Ship the changes, not the table

with leech2

Lars Erik Wik

Northern.tech

2026

What is leech2?

leech2 tracks changes to tables and turns them into small patches that can be applied to a database.

- Computes deltas and stores them in linked blocks
- Consolidates multiple blocks into a patch
- Converts patches into SQL statements

Written in Rust and ships with a C FFI library and a CLI tool.

Documentation

See `lch(1)` and `leech2.h(3)`.

Initializing a project

The `lch init` command creates a work directory with a starter CSV table and a config file telling leech2 how to interpret it.

```
$ lch init
Initialized ./.leech2
$ ls .leech2/
config.toml  products.csv
```

The config file

The leech2 config file can contain:

- Configuration options
- Table definitions
- Paths to drop-in fragments

Note

Both JSON and TOML syntax are supported.

Creating the first block

The `lch block create` command:

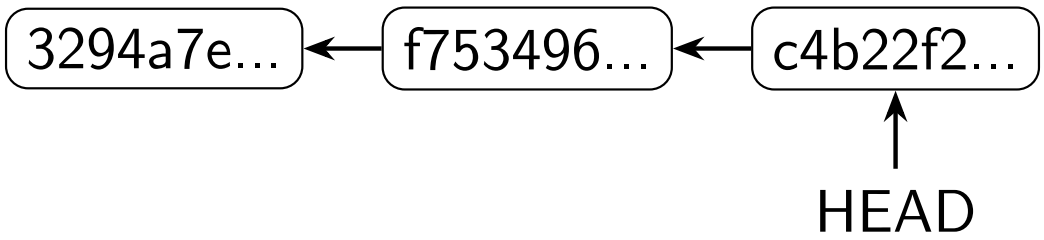
- Creates a new block
- Records the current state

```
$ lch block create
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
$ ls .leech2/state/
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786  HEAD  STATE
```

The block files

Blocks are content-addressed; the name is the SHA-1 digest of the contents. Each block contains:

- Reference to its predecessor
- Creation timestamp
- Payload



The first block

The first block always contains:

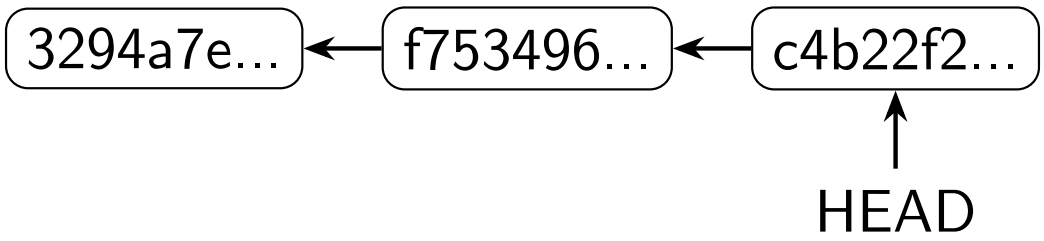
- Reference to the genesis block
- Empty payload

```
$ lch block show
Block:
  Parent: 0000000000000000000000000000000000000000
  Created: 2026-08-25 10:32:44 UTC
  Payload (0 tables)
```

The HEAD file

HEAD is a plain text file that points to the last block in the chain. It's updated whenever a new block is added.

```
$ cat .leech2/state/HEAD
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
```



The STATE file

STATE holds snapshots of the tables as of the most recent block.

```
$ lch state show
State (1 tables):
  'products' [id, name, price]
    (2) "Mouse", 34.5
    (3) "Monitor", 249.95
    (1) "Keyboard", 79.99
```

Note

Primary key columns are shown in parentheses.

Creating a patch

The `lch patch create` command consolidates the changes and stores them in the `PATCH` file.

```
$ lch patch create
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
$ ls .leech2/state/
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786  PATCH  STATS
HEAD                                      STATE
```

Note

Performance statistics are tracked in the `STATS` file.

The PATCH file

The patches contain:

- Reference and timestamp of the block at the head of the chain
- Number of consolidated blocks
- Full table contents or deltas

Note

A patch may contain both deltas and full table contents. For each table, leech2 uses whichever is smaller.

The first patch

The first patch contains only full table contents, since there is no earlier state to diff against. Nothing was consolidated, hence `Blocks: 0`.

```
$ lch patch show
Patch:
  Head: 3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
  Created: 2026-08-25 10:32:44 UTC
  Blocks: 0
  States (1):
    'products' [id, name, price]
      (2) "Mouse", 34.5
      (3) "Monitor", 249.95
      (1) "Keyboard", 79.99
```

Rendering the patch as SQL

`lch patch sql` renders the patch as SQL statements that replicate the changes in the database.

```
$ lch patch sql
TRUNCATE "products";
INSERT INTO "products" ("id", "name", "price")
VALUES (2, 'Mouse', 34.5);
INSERT INTO "products" ("id", "name", "price")
VALUES (3, 'Monitor', 249.95);
INSERT INTO "products" ("id", "name", "price")
VALUES (1, 'Keyboard', 79.99);
```

Marking the patch as applied

`lch patch applied` records the last successful patch in the `REPORTED` file.

```
$ lch patch applied
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
$ cat .leech2/state/REPORTED
3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
```

Note

Next patch will contain the consolidated data from `REPORTED` (exclusive) to `HEAD`.

Marking the patch as failed

`lch patch failed` removes the `REPORTED` file, so the next patch contains full table contents instead of deltas.

```
$ lch patch failed --dry-run
Would have removed './.leech2/state/REPORTED'
0000000000000000000000000000000000000000
```

Note

`--dry-run` reports what would happen without changing anything.

