CronDB

Developer Documentation

A temporal-native relational database engine with Finite State Machine cells, webhook automation, WAL durability, and multi-language SDKs.

Temporal FSM | Webhook Engine | WAL Durability | JS & Python SDKs | DLL Embedding

v1.0 · 2026

Table of Contents

1. Introduction & Feature Overview

CronDB is a self-contained, high-performance relational database engine written in C++. What sets it apart from traditional databases is its Temporal FSM (Finite State Machine) cell model: individual data cells can be programmed to transition through a sequence of values automatically over time — no application-side cron jobs, background workers, or external schedulers required. The engine handles it natively.

CronDB ships as both an HTTP server executable and a compiled shared library (.dll), making it easy to drop into any stack — from a web backend to an embedded game engine.

Core Features

Temporal FSM Cells	Columns can hold timed state sequences — e.g. 'Trial' → 'Expired' after 30 days. The engine evaluates and transitions them in real time, automatically.
Webhook Engine	Register HTTP listeners on any column. When an FSM fires a state change, CronDB POSTs a JSON payload to your endpoint — enabling fully event-driven architectures.
SQL-Like Query Language	A familiar typed language: CREATE, INSERT, SELECT, UPDATE, DELETE, JOIN, LINK, LISTEN, BEGIN/COMMIT/ROLLBACK, CHECKPOINT. Parsed by a hand-written Lexer + Parser.
Binary Storage Engine	Fixed-width binary .bin files with companion .del soft-delete maps for O(1) random-access to any row by index offset.
Write-Ahead Log (WAL)	Every mutating command is appended to cron.wal before execution. On crash, CronDB replays the log to restore the last committed state.
ACID Transactions	BEGIN / COMMIT / ROLLBACK with an in-memory undo log. Changes stay sandboxed in RAM until COMMIT writes them to disk.
Foreign Key Relations	LINK establishes referential integrity between tables. Parent rows referenced by children cannot be deleted, preventing orphaned records.
FSM + Foreign Key Composition	A foreign key column can itself be an FSM — meaning the row it points to changes over time automatically. For example, a child record with user_id = 1 can transition to user_id = 3 after a set duration, re-routing the reference to a different parent row at the engine level, with no application code required.
Auto-Increment Primary Keys	ApexInt columns are engine-managed auto-increment integers backed by an O(log n) B-Tree for fast key lookups.
Multi-Language SDKs	Official SDKs on npm (JavaScript) and PyPI (Python) wrap the HTTP API with ORM-style insert, select, update, delete, and listen methods.
DLL / Shared Library Embedding	Load CronDB directly into any application via ctypes, FFI, or a native C bridge — no HTTP overhead, no separate process.
Concurrent Reader-Writer Locking	The engine allows many simultaneous readers while serialising writes. A dedicated thread drives the FSM clock watcher every 500 ms.

2. Architecture Overview

CronDB is structured into five cooperating layers. A query travels top-to-bottom; the webhook clock thread operates independently in the background.

Layer	Component	Responsibility
1 — Transport	HTTP Server (httplib)	Receives raw query strings over TCP. Routes to the Parser. Returns a JSON ResultSet.
2 — Language	Lexer + Parser	Tokenises the query string into typed tokens, validates syntax, and dispatches to the correct Database method.
3 — Engine	Database	Holds all Table objects in memory. Executes DML/DDL, manages transactions and the webhook registry.
4 — Clock	Webhook Thread	Background thread polling a min-heap priority queue every 500 ms for pending FSM transitions.
5 — Storage	Storage	Binary .bin/.del/.tmp files on disk, WAL append log, checkpoint compaction, and boot-time WAL replay.

Request Lifecycle

1. The client sends an HTTP POST to `/query` with a raw query string.
2. The Lexer scans the string and emits a stream of typed tokens.
3. The Parser consumes the tokens and dispatches to a Database method.
4. The Database acquires the appropriate lock and executes the operation, returning a ResultSet.
5. Mutating operations are appended to the WAL before the response is sent.
6. The ResultSet is serialised to JSON and returned to the client.

3. Data Types

CronDB is strictly typed. Every column must declare one of the following types at table creation time. Type mismatches in INSERT or UPDATE are rejected at parse time.

