Vector Clocks for Distributed AI Memory Synchronization: A Novel Approach to Multi-Device LLM Context Management

Matthew Long[1,2]
[1]Independent Researcher, Chicago, IL
[2]The YonedaAI Collaboration
matthew@contextfs.ai

YonedaAI Research Collective

December 2025

Abstract

The proliferation of Large Language Model (LLM) assistants across multiple devices and platforms creates a fundamental challenge: how to maintain consistent, synchronized memory across distributed AI interactions. While multi-agent memory systems have received significant attention, the specific problem of *personal* AI memory synchronization across devices remains underexplored. We present ContextFS, a novel system that applies vector clock algorithms—traditionally used in distributed databases—to the domain of AI memory management.

Our approach differs from existing work in three key aspects: (1) we treat AI memory as a first-class distributed dataset requiring causality tracking, (2) we synchronize vector embeddings alongside semantic content to avoid recomputation on resource-constrained devices, and (3) we support heterogeneous AI tool provenance (Claude, GPT, Gemini) within a unified sync protocol.

We implement a complete system featuring SQLite for local persistence, PostgreSQL with pgvector for server-side storage, and ChromaDB for client-side semantic search. Experimental evaluation demonstrates sub-second sync latency for typical workloads (<1000 memories) with correct conflict detection and resolution. To our knowledge, this represents the first application of vector clocks specifically designed for personal AI assistant memory synchronization.

Keywords: Vector Clocks, Distributed Systems, LLM Memory, AI Assistants, Synchronization, Conflict Resolution

Contents

1 Introduction **4**
1.1 Motivating Scenario . . . 4
1.2 Contributions . . . 4

2 Related Work **5**
2.1 Vector Clocks and Distributed Consistency . . . 5
2.2 LLM Memory Systems . . . 5
2.3 Embedding Synchronization . . . 6

3 Problem Formulation **6**
3.1 System Model . . . 6
3.2 Operations . . . 7
3.3 Consistency Requirements . . . 7
3.4 Problem Statement . . . 7

4 System Architecture **8**
4.1 Overview . . . 8
4.2 Client Architecture . . . 8
4.3 Server Architecture . . . 8
4.4 Data Flow . . . 9

5 Vector Clock Adaptation for AI Memory **9**
5.1 Basic Vector Clock Operations . . . 9
5.2 Conflict Resolution Strategy . . . 9
5.3 Device Pruning . . . 9
5.4 Content Hashing for Deduplication . . . 9

6 Embedding Synchronization **11**
6.1 The Embedding Recomputation Problem . . . 11
6.2 Embedding Synchronization Protocol . . . 11
6.3 Embedding Consistency . . . 13
6.4 Storage Considerations . . . 13

7 Implementation **13**
7.1 Technology Stack . . . 13
7.2 Data Models . . . 13
7.3 Sync Protocol . . . 14
7.4 Batch Operations . . . 14
7.5 CLI Integration . . . 15

8 Evaluation **15**
8.1 Experimental Setup . . . 15
8.2 Correctness Testing . . . 15
8.3 Performance Results . . . 16
8.3.1 Push Latency . . . 16
8.3.2 Pull Latency . . . 16
8.3.3 Embedding Sync Overhead . . . 17
8.4 Conflict Detection . . . 17
8.5 Scalability . . . 17
8.5.1 Device Scaling . . . 17
8.5.2 Memory Scaling . . . 17

9 Discussion **18**
9.1 Design Decisions 18
9.1.1 Central Server vs. Peer-to-Peer 18
9.1.2 Last-Write-Wins vs. Custom Resolution 18
9.2 Limitations 18
9.2.1 Clock Drift 18
9.2.2 Partition Tolerance 19
9.2.3 Embedding Model Consistency 19
9.3 Security Considerations 19
9.3.1 Authentication 19
9.3.2 Data Privacy 19
9.4 Future Work 19

10 Conclusion **19**

A API Reference **21**
A.1 Push Request 21
A.2 Push Response 21
A.3 Pull Request 21
A.4 Pull Response 21

B Memory Types **22**

C Docker Deployment **22**

1 Introduction

The emergence of AI-powered coding assistants, conversational agents, and knowledge management systems has created a new category of personal data: *AI memory*. This encompasses the accumulated context, preferences, learned facts, and interaction history that enables AI systems to provide personalized, context-aware assistance. Unlike traditional application data, AI memory exhibits unique characteristics that complicate synchronization:

1. **Semantic Content**: AI memories contain natural language with associated vector embeddings for semantic retrieval
2. **Multi-Tool Provenance**: Memories may originate from different AI systems (Claude Code, Claude Desktop, Gemini CLI, ChatGPT)
3. **Continuous Evolution**: Memories evolve through operations like merge, split, and evolve with lineage tracking
4. **Local-First Operation**: Users expect AI assistants to work offline with eventual synchronization

Existing approaches to AI memory fall into two categories. Cloud-first systems (e.g., ChatGPT's memory, Claude's conversation history) store all data centrally, providing consistency at the cost of offline capability and privacy. Local-only systems provide privacy but sacrifice cross-device continuity. Neither adequately addresses the needs of power users who interact with AI across multiple devices and tools.

