data structure. Or, if you use the DDS keyword RTNDTA, you can continue to use externally described data. The RTNDTA keyword allows your program to reread data on the display without the operating system having to move data from the display to the program. This allows the program to perform the following tasks:

1. Prompt for the record identification.
2. Retrieve the requested record from the database.
3. Release the record.

4. Save the field or fields used to determine if the record was changed.
5. Display the record and wait for the workstation user to respond.

If the workstation user changes the record on the display, the program uses the following sequence:
1. Retrieve the record from the database again.
2. Compare the saved fields to determine if the database record has been changed. If it has been changed, the program releases the record and sends a message when the record appears.
3. Retrieve the record from the display by running a read operation with the RTNDTA keyword and updates the record in the database record.
4. Proceed to the next logical prompt because there are no additional records to be released if the workstation user cancels the request.

LCKLVL(*CHG) and LCKLVL(*CS) work in this situation. If LCKLVL(*ALL) is used, you must release the record lock by using a commit or rollback operation.

Related tasks

"Detecting deadlocks" on page 106
A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

Managing transaction size

Another way to minimize record locks is to manage the size of the transaction.

For this discussion, a transaction is interactive. (Commitment control can also be used for batch applications, which often can be considered a series of transactions.) Many of the same considerations apply to batch applications, which are discussed in Commitment control for batch applications.

You can lock a maximum of 500 000 000 records during a transaction for each journal associated with the transaction. You can reduce this limit by using a Query Options File (QAQQINI). Use the QRYOPTLIB parameter of the Change Query Attributes (CHGQRYA) command to specify a Query Options File for a job to use. Use the COMMITMENT_CONTROL_LOCK_LEVEL value in the Query Options File as the lock limit for the job.

When choosing the lock level for your records, consider the size of your transactions. Use size to determine how long records are locked before a transaction ends. You must decide if a commit or rollback operation for commitment control is limited to a single use of the Enter key, or if the transaction consists of many uses of the Enter key.

Note: The shorter the transaction, the earlier the job waiting to start save-while-active checkpoint processing can continue and complete.

For example, for an order entry application, a customer might order several items in a single order requiring an order detail record and an inventory master record update for every item in the order. If the transaction is defined as the entire order and each use of the Enter key orders an item, all records involved in the order are locked for the duration of the entire order. Therefore, often used records (such as inventory master records) might be locked for long periods of time, preventing other work from progressing. If all items are entered with a single Enter key using a subfile, the duration of the locks for the entire order is minimized.

In general, the number and duration of locks must be minimized so several workstation users can access the same data without long waiting periods. You can do this by not holding locks while the user is entering data on the display. Some applications might not require more than one workstation user accessing the same data. For example, in a cash posting application with many open item records per customer, the typical approach is to lock all the records and delay them until a workstation user completes posting the cash for a given receipt.

If the workstation user presses the Enter key several times for a transaction, it is possible to perform the transaction in a number of segments. For example:

- The first segment is an inquiry in which the workstation user requests the information.
- The second segment is a confirmation of the workstation user's intent to complete the entire transaction.
- The third segment is retrieval and update of the affected records.

This approach allows record locking to be restricted to a single use of Enter.

This inquiry-first approach is normally used in applications where a decision results from information displayed. For example, in an airline reservation application, a customer might want to know what flight times, connecting flights, and seating arrangements are available before making a decision on which flight to take. After the customer makes a decision, the transaction is entered. If the transaction fails (the flight is now full), the rollback function can be used and a different request entered. If the records are locked from the first inquiry until a decision is made, another reservation clerk will be waiting until the other transaction is complete.

Related concepts

"Commitment control for batch applications" on page 24
Batch applications might or might not need commitment control. In some cases, a batch application can perform a single function of reading an input file and updating a master file. However, you can use commitment control for this type of application if it is important to start it again after an abnormal end.

Soft commit

Soft commit is a form of commitment control that limits the number of times that the system writes journal entries associated with a transaction to disk.

Soft commit can improve transaction performance, but it might cause one or more transactions to be lost in the event of a system failure. Traditional commitment control on DB2 Universal Database for iSeries ensures transaction durability, which means that when a transaction has been committed, the transaction persists on the system. Soft commit does not provide this durability, although it still ensures the atomicity of the transaction. In other words, the system guarantees a commit boundary, but one or more complete transactions might be lost in the event of a system failure.

To use soft commit, both for a particular job or across the system, specify *NO on the *QIBM_TN_COMMIT_DURABLE* environment variable. You can change this variable with the Add Environment Variable (ADDENVVAR) command.

For example, to request soft commit from a particular job, run the following command from the job:

```
ADDENVVAR ENVVAR (QIBM_TN_COMMIT_DURABLE) VALUE (*NO)
```

To request soft commit across the system, run the following command:

```
ADDENVVAR ENVVAR (QIBM_TN_COMMIT_DURABLE) VALUE (*NO) LEVEL (*SYS)
```

Note: You must have *JOBCTL authority to set this environment variable system wide.

If the *QIBM_TN_COMMIT_DURABLE* environment variable has not been added, or if the environment variable has been set to any value other than *NO, the system does not use soft commit; instead, the system uses traditional commitment control so that the durability of transactions will be ensured.

You can check the existence of this new environment variable, and its value and level if it exists, using the Work with Environment Variables (WRKENVVAR) command.

For some transactions, the operating system chooses to ignore your request for soft commit, and instead, performs traditional commitment. This happens in some complex environments, where multiple database connections are required or DDL operations are underway. The operating system can determine when it is appropriate to perform the request and when it makes more sense to perform a traditional commitment operation. So it is not harmful to request soft commit in such environments.

Scenarios and examples: Commitment control

These scenarios and examples show how one company sets up commitment control. Code examples for programs that use commitment control are also included in this topic collection.

The following scenario shows how the JKL toy company implements commitment control to track transactions on its local database.

The following examples provide sample code for commitment control. The practice problem is an RPG program that implements commitment control. It includes a logic flow that shows what is happening each step of the way.

Scenario: Commitment control

The JKL Toy Company uses commitment control to protect the database records for manufacturing and inventory. This scenario shows how JKL toy company uses commitment control to transfer a part from its inventory department to its manufacturing department.

The Scenario: Journal management topic includes a description of JKL Toy Company’s network environment. The scenario that follows shows how commitment control works on its production system JKLPROD.

This scenario illustrates the advantages of using commitment control in two examples. The first example shows how the company’s inventory program, Program A, might work without commitment control, and the possible problems that can occur. The second example shows how the program works with commitment control.

The JKL Toy Company uses an inventory application program, Program A, on its system JKLPROD. Program A uses two records. One record tracks items that are stored in the stock room. Another record keeps track of items that are removed from the stock room, and used in production.

Program A without commitment control

Assume that the following application program does not use commitment control. The system locks records read for updating. The following steps describe how the application program tracks a diode as it is removed from the stock room and transferred to checking account:

- Program A locks and retrieves the stock room record. (This action might require a wait if the record is locked by another program.)
- Program A locks and retrieves the production record. (This might also require a wait.) Program A now has both records locked, and no other program can change them.
- Program A updates the stock room record. This causes the record to be released so it is now available to be read for update by any other program.
- Program A updates the production record. This causes the record to be released so it is now available to be read for update by any other program.

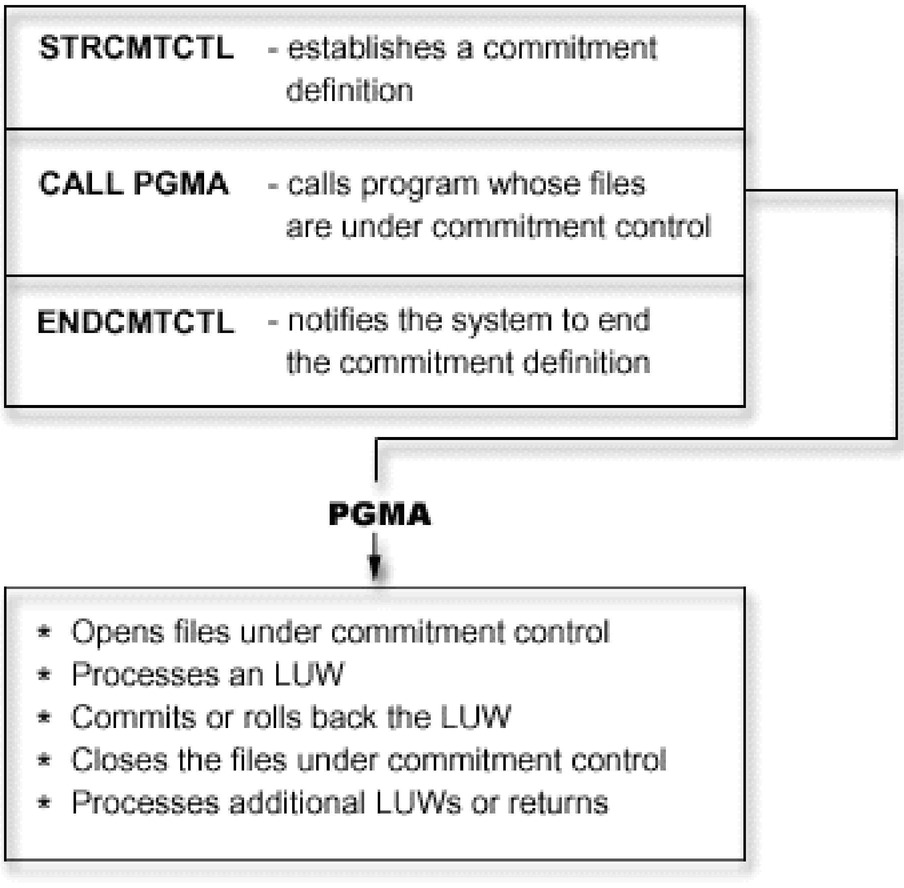
Without using commitment control, a problem needs to be solved to make this program work properly in all circumstances. For example, a problem occurs if program A does not update both records because of a job or system failure. In this case, the two files are not consistent -- diodes are removed from the stock room record, but they are not added to the production record. Using commitment control allows you to ensure that all changes involved in the transaction are completed, or that the files are returned to their original state if the processing of the transaction is interrupted.

Program A with commitment control

If commitment control is used, the preceding example is changed as follows:
1. Commitment control is started.
2. Program A locks and retrieves the stock room record. (This action can require a wait if the record is locked by another program.)
3. Program A locks and retrieves the production record. (This can also require a wait.) Program A now has both records locked, and no other program can change them.
4. Program A updates the stock room record, and commitment control keeps the lock on the record.
5. Program A updates the production record, and commitment control keeps the lock on the record.
6. Program A commits the transaction. The changes to the stock room record and the production record are made permanent in the files. The changes are recorded in the journal, which assumes they will appear on disk. Commitment control releases the locks on both records. The records are now available to be read for update by any other program.

Because the locks on both records are kept by commitment control until the transaction is committed, a situation cannot arise in which one record is updated and the other is not. If a routing step or system failure occurs before the transaction is committed, the system removes (rolls back) the changes that have been made so that the files are updated to the point where the last transaction was committed.

For each routing step in which files are to be under commitment control, the steps shown in the following figure occur:


STRCMTCTL - establishes a commitment definition
CALL PGMA - calls program whose files are under commitment control
ENDCMTCTL - notifies the system to end the commitment definition
PGMA
* Opens files under commitment control
* Processes an LUW
* Commits or rolls back the LUW
* Closes the files under commitment control
* Processes additional LUWs or returns

The operations that are performed under commitment control are journaled to the journal. The start commitment control journal entry appears after the first file open entry under commitment control. This is because the first file open entry determines what journal is used for commitment control. The journal entry from the first open operation is then used to check subsequent open operations to ensure that all files are using the same journal.

