readme.md for @push.rocks/levelcache

Supercharged Multi-Level Caching for Modern Applications

A high-performance, tiered caching solution that intelligently leverages memory, disk, and S3 storage to deliver blazing-fast data access with reliable persistence and backup capabilities.

Highlights

 Intelligent Tiered Storage - Automatically routes data between memory, disk, and S3 based on size and access patterns
⚡ **Lightning Fast** - Memory-first architecture ensures microsecond access times for hot data
□□ **Persistent & Durable** - Optional disk and S3 layers provide data durability across restarts
□□ **TTL Support** - Built-in time-to-live for automatic cache expiration
□□ **TypeScript First** - Full type safety and excellent IDE support
☁ **S3 Ready** - Seamless integration with Amazon S3 for massive scale caching

Install

```
# Using npm
npm install @push.rocks/levelcache --save

# Using yarn
yarn add @push.rocks/levelcache

# Using pnpm (recommended)
pnpm add @push.rocks/levelcache
```

Usage

Quick Start

Get up and running with just a few lines:

```
import { LevelCache, CacheEntry } from '@push.rocks/levelcache';

// Initialize cache with minimal config
const cache = new LevelCache({
  cacheId: 'myAppCache'
});

// Wait for cache to be ready
await cache.ready;

// Store data
const entry = new CacheEntry({
  contents: Buffer.from('Hello Cache World! □□'),
  ttl: 60000 // 1 minute TTL
});

await cache.storeCacheEntryByKey('greeting', entry);

// Retrieve data
const retrieved = await cache.retrieveCacheEntryByKey('greeting');
console.log(retrieved.contents.toString()); // "Hello Cache World! □□
```

Advanced Configuration

`LevelCache` offers granular control over storage tiers and behavior:

```
const cache = new LevelCache({
  cacheId: 'productionCache',

  // Storage Limits
  maxMemoryStorageInMB: 128,      // 128MB RAM cache (default: 0.5)
  maxDiskStorageInMB: 1024,        // 1GB disk cache (default: 10)
  maxS3StorageInMB: 10240,         // 10GB S3 storage (optional)
```

```
  // Disk Configuration
  diskStoragePath: './cache-data', // Custom disk location (default: '.nogit')

  // S3 Configuration (optional)
  s3Config: {
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
    region: 'us-east-1'
  },
  s3BucketName: 'my-cache-bucket',

  // Behavior Options
  forceLevel: 'memory',            // Force specific tier (optional)
  immutableCache: false,           // Prevent cache mutations
  persistentCache: true            // Persist cache on restarts
});
```

Core Operations

Storing Data

```
// Store text data
const textEntry = new CacheEntry({
  contents: Buffer.from('Important text data'),
  ttl: 3600000, // 1 hour
  typeInfo: 'text/plain' // Optional metadata
});
await cache.storeCacheEntryByKey('document:123', textEntry);

// Store JSON data
const jsonData = { user: 'john', role: 'admin' };
const jsonEntry = new CacheEntry({
  contents: Buffer.from(JSON.stringify(jsonData)),
  ttl: 7200000, // 2 hours
  typeInfo: 'application/json'
});
await cache.storeCacheEntryByKey('user:john', jsonEntry);

// Store binary data (images, files, etc.)
```

```
const imageBuffer = await fs.readFile('./logo.png');
const imageEntry = new CacheEntry({
  contents: imageBuffer,
  ttl: 86400000, // 24 hours
  typeInfo: 'image/png'
});
await cache.storeCacheEntryByKey('assets:logo', imageEntry);
```

Retrieving Data

```
// Basic retrieval
const entry = await cache.retrieveCacheEntryByKey('user:john');
if (entry) {
  const userData = JSON.parse(entry.contents.toString());
  console.log(userData); // { user: 'john', role: 'admin' }
} else {
  console.log('Cache miss or expired');
}

// Check if key exists
const exists = await cache.checkKeyPresence('user:john');
console.log(`Key exists: ${exists}`);

// Handle cache misses gracefully
async function getUser(userId: string) {
  const cacheKey = `user:${userId}`;
  let entry = await cache.retrieveCacheEntryByKey(cacheKey);

  if (!entry) {
    // Cache miss - fetch from database
    const userData = await database.getUser(userId);

    // Store in cache for next time
    entry = new CacheEntry({
      contents: Buffer.from(JSON.stringify(userData)),
      ttl: 600000 // 10 minutes
    });
    await cache.storeCacheEntryByKey(cacheKey, entry);
  }
```

```
  return JSON.parse(entry.contents.toString());
}
```

