@push.rocks/smartclickhouse

A TypeScript-based ODM (Object-Document Mapper) for ClickHouse databases, with support for creating and managing tables and their data.

- readme.md for @push.rocks/smartclickhouse
- changelog.md for @push.rocks/smartclickhouse

readme.md for @push.rocks/smartclickhouse

A TypeScript-based ODM for ClickHouse databases with full CRUD support, a fluent query builder, configurable engines, and automatic schema evolution.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Install

```
pnpm install @push.rocks/smartclickhouse
```

Usage

□□ Connecting to ClickHouse

```
import { SmartClickHouseDb } from '@push.rocks/smartclickhouse';

const db = new SmartClickHouseDb({
  url: 'http://localhost:8123',
  database: 'myDatabase',
```

```
  username: 'default',           // optional
  password: 'secret',            // optional
  unref: true,                   // optional — allow process exit during startup
});

await db.start();       // pings until available, creates database if needed
await db.start(true);   // drops and recreates database (useful for test suites)
```

The library communicates with ClickHouse over its HTTP interface — no native protocol driver required.

□□ Creating a Typed Table

Use `db.createTable<T>()` with full control over engine, ordering, partitioning, and TTL:

```
interface ILogEntry {
  timestamp: number;
  level: string;
  message: string;
  service: string;
  duration: number;
}

const logs = await db.createTable<ILogEntry>({
  tableName: 'logs',
  orderBy: ['timestamp', 'service'],
  partitionBy: "toYYYYMM(timestamp)",
  columns: [
    { name: 'timestamp', type: "DateTime64(3, 'Europe/Berlin')" },
    { name: 'level', type: 'String' },
    { name: 'message', type: 'String' },
    { name: 'service', type: 'String' },
    { name: 'duration', type: 'Float64' },
  ],
  ttl: { column: 'timestamp', interval: '90 DAY' },
});
```

⚙ Engine Configuration

Supports the full MergeTree family:

Engine	Use Case
`MergeTree`	Default — append-only, great for logs and events
`ReplacingMergeTree`	Upsert-style mutable data (deduplicates on `OPTIMIZE`)
`SummingMergeTree`	Pre-aggregated counters and metrics
`AggregatingMergeTree`	Materialized aggregate states
`CollapsingMergeTree`	Mutable rows via sign-based collapsing
`VersionedCollapsingMergeTree`	Versioned collapsing for concurrent updates

```
// ReplacingMergeTree for upsert-style mutable data
const users = await db.createTable<IUser>({
  tableName: 'users',
  engine: { engine: 'ReplacingMergeTree', versionColumn: 'updatedAt' },
  orderBy: 'userId',
});

// SummingMergeTree for pre-aggregated metrics
const metrics = await db.createTable<IMetric>({
  tableName: 'metrics',
  engine: { engine: 'SummingMergeTree' },
  orderBy: ['date', 'metricName'],
});
```

□□ Auto-Schema Evolution

When `autoSchemaEvolution` is enabled (default), new columns are created automatically from your data via `ALTER TABLE ADD COLUMN`:

```
const flexTable = await db.createTable<any>({
  tableName: 'events',
  orderBy: 'timestamp' as any,
  autoSchemaEvolution: true,
});

// First insert creates the base schema
await flexTable.insert({ timestamp: Date.now(), message: 'hello' });

// New fields trigger ALTER TABLE ADD COLUMN automatically
```

```
await flexTable.insert({
  timestamp: Date.now(),
  message: 'world',
  userId: 'u123',         // → new String column
  responseTime: 150.5,    // → new Float64 column
  tags: ['a', 'b'],       // → new Array(String) column
});
```

Nested objects are automatically flattened (e.g. `{ deep: { field: 'value' } }` becomes column `deep_field`).