When a job failure or system failure occurs, the resources under commitment control are updated to a commitment boundary. If a transaction is started but is not completed before a routing step ends, that transaction is rolled back by the system and does not appear in the file after the routing step ends. If the system abnormally ends before a transaction is completed, that transaction is rolled back by the system and does not appear in the file after a subsequent successful initial program load (IPL) of licensed internal code. Anytime a rollback occurs, reversing entries are placed in the journal.

For example, assume that JKL company has 100 diodes in stock. Manufacturing takes out 20 from stock, for a new balance of 80. The database update causes both before-image (100) and after-image (80) journal entries.

Assume the system abnormally ended after journaling the entries, but before reaching the commitment point or rollback point. After the IPL, the system reads the journal entry and updates the corresponding database record. This update produces two journal entries that reverse the update: the first entry is the before-image (80) and the second entry is the after-image (100).

When the IPL is successfully completed after the abnormal end, the system removes (or rolls back) any database changes that are not committed. In the preceding example, the system removes the changes from the stock room record because a commit operation is not in the journal for that transaction. In this case, the before-image of the stock room record is placed in the file. The journal contains the rolled back changes, and an indication that a rollback operation occurred.

Related concepts

Scenario: Journal management

Practice problem for commitment control

This practice problem might help you understand commitment control and its requirements. These steps assume that you are familiar with the i5/OS licensed program, the data file utility (DFU), and this topic collection.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 111.

Before beginning this problem, follow these prerequisite steps:

1. Create a special library for this practice problem. In the instructions, the library is called CMTLIB. Substitute the name of your library where you see CMTLIB.
2. Create source files and a job description.

To use commitment control, follow these steps:

1. Create a physical file named ITMP (item master file). The data description specification (DDS) for this file is as follows:

```
10    A    R ITMR
20    A      ITEM     2
30    A      ONHAND   5   0
40    A    K ITEM
```

2. Create a physical file named TRNP (transaction file). This file is used as a transaction log file. The DDS for this file is as follows:

```
10    A    R TRNR
20    A      QTY     5  0
30    A      ITEM    2
40    A      USER    10
```

3. Create a logical file named TRNL (transaction logical). This file is used to assist in starting the application again. The *USER* field is the type LIFO sequence. The DDS for this file is as follows:

```
10                        LIFO
20    A     R TRNR        PFILE (TRNP)
30    A     K USER
```

4. Enter the STRDFU command, and create a DFU application named ITMU for the ITMP file. Accept the defaults offered by DFU during the application definition.
5. Type the command CHGDTA ITMU and enter the following records for the ITMP file:

Item	On hand
AA	450
BB	375
CC	4000

6. End the program using F3. This entry provides some data against which the program will operate.
7. Create the CL program Item Process (ITMPCSC) as follows:

```
PGM
DCL &USER *CHAR LEN(10)
RTVJOBA USER(&USER)
CALL ITMPCS PARM(&USER)
ENDPGM
```

This is the control program that calls the ITMPCS program. It retrieves the user name and passes it to the processing program. This application assumes that unique user names are used.

8. Create a display file named ITMPCSD from the DDS as follows.

   There are two formats, the first for the basic prompt display and the second to allow the operator to review the last transaction entered. This display file is used by the ITMPCS program.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

A          R PROMPT
A                                      CA03(93 'End of program')
A                                      CA04(94 'Review last')
A                                      SETOFF(64 'No rcd to rvw')
A                                  1  2'INVENTORY TRANSACTIONS'
A                                  3  2'Quantity'
A            QTY            5  0I    +1
A  61                                  ERRMSG('Invalid +
A                                      quantity' 61)
A                                    +5'ITEM'
A            ITEM           2    I   +1
A  62                                  ERRMSG('Invalid +
A                                      Item number' 62)
A  63                                  ERRMSG('Rollback +
A                                      occurred' 63)
A  64                             24  2'CF4 was pressed and +
A                                      there are no +
A                                      transactions for +
A                                      this user'
A                                      DSPATR(HI)
A                                 23  2'CF4 Review last +
A                                      transaction'
A          R REVW
A                                  1  2'INVENTORY TRANSACTIONS'
A                                    +5'REVIEW LAST TRANSACTION'
A                                  3  2'Quantity'
A            QTY            5  0     +1EDTCDE(Z)
A                                    +5'Item'
A            ITEM           2        +1
```

9. Study the logic flow provided in Logic flow for the practice program for commitment control.

10. Enter the STRSEU command and type the source as follows:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

FITMP    UF  E           K        DISK
F*                                             KCOMIT
FTRNP    O   E                    DISK
F*                                             KCOMIT
FTRNL    IF  E           K        DISK
F            TRNR                              KRENAMETRNR1
FITMPCSD  CF E                    WORKSTN
C* Enter parameter with User name for -TRNP- file
C           *ENTRY    PLIST
C                     PARM           USER   10
C           LOOP      TAG
C                     EXFMTPROMPT
C* Check for CF3 for end of program
C   93                DO                              End of Pgm
C                     SETON                     LR
C                     RETRN
C                     END
C* Check for CF4 for review last transaction
C   94                DO                              Review last
C* Check for existence of a record for this user in -TRNL- file
C           USER      CHAINTRNR1                64    Not found
C   64                GOTO LOOP
C                     EXFMTREVW
C                     GOTO LOOP
```

```
C                     END
C* Access Item record
C           ITEM      CHAINITMR                 62    Not found
C* Handle -not found- Condition
C   62                GOTO LOOP
C* Does sufficient quantity exist
C           ONHAND    SUB  QTY       TEST    50   61  Minus
C* Handle insufficient quantity
C    61               DO
C* Release Item record which was locked by the CHAIN for update
C                     EXCPTRLSITM
C                     GOTO LOOP
C                     END
C* Change ONHAND and update the Item record
C                     Z-ADDTEST      ONHAND
C                     UPDATITMR
C* Test for Special Simulation Conditions
C           ITEM      IFEQ 'CC'
C*      Simulate program need for rollback
C           QTY       IFEQ 100
C                     SETON                     63    Simult Rlbck
C*                    ROLBK
C                     GOTO LOOP
C                     END
C*      Simulate an abnormal program cancellation by Div by zero
C*        Operator Should respond -C- to inquiry message
C           QTY       IFEQ 101
C                     Z-ADD0         ZERO    30
C           TESTZ     DIV  ZERO      TESTZ   30       Msg occurs
C                     END
C*      Simulate an abnormal job cancellation by DSPLY.
C*        Operator Should System Request to another job
C*         and cancel this one with OPTION(*IMMED)
C           QTY       IFEQ 102
C           'CC=102'  DSPLY                         Msg occurs
C                     END
C                     END                           ITEM=CC
C* Write the -TRNP- file
C                     WRITETRNR
C* Commit the update to -ITMP- and write to -TRNP-
C*                    COMIT
C                     GOTO LOOP
OITMR  E                  RLSITM
```

11. Enter the CRTRPGPGM command to create program ITMPCS from the source entered in the previous step.
12. Type the command CALL ITMPCSC, press Enter, and press F4. A message states that there are no entries for this operator.
13. Enter the following data to see if the program operates correctly:

Quantity	Item
3	AA
4	BB

14. Press F4. The review display shows the BB item last entered. Enter the following data:

Quantity	Item
5	FF (Invalid item number message occurs.)
9000	BB (Insufficient quantity error message occurs.)
100	CC (Rollback message occurs.)
102	CC (RPG DSPLY operation must occur. Press the Enter key.)

Quantity	Item
101	CC (The program must display an inquiry message stating that a divide by zero condition has occurred or end, depending on the setting of job attribute INQMSGRPY. If the inquiry message appears, enter C to cancel the RPG program and then C to cancel the CL program on the subsequent inquiry. This simulates an unexpected error condition.)

15. Type the Display Data command DSPDTA ITMP.

    See if the records AA and BB have been updated correctly. The values must be AA = 447, BB = 371, and CC = 3697. Note that the quantities subtracted from CC occurred, but the transaction records were not written.
16. Create a journal receiver for commitment control. Use the Create Journal Receiver (CRTJRNRCV) command to create a journal receiver called RCVR1 in the CMRLIB library. Specify a threshold of at least 5000KB. A larger threshold is recommended if your system has sufficient space in order to maximize the time between generation of new journal receivers to minimize the performance impacts of too frequent change journals.
17. Create a journal for commitment control. Use the Create Journal (CRTJRN) command to create a journal called JRNTEST in the CMTLIB library. Because this journal is used only for commitment control, specify `MNGRCV(*SYSTEM) DLTRCV(*YES)`. For the JRNRCV parameter, specify the journal receiver that you created in step 16.
18. Use the Start Journal Physical File (STRJRNPF) command with the parameters FILE(CMTLIB/ITMP CMTLIB/TRNP) JRN(CMTLIB/JRNTEST) to journal the files to be used for commitment control.

    The IMAGES parameter uses a default of *AFTER, meaning that only after-image changes of the records appear in the journal. The files ITMP and TRNP have now started journaling.

    Normally, you save the files after starting journaling. You cannot apply journaled changes to a restored file that does not have the same JID as the journal entries. Because this practice problem does not require you to apply journaled changes, you can skip saving the journaled files.
19. Type the command CALL ITMPCSC and enter the following transactions:

Quantity	Item
5	AA
6	BB

    End the program by pressing F3.
20. Type the Display Journal command: DSPJRN CMTLIB/JRNTEST.

    Note the entries appearing in the journal. The same sequence of entries (R UP = update of ITMP followed by R PT = record added to TRNP) occurs in the journal as was performed by the program. This is because a logical file is defined over the physical file TRNP and the system overrides the RPG default. If no logical file existed, the RPG assumption of SEQONLY(*YES) is used, and a block of PT entries appear because the records are kept in the RPG buffer until the block is full.
21. Change the CL program ITMPCSC as follows (the new statements are shown with an asterisk).

```
     PGM
     DCL &USER *CHAR LEN(10)
     RTVJOBA USER(&USER)
*    STRCMTCTL LCKLVL(*CHG)
     CALL ITMPCS PARM(&USER)
*    MONMSG MSGID(RPG9001) EXEC(ROLLBACK)
*    ENDCMTCTL
     ENDPGM
```

    The STRCMTCTL command sets up the commitment control environment. The LCKLVL word specifies that records read for update but not updated can be released during the transaction. The

MONMSG command handles any RPG escape messages and performs a ROLLBACK in case the RPG program abnormally ends. The ENDCMTCTL command ends the commitment control environment.

22. Delete the existing ITMPCSC program and create it again.
23. Change the RPG program to remove the comment symbols at statements 2.00, 4.00, 46.00, and 65.00. The source is now ready for use with commitment control.
24. Delete the existing ITMPCS program and create it again. The program is now ready to operate under commitment control.
25. Type the command CALL ITMPCSC and the following transactions:

Quantity	Item
7	AA
8	BB

26. Use System Request and request the option to display the current job. When the Display Job display appears, select option 16 to request the display of the commitment control status.

    Note the values on the display. There must be two commits because two commit statements were run in the program.
27. Press F9 to see a list of the files under commitment control and the amount of activity for each file.
28. Return to the program and end it by pressing F3.
29. Type `DSPJRN CMTLIB/JRNTEST` and note the entries for the files and the special journal entries for commitment control:

Entry	Meaning
C BC	STRCMTCTL command occurred.
C SC	Start commit cycle. This occurs whenever the first database operation in the transaction causes a record to be inserted, updated, or deleted as part of commitment control.
C CM	Commit operation has occurred.
C EC	ENDCMTCTL command occurred.

    The commitment control before-images and after-images (R UB and R UP types) automatically occur even though you had originally requested IMAGES(*AFTER) for the journal.
30. Type the command CALL ITMPCSC and the following transactions:

Quantity	Item
12	AA
100	CC (This is the condition to simulate the need for an application use of rollback. The CC record in the ITMP file, which was updated by RPG statement 40.00 is rolled back.)