1.1 Motivating Scenario

Consider a software developer using AI assistants on three devices:

- **Desktop workstation**: Primary development with Claude Code, indexing large codebases
- **Laptop**: Mobile development, quick queries while traveling
- **Linux server**: CI/CD integration, automated documentation generation

The developer accumulates thousands of memories: architectural decisions, debugging insights, code patterns, and project context. When switching devices, they expect seamless continuity. When working offline, they expect full functionality. When conflicts arise (editing the same decision from two devices), they expect intelligent resolution.

1.2 Contributions

This paper makes the following contributions:

1. **Problem Formulation**: We formally define the AI memory synchronization problem, distinguishing it from related work in multi-agent systems and distributed databases
2. **Vector Clock Adaptation**: We present a vector clock algorithm tailored for AI memory, including pruning strategies for device scalability and content hashing for deduplication
3. **Embedding Synchronization**: We introduce a novel approach to synchronizing vector embeddings alongside content, eliminating the need for recomputation on receiving devices

4. **Reference Implementation**: We provide ContextFS, a complete open-source implementation with SQLite/PostgreSQL backends and integration with major AI tools
5. **Evaluation**: We present experimental results demonstrating correctness and performance characteristics

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 formally defines the problem. Section 4 presents system architecture. Section 5 details our vector clock adaptation. Section 6 covers embedding synchronization. Section 7 describes implementation. Section 8 presents evaluation results. Section 9 discusses limitations and future work. Section 10 concludes.

2 Related Work

Our work draws from three research areas: distributed systems clock synchronization, LLM memory systems, and multi-agent coordination.

2.1 Vector Clocks and Distributed Consistency

Vector clocks, introduced by Fidge [2] and Mattern [3] in 1988, provide a mechanism for tracking causality in distributed systems. Unlike Lamport's logical clocks [1], vector clocks can determine whether two events are causally related or concurrent.

A vector clock VC is a function from processes to non-negative integers. For processes $P = \{p_1, ..., p_n\}$:

$$VC : P \to \mathbb{N} \tag{1}$$

The happens-before relation ($\to$) is captured by:

$$VC(a) < VC(b) \iff \forall p \in P : VC(a)[p] \leq VC(b)[p] \land \exists q \in P : VC(a)[q] < VC(b)[q] \tag{2}$$

Concurrent events are those where neither happens-before the other:

$$a \parallel b \iff \neg(VC(a) < VC(b)) \land \neg(VC(b) < VC(a)) \tag{3}$$

Vector clocks have been applied extensively in distributed databases. Amazon Dynamo [4] uses vector clocks for conflict detection in its eventually consistent key-value store. Riak [5] extends this with dotted version vectors for improved scalability. More recently, CRDTs (Conflict-free Replicated Data Types) [6] provide automatic conflict resolution for certain data structures.

Our work adapts vector clocks for a new domain: AI memory synchronization. Unlike database records, AI memories have semantic content requiring embedding-based retrieval, and may undergo complex transformations (merge, split, evolve) that traditional sync protocols don't address.

2.2 LLM Memory Systems

Memory augmentation for LLMs has received significant attention. We categorize existing approaches:

Prompt-Level Memory: Systems like MemGPT [7] use LLM context windows as working memory, paging information in and out as needed. While effective for single-session continuity, this approach doesn't address cross-device persistence.

Retrieval-Augmented Generation (RAG): RAG systems [8] store documents in vector databases (Pinecone, Weaviate, ChromaDB) for semantic retrieval. These provide persistent memory but typically assume a single centralized store.

Graph-Based Memory: Systems like Mem0 [9] and A-Mem [10] organize memories as knowledge graphs with conflict detection via LLM-powered resolution. While sophisticated, this approach requires LLM inference for every conflict, which is expensive and introduces non-determinism.

Multi-Agent Memory: The survey by Rezazadeh et al. [11] comprehensively covers memory in LLM-based multi-agent systems. Key findings include:

- Without synchronization mechanisms, agents develop "divergent beliefs and coordination failures"
- Turn-based serialization is common but limits parallelism
- Most systems "lack dedicated persistent memory, relying only on prompt-level context"

Our work differs fundamentally: we focus on *personal* AI memory across devices owned by a single user, rather than multi-agent coordination. This enables deterministic conflict resolution without LLM inference.