Type	Description	Storage	Notes
int	Signed 32-bit integer	Fixed width	Supports <, >, =, <=, >= in WHERE clauses.
float	32-bit floating-point number	Fixed width	Supports all numeric comparisons.
bool	Boolean true or false	Fixed width	Values must be the literal identifiers true or false.
string	UTF-8 text, 60 chars by default	Fixed width	Override size with string(N). All rows share the same fixed width.
ApexInt	Auto-increment primary key	Fixed (int)	Engine-managed. Never include this column in INSERT commands.

■ The ApexInt column must always be declared first in CREATE TABLE. It acts as the auto-incrementing primary key and is backed by an O(log n) B-Tree for fast key lookups.

String Sizing

By default, string columns are allocated 60 bytes of fixed storage. You can override this per-column using the string(N) syntax. Because CronDB uses fixed-width binary rows, choosing appropriate sizes up front is important — all rows in a table consume the same number of bytes on disk regardless of actual content length.

EXAMPLE — Custom String Sizes

```
CREATE TABLE users uid ApexInt name string email string 0
CREATE TABLE profiles pid ApexInt username string bio string(200) 0
```

FSM State Limit (maxStates)

The trailing integer at the end of a CREATE TABLE command sets the maximum number of FSM states any cell in that column may define. INSERT and UPDATE commands that exceed this limit are rejected by the engine. Use 0 for columns that should never hold temporal values.

EXAMPLE — maxStates declaration (status column allows up to 4 FSM states)

```
CREATE TABLE warriors wid ApexInt name string hp int status string 4
```

4. Query Language Reference

CronDB uses a typed, SQL-inspired query language. Keywords are case-insensitive. String literals must be wrapped in single or double quotes. Table and column names are case-sensitive identifiers. The CronDB Lexer does not support comment syntax — do not include -- in queries.

4.1 CREATE TABLE

Creates a new persistent table. Column definitions are space-separated name/type pairs. The first column must be the primary key (recommended: ApexInt). An optional integer suffix per column sets its FSM state limit.

Syntax / Example	Description
CREATE TABLE <name> <col> <type> [maxStates] ...	Full CREATE syntax
CREATE TABLE users uid ApexInt name string 0	Table with no temporal columns
CREATE TABLE subs sid ApexInt plan string expires string 3	expires column allows up to 3 FSM states
CREATE TABLE logs lid ApexInt msg string(500) 0	Custom 500-char string column

■ Reserved table names: **webhooks** and **listeners** are used internally by the engine and cannot be created by users.

4.2 DROP TABLE

Permanently deletes a table and all its associated data from disk.

Syntax / Example	Description
DROP TABLE <name>	Delete a table and all its data
DROP TABLE users	Example: delete the users table

4.3 INSERT INTO

Inserts a new row. Columns are assigned with col = value pairs. The ApexInt primary key is always omitted from inserts — the engine assigns it automatically. Temporal FSM values use the --> chain syntax described in Section 5.

Syntax / Example	Description
INSERT INTO <table> <col> = <val> ...	Full INSERT syntax
INSERT INTO users name = 'Alice' age = 28	Simple static row insert
INSERT INTO subs plan = 'Pro' expires = ('Active' --> 'Expired' @ 30 d)	Insert with one-way FSM transition
INSERT INTO items name = 'Sword' state = ('Locked' --> 'Unlocked' @ 7 d c)	Cyclic FSM — loops forever

4.4 SELECT

Retrieves rows from a table. Supports the wildcard *, specific column lists, WHERE filtering with AND (&&) and OR (||) logic, and INNER JOINs across two tables.

Syntax / Example	Description
SELECT * FROM <table>	Select all columns from a table
SELECT col1, col2 FROM <table> WHERE col = 'val'	Select specific columns with a filter
SELECT * FROM <table> WHERE age >= 18 && active = true	Compound AND filter
SELECT * FROM <t> WHERE plan = 'free' || plan = 'trial'	OR filter across two conditions
SELECT * FROM orders JOIN users ON orders.uid = users.uid	Inner join on a matching column pair

■ Temporal FSM cells always return their current computed state based on the elapsed time since insertion. A SELECT always reflects the live, real-time value.

4.5 UPDATE

Modifies one or more columns on rows matching a WHERE clause. Omitting WHERE updates every row in the table. FSM expressions can be assigned on UPDATE using the same chain syntax as INSERT.

