Manual 10 · commit, rollback and everything in between

Transactions

How Stilton groups database writes so that they succeed or fail together — native and XA transactions, the exact rules that decide commit or rollback, what each database supports, how transactions meet catch ports, the error handler, the Stop node, bridges and continuous flows, and how to read the outcome of every run.



CONTENTS

01 Why transactions

02 The model in one page

03 Setting up a transaction

04 Native transactions
- 4.1 What happens, step by step
- 4.2 One connection, shared
- 4.3 Example — order header and lines

05 XA transactions
- 5.1 Prerequisite on the PostgreSQL servers
- 5.2 Transactions in doubt

06 What decides the outcome
- 6.1 Scenario by scenario
- 6.2 Interruptions half-way

07 Database support
- 7.1 PostgreSQL
- 7.2 Oracle, Informix and generic ODBC
- 7.3 MySQL, MariaDB and SQLite

08 Master-detail inside a transaction

09 Visibility and locks

10 Lanes, bridges and continuous flows

11 Reading the outcome
- 11.1 In the studio and the Monitor
- 11.2 In the runner log

12 Patterns
- 12.1 Staging table and swap
- 12.2 Upsert for safe repetition
- 12.3 Audit outside, data inside
- 12.4 Short transactions

13 Troubleshooting

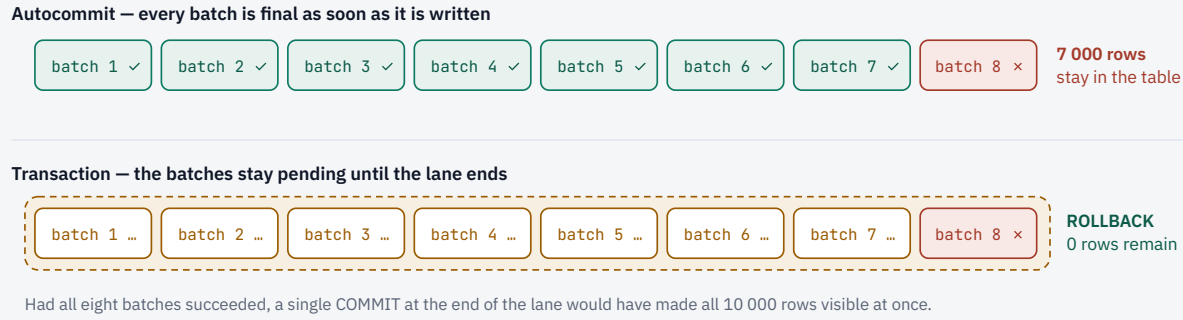
Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 Why transactions

A **DB Sink** on its own works in *autocommit*: each batch it writes is committed by the database as soon as it is written. That is the fastest way to load data, and for many flows it is exactly right. It has one consequence you must know: **if the flow fails half-way, what was already written stays written.**

Example. A flow loads 10 000 orders into `orders` with a batch size of 1 000. Row 7 412 breaks a check constraint. In autocommit the first seven batches — 7 000 rows — are already in the table, the eighth batch is refused, and the run fails. Someone now has to work out which orders are in and which are not.

A **transaction** changes the rule: the writes of a group of DB Sinks are kept pending until the lane ends, and then they are either **all committed** or **all rolled back**. In the example above, the 7 000 rows disappear with the failure and the table is exactly as it was before the run.



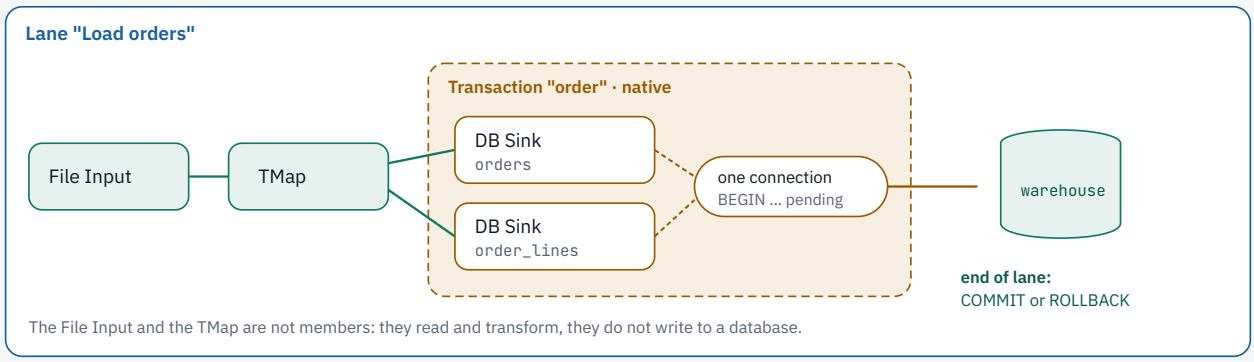
Same flow, same failure: autocommit leaves a partial load, a transaction leaves nothing.