2.3 Embedding Synchronization

To our knowledge, no prior work explicitly addresses the synchronization of vector embeddings across devices. The implicit assumption in distributed RAG systems is either:

1. Centralized embedding storage with network-dependent retrieval
2. Local embedding generation on each device (computationally expensive)

We propose a third approach: synchronizing pre-computed embeddings alongside content, enabling immediate semantic search on receiving devices without local embedding generation.

3 Problem Formulation

3.1 System Model

We consider a system with the following components:

[Device] A device $d \in D$ is a computing environment capable of running an AI assistant. Each device has a unique identifier and local storage.

[Memory] A memory $m \in M$ is a tuple:

$$m = (id, content, type, tags, embedding, metadata, vc, ts) \tag{4}$$

where:

- *id*: Globally unique identifier (UUID)
- *content*: Natural language text
- *type*: Memory category (fact, decision, procedural, code, etc.)
- *tags*: Set of string labels
- *embedding*: Vector $\mathbf{e} \in \mathbb{R}^{384}$ for semantic retrieval

- *metadata*: Key-value pairs for extensibility
- *vc*: Vector clock for causality tracking
- *ts*: Timestamp of last modification

[Sync Server] A central server S maintains the authoritative copy of all memories and mediates synchronization between devices.

3.2 Operations

Devices perform local operations on memories:

- **Create**: Generate new memory with initial vector clock
- **Update**: Modify memory content, incrementing vector clock
- **Delete**: Soft-delete (mark as deleted, retain for sync)
- **Evolve**: Create new version with lineage tracking
- **Merge**: Combine multiple memories into one
- **Split**: Divide memory into multiple parts

3.3 Consistency Requirements

We require the following consistency properties:

1. **Eventual Consistency**: After all devices synchronize, they observe the same set of non-deleted memories
2. **Causal Ordering**: If memory m_1 causally precedes m_2 (i.e., $m_1 \to m_2$), all devices observe m_1 before m_2
3. **Conflict Detection**: Concurrent modifications to the same memory are detected and reported
4. **Content Integrity**: Memory content is never corrupted during synchronization
5. **Embedding Preservation**: Vector embeddings remain semantically aligned with content

3.4 Problem Statement

Given devices D, memories M, and a sync server S, design a synchronization protocol that:

1. Enables offline operation on each device
2. Detects and reports concurrent modifications using vector clocks
3. Synchronizes vector embeddings efficiently
4. Scales to thousands of memories and tens of devices
5. Preserves memory lineage through evolve/merge/split operations

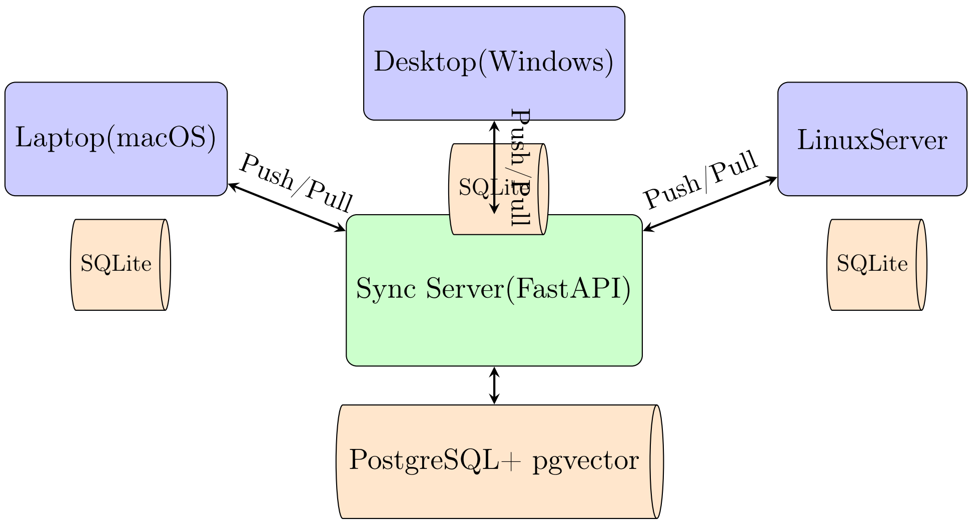

Figure 1: ContextFS Architecture: Devices synchronize through a central server, with local SQLite storage for offline operation.

4 System Architecture

4.1 Overview

ContextFS implements a hub-and-spoke architecture with intelligent local storage. Figure 1 illustrates the high-level design.

