DataSets and Transactions

Module Objectives

In this module, you will look at:

- The need for transactions
- OCC: Optimistic Concurrency Control
- Rules for transactions
- Disabling transactions: when and where

The Need for Transactions (1 of 2)

A Flowlet processes the data objects received on its inputs one at a time

While processing a single input object, all operations, including the removal of the data from the input, and emission of data to the outputs, are executed in a **transaction**

This provides us with **ACID:**

- Atomicity
- Consistency
- Isolation, and
- Durability properties

The Need for Transactions (2 of 2)

To create **ACID**:

- The process method runs under read isolation to ensure that it does not see dirty writes (uncommitted writes from concurrent processing) in any of its reads; it does see, however, its own writes
- A failed attempt to process an input object leaves the data in a consistent state; it does not leave partial writes behind
- All writes and emission of data are committed atomically; either all of them or none of them are persisted
- After processing completes successfully, all its writes are persisted in a durable way

In case of failure, the state of the data is unchanged and processing of the input object can be reattempted

This ensures "exactly-once" processing of each object

OCC: Optimistic Concurrency Control (1 of 2)

The Continuuity Reactor uses **Optimistic Concurrency Control** (OCC) to implement transactions

- Most relational databases use locks to prevent conflicting operations between transactions
- Under OCC we allow conflicting writes to happen
- When the transaction is committed, we can detect whether it has any conflicts: namely, if during the lifetime of the transaction, another transaction committed a write for one of the same keys that the transaction has written

In last case, the transaction is aborted and all of its writes are rolled back

---

OCC: Optimistic Concurrency Control (2 of 2)

In other words:

- If two overlapping transactions modify the same row, then the transaction that commits first will succeed, but the transaction that commits last is rolled back due to a write conflict

Optimistic Concurrency Control

- Is lockless and therefore avoids problems such as idle processes waiting for locks, or even worse, deadlocks
- Comes at the cost of rollback in case of write conflicts
- We can only achieve high throughput with OCC if the number of conflicts is small
- Good practice to reduce the probability of conflicts wherever possible

---

Rules for Flows, Flowlets and Procedures (1 of 2)

Keep transactions short: minimize the duration of transactions

- Reactor attempts to delay the beginning of each transaction as long as possible
- **If your Flowlet only performs write operations**, but no read operations, then all writes are deferred until the process method returns
- They are then performed and transacted, together with the removal of the processed object from the input, in a single batch execution
- **If your Flowlet performs a read**, then the transaction must begin at the time of the read
- If your Flowlet performs long-running computations after that read, then the transaction runs longer and the risk of conflicts increases
- Good practice: **perform reads as late as possible** in the process method

---

Rules for Flows, Flowlets and Procedures (2 of 2)

There are two ways to perform an increment

- As a write operation that returns nothing, or
- As a read-write operation that returns the incremented value
- If you perform the read-write operation, then that forces the transaction to begin, and the chance of conflict increases
- Unless you depend on that return value, always perform an increment only as a write operation

Use hash-based partitioning

- Use with the inputs of highly concurrent Flowlets that perform writes
- Reduces concurrent writes to same key from different instances of a Flowlet

These guidelines will help you write more efficient and faster-performing code

---

The Need for Disabling Transactions (1 of 2)

Transactions providing **ACID** (atomicity, consistency, isolation, and durability) guarantees are useful in applications where data accuracy is critical:

- Billing applications
- Computing click-through rates

However, some applications—such as trending—might not need it

Applications that do not strictly require accuracy can trade off accuracy against increased throughput by taking advantage of not having to write/read all the data in a transaction

---

The Need for Disabling Transactions (2 of 2)

Transaction can be disabled for a Flow by annotating the Flow class with the `@DisableTransaction` annotation:

```
@DisableTransaction
class MyExampleFlow implements Flow {
  ...
}
```

While this may speed up performance, if—for example—a Flowlet fails, the system would not be able to roll back to its previous state

You will need to judge whether the increase in performance offsets the increased risk of inaccurate data

---

Module Summary

You should be able to explain:

- The need for transactions
- OCC: Optimistic Concurrency Control
- Rules for transactions
- Disabling transactions: when and where

Module Completed