InterSystems™
IRIS Data Platform

Developing InterSystems Applications

Version 2025.1
2025-06-03

InterSystems Corporation   1 Memorial Drive   Cambridge MA 02142   www.intersystems.com

Table of Contents

1 Transaction Processing

A *transaction* is a logical unit of work that groups multiple atomic operations into a single, indivisible action. An atomic operation is always fully executed in any circumstance, including if an error occurs. Typically, a transaction consists of several atomic operations executed in a specific order and treated as a single action.

This document provides an overview of transaction processing in the InterSystems IRIS® data platform. It explains how to define and manage transactions, handle errors using rollbacks, and implement strategies for disaster recovery.

1.1 About Transactions in InterSystems IRIS

In InterSystems IRIS, an *atomic operation* consists of a single operation that changes an object or row, including creation, deletion, and modification.

However, applications often require combining multiple atomic operations to complete a task. For example, consider a bank transferring money from one account to another. This task involves at least two separate operations: subtracting the transfer amount from the sender's account balance and adding the same amount to the recipient's account balance. Each of these updates is an atomic operation on its own. However, to maintain accurate financial records, they must be treated as a single unit—either both happen together, or neither happens at all. By grouping these operations within a single transaction, the system ensures that if something goes wrong after the first update but before the second, it can undo the changes, returning both accounts to their original state.

Transaction processing commands allow you to specify the sequence of operations that constitute a complete transaction. One command marks the beginning of the transaction, and after executing a series of operations, another command marks the end. If any part of the transaction fails, a rollback — either from developer-defined rollback logic or triggered automatically in cases like system failure or process termination— reverses the entire sequence.

1.2 Managing Transactions Within Applications

- Transaction Processing Commands
- Transaction Processing Details
- Examples of Transaction Processing Within Applications

1.2.1 Transaction Processing Commands

The available transaction processing commands are summarized in the following sections. These Python, SQL, and ObjectScript commands are identical in functionality, with any exceptions noted in the command definition below.

1.2.1.1 Python Transaction Commands

iris.tstart()

Begins a new transaction. Each call increases the transaction level.

iris.gettlevel()

Detects whether a transaction is currently in progress. The returned value reflects the current transaction level—the number of nested transactions opened.

Returns the current transaction level:

- >0: In a transaction; value indicates nesting level (for example, 2 means two nested **iris.tstart()** commands are active).
- 0: No transaction active

iris.tcommit()

Commits the current transaction level.

iris.trollbackone()

Rolls back changes made during the most recent nested transaction only. Outer transactions are unaffected.

iris.trollback()

Rolls back all active transactions. Resets the transaction level to 0.

Important: Using the **iris.trollback()** command can be potentially destructive, as it rolls back all active transactions for the current process. To avoid unintentionally affecting transactions beyond the one you are currently working on, InterSystems strongly recommends using **iris.trollbackone()**.

1.2.1.2 SQL Transaction Commands

InterSystems IRIS supports the ANSI SQL operations **COMMIT WORK** and **ROLLBACK WORK** (in InterSystems SQL, the keyword WORK is optional). It also supports the InterSystems SQL extensions **SET TRANSACTION**, **START TRANSACTION**, **SAVEPOINT**, and **%INTRANSACTION**.

SET TRANSACTION

Sets transaction parameters without starting a transaction.

START TRANSACTION

Begins a new transaction.

%INTRANSACTION

Determines whether a transaction is currently in progress. This command sets the *SQLCODE* variable based on the transaction state, but does not return a value.

After invoking **%INTRANSACTION**, the value of *SQLCODE* is:

- 0: A transaction is in progress.
- 100: No transaction is in progress.
- <0: In transaction, journaling disabled.

SAVEPOINT name

Marks a named point within a transaction for partial rollback.

ROLLBACK TO SAVEPOINT name

Rolls back to the named savepoint without ending the entire transaction.

ROLLBACK

Rolls back *all* changes made since the transaction began. Releases all locks and resets **%INTRANSACTION** to 0.

Important: Using the **ROLLBACK** command can be potentially destructive, as it rolls back all active transactions for the current process. To avoid unintentionally affecting transactions beyond the one you are currently working on, InterSystems strongly recommends using **ROLLBACK TO SAVEPOINT**.