✏ Inserting Data

```
// Single row
await logs.insert({
  timestamp: Date.now(),
  level: 'info',
  message: 'Request processed',
  service: 'api',
  duration: 42.5,
});

// Multiple rows
await logs.insertMany([
  { timestamp: Date.now(), level: 'info', message: 'msg1', service: 'api', duration: 10 },
  { timestamp: Date.now(), level: 'error', message: 'msg2', service: 'worker', duration: 500
},
]);

// Large batch with configurable chunk size
await logs.insertBatch(largeArray, { batchSize: 50000 });
```

□□ Streaming Inserts

Use `createInsertStream()` for push-based insert buffering with automatic batch flushing:

```
const stream = logs.createInsertStream({ batchSize: 100, flushIntervalMs: 1000 });

stream.push({ timestamp: Date.now(), level: 'info', message: 'event1', service: 'api',
```

```
duration: 10 });
stream.push({ timestamp: Date.now(), level: 'info', message: 'event2', service: 'api',
duration: 20 });

// Signal end-of-stream and wait for final flush
stream.signalComplete();
await stream.completed;
```

□□ Querying with the Fluent Builder

The query builder provides type-safe, chainable query construction:

```
// Basic filtered query
const errors = await logs.query()
  .where('level', '=', 'error')
  .orderBy('timestamp', 'DESC')
  .limit(100)
  .toArray();

// Multiple conditions with AND / OR
const result = await logs.query()
  .where('service', '=', 'api')
  .and('duration', '>', 1000)
  .and('level', 'IN', ['error', 'warn'])
  .orderBy('timestamp', 'DESC')
  .limit(50)
  .toArray();

// OR conditions
const mixed = await logs.query()
  .where('level', '=', 'error')
  .or('duration', '>', 5000)
  .toArray();

// Get first match
const latest = await logs.query()
  .orderBy('timestamp', 'DESC')
  .first();
```

```
// Count
const errorCount = await logs.query()
  .where('level', '=', 'error')
  .count();

// Pagination with limit/offset
const page2 = await logs.query()
  .orderBy('timestamp', 'DESC')
  .limit(20)
  .offset(20)
  .toArray();

// Aggregation with raw expressions
const stats = await logs.query()
  .selectRaw('service', 'count() as requests', 'avg(duration) as avgDuration')
  .groupBy('service')
  .having('requests > 100')
  .orderBy('requests' as any, 'DESC')
  .toArray();

// Select specific columns
const names = await logs.query()
  .select('service', 'level')
  .limit(10)
  .toArray();

// Raw WHERE expression for advanced use cases
const advanced = await logs.query()
  .whereRaw("toHour(timestamp) BETWEEN 9 AND 17")
  .toArray();

// Debug — inspect generated SQL without executing
console.log(logs.query().where('level', '=', 'error').limit(10).toSQL());
// → SELECT * FROM mydb.logs WHERE level = 'error' LIMIT 10 FORMAT JSONEachRow
```

Supported Operators

`=`, `!=`, `>`, `>=`, `<`, `<=`, `LIKE`, `NOT LIKE`, `IN`, `NOT IN`, `BETWEEN`

□□ Result Sets

Use `.execute()` to get a `ClickhouseResultSet` with convenience methods:

```
const resultSet = await logs.query()
  .orderBy('timestamp', 'DESC')
  .limit(100)
  .execute();

resultSet.isEmpty();      // boolean
resultSet.rowCount;       // number
resultSet.first();        // T | null
resultSet.last();         // T | null
resultSet.map(r => r.service);               // string[]
resultSet.filter(r => r.duration > 100);     // ClickhouseResultSet<T>
resultSet.toObservable();                    // RxJS Observable<T>
resultSet.toArray();                         // T[]
```

□□ Updating Data

Updates use ClickHouse mutations (`ALTER TABLE UPDATE`). The library automatically waits for mutations to complete.

> □□ For frequently updated data, consider using `ReplacingMergeTree` instead — it's the idiomatic ClickHouse approach for mutable rows.

```
await logs.update(
  { level: 'warn' },                          // SET clause
  (q) => q.where('level', '=', 'warning'),    // WHERE clause
);
```

A WHERE clause is **required** — you can't accidentally update every row.

□□ Deleting Data

```
// Targeted delete with builder
await logs.deleteWhere(
  (q) => q.where('level', '=', 'debug').and('timestamp', '<', cutoffDate),
);

// Delete by age (interval syntax)
await logs.deleteOlderThan('timestamp', '30 DAY');

// Drop entire table
await logs.drop();
```

□□ Watching for New Data

Stream new entries via polling with an RxJS Observable:

```
const subscription = logs.watch({ pollInterval: 2000 }).subscribe((entry) => {
  console.log('New entry:', entry);
});

// Stop watching
subscription.unsubscribe();
```