31. Press F4 to determine the last transaction entered.

    The last committed transaction is the entry for item AA.
32. Use System Request and request the Display Current Job option. When the Display Job display appears, request the display of the commitment control status.

    Note the values on the display and how they have been changed by the rollback.
33. Return to the program.
34. Return to the basic prompt display and end the program by pressing F3.
35. Type the command DSPJRN CMTLIB/JRNTEST.

Note the additional entries that appear in the journal for the use of the rollback entry (C RB entry). When the ITMP record is rolled back, three entries are placed in the journal. This is because any change to the database file under commitment control produces a before (R BR) and after (R UR) entry.

36. Display the entries with journal code R and these entry types: UB, UP, BR, and UR. Use option 5 to display the full entries. Because the *Quantity* field is in packed decimal, use F11 to request a hex display. Note the following conditions:
    - The on-hand value of the ITMP record in the UB record
    - How the on-hand value is reduced by the UP record
    - How the BR record is the same as the UP record
    - How the UR record returns the value as originally displayed for the UB record

    The last entry is the RB entry for the end of the rollback.
37. Type the command CALL ITMPCSC; press Enter; and press F4. Note the last transaction entered.
38. Type the following transactions:

Quantity	**Item**
13	AA
101	CC (This is the condition to simulate an unexpected error condition, which causes the program to end. The simulation occurs by dividing a field by 0. The program will display an inquiry message or end, depending on the setting of the job attribute INQMSGRPY. If the inquiry message appears, enter C to end the program. Because the CL program was changed to monitor for RPG program errors, the second inquiry which occurred does not occur.)

39. Type the command DSPJRN CMTLIB/JRNTEST.

    The same type of rollback handling has occurred, but this time the rollback was caused by the EXEC parameter of the MONMSG command in the CL program instead of the RPG program. Display the two RB entries to see which program caused them.
40. Type the command WRKJOB and write down the fully qualified job name to be used later.
41. Type the command CALL ITMPCSC and enter the following transaction:

Quantity	**Item**
14	AA
102	CC (The RPG DSPLY operation must occur to the external message queue. Use the System Request key and select option 1 on the system request menu to transfer to a secondary job.)

42. Sign on to the second job and reestablish your environment.
43. Type the command ENDJOB and specify the fully qualified job name identified earlier and OPTION(*IMMED). This simulates an abnormal job or system end.
44. Wait about 30 seconds, type the command CALL ITMPCSC and press F4.

    Note the last committed transaction. It must be the AA item entered earlier.
45. Return to the basic prompt display and end the program by pressing F3.
46. Type the command DSPJRN CMTLIB/JRNTEST.

    The same type of rollback handling has occurred, but this time the rollback was caused by the system instead of one of the programs. The RB entry was written by the program QWTPITPP, which is the work management abnormal end program.

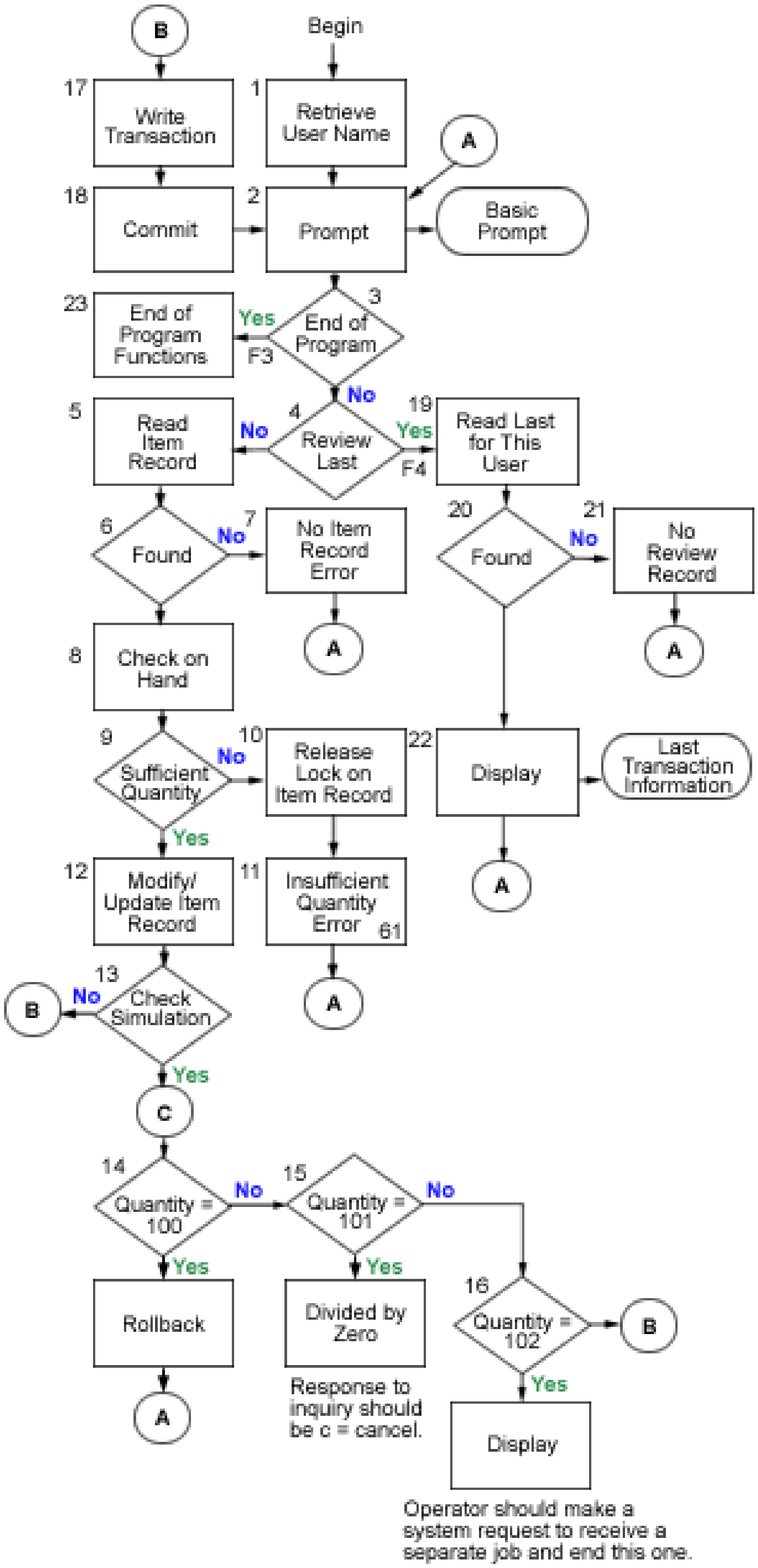
You have now used the basic functions of commitment control. You can proceed with commitment control on your applications or try some of the other functions such as:

- Using a notify object
- Locking records that are only read with LCKLVL(*ALL)
- Locking multiple records in the same file with LCKLVL(*ALL)

Logic flow for practice problem

The logic flow might help you further understand this practice program for commitment control. The following figure shows the flow of the practice problem for commitment control.

See “Steps associated with the logic flow for the practice program” on page 81 for details about each of the steps shown in the figure.


B
Begin
17
Write Transaction
1
Retrieve User Name
A
18
Commit
2
Prompt
Basic Prompt
23
End of Program Functions
Yes
F3
3
End of Program
No
5
Read Item Record
No
4
Review Last
19
Yes
F4
Read Last for This User
6
Found
No
7
No Item Record Error
A
20
Found
21
No
No Review Record
A
8
Check on Hand
9
Sufficient Quantity
No
10
Release Lock on Item Record
22
Display
Last Transaction Information
Yes
12
Modify/ Update Item Record
11
Insufficient Quantity Error
61
A
A
13
B
No
Check Simulation
Yes
C
14
Quantity = 100
No
15
Quantity = 101
No
Yes
Rollback
Yes
Divided by Zero
16
Quantity = 102
B
A
Response to inquiry should be c = cancel.
Yes
Display
Operator should make a system request to receive a separate job and end this one.

Steps associated with the logic flow for the practice program

These steps are associated with the logic flow for the practice problem.

1. Retrieve the user name that is passed in as a parameter. This is used to write to the TRNP file and also used to retrieve the last transaction entered by each operator. This application assumes unique user names for operators.
2. Prompt for the basic display using the format name PROMPT.
3. If F3 is pressed, start an end of program function.
4. If F4 is pressed, start a routine to access the last transaction entered by the operator.
5. Read the item record using the field *ITEM*. Because the file is an update file, this request locks the record.
6. Check for a not found condition in the file ITMP.
7. If no ITMP record exists, set on indicator 62 to cause the error message and return to step 2.
8. Subtract the quantity requested (QTY) from the on hand balance (ONHAND) into a work area.
9. Check to see if sufficient quantity exists to meet the request.
10. If insufficient quantity exists, release the lock on the record in the ITMP file. This step is needed because of insufficient quantity.
11. Set on indicator 61 to signal an insufficient quantity display error message and return to step 2.
12. Change the ONHAND field for the new balance and update the ITMR record.
13. Check for special entry in the ITEM field that can be used to simulate conditions where ROLLBACK is required.
14. Check for QTY=100. Issue a ROLLBACK operation. This simulates a condition where the program senses a need for rollback.
15. Check for QTY=101. Cause an exception in the program that will produce an inquiry message. Use divide by zero for this function. The operator should enter C to cancel the program unless the job description INQMSGRPH option provides an automatic reply. This simulates a condition where an unexpected error has occurred and the operator cancels the program.
16. Check for QTY=102. Issue a display with inquiry operation. This stops the program at this step and allows the use of the System Request key to get to a different job. Cancel the updating job. This simulates a condition where an abnormal job or system end has occurred in the middle of a commit boundary.
17. Write the transaction record to TRNP.
18. Commit the records for the transaction and return to step 2.
19. Read the first record on the access path for file TRNL, using USER as the key. Because this file is in LIFO sequence, this will be the last transaction record entered by this user.
20. Check for a record not found condition in the TRNL file that is caused if the file does not contain entries for this user.
21. If there is no record for this user, set on indicator 64 to cause an error message and return to step 2.
22. Display the last transaction entered for this user. This information can be used if the operator forgets what was previously entered or when the transaction is restarted. When the operator responds, return to step 2.
23. Perform any of the program functions.

Example: Using a transaction logging file to start an application

This example provides sample code and instructions of how to use a transaction logging file to start an application after an abnormal end.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 111.

A *transaction logging file* is used to start an application again after a system or job failure when a notify object is not used. A transaction logging file is often used in interactive applications to summarize the effects of a transaction.

For example, in an order entry application, a record is typically written to a transaction logging file for each item ordered. The record contains the item ordered, the quantity, and the price. In an accounts payable application, a record is written to a transaction logging file for each account number that is to receive a charge. This record normally contains such information as the account number, the amount charged, and the vendor.

In many of the applications where a transaction logging file already exists, a workstation user can request information about the last transaction entered. By adding commitment control to the applications in which a transaction logging file already exists, you can:
- Ensure that the database files are updated to a commitment boundary.
- Simplify the starting of the transaction again.

You must be able to uniquely identify the workstation user if you use a transaction logging file for starting applications again under commit control. If unique user profile names are used on the system, that profile name can be placed in a field in the transaction logging record. This field can be used as the key to the file.

The following examples assume that an order inventory file is being used to perform transactions and that a transaction logging file already exists. The program does the following tasks:
1. Prompt the workstation user for a quantity and item number.
2. Update the quantity in the production master file (PRDMSTP).
3. Write a record to the transaction logging file (ISSLOGL).

If the inventory quantity on hand is insufficient, the program rejects the transaction. The workstation user can ask the program where the data entry was interrupted, because the item number, description, quantity, user name, and date are written to the transaction logging file.