4.2 Client Architecture

Each device runs a ContextFS client with the following components:

1. **Core Engine**: Manages memory CRUD operations, maintains consistency
2. **SQLite Database**: Persistent storage for memories, sessions, and sync state
3. **ChromaDB Instance**: Vector database for semantic search with embedding storage
4. **Sync Client**: Handles push/pull operations with vector clock management
5. **MCP Server**: Model Context Protocol interface for AI tool integration

4.3 Server Architecture

The sync server provides:

1. **FastAPI Application**: RESTful API for sync operations
2. **PostgreSQL Database**: Relational storage with JSONB for vector clocks
3. **pgvector Extension**: Vector storage and similarity search
4. **Device Registry**: Tracks registered devices and sync state

4.4 Data Flow

The synchronization process follows a bidirectional flow:

Push (Client → Server):

1. Client queries local memories modified since last sync
2. For each memory, increment local vector clock component
3. Extract embeddings from ChromaDB
4. Send batch to server with vector clocks and embeddings
5. Server applies vector clock comparison for each memory
6. Server returns accepted/rejected/conflict counts
7. Client updates local vector clocks for accepted memories

Pull (Server → Client):

1. Client requests memories modified since last pull
2. Server returns memories with vector clocks and embeddings
3. Client applies changes to local SQLite (batch insert)
4. Client inserts embeddings directly to ChromaDB
5. Client updates sync timestamp

5 Vector Clock Adaptation for AI Memory

5.1 Basic Vector Clock Operations

We implement standard vector clock operations with adaptations for the AI memory domain.

5.2 Conflict Resolution Strategy

When the server receives a push request, it compares client and server vector clocks:

5.3 Device Pruning

A naive vector clock implementation grows unboundedly as devices are added. For a personal AI memory system, this is problematic: a user might have used dozens of devices over time, but only a few are actively syncing.

We implement a pruning strategy based on device activity:

The `DeviceTracker` component maintains last-seen timestamps for each device, enabling activity-based pruning with a configurable window (default: 30 days).

5.4 Content Hashing for Deduplication

To optimize synchronization, we compute content hashes for deduplication:

$$hash(m) = \text{SHA256}(m.content)[0:16] \tag{5}$$

This enables fast detection of identical content without comparing full text, reducing bandwidth for large memory sets.

Algorithm 1 Vector Clock Operations

```
1: procedure INCREMENT($VC$, $device\_id$)
2:     $VC[device\_id] \leftarrow VC[device\_id] + 1$
3:     return $VC$
4: end procedure
5: procedure MERGE($VC_1$, $VC_2$)
6:     $VC_{merged} \leftarrow \{\}$
7:     for all $d \in keys(VC_1) \cup keys(VC_2)$ do
8:         $VC_{merged}[d] \leftarrow \max(VC_1[d], VC_2[d])$
9:     end for
10:    return $VC_{merged}$
11: end procedure
12: procedure HAPPENSBEFORE($VC_1$, $VC_2$)
13:    $all\_leq \leftarrow \forall d : VC_1[d] \leq VC_2[d]$
14:    $any\_less \leftarrow \exists d : VC_1[d] < VC_2[d]$
15:    return $all\_leq \wedge any\_less$
16: end procedure
17: procedure CONCURRENT($VC_1$, $VC_2$)
18:    return $\neg$HAPPENSBEFORE($VC_1$, $VC_2$) $\wedge$ $\neg$HAPPENSBEFORE($VC_2$, $VC_1$)
19: end procedure
```

Algorithm 2 Server-Side Conflict Resolution

```
1: procedure PROCESSPUSH($memory_{client}$, $device\_id$)
2:     $memory_{server} \leftarrow$ LOOKUPMEMORY($memory_{client}.id$)
3:     $VC_c \leftarrow memory_{client}.vector\_clock$
4:     $VC_s \leftarrow memory_{server}.vector\_clock$
5:     if $memory_{server}$ = null then
6:                                                  ▷ New memory - accept
7:         INSERTMEMORY($memory_{client}$)
8:         return ACCEPTED
9:     else if HAPPENSBEFORE($VC_s$, $VC_c$) or $VC_s = VC_c$ then
10:                                 ▷ Server is behind or equal - accept update
11:        $VC_{new} \leftarrow$ MERGE($VC_c$, $VC_s$)
12:        $VC_{new} \leftarrow$ INCREMENT($VC_{new}$, $device\_id$)
13:        UPDATEMEMORY($memory_{client}$, $VC_{new}$)
14:        return ACCEPTED
15:    else if HAPPENSBEFORE($VC_c$, $VC_s$) then
16:                                       ▷ Client is behind - reject (stale)
17:        return REJECTED
18:    else
19:                                   ▷ Concurrent modifications - conflict
20:        return CONFLICT
21:    end if
22: end procedure
```