COMMIT

Commits all changes made during the transaction, including those after any savepoints. **COMMIT** always finalizes the entire transaction, including any savepoints or nested transactions.

Tip: SQL does not support committing part of a nested transaction. If you use **SAVEPOINT**, do not use **COMMIT** unless you intend to finalize the entire transaction.

1.2.1.3 ObjectScript Transaction Commands

tstart

Begins a new transaction. Each call increases the transaction level.

$TLEVEL

Detects whether a transaction is currently in progress. The returned value reflects the current transaction level—the number of nested transactions opened.

Returns the current transaction nesting level.

- >0: In a transaction; value indicates nesting level (for example, 2 means two nested **tstart** commands are active).
- 0: No transaction.

tcommit

Commits the current transaction level only.

trollback 1

Rolls back the current nested transaction level only. Outer transactions are unaffected.

trollback

Rolls back all active transactions. Resets **$TLEVEL** to 0.

Important: Using the **trollback** command without an argument can be potentially destructive, as it rolls back all active transactions for the current process. To avoid unintentionally affecting transactions beyond the one you are currently working on, InterSystems strongly recommends using **trollback 1**.

1.2.1.4 Nested Transactions

While transactions should ideally be managed within a single language, it is possible to call transaction commands across languages. For example, from an SQL statement, you can call a stored procedure written in Python that uses Python transaction commands. Similarly, a Python method can call a stored procedure written in SQL that uses SQL transaction commands.

It's important to note that nested transactions behave differently depending on the language. In ObjectScript, you nest transactions by issuing **tstart** multiple times, and you can commit or roll back at specific levels using **tcommit** or **trollback 1**. In contrast, SQL uses the **SAVEPOINT** command to create nested rollback points. You can roll back to a specific savepoint using **ROLLBACK TO SAVEPOINT**, but any **COMMIT** statement ends all active transactions, including those started with **SAVEPOINT**. When mixing languages, be mindful of these behavioral differences to avoid unintended commits or rollbacks.

1.2.2 Transactions and Journaling

Transaction commands, such as begin, commit, and rollback, are recorded in the journal and can be accessed via the *Management Portal* (System Operation > Journals). Journaling plays a critical role in supporting backups and ensuring disaster recovery. Read about backups and journaling to learn more.

1.2.3 Examples of Transaction Processing Within Applications

The following are examples of transaction processing. The code below performs database modifications and then transfers funds from one account to another:

Python

```
# Transfer funds from one account to another in Python with SQL

def transfer(from_account, to_account, amount):
  try:
    iris.tstart()
    iris.sql.exec("UPDATE Bank.Account SET Balance = Balance - ? WHERE AccountNum = ?", (amount,
from_account))
    iris.sql.exec("UPDATE Bank.Account SET Balance = Balance + ? WHERE AccountNum = ?", (amount,
to_account))
    iris.tcommit()
    return "Transfer succeeded"
  except Exception as e:
    iris.trollbackone()
    return f"Transaction failed: {e}"
```

SQL

```
START TRANSACTION;

UPDATE Bank.Account
SET Balance = Balance - 500
WHERE AccountNum = '12345';

UPDATE Bank.Account
SET Balance = Balance + 500
WHERE AccountNum = '67890';

COMMIT;
```

Class Member

```
 ClassMethod TransferFunds(from As %String, to As %String, amount As %Double) As %Status
 {
    // Transfer funds from one account to another in ObjectScript
    tstart
    try {
        set acctFrom = ##class(Bank.Account).%OpenId(from)
        set acctTo = ##class(Bank.Account).%OpenId(to)
        if acctFrom = "" || acctTo = "" {
            throw ##class(%Exception.StatusException).CreateFromStatus($$$ERROR("Account not found"))
        }

        set acctFrom.Balance = acctFrom.Balance - amount
        set acctTo.Balance = acctTo.Balance + amount

        set status = acctFrom.%Save()
        $$$ThrowOnError(status)
        set status = acctTo.%Save()
        $$$ThrowOnError(status)

        tcommit
        return $$$OK
    } catch ex {
        trollback 1
        return ex.AsStatus()
    }
 }
```

