Contents

ADR-006: Global Memory with DynamoDB Global Tables

Status

Accepted

Context

Cato is a **single global brain** serving all users worldwide. This creates unique memory requirements:

1. **Global consistency**: A fact learned from a user in Tokyo must be available to a user in New York
2. **Low latency**: Memory retrieval must not add significant latency to user interactions
3. **High scale**: 10MM+ users generating billions of memory entries
4. **Multi-type**: Different memory types (semantic, episodic, working) have different access patterns

Memory Types

Type	Purpose	Access Pattern	Consistency Need
Semantic	Facts, concepts, relationships	Read-heavy, rare writes	Strong

Type	Purpose	Access Pattern	Consistency Need
Episodic	User interactions, experiences	Write-heavy, recent reads	Eventual
Working	Current context, active goals	Read/write balanced	Strong
Knowledge Graph	Concept relationships	Graph traversal	Eventual

Decision

Implement a **polyglot persistence** architecture using AWS managed services:

1. DynamoDB Global Tables (Semantic + Working Memory)

- **Purpose**: Core fact storage with global replication
- **Consistency**: MRSC for semantic, MREC for high-volume updates
- **Access**: DAX for sub-millisecond reads

2. OpenSearch Serverless (Episodic Memory)

- **Purpose**: Vector similarity search for experience retrieval
- **Scale**: Billions of embeddings
- **Access**: k-NN search with filters

3. Neptune (Knowledge Graph)

- **Purpose**: Concept relationships and reasoning
- **Access**: Gremlin/SPARQL queries
- **Use case**: "What concepts are related to X?"

4. ElastiCache Redis (Working Memory Cache)

- **Purpose**: Active session context, hot data
- **TTL**: 24-hour decay for working memory
- **Access**: Sub-millisecond reads

Architecture

```
                         CATO GLOBAL MEMORY



                    SEMANTIC MEMORY (DynamoDB Global Tables)
     Facts: Subject-Predicate-Object triples
     Concepts: Domain knowledge with confidence
     Sources: Attribution for each fact
```

```
   Versioning: Optimistic locking for updates

Regions: us-east-1, eu-west-1, ap-northeast-1
Consistency: MRSC for writes, MREC for reads
Access: DAX cluster for sub-ms reads



                  EPISODIC MEMORY (OpenSearch Serverless)
   Interactions: User queries and responses
   Embeddings: 768-dim vectors for similarity
   Metadata: Timestamp, user, domain, satisfaction
   TTL: 90-day retention for compliance

Scale: 1B+ vectors
Search: k-NN with 10ms p99 latency



                  KNOWLEDGE GRAPH (Neptune)
   Concepts: Nodes with properties
   Relationships: Typed edges (is-a, part-of, causes, etc.)
   Weights: Relationship strength
   Domains: Subgraph per knowledge domain

Query: Gremlin for traversal
Use: "Find related concepts within 3 hops"



                  WORKING MEMORY (ElastiCache Redis)
   Sessions: Active conversation context
   Goals: Current autonomous objectives
   Attention: What Cato is "thinking about"
   Mood: Current meta-cognitive state

TTL: 24-hour decay
Access: Sub-ms reads via cluster mode
```

DynamoDB Schema

Semantic Memory Table

```
Table: cato-semantic-memory
```

```
Primary Key:
  pk (Partition Key): "FACT#{domain}" or "CONCEPT#{id}"
  sk (Sort Key): "{subject}#{predicate}#{object}" or "{timestamp}"

GSI1 (Subject Index):
  gsi1pk: "SUBJECT#{subject}"
  gsi1sk: "{predicate}#{object}"

GSI2 (Domain Index):
  gsi2pk: "DOMAIN#{domain}"
  gsi2sk: "{confidence}#{timestamp}"

Attributes:
  - fact_id: UUID
  - subject: string
  - predicate: string
  - object: string
  - domain: string
  - confidence: number (0-1)
  - sources: string[] (URLs, references)
  - created_at: ISO timestamp
  - updated_at: ISO timestamp
  - version: number (for optimistic locking)
```