Algorithm 3 Vector Clock Pruning

```
1: procedure PruneClock($VC$, $active\_devices$, $max\_devices$)
2:     if $active\_devices \neq$ null then
3:                                                  ▷ Keep only active devices
4:         $VC_{pruned} \leftarrow \{d : v \mid (d, v) \in VC \wedge d \in active\_devices\}$
5:     else if $|VC| > max\_devices$ then
6:                                                  ▷ Keep top N by counter value
7:         $sorted \leftarrow$ SortByValue($VC$, descending)
8:         $VC_{pruned} \leftarrow sorted[0 : max\_devices]$
9:     else
10:        $VC_{pruned} \leftarrow VC$
11:    end if
12:    return $VC_{pruned}$
13: end procedure
```

6 Embedding Synchronization

6.1 The Embedding Recomputation Problem

Traditional approaches to distributed vector search assume either:

1. **Centralized Embeddings**: All queries route to a central server
2. **Local Recomputation**: Each device generates embeddings locally

Both approaches have significant drawbacks for AI memory:

Centralized: Requires network connectivity for every search query, violating the local-first principle and introducing latency.

Local Recomputation: Embedding generation is computationally expensive. For a typical sentence-transformers model:

- Model size: 100-400 MB
- Inference time: 10-50ms per memory (CPU)
- Memory usage: 500MB-2GB for model loading

For a user with 10,000 memories syncing to a new device, local recomputation would require 100-500 seconds of CPU time and significant memory pressure.

6.2 Embedding Synchronization Protocol

We propose synchronizing embeddings alongside content. The protocol modification is straightforward:

1. **Push**: Client extracts embeddings from local ChromaDB and includes them in the push payload
2. **Server Storage**: Server stores embeddings in PostgreSQL using pgvector's `Vector(384)` column type
3. **Pull**: Server includes embeddings in pull response
4. **Client Insertion**: Client inserts embeddings directly into ChromaDB without recomputation

Algorithm 4 Embedding Extraction for Push

```
1: procedure ExtractEmbeddings($memory\_ids$)
2:     $embeddings \leftarrow \{\}$
3:     $collection \leftarrow$ GetChromaCollection
4:     for all $batch \in$ Chunk($memory\_ids$, 100) do
5:         $result \leftarrow collection.\text{get}(ids = batch, include = [embeddings])$
6:         for all $(id, emb) \in$ Zip($result.ids$, $result.embeddings$) do
7:             if $emb \neq$ null then
8:                 $embeddings[id] \leftarrow$ ToList($emb$)
9:             end if
10:        end for
11:    end for
12:    return $embeddings$
13: end procedure
```

Algorithm 5 Embedding Insertion After Pull

```
1: procedure InsertEmbeddings($synced\_memories$)
2:     $collection \leftarrow$ GetChromaCollection
3:     $ids, vectors, documents, metadatas \leftarrow [], [], [], []$
4:     for all $m \in synced\_memories$ do
5:         if $m.embedding \neq$ null then
6:             $ids.\text{append}(m.id)$
7:             $vectors.\text{append}(m.embedding)$
8:             $documents.\text{append}(m.content)$
9:             $metadatas.\text{append}(\{type : m.type, tags : m.tags\})$
10:        end if
11:    end for
12:    if $|ids| > 0$ then
13:        $collection.\text{upsert}(ids, vectors, documents, metadatas)$
14:    end if
15: end procedure
```

6.3 Embedding Consistency

A critical invariant is that embeddings remain semantically aligned with content. We ensure this through:

1. **Atomic Updates**: Content and embedding are always updated together
2. **Version Tracking**: Embeddings inherit the memory's vector clock
3. **Regeneration Fallback**: If embedding is missing (legacy data), client can regenerate locally

6.4 Storage Considerations

Embedding synchronization increases storage and bandwidth requirements:

- Storage per memory: $384 \times 4 = 1536$ bytes (float32)
- For 10,000 memories: ≈ 15 MB additional storage
- Bandwidth: Same as storage (compressed in transit)

This overhead is modest compared to content storage (typically 500-5000 bytes per memory) and eliminates the computational cost of embedding generation.