DDS for physical file PRDMSTP

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7

  1.00      A          R PRDMSTR                  TEXT('Master record')
  2.00      A            PRODCT         3         COLHDG('Product' 'Number')
  3.00      A            DESCRP        20         COLHDG('Description')
  4.00      A            ONHAND         5  0      COLHDG('On Hand' 'Amount')
  5.00      A                                     EDTCDE(Z)
  6.00      A          K PRODCT
```

DDS for physical file ISSLOGP used by ISSLOGP

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7

  1.00      A          R ISSLOGR                  TEXT('Product log record')
  2.00      A            PRODCT         3         COLHDG('Product' 'Number')
  3.00      A            DESCRP        20         COLHDG('Description')
  4.00      A            QTY            3  0      COLHDG('Quantity')
  5.00      A                                     EDTCDE(Z)
  6.00      A            USER          10         COLHDG('User' 'Name')
  7.00      A            DATE           6  0      EDTCDE(Y)
  8.00      A                                     COLHDG('Date')
```

DDS for logical file ISSLOGL

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7

  1.00      A                                    LIFO
  2.00      A          R ISSLOGR                 PFILE(ISSLOGP)
  3.00      A          K USER
```

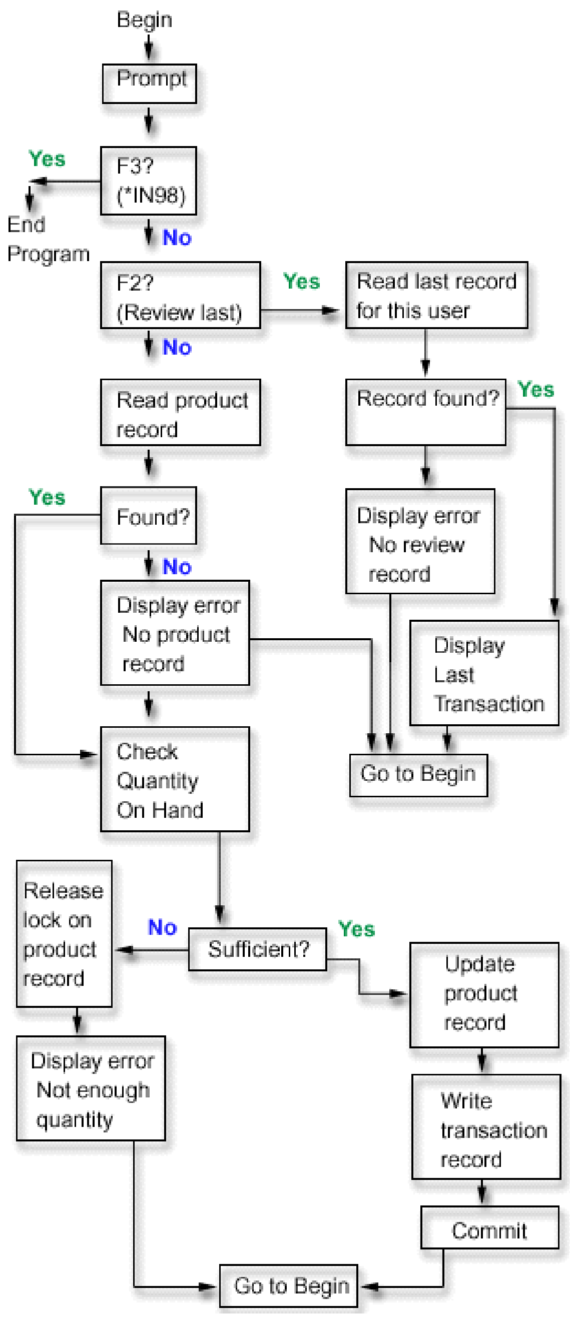
DDS for display file PRDISSD used in the program

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 1.00       A                                    REF(ISSLOGP)
 2.00       A          R PROMPT
 3.00       A                                    CA03(98 'End of program')
 4.00       A                                    CA02(97 'Where am I')
 5.00       A                                1 20'ISSUES PROCESSING'
 6.00       A                                3  2'Quantity'
 7.00       A            QTY       R        I    +1
 8.00       A  62                                ERRMSG('Not enough +
 9.00       A                                    Qty' 62)
 10.00      A                                  +6'Product'
 11.00      A            PRODCT    R        I    +1
 12.00      A  61                                ERRMSG('No Product +
 13.00      A                                    record found' 62)
 14.00      A  55                           15  2'No Previous record exists'
 15.00      A                               24  2'CF2 Last transaction'
 16.00      A          R RESTART
 17.00      A                                1 20'LAST TRANSACTION +
 18.00      A                                    INFORMATION'
 19.00      A                                5  2'Product'
 20.00      A            PRODCT    R           +1
 21.00      A                                7  2'Description'
 22.00      A            DESCRP    R           +1
 23.00      A                                9  2'Qty'
 24.00      A            QTY       R           +1REFFLD(QTY)
```

This process is outlined in the **Program flow**.

Program flow



The RPG COMMIT operation code is specified after the PRDMSTP file is updated and the record is written to the transaction logging file. Because each prompt to the operator represents a boundary for a new transaction, the transaction is considered a single Enter transaction.

The user name is passed to the program when it is called. The access path for the transaction logging file is defined in last-in-first-out (LIFO) sequence so the program can easily access the last record entered.

The workstation user can start the program again after a system or job failure by using the same function that identified where data entry was stopped. No additional code needs to be added to the program. If you are currently using a transaction logging file but are not using it to find out where you are, add the user name to the transaction logging file (assuming that user names are unique) and use this approach in the program.

The following example shows the RPG program used. Statements required for commitment control are marked with arrows (==>).

RPG Program

```
 SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... .. 7 ..
 =>1.00      FPRDMSTP UP  E           K        DISK        KCOMIT
 =>2.00      FISSLOGL IF  E           K        DISK        KCOMIT
   3.00      PRDISSD  CP  E                    WORKSTN
   4.00                  *ENTRY    PLIST
   5.00                            PARM           USER    10
   6.00      C*
   7.00      C*  Initialize fields used in Trans Log Rcd
   8.00      C*
   9.00      C                     MOVE UDATE     DATE
  10.00      C*
  11.00      C*  Basic processing loop
  12.00      C*
  13.00      C           LOOP      TAG
  14.00      C                     EXFMTPROMPT
  15.00      C   98                GOTO END                       End of pgm
  16.00      C   97                DO                             Where am I
  17.00      C                     EXSR WHERE
  18.00      C                     GOTO LOOP
  19.00      C                     END
  20.00      C           PRODCT    CHAINPRDMSTR              61     Not found
  21.00      C   61                GOTO LOOP
  22.00      C           ONHAND    SUB  QTY      TEST    50    62   Less than
  23.00      C   62                DO                               Not enough
  24.00      C                     EXCPTRLSMST                      Release lock
  25.00      C                     GOTO LOOP
  26.00      C                     END
  27.00      C*
  28.00      C*  Update master record and output the Transaction Log Record
  29.00      C*
  30.00      C                     Z-ADDTEST     ONHAND
  31.00      C                     UPDATPRDMSTR
  32.00      C                     WRITEISSLOGR
=>33.00      C                     COMIT
  34.00      C                     GOTO LOOP
  35.00      C*
  36.00      C*  End of program processing
  37.00      C*
  38.00      C           END       TAG
  39.00      C                     SETON                     LR
  40.00      C*
  41.00      C* WHERE subroutine for "Where am I" requests
  42.00      C*
  43.00      C           WHERE     BEGSR
  44.00      C           USER      CHAINISSLOGL              55    Not found
  45.00      C  N55                EXFMTRESTART
  46.00      C                     ENDSR
  47.00      OPRDMSTR E                RLSMST
```

CL program used to call RPG program PRDISS

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

  1.00            PGM
  2.00            DCL          &USER *CHAR LEN(10)
  3.00            STRCMTCTL    LCKLVL(*CHG)
  4.00            RTVJOBA      USER(&USER)
```

```
 5.00           CALL         PRDISS PARM(&USER)
 6.00           MONMSG       MSGID(RPG9001) EXEC(ROLLBACK)
 7.00           ENDCMTCTL
 8.00           ENDPGM
```

To use commitment control in this program, a lock level of *CHG is normally specified. The record is locked by the change until a commit operation is run. Note that if there is an insufficient quantity of inventory, the record is explicitly released. (If the record are not explicitly released in the program, it is released when the next record is read for update from the file.)

In this example, there is no additional advantage to using the lock level *ALL. If *ALL is used, a rollback or commit operation must be used to release the record when an insufficient quantity existed.

The previous code is a CL program that calls the RPG program PRDISS. Note the use of STRCMTCTL/ENDCMTCTL commands. The unique user name is retrieved (RTVJOBA command) and passed to the program. The use of the MONMSG command to cause a rollback is described in Example: Use a standard processing program to start an application.

Related concepts

"Example: Using a standard processing program to start an application" on page 93
A standard processing program is one way to start your application again using one database file as the notify object for all applications. This approach assumes that user profile names are unique by user for all applications using the standard program.

"Example: Unique notify object for each program" on page 87
Using a single, unique notify object for each job allows use of an externally described commit identification even though there might be multiple users of the same program.

Example: Using a notify object to start an application

When a program is started after an abnormal end, it can look for an entry in the notify object. If the entry exists, the program can start a transaction again. After the transaction has been started again, the notify object is cleared by the program to prevent it from starting the same transaction yet another time.

You can use a notify object in the following ways:

- If the commit identification is placed in a database file, query this file to determine where to start each application or workstation job again.
- If the commit identification is placed in a message queue for a particular workstation, a message can be sent to the work station users when they sign on to inform them of the last transaction committed.
- If the commit identification is placed in a database file that has a key or user name, the program can read this file when it is started. If a record exists in the file, start the program again. The program can send a message to the workstation user identifying the last transaction committed. Any recovery is performed by the program. If a record existed in the database file, the program deletes that record at the end of the program.
- For a batch application, the commit identification can be placed in a data area that contains totals, switch settings, and other status information necessary to start the application again. When the application is started, it accesses the data area and verifies the values stored there. If the application ends normally, the data area is set up for the next run.
- For a batch application, the commit identification can be sent to a message queue. A program that is run when the application is started can retrieve the messages from the queue and start the programs again.

You can use several techniques for starting your applications again, depending on your application needs. In choosing the technique, consider the following information:

- When there are multiple users of a program at the same time, a single data area cannot be used as the notify object because after an abnormal system end, the commit identification for each user will overlay each other in the data area.

- Your design for deleting information in the notify object must handle the situation when a failure occurs immediately following use of the information:
  - If information is deleted immediately, it does not exist if another failure occurs before processing the interrupted transaction.
  - The information in the notify object must not be deleted until the successful processing of the interrupted transaction. In this case, more than one entry will exist in the notify object if it is a database file or message queue.
  - The program must access the last record if there is more than one entry.
- A notify object cannot be used to provide the work station user with the last transaction committed because the notify object is updated only if a system or job failure occurs or if uncommitted changes exist at the normal end of a job.
- If information is displayed to the workstation user, it must be meaningful. This might require that the program translate codes kept in the notify object into information that will help the user start again.
- Information for starting again must be displayed if the work station user needs it. Additional logic in the program is required to prevent information from being displayed again when it is no longer meaningful.
- A single notify object and a standard processing program can provide a starting again function if the notify object is a database file. This standard processing program is called by the programs that require the ability to start again to minimize the changes to each individual program.

Related concepts

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Example: Unique notify object for each program

Using a single, unique notify object for each job allows use of an externally described commit identification even though there might be multiple users of the same program.

In the following examples, a database file is used as a notify object and it is used only by this program.

The program has two database files (PRDMSTP and PRDLOCP) that must be updated for receipts to inventory. The display file used by the program is named PRDRCTD. A database file, PRDRCTP, is used as the notify object. This notify object is defined to the program as a file and is also used as the definition of a data structure for the notify function.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 111.

DDS for physical file PRDLOCP

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7

  1.00      A          R PRDLOCR                     TEXT('Location record')
  2.00      A            PRODCT         3            COLHDG('Product' 'Number')
  3.00      A            LOCATN         6            COLHDG('Location')
  4.00      A            LOCAMT         5  0         COLHDG('Location' 'Amount')
  5.00      A                                        EDTCDE(Z)
  6.00      A          K PRODCT
  7.00      A          K LOCATN
```

