BeaverDB: The Complete Guide

SYALIA S.R.L.

Table of contents

1 Introduction

Welcome to BeaverDB!

If you've ever found yourself building a Python application that needs to save some data, but setting up a full-blown database server felt like massive overkill, you're in the right place. BeaverDB is designed to be the "Swiss Army knife" for embedded Python data. It's built for those exact "just right" scenarios: more powerful than a simple pickle file, but far less complex than a networked server like PostgreSQL or MySQL.

1.1 What is BeaverDB?

At its heart, BeaverDB is a multi-modal database in a single SQLite file.

It's a Python library that lets you manage modern, complex data types without ever leaving the comfort of a local file. The name itself tells the story:

B.E.A.V.E.R. stands for **B**ackend for mbedded, **A**ll-in-one **V**ector, **E**ntity, and **R**elationship storage.

This means that inside that one `.db` file, you can seamlessly store and query:

- Key-Value Pairs (like a dictionary for your app's configuration)
- Lists (like a persistent to-do list)
- Vector Embeddings (for AI and semantic search)
- Documents & Text (for full-text search)
- Graph Relationships (to connect your data together)
- …and much more.

All this power comes from building on top of the world's most deployed database engine: SQLite.

1.2 The BeaverDB Philosophy

BeaverDB is built on a few core principles. Understanding these will help you know when and why to choose it for your project.

Robust, Safe, and Durable

Your data should be safe, period. BeaverDB is built to be resilient. Thanks to SQLite's atomic transactions and Write-Ahead Logging (WAL) mode, your database is crash-safe.

If your program crashes mid-operation, your data is never lost or corrupted; the database simply rolls back the incomplete transaction.

Furthermore, it's designed for concurrency. It's both thread-safe (different threads can share one BeaverDB object) and process-safe (multiple, independent Python scripts can read from and write to the same database file at the same time). For tasks that require true coordination, it even provides a simple, built-in distributed lock.

Performant by Default

BeaverDB is fast. It's not just "fast for a small database"–it's genuinely fast for the vast majority of medium-sized projects. Because it's an embedded library, there is zero network latency for any query.

Let's be clear: if you're building the next X (formerly Twitter) and need to handle millions of documents and thousands of queries per second, you'll need a distributed, networked database. But for almost everything else? BeaverDB is more than fast enough. If your project is in the thousand to tens-of-thousands of documents range, you'll find it's incredibly responsive.

Local-First & Embedded

The default, primary way to use BeaverDB is as a single file right next to your code. This means your entire database—users, vectors, chat logs, and all—is contained in one portable file (e.g., my_app.db). You can copy it, email it, or back it up. This "local-first" approach is what makes it so fast and simple to deploy.

Minimal & Optional Dependencies

The core BeaverDB library has zero external dependencies. You can get started with key-value stores, lists, and queues right away.

Want to add vector search? Great! Install the [vector] extra, and BeaverDB will activate its faiss integration. Need a web server? Install the [server] extra, and it unlocks a fastapi-based REST API. This "pay-as-you-go" approach keeps your project lightweight.

Pythonic API

BeaverDB is designed to feel like you're just using standard Python data structures. You shouldn't have to write complex SQL queries just to save a Python dict or list. The goal is to make the database feel like a natural extension of your code.

Standard SQLite Compatibility

This is the "no-magic" rule. The my_app.db file that BeaverDB creates is a 100% standard, valid SQLite file. You can open it with any database tool (like DB Browser for SQLite) and see your data in regular tables. This ensures your data is never locked into a proprietary format.

Synchronous Core with Async Potential

The core library is synchronous, which makes it simple and robust for multi-threaded applications. However, BeaverDB is fully aware of the modern asyncio world. For every data structure, you can call .as_async() to get a fully awaitable version that runs its blocking operations in a background thread, keeping your asyncio event loop from getting blocked.

1.3 Ideal Use Cases

BeaverDB shines in scenarios where simplicity, robustness, and local performance are more important than massive, web-scale concurrency.

- **Local AI & RAG:** Perfect for building Retrieval-Augmented Generation (RAG) applications that run on your local machine. You can store your vector embeddings and their corresponding text right next to each other.
- **Desktop Utilities & CLI Tools:** The ideal companion for a custom tool that needs to remember user preferences, manage a history, or cache results.
- **Chatbots:** A persistent list is a perfect, simple way to store a chatbot's conversation history for a user.
- **Rapid Prototyping:** Get your idea up and running in minutes. Start with a local .db file, and if your project grows, you can deploy it as a REST API without changing your application logic.

1.4 How This Guide is Structured

We've designed this documentation to get you the information you need, whether you're building your first script or contributing to the core.

This guide is split into two main parts:

- **Part 1: The User Guide**

  This is your starting point. After the Quickstart, this is where you'll find an in-depth guide that walks you through how to use every single feature of BeaverDB. We'll explore each "modality" one by one with practical examples.