1.3 Handling Transaction Errors with Rollbacks

- Understanding Rollbacks
- Rollback Commands
- Viewing Rollback Logs
- Rollback Example

1.3.1 Understanding Rollbacks

A transaction typically consists of a sequence of atomic operations that either completes entirely or not at all. If an error or system malfunction interrupts the transaction, the system uses *rollback* logic you've defined to undo any completed operations to restore the state before the failure. To cite the example of a bank transaction, rolling back an incomplete transaction prevents money from being removed from one account but not credited to another in the case of a system crash mid-process. As long as the removal of money from one account is grouped in the same transaction as depositing the money in another, a rollback ensures that each account is credited appropriately.

When developing your transaction, include a rollback command for error handling. Using structured error-handling mechanisms — such as **TRY-CATCH** in ObjectScript or **try except** in Python — is best practice. InterSystems IRIS is equipped to manage rollbacks automatically in cases of system failure or process termination. For more information, see system automated rollbacks.

1.3.2 Rollback Commands

Applications typically implement a rollback command in the error-handling block, such as CATCH in **TRY/CATCH** or EXCEPT in **TRY/EXCEPT**. To roll back the current nested level of transactions:

- *Python:* **iris.trollbackone()**
- *SQL:* **ROLLBACK TO SAVEPOINT**
- *ObjectScript:* **trollback 1**

Note: These commands will also work if called in the Terminal while a transaction runs.

1.3.2.1 Viewing Rollback Logs

After a rollback occurs, if you have enabled the *LogRollback* configuration option, the system logs the details of the rollback in the messages.log file, which you can view in the *Management Portal* (System Operation > System Logs > Messages Log).

1.3.3 Rollback Example

The following code samples illustrate how to use rollback commands within transactions, along with error handling to maintain data integrity.

Each sample starts a transaction and sets up an error handler. Operations on data structures or variables are executed, including an intentional error to trigger the rollback. If an error occurs, the handler undoes all changes and displays "Transaction Failed." If the line triggering an error were deleted and no error occurs, the transaction is committed successfully with a commit command, and a success message, “Transaction Committed," is displayed.

Python

```
def rollback_example():
    try:
        iris.tstart()
        # Withdraw too much from the account to simulate a failure
        result1 = iris.sql.exec("UPDATE Bank.Account SET Balance = Balance - 1000 WHERE AccountNum = 
?", "12345")
        result2 = iris.sql.exec("UPDATE Bank.Account SET Balance = Balance + 1000 WHERE AccountNum = 
?", "67890")

        # Simulate a failure if the balance drops below zero
        balance = iris.sql.exec("SELECT Balance FROM Bank.Account WHERE AccountNum = ?", 
"12345").first()[0]
        if balance < 0:
            raise Exception("Insufficient funds")

        iris.tcommit()
        print("Transaction Committed")
    except Exception as e:
        iris.trollbackone()
        print(f"Transaction Failed: {e}")
```

ObjectScript

```
TRY {
   NEW balance
   tstart
   &sql(UPDATE Bank.Account SET Balance = Balance - 1000 WHERE AccountNum = '12345')
   &sql(UPDATE Bank.Account SET Balance = Balance + 1000 WHERE AccountNum = '67890')

   &sql(SELECT Balance INTO :balance FROM Bank.Account WHERE AccountNum = '12345')
   IF balance < 0 {
       THROW ##class(%Exception.StatusException).CreateFromStatus($$$ERROR("Insufficient funds"))
   }

   tcommit
   WRITE !, "Transaction Committed"
} CATCH ex {
   trollback 1
   WRITE !, "Transaction Failed: ", ex.DisplayString()
}
```