DDS for display file PRDRCTD

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

  1.00      A                                        REF(PRDMSTP)
  2.00      A          R PROMPT
```

```
A                                      CA03(98 'End of program')
A                                      SETOFF(71 'RESTART')
A                                 1  20'PRODUCT RECEIPTS'
A                                 3   2'Quantity'
A            QTY           3  OI     +1
A                                    +6'Product'
A            PRODCT   R        I     +1
A  61                                  ERRMSG('No record +
A                                      found in the +
A                                      master file' 62)
A                                    +6'Location'
A            LOCATN   R        I     +1REFFLD(LOCATN PRDLOCP)
A  62                                  ERRMSG('No record +
A                                      found in the +
A                                      location file' 62)
A                                 9   2'Last Transaction'
A  71                                +6'This is restart +
A                                       information'
A                                      DSPATR(HI BL)
A                                12   2'Quantity'
A                                12  12'Product'
A                                12  23'Location'
A                                12  35'Description'
A            LSTPRD   R          14  15REFFLD(PRODCT)
A            LSTLOC   R          14  26REFFLD(LOCATN *SRC)
A            LSTQTY   R          14   5REFFLD(QTY *SRC)
A                                      EDTCDE(Z)
A            LSTDSC   R          14  35REFFLD(DESCRP)
```

DDS for notify object and externally described data structure (PRDRCTP)

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

A                                      LIFO
A                                      REF(PRDMSTP)
A          R PRDRCTR
A            USER          10
A            PRODCT    R
A            DESCRP    R
A            QTY            3  0
A            LOCATN    R               REFFLD(LOCATN PRDLOCP)
A          K USER
```

The program processes the notify object as follows:

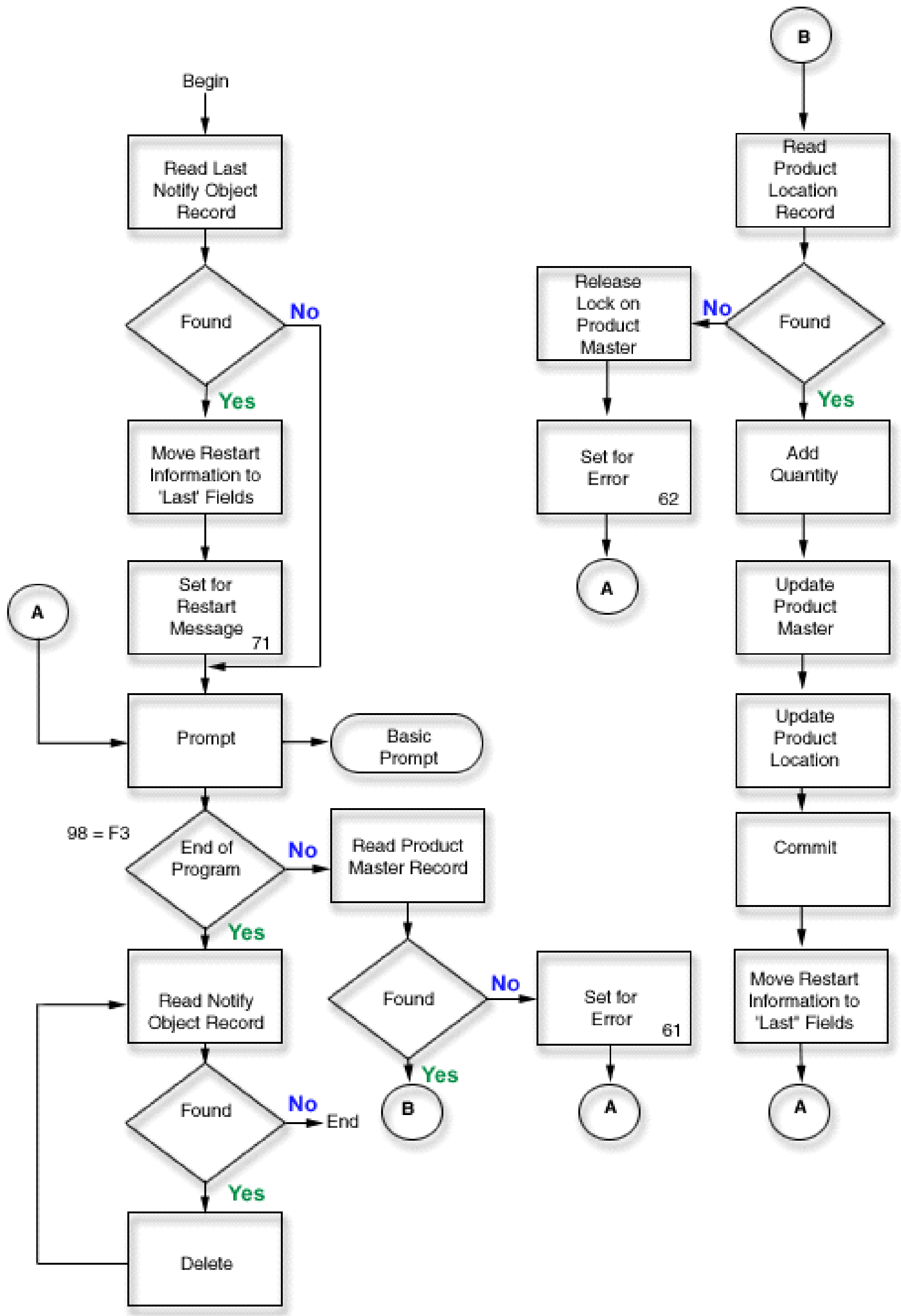
- At the beginning, the program randomly processes the notify object and displays a record if it exists for the specific key:
  - If multiple records exist, the last record for this key is used because the PRDRCTP file is in LIFO sequence.
  - If no record exists, a transaction was not interrupted so it is not necessary to start again.
  - If the program fails before the first successful commit operation, it does not consider that starting again is required.
- The routine to clear the notify object occurs at the end of the program:
  - If there were multiple failures, the routine can handle deletion of multiple records in the notify object.
  - Although the system places the commit identification in a database file, the commit identification must be specified as a variable in the RPG program.
  - Because RPG allows a data structure to be externally described, a data structure is a convenient way of specifying the commit identification. In this example, the data structure uses the same external description that the database file used as the notify object.

The processing for this program prompts the user for a product number, a location, and a quantity:

- Two files must be updated:
  - Product master file (PRDMSTP)
  - Product location file (PRDLOCP)
- A record in each file must exist before either is updated.
- The program moves the input fields to corresponding last fields after each transaction is successfully entered. These last fields are displayed to the operator on each prompt as feedback for what was last entered.
- If information for starting again exists, it is moved to these last fields and a special message appears on the display.

This process is outlined in the following figure. The user name is passed to the program to provide a unique record in the notify object.

Program flow


Begin
Read Last Notify Object Record
Found
No
Yes
Move Restart Information to 'Last' Fields
Set for Restart Message 71
A
Prompt
Basic Prompt
98 = F3
End of Program
No
Yes
Read Product Master Record
Read Notify Object Record
Found
No
End
Yes
Delete
Found
No
Yes
B
Set for Error 61
A
B
Read Product Location Record
Found
No
Yes
Release Lock on Product Master
Set for Error 62
A
Add Quantity
Update Product Master
Update Product Location
Commit
Move Restart Information to 'Last' Fields
A

The following example is about the RPG source code. The notify object (file PRDRCTP) is used as a normal file at the beginning and end of the program, and is also specified as the notify object in the CL (STRCMTCTL command) before calling the program.

RPG source

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

            FPRDMSTP UF  E           K        DISK         KCOMIT
            FPRDLOCP UF  E           K        DISK         KCOMIT
            FPRDRCTD CF  E                    WORKSTN
            F*
            F* The following file is the specific notify object for this pgm.
            F*      It is accessed only in a restart situation and at the
            F*        end of the program to delete any records.  The records
            F*        are written to the notify object by Commitment Control.
            F*
            FPRDRCTP UF  E           K        DISK
            ICMTID     E DSPRDRCTP
            C           *ENTRY    PLIST
            C                     PARM           USER10 10
            C                     MOVE USER10    USER
            C*
            C*  Check for restart information - get last rcd per user
            C*    PRDRCTP file access path is in LIFO sequence
            C*
            C           USER      CHAINPRDRCTR            20     Not found
            C  N20                DO                             Restart
            C                     EXSR MOVLST                    Move to last
            C                     SETON                  71     Restart
            C                     END
            C*
            C*  Basic processing loop
            C*
            C           LOOP      TAG
            C                     EXFMTPROMPT
            C   98                GOTO END                       End of pgm
            C           PRODCT    CHAINPRDMSTR            61     Not found
            C   61                GOTO LOOP
            C           KEY       KLIST
            C                     KFLD           PRODCT
            C                     KFLD           LOCATN
            C           KEY       CHAINPRDLOCR            62     Not found
            C   62                DO
            C                     EXCPTRLSMST                    Release lck
            C                     GOTO LOOP
            C                     END
            C                     ADD  QTY       ONHAND           Add
            C                     ADD  QTY       LOCAMT
            C                     UPDATPRDMSTR                   Update
            C                     UPDATPRDLOCR                   Update
            C*
            C*  Commit and move to previous fields
            C*
            C           CMTID     COMIT
            C                     EXSR MOVLST                    Move to last
            C                     GOTO LOOP
            C*
            C*  End of program processing
            C*
            C           END       TAG
            C                     SETON                  LR
            C*56.00      C*  Delete any records in the notify object
            C*
            C           DLTLP     TAG
            C           USER      CHAINPRDRCTR            20     Not found
            C  N20                DO
            C                     DELETPRDRCTR                   Delete
            C                     GOTO DLTLP
            C                     END
            C*
            C*  Move to -Last Used- fields for operator feedback
```

```
C*
C           MOVLST    BEGSR
C                     MOVE PRODCT    LSTPRD
C                     MOVE LOCATN    LSTLOC
C                     MOVE QTY       LSTQTY
C                     MOVE DESCRP    LSTDSC
C                     ENDSR
OPRDMSTR E                RLSMST
```

Related concepts

"Example: Using a transaction logging file to start an application" on page 81
This example provides sample code and instructions of how to use a transaction logging file to start an application after an abnormal end.

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

"Example: Single notify object for all programs"
Using a single notify object for all programs is advantageous. It is because all information required to start again is in the same object, and a standard approach to the notify object can be used in all programs.

Example: Single notify object for all programs

Using a single notify object for all programs is advantageous. It is because all information required to start again is in the same object, and a standard approach to the notify object can be used in all programs.

In this situation, use a unique combination of user and program identifications to make sure that the program accesses the correct information when it starts again.

Because the information required to start again might vary from program to program, do not use an externally described data structure for the commit identification. If a single notify object is used, the preceding program can describe the data structure within the program rather than externally. For example:

```
 1   10      USER
11   20      PGMNAM
21   23      PRODCT
24   29      LOCATN
30   49      DESC
50   51  0   QTY
52  220      DUMMY
```

In each program that uses this notify object, the information specified for the commit identification is unique to the program (the user and program names are not unique). The notify object must be large enough to contain the maximum information that any program can place in the commit identification.

Related concepts

"Example: Unique notify object for each program" on page 87
Using a single, unique notify object for each job allows use of an externally described commit identification even though there might be multiple users of the same program.

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Example: Using a standard processing program to start an application

A standard processing program is one way to start your application again using one database file as the notify object for all applications. This approach assumes that user profile names are unique by user for all applications using the standard program.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 111.

For this approach, the physical file NFYOBJP is used as the notify object and defined as:

```
Unique user profile name                 10 characters
Program identification                   10 characters
Information for starting again           Character field
                                        (This must be large
                                         enough to contain the maximum
                                         amount of information for starting
                                         programs again that require
                                         information for starting again.
                                         This field is required by
                                         the application programs.
                                         In the example, it is
                                         assumed to be a length of 200.)
```

The file is created with SHARE(*YES). The first two fields in the file are the key to the file. (This file can also be defined as a data structure in RPG programs.)

Related concepts

"Example: Using a transaction logging file to start an application" on page 81
This example provides sample code and instructions of how to use a transaction logging file to start an application after an abnormal end.

Example: Code for a standard processing program

This example shows the code for a standard processing program that can be used to start your application again using one database file as the notify object for all applications.

The application shown in the following code example performs as follows:

1. The application program receives the user name in a parameter and uses it with the program name as a unique identifier in the notify object.
2. The application program passes a request code of R to the standard commit processing program, which determines if a record exists in the notify object.
3. If the standard commit processing program returns a code of 1, a record was found and the application program presents the information needed to start again to the user.
4. The application program proceeds with normal processing.
5. When a transaction is completed, values are saved for reference so the workstation user can see what was done for the previous transaction.

   The information saved is not provided by the notify object because the notify object is updated only if a job or system failure occurs.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 111.

Application program example

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

  1.00      FPRDMSTP UF  E           K        DISK         KCOMIT
  2.00      FPRDLOCP UF  E           K        DISK         KCOMIT
  3.00      FPRDRCTD CF  E                    WORKSTN
  4.00      F*
```

```
F* The following is a compile time array which contains the
F*    restart information used in the next example
F*
E                    RTXT   50  50  1               Restart text
I*
I* Data structure used for info passed to notify object
I*
ICMTID       DS
I                                       1  10 USER
I                                      11  20 PGMNAM
I                                      21  23 PRODCT
I                                      24  29 LOCATN
I                                      30  49 DESCRP
I                                   P  50  510QTY
I                                      52 170 DUMMY
I                                     171 220 RSTART
C           *ENTRY    PLIST
C                     PARM          USER10 10
C*
C*  Initialize fields used to communicate with std program
C*
C                     MOVE USER10    USER
C                     MOVEL'PRDRC2'  PGMNAM
C                     MOVE 'R'       RQSCOD           Read Rqs
C                CALL 'STDCMT'
C                     PARM           RQSCOD  1
C                     PARM           RTNCOD  1
C                     PARM           CMTID 220        Data struct
C            RTNCOD   IFEQ '1'                        Restart
C                     EXSR MOVLST                     Move to last
C SETON                                  71    Restart
C                     END
C*
C*  Initialize fields used in notify object
C*
C                     MOVEARTXT,1    RSTART           Move text
C*
C* Basic processing loop
C*
C            LOOP     TAG
C                     EXFMTPROMPT
C   98                GOTO END
C            PRODCT   CHAINPRDMSTR            61    Not found
C   61                GOTO LOOP
C            KEY      KLIST
C                     KFLD           PRODCT
C                     KFLD           LOCATN
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..
C           KEY       CHAINPRDLOCR            62    Not found
C   62                DO
C                     EXCPTRLSMST                     Release lck
C                     GOTO LOOP
C                     END
C                     ADD  QTY       ONHAND           Add
C                     ADD  QTY       LOCAMT
C                     UPDATPRDMSTR                    Update
C                     UPDATPRDLOCR                    Update
C*
C*  Commit and move to previous fields
C*
C           CMTID     COMIT
C                     EXSR MOVLST                     Move to last
C                     GOTO LOOP
C*  End of program processing
C*
C           END       TAG
```

```
     C                     MOVE 'D'       RQSCOD           Dlt Rqs
     C                     CALL 'STDCMT'
     C                     PARM           RQSCOD
     C                     PARM           RTNCOD
     C                     PARM           CMTID
     C                     SETON                     LR
     C*
     C*  Move to -Last Used- fields for operator feedback
     C*
     C           MOVLST    BEGSR
     C                     MOVE PRODCT    LSTPRD
     C                     MOVE LOCATN    LSTLOC
     C                     MOVE DESCRP    LSTDSC
     C                     MOVE QTY       LSTQTY
     C                     ENDSR
     OPRDMSTR E                RLSMST
** RTXT       Restart Text
Inventory Menu - Receipts Option
```

Processing flow:

The standard program is called from applications that must start again.

The application programs pass this parameter list to the standard program:
- Request code
- Return code
- Data structure name (the contents of the notify object)

Request codes perform the following operations:
- R (Read)

  Retrieves the last record added to the notify object with the same key. The return code is set as:

  0 No record is available (no start again required).

  1 Record returned in the information field for starting again (start again required).
- WA (Write)

  Writes a record to the file. This code can be used if you use a notify object for your own purposes. For example, if the program determines that the transaction needs to be started again, the program can write a record to the notify object to simulate what the system will do if a job or the system fails.
- DE (Delete)

  Deletes all records in the notify object with the same key. The return code is set as:

  0 No records exist to be deleted.

  1 One or more records were deleted.
- OE (Open)

  The O request code is optional and is used to avoid having to start the processing program each time it is called.
- CA (Close)

  After the open request code is used, using the close request code ensures that the file is closed.
- SA (Search)

  Returns the last record for this user. The program name is not used. This code can be used in an initial program to determine whether starting again is required.

Example: Code for a standard commit processing program

The standard commit (STDCMT) processing program performs the functions required to communicate with a single notify object used by all applications.

While the commitment control function automatically writes an entry to the notify object, a user-written standard program must process the notify object. The standard program simplifies and standardizes the approach.

The program is written to verify the parameters that were passed and perform the appropriate action as follows:

O=Open
: The calling program requests the notify object file be kept open on return. Because the notify object is opened implicitly by the RPG program, the program must not close it. Indicator 98 is set so that the program returns with LR off to keep the program's work areas and leaves the notify object open so it can be called again without excess overhead.

C=Close
: The calling program has determined that it no longer needs the notify object and requests a close. Indicator 98 is set off to allow a full close of the notify object.

R=Read
: The calling program requests that a record with matching key fields be read and passed back. The program uses the passed key fields to attempt to retrieve a record from NFYOBJP. If duplicate records exist for the same key, the last record is returned. The return code is set accordingly and, if the record existed, it is passed back in the data structure CMTID.

W=Write
: The calling program requests a record to be written to the notify object to allow the calling program to start again the next time it is called. The program writes the contents of the passed data as a record in NFYOBJP.

D=Delete
: The calling program requests that records for this matching key be deleted. This function is typically performed at the successful completion of the calling program to remove any information about starting again. The program attempts to delete any records for passed key fields. If no records exist, a different return code is passed back.

S=Search
: The calling program requests a search for a record for a particular user regardless of which program wrote it. This function is used in the program for sign-on to indicate that starting again is required. The program uses only the user name as the key to see if records exist. The return code is set appropriately, and the contents of the last record for this key (if it exists) are read and passed back.

The following example shows the standard commit processing program, STDCMT.

Standard commit processing program

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 111.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

           FNFYOBJP UF   E           K        DISK                      A
           ICMTID        DS
           I                                         1  10 UNQUSR
           I                                        11  20 UNQPGM
           I                                        21 220 BIGFLD
           C            *ENTRY    PLIST
           C                      PARM           RQSCOD  1
           C                      PARM           RTNCOD  1
           C                      PARM           CMTID 220
           C             UNQUSR   CABEQ*BLANKS   BADEND         H1 Invalid
           C             UNQPGM   CABEQ*BLANKS   BADEND         H2 Invalid
           C*
```

```
C*  'O' for Open
C*
C            RQSCOD   IFEQ 'O'                         Open
C                     SETON                     98    End LR
C                     GOTO END
C                     END
C*
C*  'C' for Close
C*
C            RQSCOD   IFEQ 'C'                         Close
C                     SETOF                     98
C                     GOTO END
C                     END
C*
C*  'R' for Read - Get last record for the key
C*
C            RQSCOD   IFEQ 'R'                        Read
C            KEY      KLIST
C                     KFLD           UNQUSR
C                     KFLD           UNQPGM
C            KEY      CHAINNFYOBJR              51    Not found
C   51                MOVE '0'       RTNCOD
C   51                GOTO END
C                     MOVE '1'       RTNCOD            Found
C            LOOP1    TAG
C            KEY      READENFYOBJR                  20 EOF
C   20                GOTO END
C                     GOTO LOOP1
C                     END
C*
C*  'W' FOR Write
C*
C            RQSCOD   IFEQ 'W'                         Write
C                     WRITENFYOBJR
C                     GOTO END
C                     END
C*
C*  'D' for Delete - Delete all records for the key
C*
C           RQSCOD    IFEQ 'D'                         Delete
C           KEY       CHAINNFYOBJR              51    Not found
C   51                MOVE '0'       RTNCOD
C   51                GOTO END
C                     MOVE '1'       RTNCOD            Found
C           LOOP2     TAG
C                     DELETNFYOBJR
C           KEY       READENFYOBJR                  20 EOF
C  N20                GOTO LOOP2
C                     GOTO END
C                     END
C*
C*  'S' for Search for the last record for this user
C*              (Ignore the -Program- portion of the key)
C*
C           RQSCOD    IFEQ 'S'                        Search
C           UNQUSR    SETLLNFYOBJR                 20 If equal
C  N20                MOVE '0'       RTNCOD
C  N20                GOTO END
C                     MOVE '1'       RTNCOD           Found
C           LOOP3     TAG
C           UNQUSR    READENFYOBJR                 20 EOF
C  N20                GOTO LOOP3
C                     GOTO END
C                     END
C*
C*  Invalid request code processing
C*
```

```
       C                     SETON                     H2    Bad RQS code
       C                     GOTO BADEND
       C*
       C*  End of program processing
       C*
       C           END       TAG
       C  N98                SETON                     LR
       C                     RETRN
       C* BADEND tag is used then fall thru to RPG cycle error return
       C           BADEND    TAG
```

Related concepts

"Example: Using a standard processing program to decide whether to restart the application"
The initial program can call the standard commit processing program to determine if it is necessary to start again. The workstation user can then decide whether to start again.

Example: Using a standard processing program to decide whether to restart the application

The initial program can call the standard commit processing program to determine if it is necessary to start again. The workstation user can then decide whether to start again.

The initial program passes a request code of S (search) to the standard program, which searches for any record for the user. If a record exists, the information for starting again is passed to the initial program and the information is displayed to the workstation user.

The commit identification in the notify object contains information that the initial program can display identifying what program needs to be started again. For example, the last 50 characters of the commit identification can be reserved to contain this information. In the application program, this information can be in a compile time array and moved to the data structure in an initialization step. Example: Code for a standard commit processing program shows how to include this in the application program.

The following example shows an initial program, which determines if a record exists in the notify object.

Example: Initial program

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 111.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7

             PGM
             DCLF       CMTINLD
             DCL        &RQSCOD *CHAR LEN(1) VALUE(S)  /* Search */
             DCL        &RTNCOD *CHAR LEN(1)
             DCL        &CMTID *CHAR LEN(220)
             DCL        &USER *CHAR LEN(10)
             DCL        &INFO *CHAR LEN(50)
             RTVJOBA    USER(&USER)
             CHGVAR     &CMTID (&USER *CAT XX)
                        /* The XX is required to prevent a blank Pgm nam */
             CALL       STDCMT PARM(&RQSCOD &RTNCOD &CMTID)
             IF         (&RTNCOD *EQ '1') DO /* RESTART REQD */
             CHGVAR     &INFO %SST(&CMTID 171 50)
             SNDRCVF    RCDFMT(RESTART)
             ENDDO
                        /*                                        */
                        /* Enter normal initial program statements  */
                        /*    or -TFRCTL- to first menu program      */
                        /*                                         */
             ENDPGM
```

Related concepts

"Example: Code for a standard commit processing program" on page 95
The standard commit (STDCMT) processing program performs the functions required to communicate with a single notify object used by all applications.