- **Part 2: The Developer Guide**

  This part is for power users and contributors. We'll go under the hood to look at the why behind the design. This is where we do deep dives into the core architecture, the concurrency model (threading and locking), and the internals of how features like vector search are implemented.

2 Quickstart Guide

This is where the fun begins. Let's get BeaverDB installed and run your first multi-modal script. You'll be up and running in about 30 seconds.

2.1 Installation

BeaverDB is a Python library, so you can install it right from your terminal using pip.

The Core Install

If you just want the core features—like key-value dictionaries, lists, and queues—you can install the zero-dependency package.

```
# This has NO external dependencies
pip install beaver-db
```

This gives you all the core data structures and is perfect for many simple applications.

Installing Optional Features

BeaverDB keeps its core light by making advanced features optional. You can install them as "extras" as needed.

- `beaver-db[vector]`: Adds AI-powered vector search (using faiss).
- `beaver-db[server,cli]`: Adds the fastapi-based REST server and the beaver command-line tool.

For this guide, we recommend installing the `beaver-db[full]` package, which includes everything, so you can follow along with all the examples.

```
# To install all features, including vector search and the server
pip install "beaver-db[full]"
```

With that, you're ready to write some code.

2.2 Your First Example in 10 Lines

Let's create a single Python script that shows off BeaverDB's "multi-modal" power. We'll use three different data types—a dictionary, a list, and a document collection—all in the same database file.

Create a new file named `quickstart.py` and add the following:

```python
from beaver import BeaverDB, Document

# 1. Initialize the database
# This creates a single file "my_data.db" if it doesn't exist
# and sets it up for safe, concurrent access.
db = BeaverDB("my_data.db")

# 2. Use a namespaced dictionary (like a Python dict)
# This is perfect for storing app configuration or user settings.
config = db.dict("app_config")
config["theme"] = "dark"
config["user_id"] = 123

# You can read the value back just as easily:
print(f"App theme is: {config['theme']}")

# 3. Use a persistent list (like a Python list)
# This is great for a to-do list, a job queue, or a chat history.
tasks = db.list("daily_tasks")
tasks.push({"id": "task-001", "desc": "Write project report"})
tasks.push({"id": "task-002", "desc": "Deploy new feature"})

# You can access items by index, just like a normal list:
print(f"First task is: {tasks[0]['desc']}")

# 4. Use a collection for rich documents and search
# This is the most powerful feature, combining data and search.
articles = db.collection("articles")

# Create a Document to store.
# We give it a unique ID and some text content.
doc = Document(
    id="sqlite-001",
    content="SQLite is a powerful embedded database ideal for local apps."
)

# 5. Index the document
# This not only saves the document but also automatically
# makes its text content searchable via a Full-Text Search (FTS) index
# with optional fuzzy matching.
articles.index(doc, fts=True, fuzzy=True)

# 6. Perform a full-text search
# This isn't a simple string find; it's a real search engine with fuzzy matching!
results = articles.match(query="datbase", fuzziness=1)

# The result is a list of tuples: (document, score)
top_doc, rank = results[0]
print(f"Search found: '{top_doc.content}' (Score: {rank:.2f})")
```

Here's a line-by-line explanation of what you just did:

- **`from beaver import BeaverDB, Document`** `BeaverDB` is the main class, your entry point to the database. A `Document` is a special data object used when you're working with `db.collection()`.
- **`db = BeaverDB("my_data.db")`** This is the most important line. It finds `my_data.db` or creates it if it's not there. It also automatically enables all the high-performance and safety features (like Write-Ahead Logging) so it's ready for use.
- **`config = db.dict("app_config")`** Here, you're asking BeaverDB for a dictionary. `"app_config"` is the "namespace." This means you can have *many* different dictionaries (`app_config`, `user_prefs`, `cache`, etc.) that won't interfere with each other. The `config` object you get back behaves just like a standard Python `dict`. When you do `config["theme"] = "dark"`, that change is instantly saved to the `my_data.db` file.
- **`tasks = db.list("daily_tasks")`** Same idea, but for a list. You get back a `tasks` object that acts like a Python `list`. You can `push` (append) items, get items by index (`tasks[i]`), or `pop` them. You can also insert and remove items at an arbitrary index, and it all works instantaneously (in CS terms, it's O(1) for all operations).
- **`articles = db.collection("articles")`** This gets you a "collection," which is the most powerful data structure. A collection is designed to hold rich data, like articles, user profiles, or AI embeddings.
- **`doc = Document(...)`** To put something in a collection, you wrap it in a `Document`. Here, we give it a unique `id` and some `content`. You can add any other fields you want just by passing them as keyword arguments.
- **`articles.index(doc, ...)`** This is where the magic happens. When you call `.index()`, BeaverDB saves your document. But it *also* reads the `content` field and automatically puts all the words into a Full-Text Search (FTS) index and a clever fuzzy index, which are optional.
- **`results = articles.match(query="database")`** This line runs a search. Because `index()` already did the work, this query is fast. It searches the FTS index for the word "database" and finds your document.