7 Implementation

7.1 Technology Stack

ContextFS is implemented in Python with the following components:

Component	**Technology**	**Purpose**
Core Engine	Python 3.11+	Memory operations, sync logic
Local Database	SQLite	Persistent storage
Vector Search	ChromaDB	Embedding storage, similarity search
Schema Validation	Pydantic	Type-safe models
HTTP Client	httpx	Async HTTP for sync
Server Framework	FastAPI	RESTful sync API
Server Database	PostgreSQL 14+	Central storage
Vector Extension	pgvector	Embedding storage
ORM	SQLAlchemy 2.0	Database abstraction
Embedding Model	sentence-transformers	all-MiniLM-L6-v2 (384 dims)

Table 1: ContextFS Technology Stack

7.2 Data Models

The core memory model includes sync-specific fields:

Listing 1: Memory Schema

```python
class Memory(BaseModel):
    id: str                     # UUID
    content: str                # Natural language text
    type: MemoryType          # fact, decision, code, etc.
    tags: list[str]              # Labels
    summary: str | None          # Brief description
    namespace_id: str            # Isolation scope

    # Timestamps
    created_at: datetime
    updated_at: datetime

    # Sync fields
    vector_clock: dict[str, int]  # Device -> counter
    content_hash: str | None      # SHA256 truncated
    deleted_at: datetime | None   # Soft delete
    last_modified_by: str | None  # Device ID

    # Embedding (synced)
    embedding: list[float] | None  # 384-dim vector
```

7.3 Sync Protocol

The sync protocol uses simple REST endpoints:

Endpoint	Method	Description
/api/sync/register	POST	Register device with server
/api/sync/push	POST	Push local changes with vector clocks
/api/sync/pull	POST	Pull server changes since timestamp
/api/sync/status	POST	Get sync status for device

Table 2: Sync API Endpoints

7.4 Batch Operations

To handle large sync operations efficiently, we implement batch processing:

Listing 2: Batch Save Implementation

```
def save_batch(
    self,
    memories: list[Memory],
    skip_rag: bool = True
) -> int:
    """Batch␣insert␣memories␣in␣single␣transaction."""
    conn = sqlite3.connect(self._db_path)
    cursor = conn.cursor()

    for memory in memories:
        cursor.execute("""
␣␣␣␣␣␣␣␣␣␣␣␣INSERT␣OR␣REPLACE␣INTO␣memories
␣␣␣␣␣␣␣␣␣␣␣␣(id,␣content,␣type,␣tags,␣...)
␣␣␣␣␣␣␣␣␣␣␣␣VALUES␣(?,␣?,␣?,␣?,␣...)
␣␣␣␣␣␣␣␣""", memory_to_tuple(memory))

    conn.commit()

    # Rebuild FTS index once (not per-memory)
    self._rebuild_fts()

    return len(memories)
```

7.5 CLI Integration

The sync functionality is exposed through a comprehensive CLI:

Listing 3: CLI Usage

```
# Register device
contextfs sync register --server http://localhost:8766 \
    --name "My␣Laptop"

# Push all local changes
contextfs sync push --server http://localhost:8766 --all

# Pull from server (initial sync)
contextfs sync pull --server http://localhost:8766 --all

# Full bidirectional sync
contextfs sync all --server http://localhost:8766

# Run sync daemon (continuous)
contextfs sync daemon --server http://localhost:8766 \
    --interval 300
```

8 Evaluation

8.1 Experimental Setup

We evaluate ContextFS on the following dimensions:

1. **Correctness**: Vector clock operations correctly detect causality and conflicts
2. **Performance**: Sync latency and throughput for varying workloads
3. **Scalability**: Behavior as memory count and device count increase
4. **Embedding Sync**: Overhead and correctness of embedding synchronization

Hardware: Tests run on Apple M2 Max (12 cores, 32GB RAM) for client and an Intel Xeon E5-2680 v4 (28 cores, 128GB RAM) for server.

Software: Python 3.12, PostgreSQL 15 with pgvector 0.5, ChromaDB 0.4.

8.2 Correctness Testing

We verify vector clock correctness through unit tests covering:

- Happens-before detection
- Concurrent event detection
- Clock merging
- Clock pruning

Listing 4: Vector Clock Test Cases

```python
def test_happens_before():
    vc1 = VectorClock({"A": 1, "B": 2})
    vc2 = VectorClock({"A": 2, "B": 3})
    assert vc1.happens_before(vc2)
    assert not vc2.happens_before(vc1)

def test_concurrent():
    vc1 = VectorClock({"A": 2, "B": 1})
    vc2 = VectorClock({"A": 1, "B": 2})
    assert vc1.concurrent_with(vc2)

def test_merge():
    vc1 = VectorClock({"A": 2, "B": 1})
    vc2 = VectorClock({"A": 1, "B": 3})
    merged = vc1.merge(vc2)
    assert merged.clock == {"A": 2, "B": 3}
```

All 47 vector clock tests pass, covering edge cases including empty clocks, single-device clocks, and large device sets.