Syntax / Example	Description
UPDATE <table> SET <col> = <val> WHERE <cond>	Full UPDATE syntax
UPDATE users SET age = 29 WHERE name = 'Alice'	Update a single field
UPDATE users SET active = false WHERE age < 18	Update with a comparison
UPDATE subs SET expires = ('Trial' --> 'Paid' @ 14 d) WHERE plan = 'free'	Assign a new FSM on update

4.6 DELETE

Removes rows matching a WHERE clause using a soft-delete mechanism. The row's byte slot in the .bin file is flagged in the companion .del file and reclaimed on the next CHECKPOINT. DELETE is blocked when a LINK constraint exists on the target row.

Syntax / Example	Description
DELETE FROM <table> WHERE <cond>	Full DELETE syntax
DELETE FROM users WHERE name = 'Bob'	Delete by a string field
DELETE FROM orders WHERE status = 'cancelled'	Delete all cancelled orders

■ If a row is referenced as a foreign key via LINK, DELETE is rejected to preserve referential integrity. Delete the child rows first.

4.7 LINK — Foreign Keys

LINK establishes a referential integrity constraint. After linking, the engine will reject any DELETE on a parent row that has matching child rows in the foreign table.

Syntax / Example	Description
LINK <foreignTable>.<fk> --> <primaryTable>.<pk>	Full LINK syntax
LINK orders.uid --> users.uid	Link orders.uid to users.uid

The following sequence sets up and exercises a foreign key constraint:

EXAMPLE — Foreign Key Setup & Enforcement (last line will be rejected if uid=1 has orders)

```
CREATE TABLE users uid ApexInt name string 0
CREATE TABLE orders oid ApexInt uid int product string 0
LINK orders.uid --> users.uid
DELETE FROM users WHERE uid = 1
```

■ **Foreign key columns can be FSM-driven.** A column that holds a foreign key reference is not required to be static — it can be declared with a non-zero maxStates limit and given a temporal chain just like any other column. This means the row a child record points to can change automatically over time. For example, a child row with `user_id = ('1' --> '3' @ 30 d)` will reference parent row 1 for the first 30 days, then switch its pointer to parent row 3 — entirely at the engine level. Referential integrity is still enforced at each state: the target row must exist at the time of the transition.

4.8 LISTEN — Webhooks

LISTEN registers an HTTP POST webhook on a specific column. When an FSM cell in that column transitions, the engine fires a POST to the specified URL. An optional WHERE clause filters which rows trigger the webhook. Listeners are persisted to disk and reloaded automatically on engine restart.

Syntax / Example	Description
LISTEN <table>.<col> --> 'http://...'	Basic LISTEN — fires for any change
LISTEN subs.status --> 'https://api.example.com/hook'	Attach a webhook to subs.status
LISTEN subs.status --> 'http://localhost:3000' WHERE plan = 'Pro'	Filtered: fires only for Pro rows

4.9 Transactions: BEGIN / COMMIT / ROLLBACK

CronDB supports explicit transactions. After BEGIN, all write operations are held in a memory-only undo log. COMMIT permanently writes the transaction to disk. ROLLBACK discards all changes since BEGIN and restores the prior state.

Syntax / Example	Description
BEGIN	Start a transaction and open an undo sandbox
COMMIT	Flush the transaction permanently to disk

Syntax / Example	Description
ROLLBACK	Discard all changes made since BEGIN

The example below shows a transaction that is safely rolled back on error:

EXAMPLE — Transaction with Rollback

```
BEGIN
INSERT INTO accounts name = 'Eve' balance = 1000
UPDATE accounts SET balance = 500 WHERE name = 'Alice'
ROLLBACK
```

4.10 CHECKPOINT

CHECKPOINT compacts the database: it flushes all in-memory table state to the binary .bin and .del files, then clears the WAL. This is an optional maintenance operation — WAL replay provides durability even without it. Run CHECKPOINT periodically in production to reclaim WAL disk space and speed up engine boot time.

Syntax / Example	Description
CHECKPOINT	Flush all tables to disk and clear the WAL

5. Temporal FSM Syntax

The Temporal Finite State Machine is CronDB's defining feature. Any column declared with a non-zero maxStates limit can hold a sequence of timed value transitions. The engine computes the active state on every read, comparing the current time against the row's insertion time and each state's configured duration.