When you run the script, you've created a single `my_data.db` file that now contains your config, your task list, and your searchable articles.

3 Feature Summary

This guide provides a comprehensive set of practical, code-first examples for every major feature of BeaverDB.

3.1 Core Data Structures

3.1.1 Key-Value Dictionaries (`db.dict`)

Use a namespaced dictionary for storing simple key-value data like application configuration or user profiles.

First, let's import `BeaverDB` and get a handle to a namespaced dictionary called `"app_config"`.

```python
from beaver import BeaverDB

db = BeaverDB("demo.db")

# Get a handle to a namespaced dictionary
config = db.dict("app_config")
```

Now you can use the `config` object just like a standard Python dictionary. Operations are saved to the database file instantly.

```python
# --- 1. Setting Values ---
config["theme"] = "dark"
config["retries"] = 3

# You can also use the .set() method
config.set("user_ids", [101, 205, 301])

print(f"Configuration dictionary has {len(config)} items.")

# --- 2. Retrieving Values ---
# Use standard dict access
theme = config["theme"]
print(f"Retrieved theme: {theme}")
```

```python
# Or use .get() with a default value
non_existent = config.get("non_existent_key", "default_value")
print(f"Result for non_existent_key: {non_existent}")
```

Iterating and deleting items also follows standard Python syntax.

```python
# --- 3. Iterating Over the Dictionary ---
print("\nIterating over config items:")
for key, value in config.items():
    print(f"  - {key}: {value}")

# --- 4. Deleting an Item ---
del config["retries"]
print(f"\nAfter deleting 'retries', config has {len(config)} items.")

db.close()
```

3.1.2 Caching with TTL (`db.dict`)

Leverage a dictionary with a **Time-To-Live (TTL)** to cache the results of slow network requests.

Let's define a mock function that simulates a slow API call and get a handle to a dictionary we'll use as our cache.

```python
import time
from beaver import BeaverDB

def expensive_api_call(prompt: str):
    """A mock function that simulates a slow API call."""
    print(f"--- Making expensive API call for: '{prompt}' ---")
    time.sleep(2)  # Simulate network latency
    return "Quito"

db = BeaverDB("demo.db")
api_cache = db.dict("api_cache")
prompt = "capital of Ecuador"
```

On our first attempt, the cache is empty, so we get a "cache miss." We call our expensive function and then use `.set()` with `ttl_seconds=10` to store the result for 10 seconds.

```python
# --- 1. First Call (Cache Miss) ---
print("\nAttempt 1: Key is not in cache.")
response = api_cache.get(prompt)
if response is None:
    print("Cache miss.")
    response = expensive_api_call(prompt)
    # Set the value in the cache with a 10-second TTL
    api_cache.set(prompt, response, ttl_seconds=10)

print(f"Response: {response}")
```

Now, if we request the same data within the 10-second window, we get a “cache hit.” The data is returned instantly from the cache, and our expensive function is never called.

```python
# --- 2. Second Call (Cache Hit) ---
print("\nAttempt 2: Making the same request within 5 seconds.")
time.sleep(5)
response = api_cache.get(prompt)
if response is None:
    # ... (this won't be called) ...
    pass
else:
    print("Cache hit!")

print(f"Response: {response}")
```

Finally, we wait 12 seconds, which is longer than our TTL. The cache key has automatically expired. When we request the data, it’s a “cache miss” again, forcing us to re-run the expensive function and update the cache for another 10 seconds.

```python
# --- 3. Third Call (Cache Expired) ---
print("\nAttempt 3: Waiting for 12 seconds for the cache to expire.")
time.sleep(12)
response = api_cache.get(prompt)
if response is None:
    print("Cache miss (key expired).")
    response = expensive_api_call(prompt)
    api_cache.set(prompt, response, ttl_seconds=10)
else:
    print("Cache hit!")

print(f"Response: {response}")
db.close()
```

3.1.3 Persistent Lists (`db.list`)

A persistent list is perfect for storing ordered data, like the history of a conversation. It supports a full Python list API.

```python
from beaver import BeaverDB

db = BeaverDB("demo.db")
tasks = db.list("project_tasks")
```

You can add items to the beginning or end using `push` (like `append`) and `prepend`.

```python
# --- 1. Pushing and Prepending Items ---
tasks.push({"id": "task-002", "desc": "Write documentation"})
tasks.push({"id": "task-003", "desc": "Deploy to production"})
tasks.prepend({"id": "task-001", "desc": "Design the feature"})
```