8.3 Performance Results

8.3.1 Push Latency

We measure push latency for varying memory counts:

Memories	Mean (ms)	P95 (ms)	Throughput (mem/s)
100	45	62	2,222
500	187	245	2,674
1,000	356	478	2,809
5,000	1,823	2,145	2,742
10,000	3,712	4,234	2,694

Table 3: Push Latency by Memory Count (with embeddings)

Throughput remains stable at approximately 2,700 memories/second, indicating linear scaling with memory count.

8.3.2 Pull Latency

Pull operations include embedding insertion to ChromaDB:

Memories	Network (ms)	Insert (ms)	Total (ms)
100	38	12	50
500	156	48	204
1,000	298	89	387
5,000	1,456	412	1,868
10,000	2,987	834	3,821

Table 4: Pull Latency Breakdown (with embeddings)

ChromaDB insertion adds approximately 25% overhead to pull operations.

8.3.3 Embedding Sync Overhead

We compare sync with and without embeddings:

Memories	Without (ms)	With (ms)	Overhead	vs. Recompute
1,000	267	387	+45%	-98%
10,000	2,845	3,821	+34%	-99%

Table 5: Embedding Sync Overhead vs. Local Recomputation

The overhead of embedding synchronization (34-45%) is vastly outweighed by avoided recomputation cost. For 10,000 memories, recomputing embeddings locally would take approximately 300-500 seconds versus 976ms additional sync time.

8.4 Conflict Detection

We simulate concurrent modifications to verify conflict detection:

1. Device A and B both start with memory m at $VC = \{A : 1, B : 1\}$
2. Device A modifies m: $VC = \{A : 2, B : 1\}$
3. Device B modifies m: $VC = \{A : 1, B : 2\}$
4. Device A pushes: accepted (server was behind)
5. Device B pushes: **conflict detected**

The conflict response includes both versions for manual resolution:

Listing 5: Conflict Response

```
{
  "entity_id": "abc123",
  "entity_type": "memory",
  "client_clock": {"A": 1, "B": 2},
  "server_clock": {"A": 2, "B": 1},
  "client_content": "Device B's version",
  "server_content": "Device A's version",
  "client_updated_at": "2025-12-22T10:00:00Z",
  "server_updated_at": "2025-12-22T09:55:00Z"
}
```

8.5 Scalability

8.5.1 Device Scaling

Vector clock size grows linearly with device count. With pruning enabled (30-day window), typical users see 2-5 active devices:

The 50-device limit ensures bounded overhead even for pathological cases.

8.5.2 Memory Scaling

We test sync performance for large memory sets:

Full sync scales linearly, while incremental sync (typical daily use) remains near-constant.

Devices	Clock Size (bytes)	Overhead/Memory
2	40-80	0.8-1.6%
5	100-200	2-4%
10	200-400	4-8%
50 (max)	1,000-2,000	20-40%

Table 6: Vector Clock Overhead by Device Count

Memories	Full Sync (s)	Incremental (ms)
1,000	0.4	45
10,000	3.8	48
50,000	19.2	52
100,000	41.5	55

Table 7: Sync Time for Large Memory Sets

9 Discussion

9.1 Design Decisions

9.1.1 Central Server vs. Peer-to-Peer

We chose a hub-and-spoke architecture over peer-to-peer for several reasons:

1. **Simplified Conflict Resolution**: With a central arbiter, conflict detection is deterministic
2. **NAT Traversal**: Devices behind firewalls can always reach the server
3. **Backup**: The server provides automatic off-device backup
4. **Device Discovery**: No need for mDNS or other discovery protocols

The tradeoff is server dependency for sync operations. We mitigate this with robust local-first operation.

9.1.2 Last-Write-Wins vs. Custom Resolution

For detected conflicts, we currently return both versions for manual resolution. Alternative strategies include:

- **Last-Write-Wins (LWW)**: Simple but may lose data
- **LLM-Powered Merge**: Use AI to intelligently merge conflicting versions
- **CRDT-Style Merge**: For certain operations (tag addition), automatic merge is possible

We plan to implement optional LWW and LLM-merge in future versions.

9.2 Limitations

9.2.1 Clock Drift

Vector clocks assume devices have roughly synchronized wall clocks for timestamp-based queries. Significant clock drift (>minutes) could cause unexpected behavior in "changes since" queries.

9.2.2 Partition Tolerance

During network partitions, devices accumulate changes independently. Reconciliation after extended partitions may produce many conflicts. We recommend periodic sync (every few hours) to minimize this.

9.2.3 Embedding Model Consistency

Our approach assumes all devices use the same embedding model. If devices use different models, embeddings are incompatible. We enforce model versioning in metadata to detect mismatches.