"Commit notify object" on page 48
A *notify object* is a message queue, data area, or database file that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally.

Troubleshooting transactions and commitment control

This information might help you troubleshoot commitment control errors, detect deadlocks, recover transactions after communications failure, know when to force commit and rollback operations, and when to cancel resynchronization and end long-running rollbacks.

Commitment control errors

When you use commitment control, it is important to understand which conditions cause errors and which do not.

In general, errors occur when commitment control functions are used inconsistently, such as running an End Commitment Control (ENDCMTCTL) command when files that use the commitment definition are still open.

Errors during commit processing

If a communications or system failure occurs during a commit operation, resynchronization might need to be performed to ensure that the transaction managers keep the data consistent on all the systems involved in the transaction. The behavior of the resynchronization and how it affects the commit operation depends on these factors :

- The `Wait for outcome` commitment option.
- The state of the transaction.

If the failure is catastrophic such that it can never be repaired, or it cannot be repaired in a timely manner, the system operators for other systems involved in the transaction must make a heuristic decision. The heuristic decision commits or rolls back the changes made on that system during the transaction. If the failure is repaired after such a decision, and the resynchronization detects that the decision caused data integrity problems, message CPD83D9 or CPD83E9 is sent to the QSYSOPR message queue.

Related concepts

"Commitment definition for two-phase commit: Not wait for outcome" on page 33
When a communication or system failure occurs during a commit operation so that resynchronization is required, the default is to wait until the resynchronization is finished before the commit operation completes.

"States of the transaction for two-phase commitment control" on page 28
A commitment definition is established at each location that is part of the transaction program network. For each commitment definition, the system keeps track of the state of its current transaction and previous transaction.

Error conditions

If an error occurs, an escape message that you can monitor for in a program is sent.

Here are some typical errors related to commitment control:

- Consecutive STRCMTCTL commands are run without an intervening ENDCMTCTL command.
- Files are opened under commitment control, but no STRCMTCTL command was run.

This is not an error condition for programs that run within an activation group that are to use the job-level commitment definition. The job-level commitment definition can be started only by a single program, but when started by a program, the job-level commitment definition is used by any program running in any activation group that is not using an activation-group-level commitment definition. Programs that run within an activation group that are to use the activation-group-level commitment definition must first start the activation-group-level commitment definition with the STRCMTCTL command.

- Files that are opened for output under commitment control are not journaled.
- The first open operation of a shared file places the file under commitment control, but subsequent open operations of the same shared file do not.
- The first open operation of a shared file does not place the file under commitment control, but subsequent open operations of the same shared file do.
- The record lock limit for the job is reached in a single transaction.
- The program issues a read operation, a commit operation, and a change to the same record. The read operation must be issued again after the commit operation because the commit operation has freed the lock on the record.
- For a one-phase location, resources placed under commitment control do not reside at the same location as resources already under commitment control for the commitment definition.
- Uncommitted changes exist when an ENDCMTCTL command is issued.

  This is not an error condition for the ENDCMTCTL command if all files are closed, any remote database is disconnected, and no API commitment resource is still associated with the commitment definition to be ended.
- A commit, rollback, or ENDCMTCTL command is run, and a STRCMTCTL command was not run.

  This is not an error condition for programs that run within an activation group and the job-level commitment definition is active. The job-level commitment definition can be started only by a single program, but when started by a program, the job-level commitment definition is used by any program running in any activation group that is not using an activation-group-level commitment definition. Programs that run within an activation group and are to use the activation-group-level commitment definition must first start the activation-group-level commitment definition with the STRCMTCTL command.
- An ENDCMTCTL command is run with files still open under commitment control for the commitment definition.
- A job performing a save operation has one or more commitment definitions that are not at a commitment boundary.
- A save-while-active operation ended because other jobs with committable resources did not reach a commitment boundary in the time specified for the SAVACTWAIT parameter.
- A save-while-active process was not able to continue because of API committable resources being added to more than one commitment definition for a single job.
- More than 1023 commitment definitions exist for a single job.
- The conversation to a remote location is lost due to a resource failure. This might cause the transaction to be rolled back.
- A one-phase resource that is opened for update is present at a node that did not initiate the commit operation. You must remove either the resource or the node that initiated the commit request.
- A commit operation is requested while the transaction is in rollback required (RBR) state. A rollback operation must be done.
- An API exit program issues a commit request or a rollback request.
- A trigger program issues a commit request or a rollback request for the commitment definition under which the trigger program was called.

The trigger program can start a separate commitment definition and issue a commit or rollback request for that definition.

Nonerror conditions

Here are some situations for commitment control in which no errors occur.

- A commit or rollback operation is run and no resources are under commitment control. This allows you to include commit or rollback operations in your program without considering whether there are resources under commitment control. It also allows you to specify a commit identification before making any committable changes.
- A commit or rollback operation is run and there are no uncommitted resource changes. This allows you to include commit and rollback operations within your program without considering whether there are uncommitted resource changes.
- A file under commitment control is closed and uncommitted records exist. This situation allows another program to be called to perform the commit or rollback operation. This occurs regardless of whether the file is shared. This function allows a subprogram to make database changes that are part of a transaction involving multiple programs.
- A job ends, either normally or abnormally, with uncommitted changes for one or more commitment definitions. The changes for all commitment definitions are rolled back.
- An activation group ends with pending changes for the activation-group-level commitment definition. If the activation group is ending normally and there are no errors encountered when closing any files opened under commitment control scoped to the same activation group that is ending, an implicit commit is performed by the system. Otherwise, an implicit rollback is performed.
- A program accesses a changed record again that has not been committed. This allows a program to:
  - Add a record and update it before specifying the commit operation.
  - Update the same record twice before specifying the commit operation.
  - Add a record and delete it before specifying the commit operation.
  - Access an uncommitted record again by a different logical file (under commitment control).
- You specify LCKLVL(*CHG or *CS) on the STRCMTCTL command and open a file with a commit operation for read-only. In this case, no locks occur on the request. It is treated as if commitment control is not in effect, but the file does appear on the WRKJOB menu option of files under commitment control.
- You issue the STRCMTCTL command and do not open any files under commitment control. In this situation, any record-level changes made to the files are not made under commitment control.

Error messages to monitor during commitment control

Several different error messages can be returned by the commit or rollback operations or sent to the job log, depending on the type of message and when the error occurred.

The error messages can occur during the following processing:

- Normal commit or rollback processing
- Commit or rollback processing during job process end
- Commit or rollback processing during activation group end

You cannot monitor for any of the following messages during activation group end or job process end. Also, you can only monitor for CPFxxxx messages. CPDxxxx messages are always sent as diagnostic messages, which cannot be monitored. Any errors encountered when ending an activation-group-level commitment definition during activation group end or ending any commitment definition during job end are left in the job log as diagnostic messages.

Error messages related to commitment control to look for are as follows:

CPD8351
Changes might not have been committed.

CPD8352
Changes not committed at remote location &3.

CPD8353
: Changes to relational database &1 might not have been committed.

CPD8354
: Changes to DDM file &1 might not have been committed.

CPD8355
: Changes to DDL object &1 might not have been committed.

CPD8356
: Rolled back changes might have been committed.

CPD8358
: Changes to relational database &1 might not have been rolled back.

CPD8359
: Changes to DDM file &1 might not have been rolled back.

CPD835A
: Changes to DDL object &3 might not have been rolled back.

CPD835C
: Notify object &1 in &2 not updated.

CPD835D
: DRDA resource does not allow SQL cursor hold.

CPF835F
: Commit or rollback operation failed.

CPD8360
: Members or files or both were already deallocated.

CPD8361
: API exit program &1 failed during commit.

CPD8362
: API exit program &1 failed during roll back.

CPD8363
: API exit program &1 ended after &4 minutes during commit.

CPD8364
: API exit program &1 ended after &4 minutes during rollback.

CPD836F
: Protocol error occurred during commitment control operation.

CPD83D1
: API resource &4 cannot be last agent.

CPD83D2
: Resource not compatible with commitment control.

CPD83D7
: Commit operation changed to rollback.

CPD83D9
: A heuristic mixed condition occurred.

CPF83DB
: Commit operation resulted in rollback.

CPD83DC
: Action If Problems Used to determine commit or rollback operation; reason &2.

CPD83DD
Conversation ended; reason &1.

CPD83DE
Return information not valid.

CPD83EC
API exit program &1 voted rollback.

CPD83EF
Rollback operation started for next logical unit of work.

CPF8350
Commitment definition not found.

CPF8355
ENDCMTCTL not allowed. Pending changes active.

CPF8356
Commitment control ended with &1 local changes not committed.

CPF8358
Notify object &1 in &2 not updated.

CPF8359
Rollback operation failed.

CPF835A
End of commitment definition &1 canceled.

CPF835B
Errors occurred while ending commitment control.

CPF835C
Commitment control ended with remote changes not committed.

CPF8363
Commit operation failed.

CPF8364
Commitment control parameter value is not valid. Reason code &3.

CPF8367
Cannot perform commitment control operation.

CPF8369
Cannot place API commitment resource under commitment control; reason code &1.

CPF83D0
Commitment operation not allowed.

CPF83D2
Commit complete == Resynchronization in progress has been returned.

CPF83D3
Commit complete == Heuristic Mixed has been returned.

CPF83D4
Logical unit of work journal entry not sent.

CPF83E1
Commit operation failed due to constraint violation.

CPF83E2
Rollback operation required.

CPF83E3
: Requested nesting level is not active.

CPF83E4
: Commitment control ended with resources not committed.

CPF83E6
: Commitment control operation completed with resynchronization in progress.

CPF83E7
: Commit or rollback of X/Open global transaction not allowed.

Monitoring for errors after a CALL command

When a program that uses commitment control is called, monitor for unexpected errors and perform a rollback operation if an error occurs.

For example, uncommitted records can exist when a program encounters an unexpected error such as an RPG divide-by-zero error.

Depending on the status of the inquiry message reply (INQMSGRPY) parameter for a job, the program sends an inquiry message or performs a default action. If the operator response or the default action ends the program, uncommitted records still exist waiting for a commit or rollback operation.

If another program is called and causes a commit operation, the partially completed transaction from the previous program is committed.

To prevent partially completed transactions from being committed, monitor for escape messages after the CALL command. For example, if it is an RPG program, use the following coding:

```
CALL RPGA
MONMSG MSGID(RPG9001)
EXEC(ROLLBACK) /*Rollback if pgm is canceled*/
```

If it is a COBOL program:

```
CALL COBOLA
MONMSG MSGID(CBE9001)
EXEC(ROLLBACK) /*Rollback if pgm is canceled*/
```

Failure of normal commit or rollback processing

Errors might occur at any time during commit or rollback processing.

The following table divides this processing into four situations. The middle column describes the actions taken by the system when it encounters errors during each situation. The third column suggests what you or your application must do in response to the messages. These suggestions are consistent with the way commitment control processing is handled by the system.