Accessing, iterating, and slicing work exactly as you'd expect.

```python
# --- 2. Iterating Over the List ---
print("\nCurrent tasks in order:")
for task in tasks:
    print(f"  - {task['id']}: {task['desc']}")

# --- 3. Accessing and Slicing ---
print(f"\nThe first task is: {tasks[0]}")
print(f"The last task is: {tasks[-1]}")
print(f"A slice of the first two tasks: {tasks[0:2]}")
```

You can also modify the list in place with standard `__setitem__` and `__delitem__` syntax, or use stack/queue methods like `pop` and `deque`.

```python
# --- 4. Updating an Item in Place ---
print("\nUpdating the second task...")
tasks[1] = {"id": "task-002", "desc": "Write and review documentation"}
print(f"Updated second task: {tasks[1]}")

# --- 5. Deleting an Item by Index ---
print("\nDeleting the first task ('task-001')...")
del tasks[0]
print(f"List length after deletion: {len(tasks)}")

# --- 6. Popping the Last Item ---
last_item = tasks.pop()
```

```python
print(f"\nPopped the last task: {last_item}")

db.close()
```

3.1.4 Priority Queues (`db.queue`)

Use a persistent priority queue to manage tasks for an AI agent or any worker system. This ensures the most important tasks are always processed first.

Items are added with a `priority` (lower numbers are processed first). The `.get()` method is atomic and blocks until an item is available, making it perfect for worker processes.

```python
from beaver import BeaverDB

db = BeaverDB("demo.db")
tasks = db.queue("agent_tasks")

# Tasks are added with a priority (lower is higher)
tasks.put({"action": "summarize_news", "topic": "AI"}, priority=10)
tasks.put({"action": "respond_to_user", "user_id": "alice"}, priority=1)
tasks.put({"action": "run_backup", "target": "all"}, priority=20)
tasks.put({"action": "send_alert", "message": "CPU at 90%"}, priority=1)

# The agent retrieves the highest-priority task
# This is a blocking call that waits for an item
item1 = tasks.get() # -> Returns "respond_to_user" (priority 1, added first)
item2 = tasks.get() # -> Returns "send_alert" (priority 1, added second)
item3 = tasks.get() # -> Returns "summarize_news" (priority 10)

print(f"Agent's first task: {item1.data['action']}")
print(f"Agent's second task: {item2.data['action']}")

db.close()
```

3.1.5 Blob Storage (`db.blob`)

Use the blob store to save binary data like user avatars, attachments, or generated reports directly in the database.

First, we'll get a handle to our blob store and prepare some sample binary data.

```python
from beaver import BeaverDB

db = BeaverDB("demo.db")
```

```
attachments = db.blobs("user_uploads")

# Create some sample binary data
file_content = "This is the content of a virtual text file."
file_bytes = file_content.encode("utf-8")
file_key = "emails/user123/attachment_01.txt"
```

Now, we store the data using `.put()` with a unique key and some JSON metadata.

```
# Store a user's avatar with metadata
attachments.put(
    key=file_key,
    data=file_bytes,
    metadata={"mimetype": "text/plain", "sender": "alice@example.com"}
)
```

We can retrieve the blob using `.get()`, which returns a `Blob` object containing the key, data, and metadata.

```
# Retrieve it later
blob = attachments.get(file_key)
if blob:
    print(f"Retrieved Blob: {blob.key}")
    print(f"Metadata: {blob.metadata}")
    print(f"Data (decoded): '{blob.data.decode('utf-8')}'")
```

Finally, we can clean up the blob using `.delete()`.

```
# Delete the blob
attachments.delete(file_key)
print(f"\nVerified deletion: {file_key not in attachments}")

db.close()
```

3.2 The Document Collection (`db.collection`)

The collection is the most powerful data structure, combining document storage with vector, text, and graph search.

3.2.1 RAG System (Hybrid Search)

This example shows how to combine keyword and vector search for a simple RAG pipeline.

```python
from beaver import BeaverDB, Document
from beaver.collections import rerank

db = BeaverDB("rag_demo.db")
articles = db.collection("articles")
```

First, we index our documents. Each `Document` contains text for FTS (in `body`) and a vector (`embedding`). We enable FTS by setting `fts=True`.

```python
# Index documents with both text and embeddings
docs_to_index = [
    Document(
        id="py-fast",
        embedding=[0.1, 0.9, 0.2],  # Vector leans towards "speed"
        body="Python is a great language for fast prototyping.",
    ),
    Document(
        id="py-data",
        embedding=[0.8, 0.2, 0.9],  # Vector leans towards "data science"
        body="The Python ecosystem is ideal for data science.",
    ),
    Document(
        id="js-fast",
        embedding=[0.2, 0.8, 0.1],  # Vector similar to "py-fast"
        body="JavaScript engines are optimized for fast execution.",
    ),
]
for doc in docs_to_index:
    articles.index(doc, fts=True) # Enable FTS
```