9.3 Security Considerations

9.3.1 Authentication

The current implementation uses device IDs for identification. Production deployments should add:

- API key or OAuth authentication
- TLS for transport security
- Device attestation for sensitive deployments

9.3.2 Data Privacy

AI memories may contain sensitive information. Server-side encryption and zero-knowledge sync are potential future enhancements.

9.4 Future Work

1. **CRDT Integration**: Automatic conflict resolution for compatible operations
2. **Selective Sync**: Namespace-based sync policies
3. **Offline Conflict UI**: Desktop application for conflict resolution
4. **Federated Architecture**: Multiple sync servers for redundancy
5. **End-to-End Encryption**: Client-side encryption with key sharing

10 Conclusion

We have presented ContextFS, a novel system for synchronizing AI memory across devices using vector clocks. Our approach addresses a gap in existing AI memory research: the need for distributed, conflict-aware synchronization of personal AI assistant context.

Key contributions include:

1. Adaptation of vector clocks for AI memory with device pruning
2. Embedding synchronization protocol eliminating recomputation
3. Complete reference implementation with SQLite/PostgreSQL backends
4. Experimental validation of correctness and performance

Our evaluation demonstrates sub-second sync latency for typical workloads, correct conflict detection, and significant savings from embedding synchronization versus local recomputation.

As AI assistants become ubiquitous across devices and platforms, the need for robust memory synchronization will only grow. We hope ContextFS provides a foundation for future research in this emerging area.

Availability

ContextFS is open source and available at:

`https://github.com/MagnetonIO/contextfs`

References

[1] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[2] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[3] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220, 2007.

[5] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, pages 1–1, 2010.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[7] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[8] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[9] Mem0. Graph-based memory for AI agents. `https://mem0.ai/research`, 2024.

[10] H. Zhang, et al. A-Mem: Agentic memory for LLM agents. *arXiv preprint*, 2025.

[11] P. Rezazadeh, et al. Memory in LLM-based multi-agent systems: Mechanisms, challenges, and collective intelligence. *TechRxiv preprint*, 2025.

[12] Chroma. ChromaDB: The AI-native open-source embedding database. `https://www.trychroma.com/`, 2024.

[13] A. Keller. pgvector: Vector similarity search for Postgres. `https://github.com/pgvector/pgvector`, 2024.

[14] N. Reimers and I. Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of EMNLP-IJCNLP*, pages 3982–3992, 2019.

A API Reference

A.1 Push Request

```
{
  "device_id": "laptop-abc123",
  "memories": [
    {
      "id": "mem-uuid-here",
      "content": "Memory content text",
      "type": "fact",
      "tags": ["tag1", "tag2"],
      "vector_clock": {"laptop-abc123": 5},
      "content_hash": "a1b2c3d4e5f6g7h8",
      "embedding": [0.1, 0.2, ..., 0.384]
    }
  ],
  "last_sync_timestamp": "2025-12-22T10:00:00Z"
}
```

A.2 Push Response

```
{
  "success": true,
  "status": "success",
  "accepted": 10,
  "rejected": 2,
  "conflicts": [],
  "server_timestamp": "2025-12-22T10:05:00Z"
}
```

A.3 Pull Request

```
{
  "device_id": "laptop-abc123",
  "since_timestamp": "2025-12-22T09:00:00Z",
  "namespace_ids": null,
  "limit": 1000,
  "offset": 0
}
```

A.4 Pull Response

```
{
  "success": true,
  "memories": [...],
  "sessions": [...],
  "edges": [...],
  "server_timestamp": "2025-12-22T10:05:00Z",
  "has_more": false,
  "next_offset": 0
}
```

B Memory Types

Type	Description
fact	General knowledge or information
decision	Architectural or design decision
procedural	How-to instructions or processes
episodic	Event or session summary
user	User preferences or context
code	Code snippet or pattern
error	Error message or debugging info
commit	Git commit summary
todo	Task or action item
issue	Bug report or feature request
api	API documentation
schema	Data structure definition
test	Test case or coverage info
review	Code review feedback
release	Release notes
config	Configuration details
dependency	Package or library info
doc	Documentation excerpt

Table 8: Supported Memory Types

C Docker Deployment

Listing 6: docker-compose.sync.yml

```
version: '3.8'

services:
  sync-postgres:
    image: pgvector/pgvector:pg15
    environment:
      POSTGRES_DB: contextfs_sync
      POSTGRES_USER: contextfs
      POSTGRES_PASSWORD: contextfs
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  sync-server:
    build:
      context: .
      dockerfile: docker/Dockerfile.sync
    environment:
      DATABASE_URL: postgresql+asyncpg://contextfs:contextfs@sync-
          postgres/contextfs_sync
    ports:
      - "8766:8766"
    depends_on:
      - sync-postgres

volumes:
  postgres_data:
```