Article
Sergey Kamenev · Nov 11, 2019 11m read

Transactions in Global InterSystems IRIS

InterSystems IRIS supports a unique data structure, called globals, for information storage. Essentially, globals are persistent arrays with multi-level indices, having several extra capabilities—transactions, quick traversal of tree structures, and a programming language known as ObjectScript.

I'd note that for the remainder of the article, or at least the code samples, we'll assume you have familiarised yourself with the basics of globals:

Globals Are Magic Swords For Managing Data. Part 1.
Globals - Magic swords for storing data. Trees. Part 2.
Globals - Magic swords for storing data. Sparse arrays. Part 3.

Globals are completely different structures for storing data than the usual tables, and operate at a much lower level. And that begs the question, how would transactions look when working with globals, and what peculiarities might you encounter in the effort?

We know from relational database theory that a good transaction implementation needs to pass the ACID test (see ACID in Wikipedia).

- *Atomicity*: All changes made in the transaction are recorded, or none at all. See Atomicity (database systems) in Wikipedia.
- *Consistency*: After the transaction is completed, the logical state of the database should be internally consistent. In many ways, this requirement applies to the programmer, but in the case of SQL databases, it also applies to foreign keys.
- *Isolation*: Transactions running in parallel shouldn ’ t affect one another.
- *Durability*: After successful completion of the transaction, low-level problems (such as a power failure) should not affect the data changed by the transaction.

Globals are non-relational data structures. They were designed to support ultra-fast work on hardware with a minimal footprint. Let's look at how transactions are implemented in globals using the IRIS/docker-image.

1. Atomicity

Consider the situation when 3 values must be saved in database together, or none of them should be recorded.

The easiest way to check atomicity is to enter the following code in terminal:

```
Kill ^a
TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
TCOMMIT
```

Then conclude with:

```
ZWRITE ^a
```

The result should be this:

```
^a(1)=1
^a(2)=2
^a(3)=3
```

As expected, Atomicity is observed. But now let's complicate the task by introducing an error and see how the transaction is saved—partially, or not at all. We ’ ll start checking atomicity as we did before, like so:

```
Kill ^a
TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
```

But this time we ’ ll forcibly stop the container using the command docker kill my-iris, which is almost equivalent to a forced power off as it sends a SIGKILL (halt process immediately) signal. After restarting the container, we check the contents of our global to see what happened. Maybe the transaction has been partially saved?

```
ZWRITE ^a
```

```
Nothing got out
```

No, nothing has been saved. So, in the case of accidental server stop, the IRIS database will guarantee the atomicity of your transactions.

But what if we want to cancel changes intentionally? So now let's try this with the rollback command, as follows:

```
Kill ^a

TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
TROLLBACK 1

ZWRITE ^a

Nothing got out
```

Once again, nothing has been saved.

2. Consistency

Recall that globals are lower-level structures for storing data than relational tables, and with a globals database, indices are also stored as globals. Thus, to meet the requirement of consistency, you need to include an index change in the same transaction as a global node value change.

Say, for example, we have a global ^person, in which we store personal data using the social security number (SSN) as the key:

```
^person(1234567, 'firstname') = 'Sergey'
^person(1234567, ‘lastname’) = ‘Kamenev’
^person(1234567, ‘phone’) = ‘+74995555555
...
```

We ’ ve created an ^index key to enable rapid search by last or last and first names, as follows:

```
^index(‘Kamenev’, ‘Sergey’, 1234567) = 1
```

To keep the database consistent, we need to add persons like this:

```
TSTART
^person(1234567, ‘firstname’) = ‘Sergey’
^person(1234567, ‘lastname’) = ‘Kamenev’
^person(1234567, ‘phone’) = ‘+74995555555
^index(‘Kamenev’, ‘Sergey’, 1234567) = 1
TCOMMIT
```

Accordingly, when deleting a person, we must use the transaction:

```
TSTART
Kill ^person(1234567)
Kill ^index(‘Kamenev’, ‘Sergey’, 1234567)
TCOMMIT
```

In other words, fulfilling the consistency requirement for your application logic is entirely up to the programmer when working with a low-level storage format such as globals.

Luckily, IRIS offers the commands to organise your transactions and deliver Consistency guarantees for your applications. When using SQL, IRIS will use these commands under the hood to ensure consistency of its underlying globals data structures when performing INSERT, UPDATE, and DELETE statements. Of course, IRIS SQL also offers corresponding SQL commands for starting and stopping transactions to leverage in your (SQL) application logic.

3. Isolation

Here ’ s where things get wild. Suppose many users are working on the same database at the same time, changing the same data. The situation is comparable to when many developers are working with the same code repository and trying to commit changes to many files at the same time.

The database needs to keep up with everything in real time. Given that serious companies typically have a person responsible for version control—merging branches, managing conflict resolution, and so forth—and that the database needs to take care of this in real time, the complexity of the problem and the importance of correctly designing the database and the code that serves it both become self-evident.