Now, we perform two separate searches. One is a keyword search for “python,” and the other is a semantic vector search for a query representing “high-performance code.”

```python
# 1. Vector Search for "high-performance code"
query_vector = [0.15, 0.85, 0.15] # A vector close to "fast"
vector_results = [doc for doc, _ in articles.search(vector=query_vector)]

# 2. Full-Text Search for "python"
text_results = [doc for doc, _ in articles.match(query="python")]
```

Finally, we use the `rerank` helper to merge these two lists. It promotes documents that appear high in *both* result sets, giving us the most relevant answer.

```
# 3. Combine and rerank to get the best context
best_context = rerank(text_results, vector_results)

print("--- Final Reranked Results ---")
for doc in best_context:
    print(f"  - {doc.id}: {doc.body}")

db.close()
```

3.2.2 Knowledge Graphs (`db.collection`)

A collection also serves as a graph. We can connect documents to build a network of relationships.

```
from beaver import BeaverDB, Document

db = BeaverDB("graph_demo.db")
net = db.collection("social_network")
```

First, we index our documents, which will act as the **nodes** in our graph.

```
# 1. Create Documents (nodes)
alice = Document(id="alice", body={"name": "Alice"})
bob = Document(id="bob", body={"name": "Bob"})
charlie = Document(id="charlie", body={"name": "Charlie"})
diana = Document(id="diana", body={"name": "Diana"})
net.index(alice); net.index(bob); net.index(charlie); net.index(diana)
```

Next, we create relationships (or **edges**) between them using the `.connect()` method. We can give each connection a `label` to define the relationship type.

```
# 2. Create Edges (relationships)
net.connect(alice, bob, label="FOLLOWS")
net.connect(alice, charlie, label="FOLLOWS")
net.connect(bob, diana, label="FOLLOWS")
net.connect(charlie, bob, label="FOLLOWS")
```

We can now query these relationships. `.neighbors()` finds all nodes one hop away.

```
# 3. Find 1-hop neighbors
print("--- Testing `neighbors` (1-hop) ---")
following = net.neighbors(alice, label="FOLLOWS")
print(f"Alice follows: {[p.id for p in following]}")
```

For multi-hop traversals (like "friends of friends"), we use the `.walk()` method.

```python
# 4. Find multi-hop connections (e.g., "friends of friends")
print("\n--- Testing `walk` (multi-hop) ---")
foaf = net.walk(
    source=alice,
    labels=["FOLLOWS"],
    depth=2,
)
print(f"Alice's extended network (friends of friends): {[p.id for p in foaf]}")

db.close()
```

3.3 Real-Time & Concurrency

3.3.1 Inter-Process Locks (`db.lock`)

Run multiple scripts in parallel and use `db.lock()` to coordinate them. This example simulates two scrapers trying to refresh a shared sitemap. The `db.lock()` ensures that only one process can enter the critical section, preventing a "thundering herd" race condition. The `timeout=1` causes other processes to skip rather than wait.

```python
import time
import os
from beaver import BeaverDB

db = BeaverDB("scraper_state.db")
scrapers_state = db.dict("scraper_state")

last_refresh = scrapers_state.get("last_sitemap_refresh", 0)
if time.time() - last_refresh > 3600: # Only refresh once per hour
    try:
        # Try to get a lock, but don't wait long
        with db.lock("refresh_sitemap", timeout=1):
            # We got the lock.
            print(f"PID {os.getpid()} is refreshing the sitemap...")
            scrapers_state["sitemap"] = ["/page1", "/page2"] # Your fetch_sitemap()
            scrapers_state["last_sitemap_refresh"] = time.time()

    except TimeoutError:
        # Another process is already refreshing
        print(f"PID {os.getpid()} letting other process handle refresh.")

sitemap = scrapers_state.get("sitemap")
```

```
print(f"PID {os.getpid()} proceeding with sitemap: {sitemap}")
db.close()
```

3.3.2 Atomic Batch Operations (`manager.acquire`)

Ensure a worker process can safely pull a *batch* of items from a queue without another worker interfering, using the built-in manager lock. By wrapping the operation in `with db.queue(...).acquire()`, we guarantee that this process can pull 10 items from the queue atomically, without another worker process stealing items in the middle of the batch.

```
from beaver import BeaverDB

db = BeaverDB("tasks.db")
tasks_to_process = []
try:
    # This lock guarantees no other process can access 'agent_tasks'
    # while this block is running.
    with db.queue('agent_tasks').acquire(timeout=5) as q:
        for _ in range(10): # Get a batch of 10
            item = q.get(block=False)
            tasks_to_process.append(item.data)
except (TimeoutError, IndexError):
    # Lock timed out or queue was empty
    print("Could not get 10 items.")
    pass

# Now process the batch outside the lock
print(f"Processing batch of {len(tasks_to_process)} items.")
db.close()
```

3.3.3 Pub/Sub Channels (`db.channel`)

BeaverDB's pub/sub system allows for real-time, multi-process communication. Here is how two separate processes can communicate.