Use a transaction when the writes **only make sense together**: an order header and its lines, a document and its attachments, the rows of a monthly close, a full reload of a table that readers must never see half-empty. Leave autocommit when every row stands on its own — an event log, a stream of measurements — and a partial load can simply be resumed.

---

02 The model in one page

A transaction in Stilton is a **group of DB Sinks in the same lane**. You declare the group on the lane, then attach DB Sinks to it; those outputs are its **members**.



Members write through one shared connection; nothing becomes visible to anyone else until the lane decides.

Five rules describe everything that follows:

01 **A group belongs to one lane.** It cannot span lanes; members are DB Sinks of that lane only.

02 **Writes are pending until the end of the lane.** No member commits on its own, not even when it finishes early.

03 **All or nothing.** At the end of the lane the group commits only if *every* member finished without failing and the lane was not interrupted; otherwise every member is rolled back.

04 **Native or XA.** A *native* group writes to one database resource through one connection. An *XA* group spans several PostgreSQL resources and commits them with a two-phase protocol.

05 **The outcome is reported.** Each member's log says whether its group committed or rolled back, and why; a commit that the database refuses makes the run fail.

03 Setting up a transaction

01 Open the **Transactions** tab of the properties panel. It lists every lane of the pool with its groups.

02 Click **+ Transaction** on the lane that does the writes. Give the group a name that says what it protects — `order`, `monthly_close` — and choose the **Mode**: *Native (one resource)* or *XA (multiple resources, PostgreSQL only)*.

03 Open each DB Sink that belongs to the group. In its **Transaction** section choose the group instead of — *none (autocommit)* —.

04 The node now shows a lock badge — green `TX native` or orange `XA xa` — and the group lists it under *Member nodes*; click a member there to jump to it.

05 Look at the **Validation** tab. The checks below catch the configurations that could not work, before anything runs.

Code	Message	What to do
`TX_UNDECLARED`	The transaction of this node does not exist in its lane	the group was deleted, or the node was moved to another lane: pick a group of its lane, or none
`TX_NO_RESOURCE`	Node in transaction … without a configured resource	choose the database resource of the DB Sink
`TX_UNSUPPORTED_DIALECT`	Transactions are not available on …	the database cannot take part (see *Database support*); remove the node from the group
`TX_XA_UNSUPPORTED_DIALECT`	XA transaction …: only PostgreSQL resources can take part	use a native group on that database instead
`TX_NATIVE_RESOURCE_MISMATCH`	Native transaction …: all members must use the same resource	point all members to one resource, or switch the group to XA (PostgreSQL)

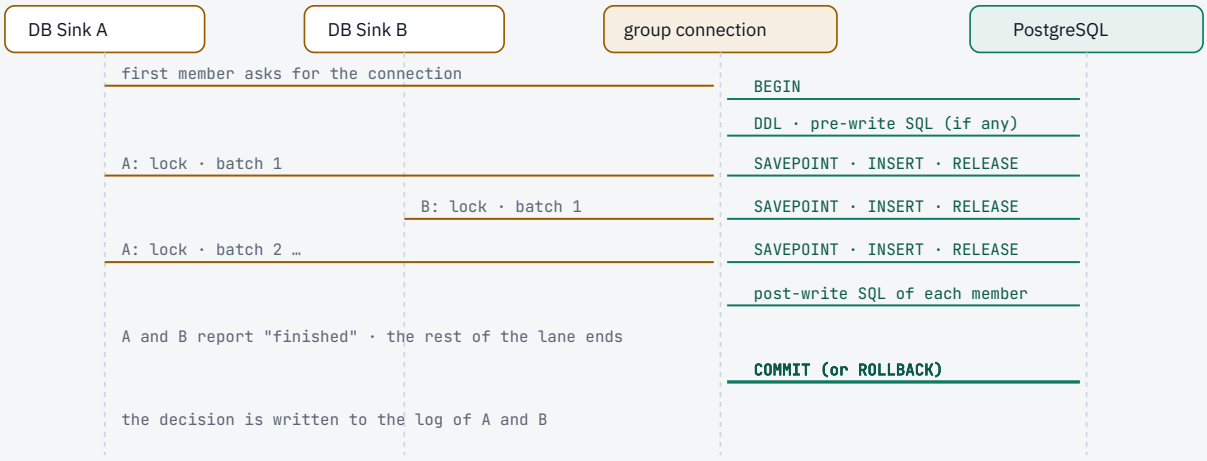
All five are **errors**: while any of them is present the flow is neither run nor compiled, because it would stop with an error anyway.