Working Memory Table

```
Table: cato-working-memory

Primary Key:
  pk: "SESSION#{session_id}" or "GOAL#{goal_id}" or "ATTENTION"
  sk: "{timestamp}" or "{priority}"

TTL: expires_at (24 hours from creation)

Attributes:
  - context: JSON (conversation history)
  - goals: string[] (current objectives)
  - attention_focus: string (current topic)
  - meta_state: enum (CONFUSED, CONFIDENT, BORED, STAGNANT)
  - created_at: ISO timestamp
```

Implementation

TypeScript Memory Service

```typescript
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, QueryCommand, PutCommand, UpdateCommand } from '@aws-sdk/lib-d
import { Client as OpenSearchClient } from '@opensearch-project/opensearch';
```

```typescript
export interface SemanticFact {
  factId: string;
  subject: string;
  predicate: string;
  object: string;
  domain: string;
  confidence: number;
  sources: string[];
  createdAt: Date;
  updatedAt: Date;
  version: number;
}

export interface EpisodicMemory {
  interactionId: string;
  userId: string;
  query: string;
  response: string;
  embedding: number[];
  domain: string;
  satisfaction: number;
  timestamp: Date;
}

export class GlobalMemoryService {
  private readonly docClient: DynamoDBDocumentClient;
  private readonly opensearch: OpenSearchClient;
  private readonly semanticTable: string;
  private readonly workingTable: string;
  private readonly episodicIndex: string;

  constructor(config: {
    semanticTable: string;
    workingTable: string;
    opensearchEndpoint: string;
    episodicIndex: string;
    region: string;
  }) {
    const dynamoClient = new DynamoDBClient({ region: config.region });
    this.docClient = DynamoDBDocumentClient.from(dynamoClient);
    this.opensearch = new OpenSearchClient({
      node: config.opensearchEndpoint
    });
    this.semanticTable = config.semanticTable;
    this.workingTable = config.workingTable;
    this.episodicIndex = config.episodicIndex;
  }
```

```
// ============================================================================
// Semantic Memory
// ============================================================================

async storeFact(fact: Omit<SemanticFact, 'factId' | 'createdAt' | 'updatedAt' | 'version'>):
  const factId = crypto.randomUUID();
  const now = new Date();

  await this.docClient.send(new PutCommand({
    TableName: this.semanticTable,
    Item: {
      pk: `FACT#${fact.domain}`,
      sk: `${fact.subject}#${fact.predicate}#${fact.object}`,
      factId,
      ...fact,
      createdAt: now.toISOString(),
      updatedAt: now.toISOString(),
      version: 1,
      gsi1pk: `SUBJECT#${fact.subject}`,
      gsi1sk: `${fact.predicate}#${fact.object}`,
      gsi2pk: `DOMAIN#${fact.domain}`,
      gsi2sk: `${fact.confidence}#${now.toISOString()}`
    },
    ConditionExpression: 'attribute_not_exists(pk)'
  }));

  return factId;
}

async getFactsByDomain(domain: string, limit: number = 100): Promise<SemanticFact[]> {
  const response = await this.docClient.send(new QueryCommand({
    TableName: this.semanticTable,
    KeyConditionExpression: 'pk = :pk',
    ExpressionAttributeValues: {
      ':pk': `FACT#${domain}`
    },
    Limit: limit
  }));

  return (response.Items || []).map(this.itemToFact);
}

async getFactsAboutSubject(subject: string, limit: number = 50): Promise<SemanticFact[]> {
  const response = await this.docClient.send(new QueryCommand({
    TableName: this.semanticTable,
    IndexName: 'gsi1',
    KeyConditionExpression: 'gsi1pk = :pk',
    ExpressionAttributeValues: {
```

```
      ':pk': `SUBJECT#${subject}`
    },
    Limit: limit
  }));

  return (response.Items || []).map(this.itemToFact);
}

async updateFactConfidence(
  domain: string,
  sk: string,
  newConfidence: number,
  source: string
): Promise<void> {
  await this.docClient.send(new UpdateCommand({
    TableName: this.semanticTable,
    Key: { pk: `FACT#${domain}`, sk },
    UpdateExpression: 'SET confidence = :conf, updatedAt = :now, version = version + :inc, so