How the Engine Evaluates State

Each FSM cell records its sequence of value/duration pairs alongside the timestamp at which the row was inserted. On every read, the engine calculates how much time has elapsed since insertion and walks the state sequence, accumulating durations until it finds the active state. For cyclic FSMs the elapsed time wraps around the total cycle length so the sequence repeats indefinitely. No application polling is needed — the engine always returns the live, real-time value.

5.1 One-Way Transition

The simplest FSM: a cell starts as one value and permanently transitions to another after a specified duration.

SYNTAX — One-Way Transition (trial for 14 days, then permanently expired)

```
INSERT INTO users name = 'Bob' status = ('trial' --> 'expired' @ 14 d)
```

5.2 Multi-Step Chain

Chain as many transitions as the column's maxStates limit allows. Each --> adds a new state with its own duration. The final state persists indefinitely.

SYNTAX — Multi-Step Chain (Idle 1h → Warming 30m → Active 2h → Cooldown forever)

```
INSERT INTO devices name = 'Turbine' state = ('Idle' --> 'Warming' @ 1 h --> 'Active' @ 30 m -->
'Cooldown' @ 2 h)
```

5.3 Cyclic Automata

Append the letter c inside the parentheses to make the FSM loop infinitely. When the last state's duration elapses, the engine wraps back to the first state. This is ideal for repeating patterns such as traffic lights, billing cycles, or equipment cooldowns.

■ The cycle flag **c** must appear inside the closing parenthesis: (... @ 5 s c). Placing it outside the parentheses will cause a parse error.

SYNTAX — Cyclic FSM (note: c is inside the closing parenthesis)

```
INSERT INTO signals name = 'Main St' color = ('Red' --> 'Green' @ 30 s --> 'Yellow' @ 25 s -->
'Red' @ 5 s c)
INSERT INTO subs plan = 'Monthly' state = ('Active' --> 'Renewal' @ 30 d --> 'Active' @ 1 d c)
```

5.4 Time Units

Unit	Suffix	Equivalent Seconds
Seconds	s	1
Minutes	m	60
Hours	h	3,600
Days	d	86,400
Months	mo	2,592,000 (30 days)

6. Webhook Engine

The webhook engine allows CronDB to push events to your application the moment an FSM state transition fires — no polling required.

How It Works

When an FSM cell is written (INSERT or UPDATE), the engine calculates the exact Unix timestamp of the next state transition and schedules it in a min-heap priority queue. A dedicated background thread wakes every 500 milliseconds, processes all transitions whose fire time has elapsed, verifies the row still exists and the FSM has not been overwritten, evaluates any WHERE filters, and fires matching HTTP POST requests with a JSON payload. Cyclic FSMs are automatically re-scheduled for the next cycle.

Webhook Payload

The engine sends an HTTP POST with the following JSON body:

PAYLOAD — HTTP POST Body

```
{
  "table": "warriors",
  "row": 2,
  "column": "status",
  "new_value": "Revealed"
}
```

Persistence

Listeners and pending webhook tasks are both persisted to disk and restored on engine boot — you never need to re-register a listener after a restart. The engine replays the webhook queue as part of its startup sequence.

✔ Webhooks are fired from the clock thread, never from the HTTP request thread. FSM transitions will never block or slow down query responses.

7. Storage Engine Internals

CronDB uses a custom binary format designed for fixed-width, O(1) random-access reads and sequential append writes.

File	Contents	Notes
<table>.bin	Binary rows, fixed width per row	Main data file. Row i is at offset: header + i × rowSize.
<table>.del	One byte per row (0=alive, 1=dead)	Soft-delete map. Physical slots are reclaimed on CHECKPOINT.
<table>.tmp	Temporal FSM state data	Stores the timed state sequences for each FSM-enabled cell in the table.
cron.wal	Append-only raw command log	Replayed on boot to recover committed but un-checkpointed operations.
webhooks	Pending scheduled transition records	Re-enqueued on boot to restore the FSM clock queue.
listeners	Registered LISTEN records	Reloaded on boot so webhooks survive restarts.

Write-Ahead Log (WAL)

Every successful INSERT, UPDATE, DELETE, CREATE, DROP, LINK, and LISTEN command is appended to `cron.wal` as a raw text line before the response is returned to the client. On startup, the engine replays each logged command in sequence to reconstruct the committed state, even if the process crashed before a CHECKPOINT could flush it to the .bin files.