The database can ’ t understand the meaning of actions performed by users and try to prevent conflicts when they ’ re working on the same data. It can only cancel one transaction that contradicts another or execute them sequentially.

Moreover, as a transaction is executing (before the commit), the state of the database may be inconsistent. Other transactions should not have access to the inconsistent database state. In relational databases, this is achieved in many ways, such as by creating snapshots or using multi-versioned rows.

When transactions execute in parallel, it ’ s important that they not interfere with each other. This is what isolation is all about.

SQL defines four levels of isolation, in order of increasing rigor. They are:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Let's consider each level separately. Note that the cost of implementing each level grows almost exponentially as you move up the stack.

READ UNCOMMITTED is the lowest level of isolation, but it ’ s also the fastest. Transactions can read the changes made by other transactions.

READ COMMITTED is the next level of isolation and represents a compromise. Transactions can ’ t read each other's changes before a commit, but can read any changes after a commit.

Say we have a long-running transaction (T1), during which commits have happened in transactions T2, T3... Tn while working on the same data as T1. In such cases, each time we request data in T1, we may well obtain a different result. This is called a non-repeatable read.

REPEATABLE READ is the next level of isolation, in which we no longer have non-repeatable reads because a snapshot of the result is taken each time we request to read data. The snapshot is used if the same data is requested again during the same transaction. However, at this isolation level, it ’ s possible that what will be read is phantom data—new strings that were added by transactions committed in parallel.

SERIALIZABLE is the highest level of isolation. It ’ s characterized by the fact that any data used in a transaction (whether read or changed) becomes accessible to other transactions only after the first transaction has finished.

First, let ’ s see whether there ’ s isolation of operations between threads with transactions and threads without transactions. Open two terminal windows and enter the following:

`Kill ^tWrite ^t(1)2`	`TSTARTSet ^t(1)=2`

There ’ s no isolation. One thread sees what the second one does when it opens a transaction.

Now let's see whether transactions in different threads can see what ’ s happening inside. Open two terminal windows and start two transactions in parallel.

`Kill ^tTSTARTWrite ^t(1)2???`	`TSTARTSet ^t(1)=2`

A 3 appears on the screen. What we have here is the simplest (but also the fastest) isolation level: READ UNCOMMITTED.

In principle, this is what we expect from a low-level data representation such as globals, which always prioritize speed. IRIS SQL provides different transaction isolation levels to choose from, but what if we need a higher level of isolation when working with globals directly?

Here we need to think about what isolation levels are actually for and how they work. For instance, lower levels of isolation are compromises designed to speed up database operations.

The highest isolation level, SERIALIZABLE, ensures that the result of transactions executed in parallel is equivalent to the result of executing them serially. This guarantees there will be no collisions. We can achieve this with properly used locks in ObjectScript, which can be applied in multiple ways. This means you can create regular, incremental, or multiple locks using the LOCK command.

Let's see how to use locks to achieve different levels of isolation. In ObjectScript, you use the LOCK operator. This operator permits not just exclusive locks, which are necessary for changing data, but also what are called shared locks. These shared locks can be accessed by several threads at once to read data that won ’ t be changed by other processes during the reading process.

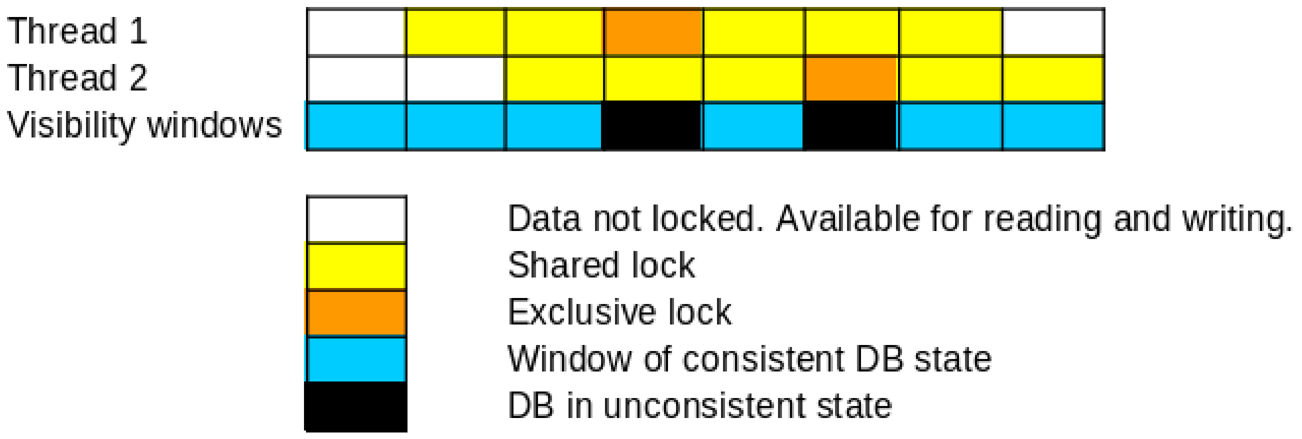
For more details about locking, see the article “ Locking and Concurrency Control”. To learn about two-phase locking, see the article "Two-phase locking" on Wikipedia.