In one process (or thread), a publisher sends a message to a named channel. This is a fast, single `INSERT` operation.

```
# --- In one process or thread (e.g., a monitoring service) ---
#
import time
from beaver import BeaverDB

db = BeaverDB("demo.db")
```

```
system_events = db.channel("system_events")
print("[Publisher] Publishing message...")
system_events.publish({"event": "user_login", "user_id": "alice"})
db.close()
```

In another process, a subscriber “listens” to that same channel. The `with ...subscribe()` block handles registration, and `listener.listen()` blocks until a message arrives.

```
# --- In another process or thread (e.g., a UI updater or logger) ---
#
from beaver import BeaverDB

db = BeaverDB("demo.db")
# The 'with' block handles the subscription lifecycle.
with db.channel("system_events").subscribe() as listener:
    for message in listener.listen():
        print(f"Event received: {message}")
        # >> Event received: {'event': 'user_login', 'user_id': 'alice'}
        break # Exit after one message for this example
db.close()
```

3.3.4 Live-Aggregating Logs (`db.log`)

Monitor your application’s health in real-time. The `.live()` method provides a continuously updating, aggregated view of your log data, perfect for terminal dashboards.

First, we define an ‘aggregator’ function that takes a list of log entries and returns a single summary dictionary.

```
from datetime import timedelta
import statistics
import time
from beaver import BeaverDB

db = BeaverDB("live_log_demo.db")
logs = db.log("system_metrics")

def summarize(window: list[dict]) -> dict:
    """Aggregator that calculates stats from a window of log data."""
    if not window:
        return {"mean": 0.0, "count": 0}
    values = [log.get("value", 0) for log in window]
    return {"mean": statistics.mean(values), "count": len(values)}
```

```
# Start a background thread to write logs (see examples/logs.py for full code)
# ...
```

Next, we get the live iterator. We tell it to maintain a 5-second rolling window of data and to compute a new summary every 1 second using our function.

```
# Get the live iterator over a 5-second rolling window, updating every 1 sec
live_summary = logs.live(
    window=timedelta(seconds=5),
    period=timedelta(seconds=1),
    aggregator=summarize
)
```

Finally, we just iterate over `live_summary`. This loop will block and yield a new summary object every 1 second, giving us a real-time view of our data.

```
print("[Main Thread] Starting live view. Press Ctrl+C to stop.")
try:
    for summary in live_summary:
        print(f"Live Stats (5s window): Count={summary['count']}, Mean={summary['mean']:.2
        time.sleep(1) # In a real app, this loop just blocks
except KeyboardInterrupt:
    print("\nShutting down.")
finally:
    db.close()
```

3.3.5 Event-Driven Callbacks (`.on` / `.off`)

Listen for database changes in real-time. You can subscribe to events on specific managers (e.g., `db.dict("config").on("set", ...)` to trigger workflows or update UIs).

First, let's define a callback function that will be triggered when a change occurs.

```
import time
from beaver import BeaverDB

db = BeaverDB("events_demo.db")
config = db.dict("app_config")
config.clear()

def on_config_change(payload):
    """This callback is triggered when a key is set or deleted."""
    print(f"EVENT RECEIVED: Key '{payload['key']}' was changed!")
```

Now, we subscribe to the 'set' and 'del' events on our `config` dictionary using the `.on()` method. This returns a handle that we can use to unsubscribe later.

```python
# Subscribe to 'set' events on this specific dict
set_handle = config.on("set", on_config_change)
del_handle = config.on("del", on_config_change)

# Give the listener thread time to start
time.sleep(0.1)
```

When we modify the dictionary, the callback is triggered automatically in a background thread.

```python
print("Setting 'theme'...")
config["theme"] = "dark"  # Triggers the 'on_config_change' callback
time.sleep(0.1) # Wait for event to process

print("Deleting 'theme'...")
del config["theme"] # Triggers the 'on_config_change' callback
time.sleep(0.1)
```

To stop listening, we call `.off()` on the handles. Subsequent changes will be silent.

```python
# Clean up the listeners
set_handle.off()
del_handle.off()

print("Listeners are off. This change will be silent.")
config["theme"] = "light" # No event will be printed
time.sleep(0.1)

db.close()
```

3.4 Advanced Features

3.4.1 Type-Safe Models with Pydantic

BeaverDB has first-class support for Pydantic. By associating a `BaseModel` with a data structure, you get automatic, recursive (de)serialization and data validation.