Tip. Resource means the *database resource* of the project, not the table. Ten DB Sinks writing to ten tables of the same database share one resource and belong happily to the same native group.

04 Native transactions

A native group uses **one connection** to **one resource**. It is the right choice for almost every case: it is simple, fast and supported by every database that supports transactions in Stilton.

What happens, step by step



On PostgreSQL every batch runs inside a savepoint, so a failing batch never leaves the shared connection unusable for the other members.

01 **The first member that writes opens the transaction.** On PostgreSQL that is `BEGIN`; on ODBC databases autocommit is switched off on the connection. Later members reuse the same connection.

02 **Table preparation and pre-write SQL** of each member run on that connection, before its first batch (see *Database support* for where DDL runs on ODBC).

03 **Batches are written one at a time.** A member takes the connection's lock, writes one batch and releases the lock. On PostgreSQL each batch is wrapped in a `SAVEPOINT`: if the batch fails, `ROLLBACK TO SAVEPOINT` cleans the connection so the other members are not affected while the lane winds down.

04 **Post-write SQL** of a member runs inside the transaction, only if all its batches succeeded.

05 **At the end of the lane** — after every node, not only the members, has finished — the group commits or rolls back according to the rules in *What decides the outcome*.

Note. A member that receives **no rows** still counts as a member that finished successfully. Its pre- and post-write SQL still run, inside the transaction.

One connection, shared

All members of a native group write through a single connection, so their batches are **serialised**: while A writes a batch, B waits for the lock. This costs less than it seems — the database would serialise the same writes on the same tables anyway — and it is what makes the group atomic. Two practical consequences:

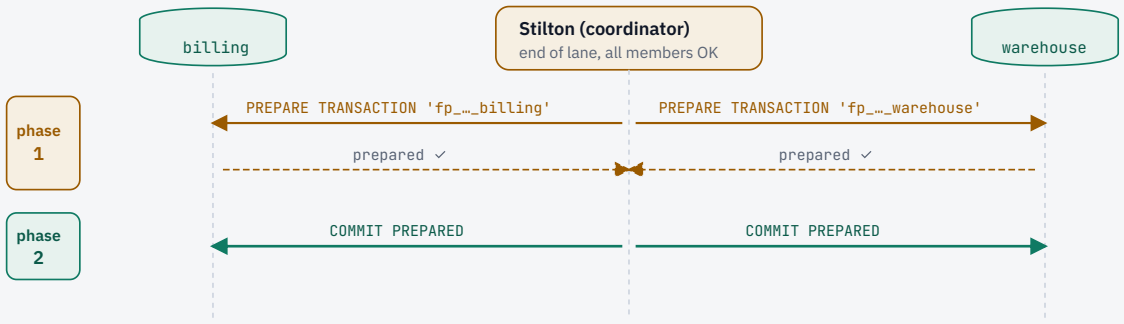
- Choose **larger batches** inside a group (1 000–5 000 rows): fewer turns of the lock, fewer savepoints. The *Flush interval* still applies.
- Do not expect a group of five members to write five times faster than one. If raw speed matters more than atomicity, autocommit is the faster choice.

Example — order header and lines

Example. A TMap splits each incoming order into a header row and its line rows. *Write orders* (DB Sink, UPSERT into `sales.orders`, key `order_id`) and *Write lines* (DB Sink, INSERT into `sales.order_lines`) both use the resource `warehouse` and the native group `order`. If a line breaks a foreign key, *Write lines* fails, and at the end of the lane both tables are rolled back: no order exists without its lines.

