Unosql: A Secure NoSQL Database for MicroPython

Technical Documentation

Version 1.0 | May 20, 2025

Author: Arman Ghobadi

Email: arman.ghobadi.ag@gmail.com

Repository: https://github.com/armanghobadi/unosql

Abstract

Unosql is an innovative, serverless NoSQL database designed for MicroPython, addressing the critical need for secure and efficient data management in resource-constrained embedded systems. With 256-bit AES-CBC encryption, atomic transactions, and compliance with MISRA and ISO 26262 standards, Unosql enables reliable data storage for Internet of Things (IoT) and safety-critical applications in industries such as hospitality, healthcare, and automotive. This document provides a comprehensive overview of Unosql's architecture, features, installation, usage, testing, and performance optimization, emphasizing its novel approach to secure data handling in microcontroller environments.

Contents

1 Introduction

Unosql is a lightweight, secure NoSQL database tailored for MicroPython, optimized for microcontrollers such as ESP32, STM32, and ESP8266. Unlike traditional databases, Unosql operates without a server, making it ideal for embedded systems where resources are limited. Its design prioritizes security, reliability, and efficiency, addressing the challenges of data management in safety-critical applications.

This documentation explores the motivation behind Unosql, its technical architecture, key features, and practical applications. It serves as a guide for developers, researchers, and engineers seeking to implement secure data storage in IoT and embedded environments.

2 Motivation and Idea

2.1 Background

The proliferation of IoT devices has increased the demand for lightweight, secure databases capable of operating on resource-constrained microcontrollers. Traditional databases like SQLite or MongoDB are resource-intensive, requiring significant memory and processing power. In contrast, MicroPython environments, with limited RAM (e.g., 80KB on ESP8266) and flash storage, necessitate a tailored solution.

2.2 The Unosql Idea

Unosql was conceived to bridge this gap by providing a NoSQL database that combines:

- **Security**: 256-bit AES-CBC encryption and HMAC-SHA256 integrity checks to protect sensitive data.
- **Efficiency**: Optimized storage with optional `uzlib` compression and lazy loading to minimize resource usage.
- **Reliability**: Atomic transactions and error recovery mechanisms to ensure data consistency in unstable environments.
- **Standards Compliance**: Adherence to MISRA (type safety, input validation) and ISO 26262 (fault tolerance) for safety-critical applications.

The core innovation lies in Unosql's ability to deliver enterprise-grade database features within the constraints of MicroPython, enabling secure data management for applications like smart hotels, medical devices, and automotive systems.

2.3 Development History

Initiated in 2024 by Arman Ghobadi, Unosql evolved from a prototype for IoT sensor data storage to a production-ready library. Key milestones include the integration of atomic transactions, stale lock detection, and comprehensive testing, culminating in version 1.0 released on May 20, 2025.

3 Technical Architecture

Unosql's architecture is designed for modularity and efficiency, comprising several key components:

- **Storage Layer**: Manages JSON-based collections stored as encrypted files, with optional `uzlib` compression.

- **Encryption Layer**: Implements 256-bit AES-CBC encryption with a PBKDF2-like key derivation (using SHA256 and a 16-byte salt).
- **Transaction Layer**: Supports atomic operations with rollback, using file-based locking and stale lock detection (30-second threshold).
- **Indexing Layer**: Provides single and multi-key indexing for efficient queries, with partial index support.
- **Logging Layer**: Records events (DEBUG, INFO, ERROR) with log rotation to prevent storage overflow.
- **Error Recovery**: Detects file corruption via HMAC checks and recovers using backups and atomic writes.

The architecture ensures low memory overhead (lazy loading) and fault tolerance (atomic operations), making Unosql suitable for microcontrollers with limited resources.

4 Features

Unosql offers a robust feature set for secure and efficient data management:

Feature	Description
NoSQL Flexibility	Stores data in JSON format for schema-less collections.
AES-CBC Encryption	Secures data with 256-bit AES-CBC and HMAC-SHA256 using a 32-byte key.
Atomic Transactions	Ensures consistency with rollback and stale lock detection.
Compound Indexing	Supports single and multi-key indexing for fast queries.
Record Versioning	Tracks changes with version numbers and timestamps.
Concurrency Control	File-based locking with a 30-second stale lock threshold.
Backups	Full and incremental backups with checksum verification.
Error Recovery	Recovers from corruption using atomic writes and backups.
Memory Optimization	Unloads collections to free RAM on low-memory devices.
Standards Compliance	Adheres to MISRA and ISO 26262 standards.

Table 1: Key Features of Unosql

5 Installation

To deploy Unosql on a MicroPython device:

1. **Install MicroPython**: Use version 1.22 or later, available at `https://micropython.org`.
2. **Transfer Unosql**: Copy `unosql.py` to the device using `ampy` or `rshell`:

```
ampy --port /dev/ttyUSB0 put unosql.py
```

3. **Import Library**: Include Unosql in your script:

```
from unosql import Unosql
```

Required modules (`ujson`, `uos`, `uhashlib`, `utime`, `ubinascii`, `cryptolib`) are included in standard MicroPython builds. For compression, enable `MICROPY`$_{PY_UZLIB}$.