By passing `model=User` to the `db.dict()` factory, BeaverDB will automatically validate data on write and, more importantly, deserialize the stored JSON back into a full `User` object on read, giving you type safety and editor autocompletion.

```python
from pydantic import BaseModel
from beaver import BeaverDB

# Define your Pydantic model
class User(BaseModel):
    name: str
    email: str
    permissions: list[str]

db = BeaverDB("user_data.db")

# Associate the User model with a dictionary
users = db.dict("user_profiles", model=User)

# BeaverDB now handles serialization automatically
users["alice"] = User(
    name="Alice",
    email="alice@example.com",
    permissions=["read", "write"]
)

# The retrieved object is a proper, validated User instance
retrieved_user = users["alice"]

# Your editor will provide autocompletion here
print(f"Retrieved: {retrieved_user.name}")
print(f"Permissions: {retrieved_user.permissions}")

db.close()
```

3.4.2 Server & CLI

You can instantly expose your database over a RESTful API and interact with it from the command line.

1. Start the Server

```bash
# Start the server for your database file
beaver serve --database data.db --port 8000
```

2. Interact with the API (e.g., from `curl`)

```bash
# Set a value in the 'app_config' dictionary
curl -X PUT [http://127.0.0.1:8000/dicts/app_config/api_key](http://127.0.0.1:8000/dicts/a
     -H "Content-Type: application/json" \
     -d '"your-secret-api-key"'

# Get the value back
curl [http://127.0.0.1:8000/dicts/app_config/api_key](http://127.0.0.1:8000/dicts/app_conf
# Output: "your-secret-api-key"
```

3. Interact with the CLI Client

The `beaver` CLI lets you call any method directly from your terminal.

```
# Get a value from a dictionary
beaver --database data.db dict app_config get theme

# Set a value (JSON is automatically parsed)
beaver --database data.db dict app_config set user '{"name": "Alice", "id": 123}'

# Push an item to a list
beaver --database data.db list daily_tasks push "Review PRs"

# Run a script protected by a distributed lock
beaver --database data.db lock my-cron-job run bash -c 'run_daily_report.sh'
```

3.4.3 Data Export & Backups (`.dump`)

All data structures support a `.dump()` method for easy backups and migration to a JSON file.

```python
import json
from beaver import BeaverDB

db = BeaverDB("my_app.db")
config = db.dict("app_config")
config["theme"] = "dark"
config["user_id"] = 456

# Dump the dictionary's contents to a JSON file
with open("config_backup.json", "w") as f:
    config.dump(f)

# You can also get the dump as a Python object
dump_data = config.dump()
print(dump_data['metadata'])

db.close()
```

You can also use the CLI to dump data:

```
beaver --database data.db collection my_documents dump > my_documents.json
```

3.5 Further Reading

Keep exploring BeaverDB with these in-depth guides:

- Key-Value Dictionaries and Blob Storage
- Persistent Lists and Priority Queues
- Document Collections (Vectors, FTS, Graphs)
- Real-Time Data (Pub/Sub and Live Logs)
- Concurrency and Inter-Process Locking
- Deployment with the REST Server and CLI

Part I

The User Guide

4 Key-Value and Blob Storage

Chapter Outline:

- **3.1. Dictionaries & Caching (`db.dict`)**
  - A Python-like dictionary interface: `config["api_key"] = ...`
  - Standard methods: `.get()`, `del`, `len()`, iterating with `.items()`.
  - **Use Case: Caching with TTL:** Using `.set(key, value, ttl_seconds=3600)`.
- **3.2. Blob Storage (`db.blobs`)**
  - Storing binary data (images, PDFs, files) with metadata.
  - API: `.put(key, data, metadata)`, `.get(key)`.
  - The `Blob` object: Accessing `.data` and `.metadata`.

5 Lists and Queues

Chapter Outline:

- **4.1. Persistent Lists (`db.list`)**
  - A full-featured, persistent Python list.
  - Full support for: `push`, `pop`, `prepend`, `deque`, slicing `my_list[1:5]`, and in-place updates `my_list[0] = ...`.
- **4.2. Priority Queues (`db.queue`)**
  - Creating a persistent, multi-process task queue.
  - Adding tasks: `.put(data, priority=N)` (lower number is higher priority).
  - Consuming tasks: The blocking `.get(timeout=N)` method.
  - **Use Case:** A multi-process producer/consumer pattern.