05 XA transactions

An XA group lets members write to **different PostgreSQL databases** and still succeed or fail together. Stilton coordinates them with PostgreSQL's two-phase commit.



Phase 1 — each database stores the changes durably and promises it can commit them: the transaction is "in doubt".
Phase 2 — only if every database prepared, all commit; if one refused, the prepared ones are rolled back (ROLLBACK PREPARED).

Members write normally during the lane; the two phases happen only at the end, and only when the group is going to commit.

During the lane each resource has its **own** connection and its own transaction; members on the same resource share it, exactly as in a native group. At the end:

- if any rule excludes the commit (a member failed, the lane was interrupted…), every connection simply rolls back — no two-phase protocol is needed;
- otherwise Stilton runs **phase 1**, `PREPARE TRANSACTION`, on every resource, one after the other. If one refuses, the resources already prepared are rolled back with `ROLLBACK PREPARED` and the others with `ROLLBACK`; the run fails;
- if every resource prepared, **phase 2** commits them all with `COMMIT PREPARED`.

Prerequisite on the PostgreSQL servers

Two-phase commit is **off by default** in PostgreSQL. On **every** server that takes part, set in `postgresql.conf`:

postgresql.conf — ini

```
# at least the number of XA groups that can be preparing at the same time
max_prepared_transactions = 20
```

Then **restart** the server (a reload is not enough). Without it, phase 1 fails with an error that mentions `max_prepared_transactions`, the group is rolled back and the run fails with a hint that says exactly this. No special privilege is needed to prepare a transaction; to commit or roll back a leftover by hand you must connect as the same database user, or as a superuser.

Transactions in doubt

Between the two phases a prepared transaction belongs to neither side: the database keeps it — with its locks — until someone commits or rolls it back. Stilton does this immediately, so a transaction stays *in doubt* only if the runner is **killed between phase 1 and phase 2**, or if a `COMMIT PREPARED` fails. In both cases:

- the runner log has a line `[xa] group '…': PREPARE TRANSACTION 'fp_…'` written **before** the command, so the identifier is never lost;
- a failure in phase 2 makes the run fail with a message ending in *transaction in doubt, resolve it by hand*.

Stilton's identifiers all start with `fp_` followed by the run, the group and the resource. To find and resolve leftovers, connect to each server:

PostgreSQL sql

```
-- leftovers of Stilton, oldest first
SELECT gid, prepared, owner, database
FROM pg_prepared_xacts
WHERE gid LIKE 'fp\_%'
ORDER BY prepared;

-- every resource prepared (look for it in the log) → finish the commit
COMMIT PREPARED 'fp_r9c1e_tx_m7q2cd_billing';

-- otherwise → undo it
ROLLBACK PREPARED 'fp_r9c1e_tx_m7q2cd_billing';
```

Warning. Decide **per group, not per resource**: look in the log for all the `PREPARE TRANSACTION` lines of that run and group. If every resource of the group prepared, commit them all; if any did not, roll back all the prepared ones. Committing one side and rolling back the other is precisely the inconsistency XA exists to prevent.

Tip. If one database can hold both sets of tables — in two schemas — a **native** group is simpler and faster than XA, and can never be left in doubt.

06 What decides the outcome

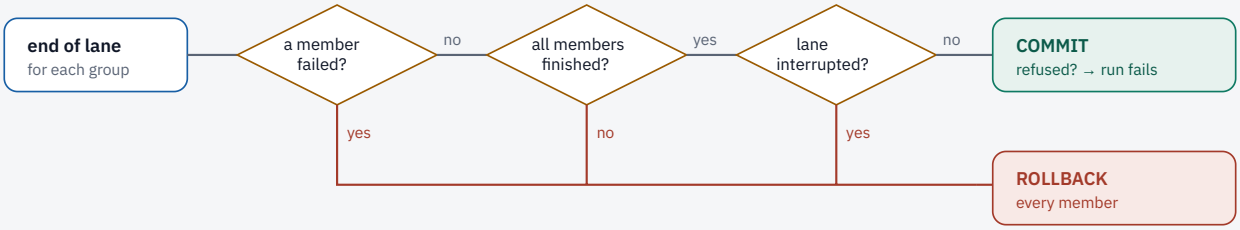
At the end of the lane each group commits only if **all three** conditions hold:

01 **No member failed** — whether the error stopped the node or was handled on its catch port.