■ Run CHECKPOINT regularly in production to keep the WAL short and minimise startup latency. A long WAL increases boot time proportionally.

8. HTTP Server & API

When CronDB runs as an executable server, it starts an embedded HTTP server (cpp-httplib) that exposes your database over a simple REST-like API. Any language that can send HTTP requests can integrate with CronDB — the official SDKs are convenience wrappers around this API.

Endpoints

Method	Path	Request Body	Response
POST	/query	Raw CronDB query string	JSON ResultSet (see schema below)

JSON ResultSet Schema

Every response shares this envelope structure:

RESPONSE — Successful SELECT

```
{
  "success": true,
  "message": "success! (1 rows affected)",
  "rows_affected": 1,
  "data": [
    { "uid": 1, "name": "Alice", "status": "Active" }
  ]
}
```

RESPONSE — Error

```
{
  "success": false,
  "message": "Error: Syntax Error: Table 'xyz' does not exist.",
  "rows_affected": 0
}
```

Direct HTTP Access (curl)

EXAMPLE — curl

```
curl -X POST http://127.0.0.1:8080/query -d "SELECT * FROM warriors"
curl -X POST http://127.0.0.1:8080/query -d "INSERT INTO warriors name = 'Paladin' hp = 300 status = 'Guard'"
```

9. SDK Reference — JavaScript

The official JavaScript SDK is available on npm as `crondb-driver`. It wraps the HTTP API in clean async/await methods and handles JSON serialisation automatically.

INSTALL

```
npm install crondb-driver
```

Package page: https://www.npmjs.com/package/crondb-driver

Initialisation

JAVASCRIPT

```
const CronDB = require('crondb-driver');
const db = new CronDB('127.0.0.1', 8080);
```

API Methods

Syntax / Example	Description
db.query(queryString)	Send a raw CronDB query. Returns the parsed JSON ResultSet.
db.insert(table, obj)	ORM insert. Maps object keys to column names.
db.select(table, where?)	ORM select. Optional WHERE string. Returns ResultSet with a data array.
db.update(table, obj, where)	ORM update. Applies obj columns to rows matching WHERE.
db.delete(table, where)	ORM delete. Removes rows matching WHERE.
db.listen(table, col, url)	Register a webhook listener on a column.

Inserting Temporal FSM Values via the ORM

Pass the raw FSM expression string as the value. The SDK forwards it verbatim to the engine:

JAVASCRIPT — ORM with Temporal FSM

```
await db.insert('subs', {
    plan: 'Pro',
    status: "('Active' --> 'Expired' @ 30 d)"
});
```

10. SDK Reference — Python

The official Python SDK is available on PyPI as `crondb-driver`. It exposes the same ORM-style interface as the JavaScript SDK.

INSTALL

```
pip install crondb-driver
```

Package page: https://pypi.org/project/crondb-driver/

Initialisation

PYTHON

```
from crondb import CronDB
db = CronDB() # defaults to 127.0.0.1:8080
db = CronDB(host='192.168.1.5', port=9090) # custom host
```

API Methods

Syntax / Example	Description
db.query(query_str)	Send a raw CronDB query. Returns a dict ResultSet.
db.insert(table, data_dict)	ORM insert from a Python dict.
db.select(table, where=None)	ORM select. Optional WHERE string.
db.update(table, data, where)	ORM update. Returns a dict ResultSet.
db.delete(table, where)	ORM delete. Returns a dict ResultSet.

Inserting Temporal FSM Values via the ORM

PYTHON — ORM with Temporal FSM

```
db.insert('robots', {
    'name': 'Terminator',
    'status': "('Searching' --> 'Found' @ 3 s)"
})
```

11. Embedded Usage (.dll / C-Shared Library)

CronDB can be compiled as a shared library (`crondb.dll`) and loaded directly into any host application. This mode bypasses the HTTP server entirely — your application and the database engine share the same process and address space, eliminating all network overhead. This is the recommended integration method for embedded tools, game engines, desktop applications, and offline-first software.

Exported C API

The library exposes three plain C functions, making it callable from any language that supports C FFI — Python (ctypes), Go (cgo), Rust (bindgen), Java (JNA), and more.

