Maintaining Data Integrity with Transactions

Xavier Morera
HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA

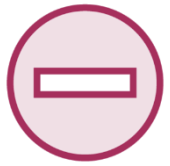
@xmorera www.xaviermorera.com

A transaction is a set of statements performed so that they are all guaranteed to succeed or fail as a single unit

What Is a Transaction?

Why Should You Use Transactions?

Imagine you are paying your credit card.



The application deducts the money from your account

The application exits unexpectedly, right before paying the credit card

Where did the money go?!!?11!!?

Benefits of Transactions

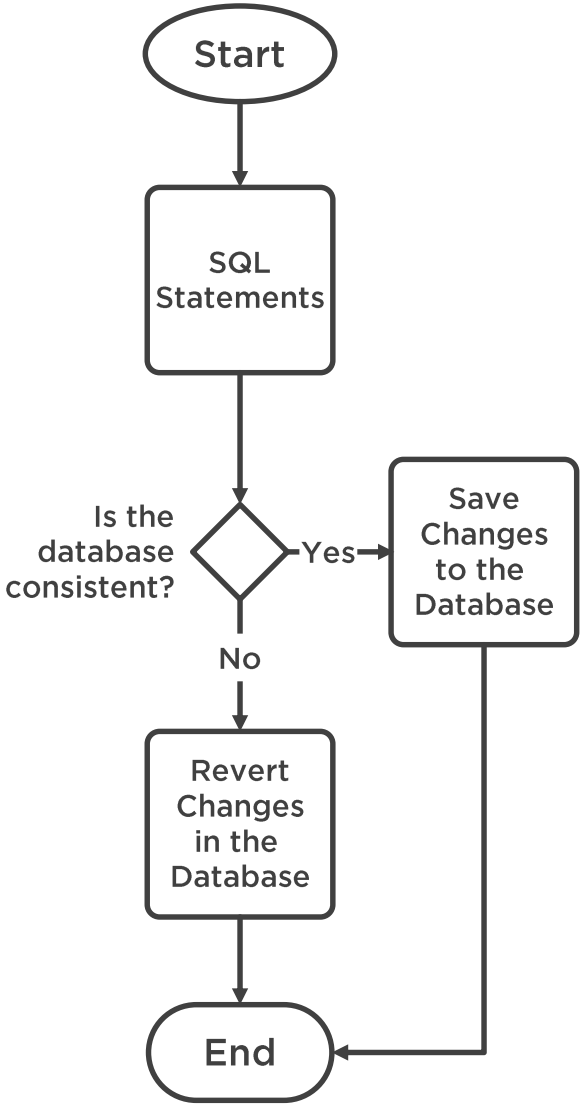
Data Integrity

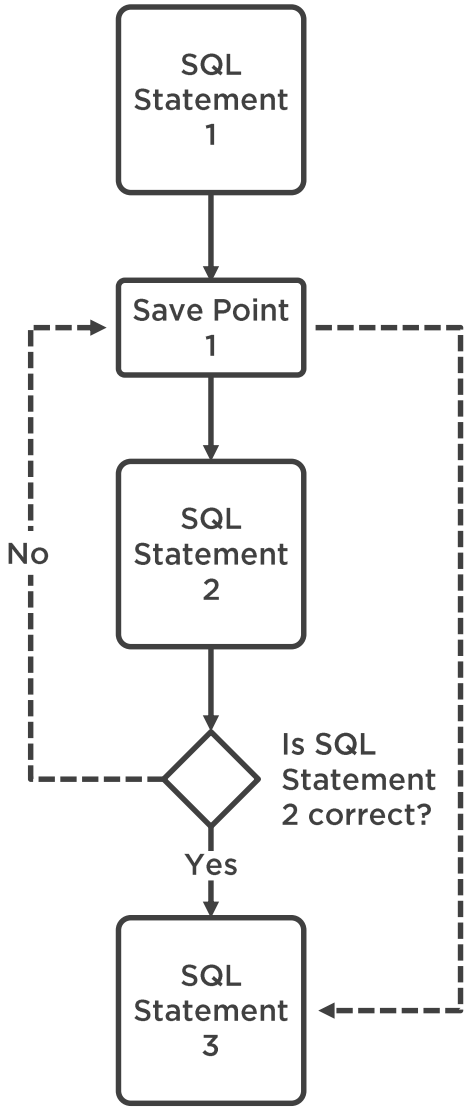
Speed

Transaction Properties

Atomicity Consistency Isolation Durability

TRANSACTIONS





Transaction Modes

Autocommit Transactions

Each individual statement is a transaction

Explicit Transactions

Each transaction is explicitly started and completed

Implicit Transactions

Transaction starts automatically after each commit

* Batch-scoped transactions for Multiple Active Result Sets (MARS)