02 **Every member finished** — none was interrupted half-way.

03 **The lane was not interrupted** — by a Stop node, a critical error, or a *Stop* rule of the error handler.

A node **outside** the group does not appear in this diagram: its failure does not roll the group back, unless it interrupts the lane (critical node, Stop rule of the error handler).



The decision is taken once, after every node of the lane has finished, and it is the same for native and XA groups.

Scenario by scenario

What happens during the lane	Group	Run
every member and every other node succeeds	**commit**	succeeds

What happens during the lane	Group	Run
a member fails (constraint, lost connection, bad SQL…)	**rollback**	fails
a row is refused and the member's *On constraint error* **is Skip**	**commit**, without that row	succeeds; the row is counted as skipped
a member fails and the error is handled on its catch port	**rollback**	continues; succeeds if nothing else fails
a member's database is unreachable at start, and its retry succeeds	**commit** — the retry only reopens the connection, before the member joins the group	succeeds
a node outside the group fails, with no stop	**commit**	fails
an error on a critical node, or an error-handler rule Stop	**rollback**	fails — *Lane interrupted*
rows reach a Stop node	**rollback**	ends as a deliberate stop, not a failure
the database refuses the COMMIT (deferred constraint, lost connection…)	nothing written	fails — *Transaction '…': NOT committed*
the run is stopped with Stop in the studio, Ctrl+C, `systemctl stop`	decided normally when the lane in progress ends	its own outcome
the runner is killed (second Ctrl+C, `kill -9`**, power loss)**	the database rolls back when the connection drops; an XA group may stay prepared	no outcome

Three of these rows deserve a closer look.

A failure outside the group commits the group. This is deliberate: a failing *Mail Sink* or *FTP/SFTP Sink* should not throw away a correct load. When a node outside the group *must* stop the load, the lane needs an **Error Handler**; in its mapping, mark the node **Critical** (or give its errors a *Stop* rule). Its error then interrupts the lane, and the group rolls back.

> **Example.** A flow writes invoices in the group `invoices` and also sends a summary to the accounting team with a Mail Sink. If the mail server is down, the invoices are still committed and the run fails because of the Mail Sink — usually what you want, since the summary can be sent again. If instead the mail is part of the contract ("no invoices without the notification"), mark the Mail Sink *Critical* in the Error Handler: an error on it rolls the invoices back. Remember that the mail may already have left when a *member* fails later: a message cannot be rolled back.

Catch does not save the group. Handling a member's error on its catch port keeps the *lane* running — you can log the error, send it somewhere, carry on with other work — but the member's writes are incomplete, so the group still rolls back. The catch is the right place to record *why*.

Retry on a member is safe, and limited. On a DB Sink, *Retry* only repeats the opening of the database connection — useful when the database is still starting. It happens before the member joins the group, so it never mixes two attempts in one transaction. A failure *while writing* is not retried: it rolls the group back, and the remedy is to run the **whole flow** again (from the scheduler, or by restarting the service), which starts from a clean transaction.

Interruptions half-way

When a Stop node, a critical error or a *Stop* rule interrupts the lane, the nodes still running are halted. A member may be in the middle of a batch, or may see its input end early and believe it has finished. Neither can lead to a commit: the lane was interrupted, so the group is rolled back — partial batch included.

> **Note.** The error handler and its sub-pipeline are excluded from the interruption: they finish their work *before* the rollback. If the error handler writes to a database, keep that DB Sink **outside** the group — inside it, the rollback would take the error records away with everything else.