Situation	**Commit or rollback processing**	**Suggested action**
Record-level I/O **commit** fails	• If the error occurs during the prepare wave, the transaction is rolled back and message CPF83DB is sent. • If the error occurs during the committed wave, commit processing continues to commit as many remaining resources as possible. Message CPF8363 is sent at the end of commit processing.	Monitor for messages; handle as you want

Situation	Commit or rollback processing	Suggested action
Object-level or commit and rollback exit program for API commitment resource fails during commit	• If the error occurs during the prepare wave, the transaction is rolled back and message CPF83DB is sent. • If the error occurs during the committed wave, processing continues to commit or roll back as many remaining resources as possible. One of the following messages is returned, depending on the commitment resource type: – CPD8353 – CPD8354 – CPD8355 – CPD8361 Message CPF8363 is sent at the end of commit processing.	Monitor for messages; handle as you want
Record-level I/O **rollback** fails	1. Returns CPD8356 2. Attempt to continue processing to rollback object-level or API commitment resources 3. Returns CPF8359 at end of processing	Monitor for messages; handle as you want
Object-level or commit and rollback exit program for API commitment resources fails during rollback	1. Returns one of the following messages depending on the commitment resource type: • CPD8358 • CPD8359 • CPD835A • CPD8362 2. Continues processing 3. Returns CPF8359 at end of processing	Monitor for messages; handle as you want

Commit or rollback processing during job end

All of the situations described in the previous table also apply when a job is ending except that one of the following messages is sent:
- CPF8356 if only local resources are registered
- CPF835C if only remote resources are registered
- CPF83E4 if both local and remote resources are registered

In addition, one of two messages might appear specific to job completion if a commit and rollback exit program for an API committable resource has been called. If the commit and rollback exit program does not complete within 5 minutes, the program is canceled; a diagnostic message CPD8363 (for commit) or CPD8364 (for rollback) is sent; and the remainder of the commit or rollback processing continues.

Commit or rollback processing during initial program load (IPL)

All of the situations described in the previous table also apply during IPL recovery for commitment definitions except that message CPF835F is sent instead of message CPF8359 or CPF8363. Messages that get sent for a particular commitment definition might appear in the job log for one of the QDBSRVxx jobs or the QHST log. In the QHST log, message CPI8356 indicates the beginning of IPL recovery for a particular commitment definition. Message CPC8351 indicates the end of IPL recovery for a particular commitment definition and any other messages regarding the recovery of that commitment definition is found between those two messages.

One of two messages might appear specific to a commitment definition if a commit and rollback exit program for an API committable resource has been called. If the commit and rollback exit program does not complete within 5 minutes, the program is canceled; a diagnostic message CPD8363 (for commit) or CPD8364 (for rollback) is sent; and the remainder of the commit or rollback processing continues.

Detecting deadlocks

A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

Do the following steps to find out if a deadlock condition has occurred and fix it if it has:

1. Locate the suspended job in the list of active jobs.
2. Look at the objects the job is waiting to lock.
3. For all the objects the job is waiting to lock, look at the list of lock holders (transactions or jobs) and try to find a conflicting lock corresponding to the level requested by the suspended job.
4. If a transaction is holding a conflicting lock, display the jobs associated with this transaction and see if one of them is waiting to lock.
5. Determine if this waiting job is trying to lock one of the objects locked by the initial suspended job. When you find the job that is trying to lock on of the objects locked by the initial suspended job, you can identify the objects in question as the trouble spots.
6. Investigate the transaction in order to determine the appropriate course of action.
   a. Look at the transaction properties to find out what application initiated it and then look at the application code.
   b. Or trace the transaction's actions up to this point by finding the Commit cycle ID in the transaction properties and then searching in a journal for entries containing this ID. To do this, you can use the Retrieve Journal Entry (RTVJRNE) command and specify the CMTCYCID parameter.
   c. After obtaining relevant information, you can choose to force a rollback or commit operation.

   Related tasks

   "Minimizing locks" on page 67
   A typical way to minimize record locks is to release the record lock. (This technique does not work if LCKLVL(*ALL) has been specified.)

   Determining the status of a job

   "Displaying locked objects for a transaction" on page 63
   You can display locked objects for global transactions with transaction-scoped locks only.

   "Displaying jobs associated with a transaction" on page 64
   To display jobs associated with a transaction, follow these steps.

   "Displaying commitment control information" on page 62
   iSeries Navigator can display information about all transactions (logical units of work) on the system. iSeries Navigator can also display information about the job, if any, associated with a transaction.

"When to force commit and rollback operations and when to cancel resynchronization"
The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

Related reference

Retrieve Journal Entry (RTVJRNE) command

Recovering transactions after communications failure

These instructions help you handle transactions performing work on a remote system after communication with that system fails.

In case of a communications failure, the system typically completes the resynchronization with any remote system automatically. However, if the failure is catastrophic such that the communications will never be reestablished to the remote system (if, for instance, the communication line is cut), you must cancel resynchronization and restore transactions yourself. The transactions also might be holding locks that need to be released.

1. In iSeries Navigator, display commitment control information for the transaction with which you are working.
2. Find the transaction of interest that is trying to resynchronize with the remote system. The **Resynchronization in Progress** field for that transaction is set to **yes**.
3. Look for transactions that had a connection to the remote system by checking the resource Status for individual transactions.
4. After identifying transactions, you must force commit or force rollback depending on the state of the transaction.
5. You can make the decision to commit or rollback after you investigate the transaction properties.
   - You can use the **Unit of Work ID** to find other parts of the transaction on other systems.
   - You can also determine to commit or rollback from the state of transaction. For example, if a database transaction is performing two-phase commit during communication failure and its state after the failure is "prepared" or "last agent pending", you might choose to force commit on the transaction.
6. After forcing a commit or rollback on the transactions in doubt, stop resynchronization on the failed connection for the identified transactions.

Related tasks

"Displaying commitment control information" on page 62
iSeries Navigator can display information about all transactions (logical units of work) on the system. iSeries Navigator can also display information about the job, if any, associated with a transaction.

"When to force commit and rollback operations and when to cancel resynchronization"
The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

When to force commit and rollback operations and when to cancel resynchronization

The decision to force a commit or rollback operation is called a *heuristic decision*. This action enables an operator to manually commit or roll back the resources for a transaction that is in a prepared state.

When you make a heuristic decision, the state of the transaction becomes heuristic mixed if your decision is inconsistent with the results of the other locations in the transaction. You must take responsibility for determining the action taken by all the other locations that participated in the transaction and resynchronizing the database records.

Before you make a heuristic decision, gather as much information as you can about the transaction. Display the jobs that are associated with the commitment definition and make a record of what journals and files are involved. You can use this information later if you need to display journal entries and apply or remove journaled changes manually.

The best place to find out information about a transaction is to look at the location that was the initiator for that transaction. However, the decision to commit or roll back might be owned by an API resource or by a last agent.

If an API resource was registered as a last agent resource, the final decision of whether to commit or roll back is owned by the API resource. You need to look at information about the application and how it uses the API resource to determine whether to commit or to roll back.

If the transaction has a last agent selected, the last agent owns the decision to commit or roll back. Look at the status of the last agent for information about the transaction.

When you must make heuristic decisions or cancel resynchronization due to a system or communications failure that cannot be repaired, you can find all transactions in doubt by using the following steps:

1. In iSeries Navigator, expand the system you want to work with.
2. Expand **Databases** and the local database for the system.
3. Expand **Transactions.**
4. Expand **Database Transactions** or **Global Transactions**.

In this display windows, you can see the commitment definition, resynchronization status, the current unit of work ID, and the current unit of work state for each transaction. Look for transactions with the following information:

- Transactions with a **Logical Unit of Work State** of Prepared or Last Agent Pending.
- Transactions that show **Resynchronization in Progress** status of yes.

To work with the job that is participating in the transaction on this system right-click the transaction and select **job**.

When you right-click the transaction, you can also select **Force Commit**, **Force Rollback**, or **Cancel Resynchronization.**

Before making a heuristic decision or canceling resynchronization, you might want to check the status of the jobs on other systems associated with the transaction. Checking the jobs on remote systems might help you avoid decisions that cause database inconsistencies between systems.

1. Right-click the transaction you want to work with.
2. Select **Resource Status**.
3. In the Resource Status dialog, select the **Conversation** tab for SNA connections; select **Connection** for TCP/IP connections.

Each conversation resource represents a remote system that is participating in the transaction. On the remote systems, you can use iSeries Navigator to see the transactions associated with the transaction.

The base portion of the unit of work ID is the same on all the systems. When you display commitment control information on the remote system, look for the base portion of the unit of work ID that is the same on the local system.

For example, if the unit of work ID on the local system starts with: `APPN.RCHASL97.X'112233445566`, look for the unit of work ID on the remote system that also starts with `APPN.RCHASL97.X'112233445566`.

Related concepts

"XA transaction support for commitment control" on page 42
DB2 Universal Database (UDB) for iSeries can participate in X/Open global transactions.

"Starting commitment control" on page 47
To start commitment control, use the Start Commitment Control (STRCMTCTL) Command.

Related tasks

"Detecting deadlocks" on page 106
A deadlock condition can occur when a job holds a lock on an object, object A, and is waiting to obtain a lock on another object, object B. At the same time, another job or transaction currently holds a lock on object B and is waiting to obtain a lock on object A.

"Recovering transactions after communications failure" on page 107
These instructions help you handle transactions performing work on a remote system after communication with that system fails.

"Displaying commitment control information" on page 62
iSeries Navigator can display information about all transactions (logical units of work) on the system. iSeries Navigator can also display information about the job, if any, associated with a transaction.

Ending a long-running rollback

You might want to end long-running rollbacks that consume critical processor time, lock resources, or take up storage space.

A rollback operation removes all changes made within a transaction since the previous commit operation or rollback operation. During a rollback operation, the system also releases locks related to the transaction. If the system contains thousands of transactions, the system can take hours to complete a rollback operation. These long-running rollbacks can consume critical processor time, lock resources or take up storage space.

Before you end a long-running rollback, you need to know which commitment definitions are being rolled back and what state the commitment definitions are in. The State field for commitment definitions that are rolling back is set to ROLLBACK IN PROGRESS.

Use the Work with Commitment Definitions (WRKCMTDFN) command to check the status of a rollback by following these steps:

1. Type WRKCMTDFN JOB(*ALL) from the character-based interface.
2. Type F11 to display the State field.

If you end a long-running rollback, files that were changed during the transaction will be left with partial transactions. You must not end a rollback if your files cannot have partial transactions. To see which files were changed during the transaction, choose option 5 to display status from the WRKCMTDFN list. Press F6 to display resource status and select Record Level.

You must have All Object (*ALLOBJ) special authority to end a long-running rollback. To end a long-running rollback, follow these steps:

1. Type WRKCMTDFN JOB(*ALL) from the character-based interface.
2. Type option 20 (End rollback) on the commitment definition you want to end.

Files with partial transactions have the Partial Transactions Exist, Rollback Ended field set to *YES in the output from the Display File Description (DSPFD) command. You must remove partial transactions before the file can be used. You can remove partial transactions by deleting the file and restoring the file from a prior save. If you do not have a prior save, you can use the Change Journaled Object (CHGJRNOBJ) command to reset the Partial Transaction Exists state so that you can open the file. Using the CHGJRNOBJ requires you to edit the file to bring the file to a consistent state. You must use the CHGJRNOBJ command only if no prior save is available.

Disabling the ability to end a long-running rollback

Users with *ALLOBJ special authority can end rollbacks by default. If you want to restrict users who have *ALLOBJ special authority from ending rollbacks, you can do this by creating data area QGPL/QTNNOENDRB.

Related reference

Create Data Area (CRTDTAARA) command

Related information for commitment control

Listed here are the product manuals and IBM Redbooks (in PDF format), Web sites, and information center topics that relate to the commitment control topic. You can view or print any of the PDFs.

Manuals

- COBOL/400® User's Guide on the V5R1 Supplemental Manuals Web site .
- RPG/400® User's Guide on the V5R1 Supplemental Manuals Web site .

IBM Redbooks

- Advanced Functions and Administration on DB2 Universal Database for iSeries (5529 KB)
- Stored Procedures, Triggers, and User Defined Functions on DB2 Universal Database for iSeries (5836 KB)
- Striving for Optimal Journal Performance on DB2 Universal Database for iSeries (3174 KB)

Web sites

- The Open Group (www.opengroup.org)

Other information

- Database programming
- SQL programming
- XA APIs
- Journal management

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF in your browser (right-click the link above).
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you would like to save the PDF.
4. Click **Save**.

Downloading Adobe Acrobat Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Code license and disclaimer information

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Programming Interface Information

This Commitment control publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

Terms and conditions

Printed in USA