Transaction Statements

BEGIN

SAVE

ROLLBACK

COMMIT

SET

BEGIN TRANSACTION Syntax

Starting point of an explicit transaction

The given name for a transaction

```
BEGIN { TRAN | TRANSACTION }    [ { transaction_name | @transaction_name_variable }
    [ WITH MARK [ 'transaction description' ] ] ];
```

Places the transaction name in the transaction log, which can be used for recovery

Demo

Creating Transactions Using BEGIN TRAN

COMMIT TRANSACTION Syntax

Marks the end of the transaction

The transaction name

```
COMMIT { TRAN | TRANSACTION }    [ { transaction_name | @transaction_name_variable }
    [ WITH ( DELAYED_DURABILITY = { OFF | ON } ) ];
```

Specifies if the transaction should be committed with delayed durability

Demo

Making Changes Permanent with COMMIT TRANSACTION

Querying Data Locked in a Transaction

Can't read data locked by a transaction

- Important for data consistency

Required to retrieve the results

- In its current state

Querying Data Locked in a Transaction

Use the NOLOCK hint

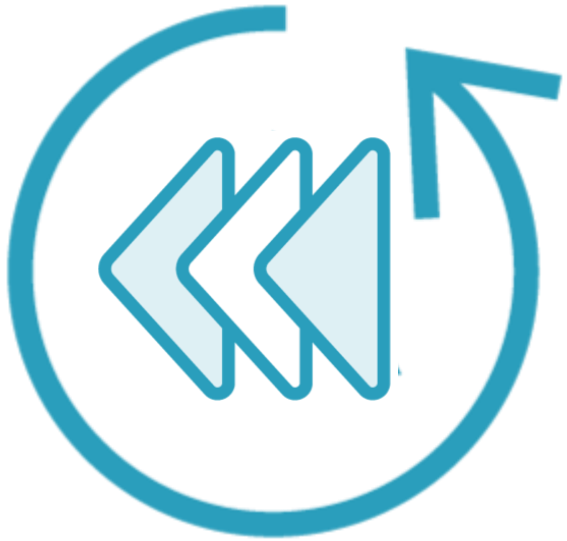
- Retrieve records regardless of locks
  - READUNCOMMITTED

Set transaction isolation level

Demo

Querying Data Locked by a Transaction Using NOLOCK

Undoing Transactions Using ROLLBACK



Returns state of data to a previous state

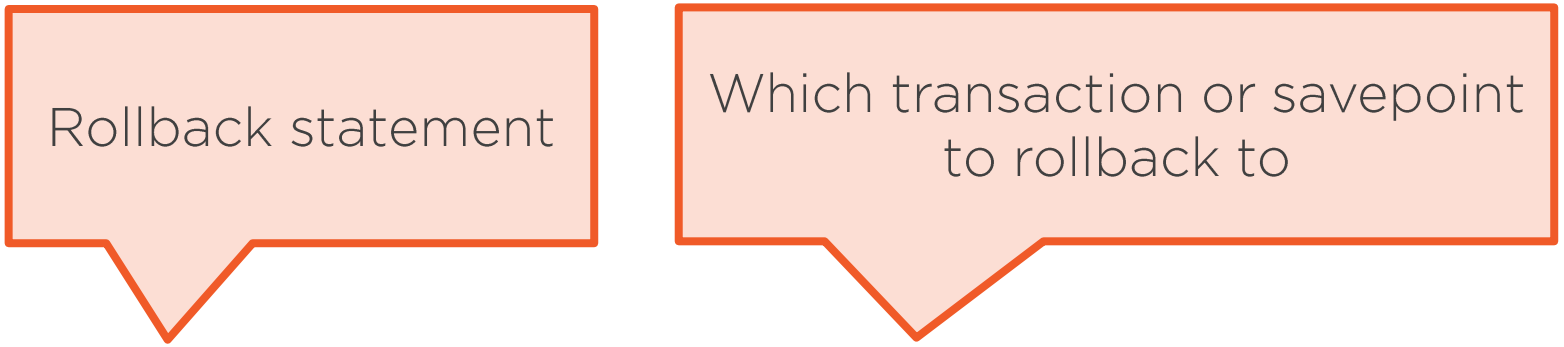
- Start of the current transaction

Because of an error

- Or a specific condition

Entire transaction or a savepoint

ROLLBACK Syntax



ROLLBACK TRANSACTION [transaction_name | savepoint_name]