07 Database support

Database	Native	XA	Notes
PostgreSQL	yes	yes	savepoint per batch; DDL and TRUNCATE inside the transaction
Oracle (ODBC)	yes	no	DDL outside the transaction; *TRUNC+INS* uses `DELETE`
Informix (ODBC)	yes	no	as Oracle
Generic ODBC (e.g. SQL Server through ODBC)	yes	no	as Oracle; behaviour depends on the driver's transaction support
MySQL / MariaDB	not yet	no	validation error `TX_UNSUPPORTED_DIALECT`
SQLite	not yet	no	validation error `TX_UNSUPPORTED_DIALECT`

PostgreSQL

Everything a member does runs inside the transaction, including its table preparation:

- **DROP + CREATE** / **Create table if it doesn't exist** (*Table options*): PostgreSQL DDL is transactional, so a table created by a member disappears again if the group rolls back.
- **TRUNC+INS** write mode: `TRUNCATE` runs inside the transaction and is rolled back with it — readers keep seeing the old content until the commit. Note that `TRUNCATE` takes an *exclusive* lock: other sessions cannot even read the table until the lane ends.
- Each batch runs inside a **savepoint**, so one failing batch does not poison the connection shared with the other members.
- With *On constraint error* set to **Skip**, each row also runs inside its own savepoint: a refused row is undone alone and the rest of the batch goes on. PostgreSQL aborts the whole transaction at the first failing statement, so without this a single refused row would invalidate everything. The price is two extra statements per row — keep *Stop* when refused rows are not expected.

Oracle, Informix and generic ODBC

On ODBC databases the group connection runs with autocommit off, and `COMMIT` / `ROLLBACK` are issued at the end of the lane. Three differences from PostgreSQL:

- **DDL runs outside the transaction**, on a separate connection, before the member joins the group. Oracle commits implicitly on every DDL statement: inside the group it would also commit the other members' rows. As a consequence, **a table created by the member stays** even if the group rolls back — it will simply be empty.
- **TRUNC+INS empties the table with** `DELETE FROM` inside the transaction, because `TRUNCATE` is DDL on Oracle and would commit implicitly. `DELETE` is slower on very large tables but is rolled back with the group.
- **No savepoint per batch**: Oracle and Informix undo only the failing statement, and a failed member rolls back the whole group anyway.

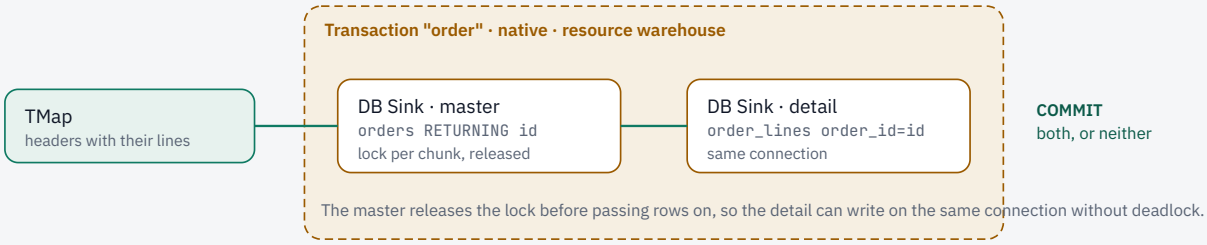
> **Warning.** Do not put DDL in the **pre-write** or **post-write SQL** of a member on Oracle (`CREATE`, `ALTER`, `TRUNCATE`, `GRANT` ...). Those statements run inside the group connection: Oracle would commit everything written so far, silently breaking the all-or-nothing rule. Use the *Table options* of the node, which run outside the transaction, or put the statement in the pre-write SQL of a DB Sink that is **not** in the group.

MySQL, MariaDB and SQLite

DB Sinks on these databases cannot join a group yet: validation reports `TX_UNSUPPORTED_DIALECT`, and an old plan that contains one stops with an error rather than writing outside the transaction. Until support arrives, the *staging pattern* in *Patterns* gives the same guarantee for full reloads.

08 Master-detail inside a transaction

A DB Sink with **Pass-through master-detail** (*Output mode*) writes each master row, reads back its generated key (for example `RETURNING id` on PostgreSQL) and passes the row on, with the key, to the detail flow. Master and detail can — and usually should — be in the same group.



Keys generated by the master are already usable by the detail inside the same transaction, because both write through the same connection.

Inside a group the master writes **row by row** (it needs the one-to-one match between row and generated key) in chunks of the batch size, under the group lock, and releases the lock before forwarding the chunk to the detail. If the detail fails, the master's rows are rolled back too.

Warning. Do not put the master in a group and leave the detail in autocommit. The detail runs on another connection, which cannot see the master's rows until the commit: a foreign key from `order_lines` to `orders` would fail on every row.

