Cloudflare D1

Version Wrangler 4 · verified 2026-11-02

SQLite databases for Workers — queries, batches as transactions, migrations, read replicas and sessions, Drizzle, limits, pricing, and Time Travel.

Contents

Mental model 2

Create and bind 2

Queries from a Worker 3

Batches and transactions 4

Migrations and local development 5

Read replication and sessions 6

Drizzle ORM 8

Limits, pricing, and Time Travel 9

D1 or Durable Object SQLite 10

Further reading 10

Mental model

A D1 database is one SQLite database with a single primary that executes queries one at a time, optionally copied to read-only replicas around the world. Throughput is therefore one over your average query time, writes always travel to the primary, and the only multi-statement transaction is a batch sent in one call. D1 is built to scale out — many small databases of up to 10 GB each, per tenant or per user — not to grow one database forever.

Create and bind

Command or key	Does
`wrangler d1 create NAME`	Creates a database
`--jurisdiction=eu`	Pins data to a region
`d1_databases` in config	Binds it as `env.DB`
`wrangler d1 info NAME`	Size, version, bookmark

The primary is created near where you ran the create command, unless you pass a location hint or a jurisdiction (`eu`, `us`, `fedramp`). A jurisdiction can be set only at creation and cannot be changed later. Workers reach the database through a binding, never a connection string.

```
npx wrangler d1 create app-db

npx wrangler d1 create eu-db --jurisdiction=eu
```

```
{
  "d1_databases": [
    {
      "binding": "DB",
      "database_name": "app-db",
      "database_id": "<UUID>"
    }
  ]
}
```

Queries from a Worker

Method	Returns
`stmt.first()`	First row or `null`
`stmt.first("col")`	One column of the first row
`stmt.all()` / `run()`	`{ results, meta }`
`stmt.raw()`	Rows as arrays
`meta.rows_read`	Rows scanned, which you pay for

Always `prepare()` and `bind()`: binding blocks SQL injection, and a prepared statement can be reused with different values. `run()` behaves like `all()`. `meta` reports `duration`, `changes`, `last_row_id`, `rows_read`, and `rows_written`.

```
const user = await env.DB
  .prepare("SELECT * FROM users WHERE id = ?")
```

```
  .bind(42)
  .first();  // { id: 42, ... } or null

const { meta } = await env.DB
  .prepare("UPDATE users SET seen = ? "
    + "WHERE id = ?")
  .bind(Date.now(), 42)
  .run();
console.log(meta.changes, meta.rows_read);
```

> **Gotcha:** Billing counts rows **read**, not rows returned. A filter on an unindexed column scans the whole table, so a query returning one row can bill 5 million rows read. Add an index and check `meta.rows_read`.

Batches and transactions

Call	Semantics
`env.DB.batch([s1, s2])`	One transaction, one round trip
Statement fails in a batch	Whole batch rolls back
`env.DB.exec(sql)`	Unprepared; maintenance only
`BEGIN TRANSACTION`	Not allowed

Every query already runs in an implicit transaction, so `BEGIN` and `COMMIT` are rejected with `cannot start a transaction within a transaction`. To make several

statements atomic, send them in one `batch()`. A batch also costs one request against the per-invocation query limit and pays the per-request overhead once — the main lever for throughput on a single-threaded database.

```
const move = (sql, ...v) =>
  env.DB.prepare(sql).bind(...v);
const [debit, credit] = await env.DB.batch([
  move("UPDATE acct SET bal = bal - ? "
    + "WHERE id = ?", 50, "a"),
  move("UPDATE acct SET bal = bal + ? "
    + "WHERE id = ?", 50, "b"),
]);
console.log(debit.meta.changes);  // 1
```

> **Gotcha:** You cannot read, decide in JavaScript, and then write inside one transaction — there is no interactive transaction. Push the condition into SQL (`WHERE bal >= ?`, `INSERT ... ON CONFLICT`) and check `meta.changes`.