The difficulty is that the state of the database may be inconsistent during the transaction, with the inconsistent data visible to other processes. How can this be avoided? For this example, we ’ ll use locks to create visibility windows within which the state of the database can be consistent. Access to any of these visibility windows will be through a lock.

Shared locks on the same data are reusable—several processes can take them. These locks prevent other processes from changing data. That is, they ’ re used to form windows of a consistent database state.

Exclusive locks, on the other hand, are used when you ’ re modifying data—only one process can take such a lock.

Exclusive locking can be employed in two scenarios. First, it can take any process if the data doesn ’ t have locks. Second, it can take only the process that has a shared lock on the data and the first one that requested an exclusive lock.



The narrower the visibility window, the longer the wait for other processes becomes—but the more consistent the state of the database in it will be.

READ COMMITTED ensures that we see only committed data from other threads. If data in another transaction hasn't yet been committed, we see the old version. This lets us parallelize the work instead of waiting for a lock to be released.

In IRIS, you can't see an old version of the data without using special tricks, so we'll have to make do with locks. We need to use shared locks to permit data to be read only at points where it ’ s consistent.

Let's say we have a database of users, ^person, who transfer money from one person to another. Here ’ s the point at which money is transferred from person 123 to person 242:

```
LOCK +^person(123), +^person(242)
TSTART
Set ^person(123, amount) = ^person(123, amount) - amount
Set ^person(242, amount) = ^person(242, amount) + amount
TCOMMIT
LOCK -^person(123), -^person(242)
```

The point where the amount is requested for person 123 before the deduction should have an exclusive lock (by default):

```
LOCK +^person(123)
Write ^person(123)
```

But if we need to display the account status in the user's personal account, we can use a shared lock, or none at all:

```
LOCK +^person(123)#”S”
Write ^person(123)
LOCK -^person(123)#”S”
```

However, if we accept that database operations are carried out virtually instantaneously (remember that globals are a much lower-level structure than a relational table), then this level is no longer so necessary in favor higher isolation levels.

Full example, for READ COMMITTED:

```
LOCK +^person(123)#”S”, +^person(242)#”S”

Read data (?oncurrent committed transactions can change the data)

LOCK +^person(123), +^person(242)
TSTART
Set ^person(123, amount) = ^person(123, amount) - amount
Set ^person(242, amount) = ^person(242, amount) + amount
TCOMMIT
LOCK -^person(123), -^person(242)

Read data (?oncurrent committed transactions can change the data)

LOCK -^person(123)#”S”, -^person(242)#”S”
```

REPEATABLE READ is the second-highest level of isolation. At this level we accept that data may be read several times with the same results in one transaction, but may be changed by parallel transactions.

The easiest way to ensure a REPEATABLE READ is to take an exclusive lock on the data, which automatically turns this isolation level into a SERIALIZABLE one.

```
LOCK +^person(123, amount)

read ^person(123, amount)
```

```
other operations (parallel streams try to change ^person(123, amount), but can't)

change ^person(123, amount)

read ^person(123, amount)

LOCK -^person(123, amount)
```

If locks are separated by commas, they are taken in sequence. But they will be taken atomically, all at once, if they ’ re listed like this:

```
LOCK +(^person(123),^person(242))
```

SERIALIZABLE is the highest level of isolation and the most costly. When working with classic locks like we did in the above examples, we have to set the locks in such a way that all transactions with data in common will end up being performed serially. For this approach, most of the locks should be exclusive and taken to the smallest fields of the global, for performance.

If we ’ re talking about deducting funds from a ^person global, then SERIALIZABLE is the only acceptable level. Money spent needs to be strictly serial, otherwise it ’ s possible to spend the same amount several times.

4. Durable

I conducted tests with a hard cut-off of the container using the docker kill my-iris command. The database stood up well to these tests. No problems were identified.

Tools to manage globals and locks

You may find useful the following tools in IRIS Management portal:

View and manage locks.

View and manage globals.

Conclusion

InterSystems IRIS has support for transactions using globals, which are atomic and durable. To ensure database consistency with globals, some programming effort and the use of transactions are necessary, since there are no complex built-in constructions like foreign keys.

Globals without locks are equivalent to the READ UNCOMMITTED level of isolation, but this can be raised to the SERIALIZABLE level using locks. The correctness and transaction speed achievable with globals depend considerably on the programmer's skill and intent. The more widely that shared locks are used when reading data, the higher the isolation level. And the more narrowly exclusive locks are used, the greater the speed.

#Databases #Database Transaction Processing #Globals #Caché #InterSystems IRIS