□□ Utilities

```
await logs.getRowCount();        // total row count
await logs.optimize(true);       // OPTIMIZE TABLE FINAL (dedup for ReplacingMergeTree)
await logs.waitForMutations();   // wait for pending mutations to complete
await logs.updateColumns();      // refresh column metadata from system.columns
```

□□ Raw Queries

Execute arbitrary SQL directly on the database:

```
const result = await db.query<{ total: string }>(
  'SELECT count() as total FROM mydb.logs FORMAT JSONEachRow'
```

```
);
```

Backward Compatibility

The legacy `getTable()` API still works exactly as before. It returns a `TimeDataTable` pre-configured with MergeTree, timestamp ordering, auto-schema evolution, and TTL:

```
const table = await db.getTable('analytics');

// Insert — accepts arbitrary JSON objects, auto-flattens nested fields
await table.addData({
  timestamp: Date.now(),
  message: 'hello',
  nested: { field: 'value' },  // stored as column `nested_field`
});

// Query
const entries = await table.getLastEntries(10);
const recent = await table.getEntriesNewerThan(Date.now() - 60000);
const range = await table.getEntriesBetween(startMs, endMs);

// Delete
await table.deleteOldEntries(30);   // remove entries older than 30 days

// Watch
table.watchNewEntries().subscribe(entry => console.log(entry));

// Drop
await table.delete();
```

You can also use the factory function directly:

```
import { createTimeDataTable } from '@push.rocks/smartclickhouse';

const table = await createTimeDataTable(db, 'analytics', 90 /* retain days */);
```

Running ClickHouse Locally

```
docker run --name clickhouse-server \
  --ulimit nofile=262144:262144 \
  -p 8123:8123 -p 9000:9000 \
  -e CLICKHOUSE_DEFAULT_ACCESS_MANAGEMENT=1 \
  clickhouse/clickhouse-server
```

The HTTP interface is available at `http://localhost:8123` with a playground at `http://localhost:8123/play`.

□□ Exported Types

The library exports all types for full TypeScript integration:

```
import type {
  TClickhouseColumnType,     // String, UInt64, Float64, DateTime64, Array(...), etc.
  TClickhouseEngine,         // MergeTree family engine names
  IEngineConfig,             // Engine + version/sign column config
  IClickhouseTableOptions,   // Full table creation options
  IColumnDefinition,         // Column name + type + default + codec
  IColumnInfo,               // Column metadata from system.columns
  TComparisonOperator,       // =, !=, >, <, LIKE, IN, BETWEEN, etc.
} from '@push.rocks/smartclickhouse';
```

Utility functions are also exported:

```
import { escapeClickhouseValue, detectClickhouseType } from '@push.rocks/smartclickhouse';

escapeClickhouseValue("O'Brien");    // → "'O\\'Brien'"
escapeClickhouseValue(42);           // → '42'
escapeClickhouseValue(['a', 'b']);   // → "('a', 'b')"

detectClickhouseType('hello');       // → 'String'
detectClickhouseType(3.14);         // → 'Float64'
detectClickhouseType([1, 2]);       // → 'Array(Float64)'
```

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

changelog.md for @push.rocks/smartclickhouse

2026-02-27 - 2.2.0 - feat(core)

introduce typed ClickHouse table API, query builder, and result handling; enhance HTTP client and add schema evolution, batch inserts and mutations; update docs/tests and bump deps

- Add generic ClickhouseTable with full table lifecycle, auto-schema-evolution and schema-sync helpers
- Add ClickhouseQueryBuilder for fluent typed queries and SQL generation (includes count/first/execute)
- Add ClickhouseResultSet with utility methods (first, last, map, filter, toObservable)
- Enhance ClickhouseHttpClient: typed query (queryTyped), robust JSONEachRow parsing, error handling, insertBatch, mutatePromise and improved request handling
- Keep backward compatibility via TimeDataTable refactor to wrap new ClickhouseTable API
- Export new modules from ts/index.ts and update README and tests to cover new features
- Bump devDependencies/dependencies, add pnpm patched dependency and patches/agentkeepalive patch, and update npmextra.json metadata

2.1.0 - feat(core): Added comprehensive support for `SmartClickHouseDb` and `TimeDataTable` with features

including time data table creation, data insertion, bulk data insertion, querying, data deletion, and real-time data observation. Included standalone Clickhouse HTTP client implementation.

Fixed

- Fixed test case for table deletion and optimized code