Class Member

```
ClassMethod RollbackExample() As %Status
{
   try {
       // Open account objects
       set acctFrom = ##class(Bank.Account).%OpenId("12345")
       set acctTo = ##class(Bank.Account).%OpenId("67890")
       if (acctFrom = "" || acctTo = "") {
           throw ##class(%Exception.StatusException).CreateFromStatus($$$ERROR("Account not found"))
       }

       // Attempt fund transfer with rollback on failure
       if (acctFrom.Balance < 1000) {
          throw ##class(%Exception.StatusException).CreateFromStatus($$$ERROR("Insufficient funds"))

       }

       tstart
       set acctFrom.Balance = acctFrom.Balance - 1000
       set acctTo.Balance = acctTo.Balance + 1000

       set status = acctFrom.%Save()
       $$$ThrowOnError(status)
       set status = acctTo.%Save()
       $$$ThrowOnError(status)

       tcommit
       write "Transaction Committed",!
       return $$$OK
   } catch ex {
       trollback 1
       write "Transaction Failed: ", ex.DisplayString(),!
       return ex.AsStatus()
   }
}
```

1.4 *Transaction Resiliency and Recovery Functionality*

- Automatic Rollbacks
- Backups and Journaling for Transaction Integrity
- Managing Concurrency with Rollbacks

1.4.1 Automatic Rollbacks

InterSystems IRIS automatically performs a rollback in cases of system failure or specific events such as process termination. Transaction rollback occurs automatically during each of the three following circumstances:

- *At the time of InterSystems IRIS startup, if recovery is needed.* When you start InterSystems IRIS and it determines that recovery is required, the system rolls back any incomplete transactions.
- **Process termination.** Halting a process using a **HALT** command (for your current process) automatically or the **^RESJOB** utility (for other running processes that are NOT your current process) affects in-progress transactions differently depending on the process type. Halting a non-interactive process (or a background job) results in the system automatically rolling back the transaction. If the process is interactive, the system displays a prompt in that process's Terminal session, asking whether to commit or rollback the transaction. This applies whether the process is halted directly or through **^RESJOB**.
- *System managers roll back incomplete transactions by running the **^JOURNAL** utility.* When you select the Restore Globals From Journal option from the **^JOURNAL** utility main menu, the journal file is restored, and all incomplete transactions are rolled back.

1.4.2 Backups and Journaling for Transaction Integrity

Journaling ensures transaction integrity by recording a time-sequenced log of database changes. Each instance of InterSystems IRIS maintains a journal that logs all **SET** and **KILL** operations made during transactions—regardless of the journal setting of the affected databases—as well as all **SET** and **KILL** operations for databases whose Global Journal State is set to "Yes.”

Backups can be performed during transaction processing; however, the resulting backup file may contain partial or uncommitted transactions, which could compromise transactional consistency if restored in isolation.

In the event of a disaster that requires restoring from a backup:

1. Restore the backup file
2. Apply journal files to the restored copy of the database

Applying journal files restores all journaled updates—from the time of the backup up to the point of failure—to the recovered database. Applying journals maintains transactional integrity by completes any partial transactions and rolls back those that were not committed.

For more information, see also:

- ECP Recovery Process, Guarantees, and Limitations
- Journaling
- Importance of Journals
- Backup and Restore

1.4.3 Managing Concurrency with Rollbacks

- $INCREMENT and $SEQUENCE in Transactions and Rollbacks
- Lock Behavior with Transactions

1.4.3.1 $INCREMENT and $SEQUENCE in Transactions and Rollbacks

The primary use case for **$INCREMENT** and **$SEQUENCE** is to increment a counter before inserting new records into a database. These functions provide a fast alternative to a lock command, allowing multiple processes to increment a counter concurrently without blocking each other.

Calls to **$INCREMENT** and **$SEQUENCE** are not considered to be part of a transaction and are not journaled, regardless of whether they are invoked explicitly or implicitly—such as through **%Save()**, **_Save()**, or **CREATE TABLE**. Their effects cannot be rolled back.

Because **$INCREMENT** and **$SEQUENCE** are not journaled, rolling back a transaction does not affect the values they have allocated. If a transaction that used **$INCREMENT** is rolled back, the counter is not decremented, as adjusting the counter retroactively could disrupt other transactions, so the next use of **$INCREMENT** will pick up where the previous left off, even if the transaction that it occurred in was reverted through a rollback. This means skipped values can occur, but avoiding potential inconsistencies takes priority. Similarly, any integer values returned by **$SEQUENCE** remain allocated and unavailable to future calls, even if the transaction that assigned them was rolled back.

Note: **%Save** and **_Save** use **$INCREMENT** by default. **CREATE TABLE** uses **$SEQUENCE** by default. Whether your class uses **$SEQUENCE** or **$INCREMENT** is defined in the IdFunction Storage Keyword, which can be configured as needed.