Migrations and local development

Command	Does
`d1 migrations create DB name`	New numbered `.sql` file
`d1 migrations list DB`	Unapplied migrations
`d1 migrations apply DB --local`	Applies to local copy

Command	Does
`d1 migrations apply DB --remote`	Applies to production
`d1 execute DB --remote --file=x.sql`	Runs a file

Migrations live in `migrations/` and are tracked in a `d1_migrations` table; `migrations_dir`, `migrations_table`, and `migrations_pattern` on the binding change that. `wrangler dev` runs D1 locally on the same engine and persists data between runs. Set `"remote": true` on the binding to develop against production data instead.

```
npx wrangler d1 migrations create app-db init
# edit migrations/0001_init.sql, then:
npx wrangler d1 migrations apply app-db --local
npx wrangler d1 migrations apply app-db --remote
npx wrangler d1 execute app-db --remote \
  --command "SELECT count(*) FROM users"
```

> **Gotcha:** Without `--remote`, `d1 execute` and `d1 migrations apply` target the local database. “The migration ran” but production has no table usually means this flag was missing.

Read replication and sessions

`withSession(arg)`	First query goes to
none or `"first-unconstrained"`	Any replica or primary
`"first-primary"`	The primary

`withSession(arg)`	First query goes to
A bookmark	A copy at least that fresh
`session.getBookmark()`	Latest bookmark to pass on

Read replicas are in beta, enabled per database in the dashboard, and cost nothing extra. Without the Sessions API every query still goes to the primary. A session gives sequential consistency: each query sees a version at least as new as the last one, even across replicas. Return the bookmark to the client and pass it back on the next request.

```
const bm = req.headers.get("x-d1-bookmark")
  ?? "first-unconstrained";
const session = env.DB.withSession(bm);
const { results } = await session
  .prepare("SELECT * FROM posts LIMIT 20")
  .run();
const res = Response.json(results);
res.headers.set("x-d1-bookmark",
  session.getBookmark() ?? "");
return res;
```

> **Gotcha:** Replicas lag the primary by an unbounded amount. A user who writes and then reads through a fresh `"first-unconstrained"` session can miss their own write — carry the bookmark forward.

Drizzle ORM

Piece	Purpose
`drizzle(env.DB)` from `drizzle-orm/d1`	Typed client over the binding
`drizzle-kit generate`	Writes migration SQL
`migrations_pattern`	Finds nested migration folders
`db.batch([...])`	Drizzle's D1 batch

Drizzle wraps the D1 binding; queries still go through prepared statements. Let Drizzle generate SQL migrations and apply them with Wrangler. Drizzle's newer layout writes one folder per migration (`migrations/0001_init/migration.sql`), which Wrangler finds only if `migrations_pattern` matches it.

```
import { drizzle } from "drizzle-orm/d1";
import { eq } from "drizzle-orm";
import { users } from "./schema.js";

export default {
  async fetch(req, env) {
    const db = drizzle(env.DB);
    const rows = await db.select().from(users)
      .where(eq(users.id, 42));
    return Response.json(rows);
```

```
  },
};
```

Limits, pricing, and Time Travel

Limit	Paid	Free
Database size	10 GB	500 MB
Databases per account	1,000,000	10
Queries per Worker invocation	1,000	50
Time Travel window	30 days	7 days
Query duration	30 s	30 s
Bound parameters per query	100	100

Workers Paid includes 25 billion rows read and 50 million rows written per month, then $0.001 per million reads and $1.00 per million writes; storage beyond 5 GB is $0.75 per GB-month. On Free, queries fail once the daily row limits are reached until midnight UTC. From 4 November 2026 a query result over 50 MB fails. Time Travel is always on and restores to any minute in the window.

```
npx wrangler d1 time-travel info app-db \
  --timestamp="2026-11-01T09:00:00Z"
npx wrangler d1 time-travel restore app-db \
  --bookmark=<BOOKMARK>
```

> **Warning:** A Time Travel restore overwrites the database in place. Note the current bookmark from `time-travel info` first, so you can roll the restore forward again.

D1 or Durable Object SQLite

Need	Choose
Familiar app-server-plus-database shape	D1
HTTP API, import/export, migrations tooling	D1
Logic running next to the data	Durable Object SQLite
Per-entity coordination or WebSockets	Durable Object SQLite

Both are SQLite with the same SQL limits and row-based pricing. D1 is the managed product: your Worker talks to it over the network, and it ships tooling. A Durable Object's SQLite lives inside the object, so queries are synchronous calls with no network hop, but you build the routing and tooling yourself.

```
// D1: a network call per query
await env.DB.prepare("SELECT 1").first();

// Durable Object: synchronous, same machine
this.ctx.storage.sql.exec("SELECT 1").one();
```

Further reading

- D1 Workers Binding API

- D1 read replication
- D1 migrations
- D1 limits
- D1 Time Travel
- Drizzle ORM — Cloudflare D1