    ExpressionAttributeValues: {
      ':conf': newConfidence,
      ':now': new Date().toISOString(),
      ':inc': 1,
      ':src': [source]
    }
  }));
}

// ============================================================================
// Episodic Memory
// ============================================================================

async storeInteraction(memory: Omit<EpisodicMemory, 'interactionId'>): Promise<string> {
  const interactionId = crypto.randomUUID();

  await this.opensearch.index({
    index: this.episodicIndex,
    id: interactionId,
    body: {
      ...memory,
      interactionId,
      timestamp: memory.timestamp.toISOString()
    }
  });

  return interactionId;
}

async searchSimilarInteractions(
```

```typescript
  embedding: number[],
  filters: { domain?: string; userId?: string },
  limit: number = 10
): Promise<EpisodicMemory[]> {
  const must: any[] = [];

  if (filters.domain) {
    must.push({ term: { domain: filters.domain } });
  }
  if (filters.userId) {
    must.push({ term: { userId: filters.userId } });
  }

  const response = await this.opensearch.search({
    index: this.episodicIndex,
    body: {
      size: limit,
      query: {
        bool: {
          must,
          should: [
            {
              knn: {
                embedding: {
                  vector: embedding,
                  k: limit
                }
              }
            }
          ]
        }
      }
    }
  });

  return response.body.hits.hits.map((hit: any) => ({
    ...hit._source,
    timestamp: new Date(hit._source.timestamp)
  }));
}

// ============================================================================
// Working Memory
// ============================================================================

async getSessionContext(sessionId: string): Promise<any | null> {
  const response = await this.docClient.send(new QueryCommand({
    TableName: this.workingTable,
```

```
      KeyConditionExpression: 'pk = :pk',
      ExpressionAttributeValues: {
        ':pk': `SESSION#${sessionId}`
      },
      ScanIndexForward: false,
      Limit: 1
    }));

    return response.Items?.[0]?.context || null;
  }

  async updateSessionContext(sessionId: string, context: any): Promise<void> {
    const now = new Date();
    const expiresAt = new Date(now.getTime() + 24 * 60 * 60 * 1000); // 24 hours

    await this.docClient.send(new PutCommand({
      TableName: this.workingTable,
      Item: {
        pk: `SESSION#${sessionId}`,
        sk: now.toISOString(),
        context,
        createdAt: now.toISOString(),
        expiresAt: Math.floor(expiresAt.getTime() / 1000) // TTL
      }
    }));
  }

  private itemToFact(item: any): SemanticFact {
    return {
      factId: item.factId,
      subject: item.subject,
      predicate: item.predicate,
      object: item.object,
      domain: item.domain,
      confidence: item.confidence,
      sources: item.sources || [],
      createdAt: new Date(item.createdAt),
      updatedAt: new Date(item.updatedAt),
      version: item.version
    };
  }
}
```

Consequences

Positive

- **Global availability**: DynamoDB Global Tables replicate across regions

- **Specialized storage**: Each memory type uses optimal database
- **Scalability**: All components auto-scale to billions of entries
- **Low latency**: DAX + Redis provide sub-ms reads

Negative

- **Cost**: ~$60K/month at 10M users for DynamoDB alone
- **Complexity**: Four different databases to manage
- **Consistency tradeoffs**: Eventual consistency for some operations
- **Operational overhead**: Multiple systems to monitor

Cost Breakdown

Component	1M Users	10M Users
DynamoDB Global Tables	$5,000	$60,000
DAX Cluster	$1,000	$5,000
OpenSearch Serverless	$8,000	$90,000
Neptune	$1,000	$12,000
ElastiCache Redis	$2,000	$10,000
Total Memory Infrastructure	**$17,000**	**$177,000**

References

- DynamoDB Global Tables
- OpenSearch Serverless
- Amazon Neptune
- ElastiCache for Redis