1.4.3.2 Lock Behavior with Transactions

Releasing a lock (**iris.lock()** in Python, **LOCK** in ObjectScript or **LOCK TABLE** in SQL) during a transaction may result in one of two possible states: the lock is fully released and immediately available to other processes, or it may enter a *delock* state. In a delock state, the lock behaves as released within the current transaction, allowing further lock operations on the same resource from within that transaction. However, to other processes, the lock remains active and unavailable until the transaction is either committed or rolled back—at which point the lock is fully released. To avoid locking conflicts, be mindful of when locks are released during a transaction and monitor for delock status using the *Monitor Locks* page.

Additionally, when configuring a lock, you can specify a timeout. If a lock attempt times out, the system sets the value of **$TEST**, which reflects the outcome of the lock attempt but is not affected by a later rollback of the transaction.

For more information about delock states, lock behavior, and best practices, refer to Managing Transactions and Locking with Python and Lock Management.

1.5 Advanced and Legacy Transaction Controls

- Suspending All Current Transactions
- The Legacy Utility ^%ETN and Transactions

1.5.1 Suspending All Current Transactions

You can temporarily suspend all current transactions within a process using the **TransactionsSuspended()** method. Changes made while transactions are suspended cannot be rolled back. Changes made before or after the suspension are still able to be rolled back. This is a potent feature that should be used with caution. If used recklessly, it can lead to incomplete and irreversible changes, which may affect data integrity. Use it only when rollback behavior is not needed and data consistency is not at risk. Suspending transactions can be appropriate in specific, controlled cases—such as bypassing rollback for audit logging, improving performance on low-risk operations, or ensuring certain changes persist during administrative tasks.

Important: If a global is modified during a transaction and then modified again while transactions are suspended, rolling back the transaction may result in an error. To prevent rollback errors while suspending transactions, avoid modifying the same global both inside a transaction and again while that transaction is suspended. If such a conflict is possible, use application-level safeguards—like a lock—to coordinate access and ensure the global isn't changed during suspension. The safest approach is to isolate operations that require transaction suspension from those that rely on rollback behavior.

To suspend all current transactions, invoke one of the following methods:

- In Python, call the **TransactionsSuspended()** method of the iris.system.Process class. This method takes a boolean argument: 1 suspends all current transactions and 0 (default) resumes them. It returns a boolean indicating the previous state.
- In ObjectScript, call the **TransactionsSuspended()** method of the %SYSTEM.Process class. This method takes a boolean argument: 1 suspends all current transactions and 0 (default) resumes them. It returns a boolean indicating the previous state.

There is no SQL equivalent to **TransactionsSuspended()**.

1.5.2 The Legacy Utility ^%ETN and Transactions

While **^%ETN** remains functional for compatibility with legacy systems, you should use structured exception handling and explicit rollback commands in modern applications. Further details are provided in the ^%ETN documentation.

^%ETN is a legacy utility that remains available for handling incomplete transactions in certain systems. If an error occurs during a transaction and you have not explicitly handled rollback using a rollback command, **^%ETN** or **FORE^%ETN** will prompt the user to commit or rollback the transaction. Committing an incomplete transaction can compromise logical database integrity. To prevent this, use structured error-handling mechanisms as recommended in the section on error handling alongside explicit rollback commands.

If your application invokes **^%ETN** or **FORE^%ETN** in an interactive process (such as running a routine in the Terminal) after an error with an active transaction, the user sees the following prompt before the process terminates:

```
You have an open transaction.
Do you want to perform a (C)ommit or (R)ollback?
R =>
```

If the user do not respond within 30 seconds, the system automatically rolls back the transaction. In a background job, the rollback happens immediately without displaying a prompt.

By default, **^%ETN** and **FORE^%ETN** exit using **HALT**. In a background job, **HALT** automatically rolls back the transaction. In an interactive process, **HALT** prompts the user to either commit or rollback. However, the **BACK^%ETN** and **LOG^%ETN** entry points do display a prompt or automatically roll back failed transactions; the user must explicitly roll back using **trollback 1** before calling these routines.