Recording an update

Editing the table

Let's change the mouse's price from 34.50 to 29.99 and record it.

```
$ cat .leech2/products.csv
id,name,price
1,Keyboard,79.99
2,Mouse,34.50
3,Monitor,249.95
$ sed -i 's/34\.50/29\.99/' .leech2/products.csv
$ cat .leech2/products.csv
id,name,price
1,Keyboard,79.99
2,Mouse,29.99
3,Monitor,249.95
```

Recording an update

Creating the block

```
$ lch block create
f75349613c3136219cca647565ad7c0c65051ede
$ lch block show
Block:
  Parent: 3294a7ec4d6f4bdb38c05fd94f0a1a8c6432f786
  Created: 2026-08-25 16:38:27 UTC
  Payload (1 tables):
    'products' [id, name, price]
      Updates (1):
        (2) _, 34.5 -> 29.99
```

Note

The `name` column is stripped because it's irrelevant for block consolidation. However, the previous value of `price` is relevant.

Recording an update

Creating the patch

```
$ lch patch create
f75349613c3136219cca647565ad7c0c65051ede
$ lch patch show
Patch:
  Head: f75349613c3136219cca647565ad7c0c65051ede
  Created: 2026-08-25 16:38:27 UTC
  Blocks: 1
  Deltas (1):
    'products' [id, name, price]
      Updates (1):
        (2) _, 29.99
```

Note

The previous value of `price` was relevant for block consolidation, but is not relevant for patches.

Recording a delete

Editing the table

Next, let's remove the monitor and record it.

```
$ cat .leech2/products.csv
id,name,price
1,Keyboard,79.99
2,Mouse,29.99
3,Monitor,249.95
$ sed -i '/^3,/d' .leech2/products.csv
$ cat .leech2/products.csv
id,name,price
1,Keyboard,79.99
2,Mouse,29.99
```

Recording a delete

Creating the block

```
$ lch block create
c4b22f2680f917bff7a61ad9f5463b52b7ee0861
$ lch block show
Block:
  Parent: f75349613c3136219cca647565ad7c0c65051ede
  Created: 2026-08-26 09:04:56 UTC
  Payload (1 tables):
    'products' [id, name, price]
      Deletes (1):
        (3) "Monitor", 249.95
```

Note

Nothing is stripped here. A `delete` needs all columns for block consolidation.

Recording a delete

Creating the patch

```
$ lch patch create
c4b22f2680f917bff7a61ad9f5463b52b7ee0861
$ lch patch show
Patch:
  Head: c4b22f2680f917bff7a61ad9f5463b52b7ee0861
  Created: 2026-08-26 09:04:56 UTC
  Blocks: 2
  Deltas (1):
    'products' [id, name, price]
      Deletes (1):
        (3) _, _
      Updates (1):
        (2) _, 29.99
```

Note

Only the primary keys are relevant for patches.

Applying the patch

Execute the following statements to mirror the table on the database and mark the patch as applied.

```
$ lch patch sql
DELETE FROM "products"
WHERE "id" = 3;
UPDATE "products"
SET "price" = 29.99
WHERE "id" = 2;
$ lch patch applied
c4b22f2680f917bff7a61ad9f5463b52b7ee0861
```

Block consolidation

Every block in `REPORTED..HEAD` is merged into a single patch.

- 15 merging rules
- `insert` and `delete` carry a primary key and values
- `update` additionally carries the previous value
- Never more than two blocks in memory at a time

No overlapping key

When a key appears in one of two subsequent blocks, there is no conflict. The operation passes through to the result unchanged.

Rule	Parent	Child	Result
1		`insert`	`insert`
2		`delete`	`delete`
3		`update`	`update`
4	`insert`		`insert`
8	`delete`		`delete`
12	`update`		`update`

Impossible combinations

Some rules exist solely to detect logical errors.

Rule	Parent	Child	Result
5	`insert`	`insert`	`error`
10	`delete`	`delete`	`error`
11	`delete`	`update`	`error`
13	`update`	`insert`	`error`

Example

You cannot `insert` something that already exists (rules 5, 13), `delete` something that is already gone (rule 10), or `update` something that has been deleted (rule 11).

`insert` then `delete`

Rule	Parent	Child	Condition	Result
6a	`insert`	`delete`	values match	
6b	`insert`	`delete`	values differ	`error`

Example

An `insert` followed by a `delete` cancels out – the net effect is the same as never inserting the row. However, if the deleted value differs from the inserted one, then this is a logical error.

`insert` then `update`

Rule	Parent	Child	Condition	Result
7a	`insert`	`update`	values match	`insert`
7b	`insert`	`update`	values differ	`error`

Example

An `insert` followed by an `update` collapses into a single `insert` carrying the updated value. However, if the inserted value differs from the update's old value, then this is a logical error.

`delete` then `insert`

Rule	Parent	Child	Condition	Result
9a	`delete`	`insert`	values match	
9b	`delete`	`insert`	values differ	`update`

Example

A `delete` followed by an `insert` cancels out – the net effect is no change at all. However, if the inserted value differs from the deleted one, the pair collapses into a single `update` from the deleted value to the inserted one.

update then delete

Rule	Parent	Child	Condition	Result
14a	update	delete	values match	delete
14b	update	delete	values differ	error

Example

An update followed by a delete collapses into a single delete carrying the update's old value. But if the deleted value differs from the update's new value, then this is a logical error.

update then update

Rule	Parent	Child	Condition	Result
15a	update	update	net change	update
15b	update	update	no net change	
15c	update	update	values differ	error

Example

Two stacked update operations collapse into a single update from the parent's old value to the child's new value. If those two values are equal, the pair cancels out. However, if intermediate values differ, then this is a logical error.