Syntax / Example	Description
StartEngine()	Initialise the database engine, load data from disk, and replay the WAL.
ExecuteQuery(char* query) → char*	Execute a CronDB query and return a JSON ResultSet string.
StopEngine()	Flush all data to disk, clear the WAL, and shut down cleanly.

Minimal Python Example

The following shows the minimum code needed to start the engine, run a few queries, and shut down cleanly:

PYTHON — Minimal DLL Integration

```
import ctypes, json

crondb = ctypes.CDLL('./crondb.dll')
crondb.StartEngine.restype = None
crondb.ExecuteQuery.argtypes = [ctypes.c_char_p]
crondb.ExecuteQuery.restype = ctypes.c_char_p
crondb.StopEngine.restype = None

def query(q):
    raw = crondb.ExecuteQuery(q.encode('utf-8'))
    return json.loads(raw.decode('utf-8'))

crondb.StartEngine()
query("CREATE TABLE tasks tid ApexInt title string done bool 0")
query("INSERT INTO tasks title = 'Write docs' done = false")
query("INSERT INTO tasks title = 'Ship it' done = false")
result = query("SELECT * FROM tasks")
for row in result['data']:
    print(row['title'], '->', row['done'])
crondb.StopEngine()
```

Extended Example: Persistent Game World

The example below demonstrates a fuller use case: a persistent game loop where hero quest timers and crop growth stages are driven entirely by CronDB's Temporal FSM engine. The game state is automatically saved between runs — no manual serialisation required.

PYTHON — Persistent Game World via DLL

```
import ctypes, json, os, time
crondb = ctypes.CDLL('./crondb.dll')
crondb.StartEngine.restype = None
crondb.ExecuteQuery.argtypes = [ctypes.c_char_p]
crondb.ExecuteQuery.restype = ctypes.c_char_p
crondb.StopEngine.restype = None
def query(q):
    raw = crondb.ExecuteQuery(q.encode('utf-8'))
    return json.loads(raw.decode('utf-8'))
is_new_game = not os.path.exists('heroes.bin')
crondb.StartEngine()
if is_new_game:
    print('No save found — creating new world...')
    query("CREATE TABLE heroes hid ApexInt name string status string 4")
    query("CREATE TABLE crops cid ApexInt name string state string 4")
    query("INSERT INTO heroes name = 'Arthur' status = 'Idle'")
    query("INSERT INTO heroes name = 'Merlin' status = 'Idle'")
else:
    print('Save found — resuming world...')
gold = 0
def draw_ui():
    os.system('cls' if os.name == 'nt' else 'clear')
    print(f' CHRONOS TAVERN | Gold: {gold}')
    heroes = query('SELECT * FROM heroes')['data']
    crops = query('SELECT * FROM crops')['data']
    print(' HEROES')
    for h in heroes:
        icon = 'Questing' if h['status'] == 'Questing' else 'Idle'
        print(f" [{h['hid']}] {h['name']}: {icon} -> {h['status']}")
    print(' GARDEN')
    for c in crops:
        print(f" [{c['cid']}] {c['name']}: {c['state']}")
while True:
    draw_ui()
    choice = input('Command [1=Quest Arthur, 2=Quest Merlin, 3=Plant, 4=Harvest, 9=Quit]: ')
    if choice == '1':
        query("UPDATE heroes SET status = ('Questing' --> 'Returning' @ 5 s --> 'Idle' @ 3 s)
WHERE name = 'Arthur'")
    elif choice == '2':
        query("UPDATE heroes SET status = ('Questing' --> 'Returning' @ 5 s --> 'Idle' @ 3 s)
WHERE name = 'Merlin'")
    elif choice == '3':
        query("INSERT INTO crops name = 'Moonweed' state = ('Seed' --> 'Sprout' @ 3 s -->
'Harvestable' @ 4 s --> 'Dead' @ 10 s)")
    elif choice == '4':
        ready = query("SELECT * FROM crops WHERE state = 'Harvestable'")
        if ready['rows_affected'] > 0:
```

```
            gold += 50 * ready['rows_affected']
            query("DELETE FROM crops WHERE state = 'Harvestable'")
    elif choice == '9':
        crondb.StopEngine()
        print('World saved. Goodbye!')
        break
```

★ CronDB uses lazy evaluation for FSM state. Timers are not actively ticked by a background thread — instead, each FSM cell records its insertion timestamp and state durations, and the active state is computed on every read by comparing elapsed time against that sequence. This means the game loop does not need to push any updates: the next call to SELECT will automatically return the correct current state, whether that is 'Questing', 'Returning', 'Harvestable', or any other step in the chain.