09 Visibility and locks

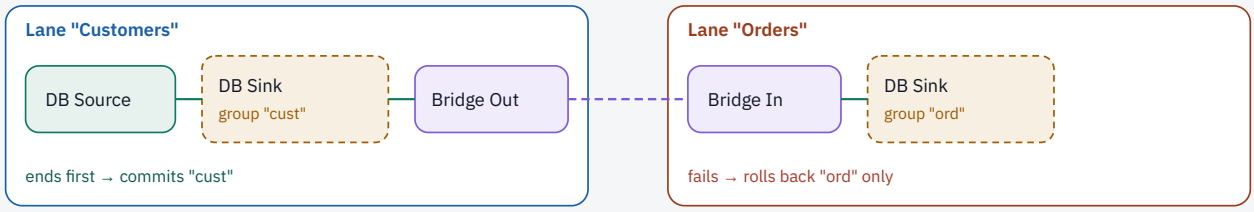
Until the commit, the rows of a group exist **only on the group connection**. Everything else — DB Sources, DB Sinks outside the group, other lanes, other applications — sees the table as it was before the run (with PostgreSQL's default *read committed* isolation, which Stilton does not change).

- A **DB Source in the same lane** that reads a table the group is writing does not see the new rows. Read before you write, or read in a later run.
- **Locks are held until the end of the lane.** Rows updated or deleted by a member stay locked for other writers; a table truncated on PostgreSQL is locked even for readers. A long lane keeps them for its whole duration.
- **Two groups, or a group and autocommit, writing the same rows** in parallel lanes can wait for each other until one of them ends, or deadlock — in which case the database aborts one side and that member fails. Keep writes to the same table in one lane, or in one group.

Tip. If a lane both loads a large file and writes the transactional result, split it: a first lane prepares the data into a staging table in autocommit, a second lane — or a later run — moves it into place in a short transaction.

10 Lanes, bridges and continuous flows

A group never crosses a lane boundary, and each lane decides about its own groups when **it** ends.



Two lanes, two independent decisions: the customers stay committed even though the orders were rolled back.

- **Writes that must be atomic belong in one lane.** If customers and orders must be all-or-nothing together, write both in the same lane and the same group; a bridge between lanes does not extend a transaction.
- **A bridge does not wait for the commit.** Rows leave the Bridge Out as the sending lane produces them; with *All at once* the receiving lane gets them only when the sender has delivered **all** of them — but that is still before the sending lane's group is decided. If the sending lane then rolls back, the receiving lane has already worked on rows that are not in the database. When the receiver depends on the sender's writes, read them in a **later run**, or write both in the same lane.
- **Dir Watcher and FTP/SFTP Source in continuous mode** run the lane once per *session* — one group of files that arrived together. **Each session is its own transaction**: earlier sessions stay committed whatever happens to a later one, and every session starts from a clean transaction.
- **Other continuous sources** — Webhook Receiver, MQTT, ActiveMQ — run **one** lane for the whole life of the service. A group in such a lane would stay open until the service stops, holding its locks, and commit (or roll back) everything at once. Do not use transactions there: write in autocommit, one event at a time, with an UPSERT so that a repeated event is harmless.

Example. A Dir Watcher in continuous mode picks up `invoices_*.csv` and loads each group of files into `invoice` and `invoice_line` in the group `file`. The file `invoices_0412.csv` has a line with an unknown product code: its session rolls back — no half-loaded invoice — while the files of earlier sessions remain committed. The failed session also ends the run; the file is not marked as processed, so it is picked up again at the next start, after you have fixed it or moved it away.

11 Reading the outcome

In the studio and the Monitor

At the end of the lane every member receives one log line with the decision of its group:

Level	Message	Meaning
info	`Transaction 'order': committed`	all writes of the group are now visible
warn	`Transaction 'order': rolled back — a member failed`	see the failing member's own error, just above
warn	`Transaction 'order': rolled back — the lane was interrupted`	a Stop node, a critical error or a *Stop* rule ended the lane
warn	`Transaction 'order': rolled back — a member did not finish`	a member was halted before it could report
error	`Transaction 'order': NOT committed — COMMIT failed on 'warehouse': …`	the database refused the commit; nothing was written and the run fails

In the Monitor, filter by a member node to see its batches (`NodeProgress`), its error if any, and this final line together.