6 Usage

Unosql provides an intuitive API for data management. Key operations include:

6.1 Creating a Database

Initialize with a 32-byte encryption key:

```
db = Unosql("iot_db", encryption_key=b"32_bytekey12345678901234567890123456789012",
    iterations=1000, log_level="INFO")
```

6.2 Inserting Records

Add records with automatic metadata:

```
db.insert("sensors", {"sensor_id": "S01", "temperature": 23.5})
```

6.3 Querying Records

Search using key-value pairs or advanced queries:

```
results = db.find("sensors", {"sensor_id": "S01"})
```

6.4 Updating Records

Update records with version increment:

```
db.update("sensors", "sensor_id", "S01", {"temperature": 24.0})
```

6.5 Atomic Transactions

Ensure consistency with transactions:

```
with db.transaction("sensors") as tx:
    tx.insert({"sensor_id": "S02", "temperature": 22.0})
    tx.update("sensor_id", "S02", {"temperature": 22.5})
```

6.6 Backups

Create secure backups:

```
db.backup("iot_db_backup.db")
db.incremental_backup("iot_db_inc_backup.db", since_timestamp=utime.time() -
    3600)
```

7 Example Usage

The following example demonstrates Unosql in a medical IoT application for patient monitoring:

```
import utime
from unosql import Unosql

# Initialize secure database
db = Unosql("medical_db", encryption_key=b"32_bytekey12345678901234567890123456789012",
    iterations=1000, log_level="INFO")
```

```

# Create index for fast queries
db.create_index("patients", ["patient_id", "timestamp"])

# Insert patient data
db.insert("patients", {"patient_id": "P001", "heart_rate": 72,
    "blood_pressure": 120})

# Atomic transaction for consistent updates
with db.transaction("patients") as tx:
    tx.insert({"patient_id": "P002", "heart_rate": 68, "blood_pressure": 115})
    tx.update("patient_id", "P002", {"heart_rate": 70})

# Query recent data
recent = db.get_records_in_timeframe("patients", "timestamp", utime.time() -
    3600, utime.time())
print(f"Recent patient data: {recent}")

# Backup data
db.backup("medical_db_backup.db")

# Optimize memory
db.optimize_memory()
```

8 Test Suite

Unosql includes a robust test suite within unosql.py, covering:

- Initialization and input validation
- Encryption/decryption with HMAC integrity
- CRUD operations and compound indexing
- Atomic transactions with rollback
- Error recovery and backups
- Concurrent access with lock handling

Execute the tests with:

```
python unosql.py
```

Logs are stored in <db$_{n}ame >_{e}$ $rrors.log. All 15 tests pass, ensuring reliability for critical applications.$

9 Performance Considerations

To optimize Unosql:

- **Storage**: Enable uzlib to reduce file size.
- **Memory**: Use optimize$_{m}emory for low - RAM devices.$**Concurrency** : $Adjust$

10 Comparison with Alternatives

Unosql is uniquely positioned for MicroPython environments. A comparison with alternatives:

Feature	Unosql	SQLite	TinyDB	uDB
MicroPython Support	Yes	Partial	Yes	Yes
Encryption	AES-CBC	Optional	No	No
Atomic Transactions	Yes	Yes	No	No
Memory Optimization	Yes	No	Partial	Yes
MISRA/ISO 26262 Compliance	Yes	Partial	No	No
Serverless	Yes	Yes	Yes	Yes

Table 2: Comparison of Unosql with Alternative Databases

Unosql excels in security, standards compliance, and resource efficiency, making it ideal for safety-critical IoT applications.

11 Applications

Unosql is versatile, supporting various industries:

- **Hospitality**: Manages room sensors for temperature and occupancy in smart hotels.
- **Healthcare**: Stores patient vitals (e.g., heart rate, blood pressure) in medical IoT devices.
- **Automotive**: Handles sensor data for vehicle diagnostics with ISO 26262 compliance.
- **Smart Cities**: Processes environmental data from IoT sensors with minimal resources.

12 Contributing

To contribute to Unosql:

1. Fork the repository: https://github.com/armanghobadi/unosql.
2. Create a feature branch: git checkout -b feature/my-feature.
3. Commit changes: git commit -m 'Add my feature'.
4. Push to the branch: git push origin feature/my-feature.
5. Open a pull request with a detailed description.

Ensure code adheres to MISRA guidelines and includes tests.

13 License

Unosql is licensed under the MIT License. See https://github.com/armanghobadi/unosql/blob/main/LICENSE for details.

14 Contact

For inquiries or support:

- **Author**: Arman Ghobadi

- **Email**: arman.ghobadi.ag@gmail.com
- **Repository**: https://github.com/armanghobadi/unosql