12. Full Integration Examples

12.1 JavaScript — Full SDK Integration Test

This walkthrough exercises every major feature: raw queries, ORM CRUD, Temporal FSM insertion, webhook receiving, and state verification.

JAVASCRIPT — Full Integration Test

```
const CronDB = require('crondb-driver');
const http = require('http');
const catcher = http.createServer((req, res) => {
    let body = '';
    req.on('data', chunk => body += chunk.toString());
    req.on('end', () => {
        console.log('[WEBHOOK]', body);
        res.writeHead(200);
        res.end();
    });
});
catcher.listen(3000);
const db = new CronDB('127.0.0.1', 8080);
const sleep = ms => new Promise(r => setTimeout(r, ms));
async function runTest() {
    await db.query('CREATE TABLE warriors wid ApexInt name string hp int status string 4');
    await db.listen('warriors', 'status', 'http://127.0.0.1:3000');
    await db.insert('warriors', { name: 'Tank', hp: 200, status: 'Defending' });
    await db.insert('warriors', {
        name: 'Assassin',
        hp: 80,
        status: "('Hidden' --> 'Revealed' @ 3 s)"
    });
    console.log('Initial:', await db.select('warriors'));
    await db.update('warriors', { hp: 150 }, "name = 'Tank'");
    await db.delete('warriors', "name = 'Tank'");
    console.log('Waiting 4 s for FSM to fire...');
    await sleep(4000);
    console.log('Final (Assassin should be Revealed):', await db.select('warriors'));
    catcher.close();
    process.exit(0);
}
runTest();
```

12.2 Python — Full SDK Integration Test

PYTHON — Full Integration Test

```
from crondb import CronDB
import time
```

```
db = CronDB()
db.query('DROP TABLE robots')
print(db.query('CREATE TABLE robots rid ApexInt name string status string 3'))
db.insert('robots', {'name': 'R2D2', 'status': 'Idle'})
db.insert('robots', {'name': 'Terminator', 'status': "('Searching' --> 'Found' @ 3 s)"})
res = db.select('robots')
print(f"Rows: {res['rows_affected']}")
for row in res['data']:
    print(' ->', row)
db.update('robots', {'status': 'Beeping'}, "name = 'R2D2'")
print('Waiting 3 s for FSM...')
time.sleep(3)
final = db.select('robots', "name = 'Terminator'")
print('Terminator status:', final['data'][0]['status']) # Expected: Found
```

13. Error Reference

All errors are returned in the JSON ResultSet with `"success": false` and a descriptive message string.

Error	Cause & Resolution
Table 'X' does not exist.	Table name is misspelled or has not been created yet. Verify with CREATE TABLE.
Primary key not found	CREATE TABLE references a primary key column name that does not exist.
Unmatching type to column	INSERT or UPDATE value type does not match the column's declared type.
Exceeded maximum allowed states limit of N	FSM chain has more states than the column's maxStates limit. Increase the limit or shorten the chain.
Unknown time period	Invalid FSM time unit. Allowed units: s, m, h, d, mo.
Dangling '&&' / '||'	WHERE clause ends with a logic operator and no trailing condition.
Expected column name in WHERE	WHERE clause token order is wrong. Expected: column op value.
Unknown command or missing parameters	First token is not a recognised keyword, or required arguments are absent.
Forbidden table name	CREATE TABLE attempted with a reserved internal name: webhooks or listeners.
Unknown type	Column type in CREATE TABLE is not one of: int, float, bool, string, ApexInt.
Unterminated string literal	A quoted string is missing its closing quote character.
Expected FROM / SET / ON	A required keyword is missing from a SELECT, UPDATE, or JOIN statement.
Foreign key points at rows that were meant for deletion	DELETE blocked by a LINK referential integrity constraint. Delete child rows first.

JavaScript SDK: https://www.npmjs.com/package/crondb-driver

Python SDK: https://pypi.org/project/crondb-driver/

CronDB Developer Documentation · v1.0 · 2026