In the runner log

The runner writes every step of the protocol to its log (standard error: the journal of a systemd service, the log file of a scheduled task). Lines start with `[tx]` for native groups and `[xa]` for XA groups:

runner log — native group, a member failed

```text
[tx] member n_write_lines failed (chunk rolled back, connection cleaned)
[tx] group 'tx_k3f9ab': rollback — a member failed (2/2 members finished)
[tx] group 'tx_k3f9ab': ROLLBACK (aborted=true, resource 'warehouse')
```

runner log — XA group, committed

```text
[xa] group 'tx_m7q2cd': PREPARE TRANSACTION 'fp_r9c1e_tx_m7q2cd_billing' (resource 'billing')
[xa] group 'tx_m7q2cd': PREPARE TRANSACTION 'fp_r9c1e_tx_m7q2cd_warehouse' (resource 'warehouse')
[xa] group 'tx_m7q2cd': COMMIT PREPARED 'fp_r9c1e_tx_m7q2cd_billing' (resource 'billing')
[xa] group 'tx_m7q2cd': COMMIT PREPARED 'fp_r9c1e_tx_m7q2cd_warehouse' (resource 'warehouse')
```

The log uses the **identifiers** of groups, nodes and resources, not their display names; the *Transactions* tab and the node panels show both.

12 Patterns

Staging table and swap

The most robust way to replace the content of a large table, on any database — including those without transaction support in Stilton — is to load a staging table and swap it in with one short statement:

01 A DB Sink writes into `dw.sales_stage` with *DROP + CREATE* (or the *TRUNC+INS* mode), in autocommit: fast, and a failure leaves the live table untouched.

02 Its **post-write SQL** — which runs only if every batch succeeded — swaps the tables.

PostgreSQL — post-write SQL

```sql
BEGIN;
ALTER TABLE dw.sales       RENAME TO sales_old;
ALTER TABLE dw.sales_stage RENAME TO sales;
DROP TABLE dw.sales_old;
COMMIT;
```

Readers see the old table until the instant of the swap, then the new one. The swap holds an exclusive lock only for the few milliseconds of the renames.

Upsert for safe repetition

Prefer **UPSERT** over INSERT in members whenever the data has a natural key. If a run is rolled back and launched again — or if, in autocommit, it is simply launched again — an UPSERT produces the same result however many times it runs.

Audit outside, data inside

Rows that must survive a failure — an audit trail, the error records of the error handler, a "last processed" checkpoint for a later restart — go to DB Sinks **outside** the group, in autocommit. Rows that must be consistent with each other go **inside**. A checkpoint inside the group would be rolled back together with the load it describes, which is usually wrong.

Short transactions

Keep the lane that owns a group short: filter and transform early, write late. Every second the lane runs is a second of locks held and, on PostgreSQL, of old row versions that vacuum cannot clean. For very large loads, consider one group per file or per day rather than one group for the whole history.

13 Troubleshooting

Symptom	Likely cause	What to do
validation error `TX_NATIVE_RESOURCE_MISMATCH`	two members use different resources	same resource for all, or an XA group (PostgreSQL)
`XA requires 'max_prepared_transactions > 0'`	two-phase commit disabled on a server	set `max_prepared_transactions` and restart that server
`transaction identifier "fp_…" is already in use`	a leftover of a killed run with the same identifiers	resolve the leftover in `pg_prepared_xacts`, then rerun
run hangs at the end of a lane, or other applications wait	a prepared XA transaction, or a long group, is holding locks	look in `pg_prepared_xacts` and `pg_stat_activity`; resolve or shorten the lane
group committed although another node failed	the failing node is outside the group and does not interrupt the lane	mark it *Critical*, or give it a *Stop* rule, if it must cancel the load
detail rows fail with a foreign-key error	master in a group, detail in autocommit (or vice versa)	put both in the same group
an empty table remains after a rollback on Oracle	DDL runs outside the transaction on ODBC	expected; use *Create table if it doesn't exist*, or create the table beforehand
the rows of the load are committed although a member failed on Oracle	DDL in the pre- or post-write SQL committed implicitly	move the DDL out of the member's SQL
`Transactions are not available on mysql` (validation), or `transactions are supported on PostgreSQL and ODBC databases … not on mysql` (old plan)	MySQL/MariaDB/SQLite members	remove them from the group; use the staging pattern