6 The Document Collection (`db.collection`)

Chapter Outline:

- **5.1. Documents & Indexing**
  - The `Document` class: `id`, `embedding`, and `metadata`.
  - Indexing and Upserting: `.index(doc)` performs an atomic insert-or-replace.
  - Removing data: `.drop(doc)`.
- **5.2. Vector Search (ANN)**
  - Adding vectors via the `Document(embedding=...)` field.
  - Querying: `.search(vector, top_k=N)`.
  - **Use Case:** Building a RAG system by combining text and vector search.
  - **Helper:** The `rerank()` function for hybrid search results.
- **5.3. Full-Text & Fuzzy Search**
  - Full-Text Search (FTS): `.match(query, on=["field.path"])`.
  - Fuzzy Search: `.match(query, fuzziness=2)` for typo-tolerance.
- **5.4. Knowledge Graph**
  - Creating relationships: `.connect(source, target, label, metadata)`.
  - Single-hop traversal: `.neighbors(doc, label)`.
  - Multi-hop (BFS) traversal: `.walk(source, labels, depth, direction)`.

7 Real-Time Data

Chapter Outline:

- **6.1. Publish/Subscribe (`db.channel`)**
  - High-efficiency, multi-process messaging.
  - Publishing: `channel.publish(payload)`.
  - Subscribing: `with channel.subscribe() as listener: for msg in listener.listen(): ...`
- **6.2. Time-Indexed Logs (`db.log`)**
  - Creating structured, time-series logs: `logs.log(data)`.
  - Querying history: `.range(start_time, end_time)`.
  - **Feature:** Creating a live dashboard with `.live(window, period, aggregator)`.

8 Concurrency

Chapter Outline:

- **7.1. Inter-Process Locks (`db.lock`)**
  - Creating a critical section: `with db.lock("my_task", timeout=10): ...`
  - Guarantees: Fair (FIFO) and Deadlock-Proof (via TTL).
- **7.2. Atomic Operations on Data Structures**
  - Locking a specific manager: `with db.dict("config") as config: ...`
  - **Use Case:** Atomically getting and processing a *batch* of items from a queue.

9 Deployment & Access

Chapter Outline:

- **8.1. The REST API Server (`beaver serve`)**
  - Exposing your database as a FastAPI application.
  - Command: `beaver serve --database my.db --port 8000`
  - Accessing the interactive OpenAPI docs at `/docs`.
- **8.2. The Command-Line Client (`beaver client`)**
  - Interacting with your database from the terminal for admin and debugging.
  - Example: `beaver client --database my.db dict config get theme`
- **8.3. Docker Deployment**
  - Running the server in a container for stable deployment.
  - `docker run -p 8000:8000 -v $(pwd)/data:/app apiad/beaverdb`

Part II

The Developer Guide

10 Core Architecture & Design

Chapter Outline:

- **9.1. Guiding Principles (Developer Focus)**
  - A deeper dive into the "Why" from `design.md`.
  - Standard SQLite Compatibility as a "no-magic" rule.
  - Convention over Configuration.
- **9.2. The Manager Delegation Pattern**
  - How `BeaverDB` acts as a factory.
  - How managers (e.g., `DictManager`) are initialized with a reference to the core `BeaverDB` connection pool.
  - How all tables are prefixed with `beaver_` to avoid user-space conflicts.
- **9.3. Type-Safe Models (`beaver.Model`)**
  - Using the `model=...` parameter for automatic serialization and deserialization.
  - Inheriting from `beaver.Model` for a lightweight, Pydantic-compatible solution.

11 Concurrency Model

Chapter Outline:

- **10.1. Thread Safety (`threading.local`)**
  - How BeaverDB provides a unique `sqlite3.Connection` for *every thread*.
  - Why this is the key to preventing thread-related errors.
  - Enabling WAL (Write-Ahead Logging) for concurrent reads.
- **10.2. Inter-Process Locking (The Implementation)**
  - How `db.lock()` works under the hood.
  - The `beaver_lock_waiters` table as a fair (FIFO) queue.
  - The `expires_at` column as a deadlock-prevention (TTL) mechanism.
- **10.3. The Asynchronous `.as_async()` Pattern**
  - How the `Async...Manager` wrappers are implemented.
  - Using `asyncio.to_thread` to run blocking I/O without blocking the event loop.

12 Search Architecture

Chapter Outline:

- **11.1. Vector Search (ANN) Internals**
  - The "Hybrid Index System": Base Index and Delta Index.
  - **Crash-Safe Logging:** How additions and deletions are written to SQLite logs (`_beaver_ann_...` tables).
  - **Background Compaction:** The `compact()` process.
- **11.2. Text Search (FTS & Fuzzy) Internals**
  - **FTS:** How `beaver_fts_index` is a `fts5` virtual table.
  - **Fuzzy Search:** How BeaverDB builds a custom trigram index (`beaver_trigrams` table) and uses Levenshtein distance.

13 Future Roadmap & Contributing

Chapter Outline:

- **12.1. The Future of BeaverDB**
  - The `BeaverClient` as a drop-in network client.
  - Replacing `faiss` with a simpler, pure-`numpy` linear search.
- **12.2. How to Contribute**
  - (Standard contribution guidelines, linking to Makefile, etc.)