Demo

Undoing Transactions Using ROLLBACK

Partially Undoing Transactions with Savepoints

Used to rollback a transaction

- Back to a specific point
- Instead of the full transaction

Useful when

- Possibility of error in a transaction
- With a previous costly operation

SAVE TRANSACTION

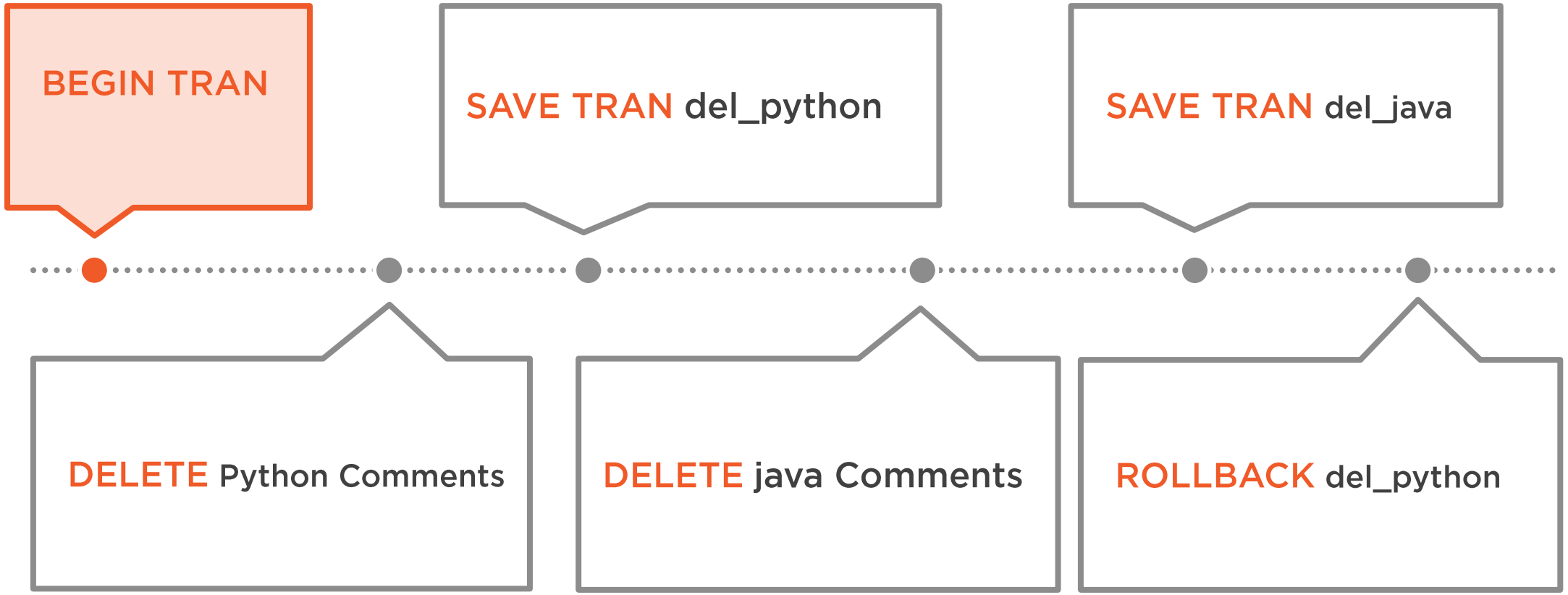
SAVE TRANSACTION Syntax

SAVE { TRAN | TRANSACTION } { savepoint_name | @savepoint_variable }

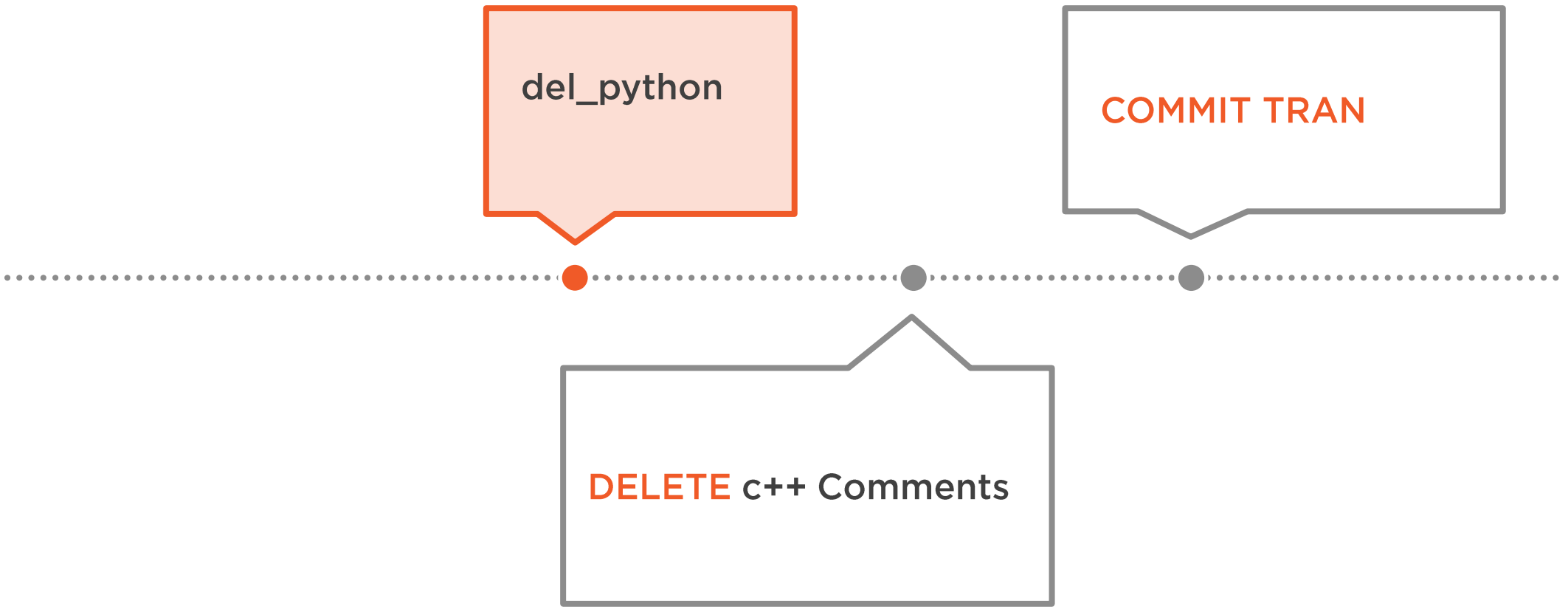
Demo

Partially Undoing Transactions Using Savepoints

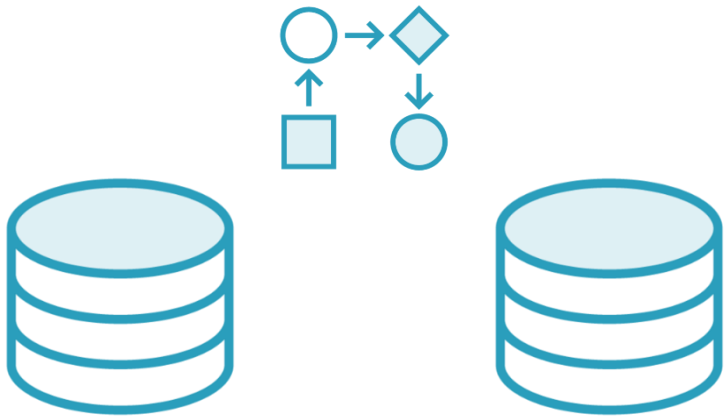
Timeline of Events



Timeline of Events



Distributed Transactions



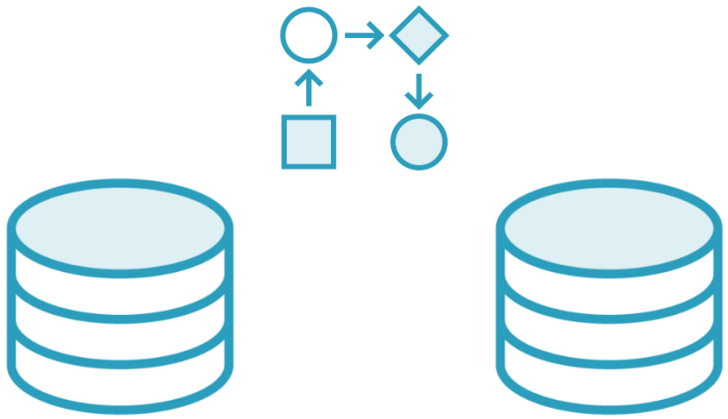
A distributed transaction

- Operation on two or more databases
- Retaining ACID properties
  - Of transaction processing

Managed by

- Microsoft Distributed Transaction Coordinator
- MS DTC

Distributed Transactions



Instance that starts the DISTRIBUTED TRAN

Called Transaction Originator

Controls completion of transaction

Any subsequent COMMIT or ROLLBACK

Sent to the controlling instance

BEGIN DISTRIBUTED TRANSACTION Syntax

Starting point of a distributed transaction

BEGIN DISTRIBUTED { TRAN | TRANSACTION }

[{ transaction_name | @transaction_name_variable }

Specifies transaction name or variable

Takeaway

Transaction: All or nothing

Important to maintain data integrity

- Speed

ACID

- Atomicity
- Consistency
- Isolation
- Durability

Takeaway

Autocommit, explicit, and implicit

Statements

- BEGIN
- COMMIT
- ROLLBACK
- SAVE

Distributed transactions