Managing Cache

```
// Delete specific entry
await cache.deleteCacheEntryByKey('user:john');

// Clean expired entries
await cache.cleanOutdated();

// Clear entire cache
await cache.cleanAll();
```

Storage Tiers Explained

`LevelCache` automatically determines the optimal storage tier based on data size and available capacity:

1. **Memory Cache** □□

- **Speed**: Microsecond access
- **Best for**: Frequently accessed, small data
- **Default limit**: 0.5MB (configurable)
- First tier checked for all operations

2. **Disk Cache** □□

- **Speed**: Millisecond access
- **Best for**: Medium-sized data, persistent storage needed
- **Default limit**: 10MB (configurable)
- Data survives process restarts when `persistentCache: true`

3. **S3 Cache** ☁

- **Speed**: Network latency (typically 50-200ms)
- **Best for**: Large data, long-term storage, distributed caching
- **Default limit**: 50MB (configurable)
- Requires S3 configuration
- Ideal for shared cache across multiple instances

Real-World Use Cases

API Response Caching

```
class ApiCache {
  private cache: LevelCache;

  constructor() {
    this.cache = new LevelCache({
      cacheId: 'apiResponses',
      maxMemoryStorageInMB: 256,
      maxDiskStorageInMB: 2048,
      persistentCache: true
    });
  }

  async getCachedResponse(endpoint: string, params: any) {
    const cacheKey = `api:${endpoint}:${JSON.stringify(params)}`;

    let cached = await this.cache.retrieveCacheEntryByKey(cacheKey);
    if (cached) {
      return JSON.parse(cached.contents.toString());
    }

    // Fetch fresh data
    const response = await fetch(endpoint, { params });
    const data = await response.json();

    // Cache for 5 minutes
    const entry = new CacheEntry({
      contents: Buffer.from(JSON.stringify(data)),
      ttl: 300000
    });
    await this.cache.storeCacheEntryByKey(cacheKey, entry);

    return data;
  }
}
```

Session Storage

```
class SessionManager {
  private cache: LevelCache;

  constructor() {
    this.cache = new LevelCache({
      cacheId: 'sessions',
      maxMemoryStorageInMB: 64,
      maxDiskStorageInMB: 512,
      immutableCache: false,
      persistentCache: true
    });
  }

  async createSession(userId: string, sessionData: any) {
    const sessionId = generateSessionId();
    const entry = new CacheEntry({
      contents: Buffer.from(JSON.stringify({
        userId,
        ...sessionData,
        createdAt: Date.now()
      })),
      ttl: 86400000 // 24 hour sessions
    });

    await this.cache.storeCacheEntryByKey(`session:${sessionId}`, entry);
    return sessionId;
  }

  async getSession(sessionId: string) {
    const entry = await this.cache.retrieveCacheEntryByKey(`session:${sessionId}`);
    return entry ? JSON.parse(entry.contents.toString()) : null;
  }

  async destroySession(sessionId: string) {
    await this.cache.deleteCacheEntryByKey(`session:${sessionId}`);
  }
}
```

Distributed Processing Cache

```
// Share computed results across multiple workers using S3
const distributedCache = new LevelCache({
  cacheId: 'mlModelResults',
  maxMemoryStorageInMB: 512,
  maxDiskStorageInMB: 5120,
  maxS3StorageInMB: 102400, // 100GB for model outputs
  s3Config: {
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
    region: 'us-west-2'
  },
  s3BucketName: 'ml-computation-cache',
  persistentCache: true
});

// Worker process can store results
async function storeComputationResult(jobId: string, result: Buffer) {
  const entry = new CacheEntry({
    contents: result,
    ttl: 604800000, // 7 days
    typeInfo: 'application/octet-stream'
  });
  await distributedCache.storeCacheEntryByKey(`job:${jobId}`, entry);
}

// Other workers can retrieve results
async function getComputationResult(jobId: string) {
  const entry = await distributedCache.retrieveCacheEntryByKey(`job:${jobId}`);
  return entry ? entry.contents : null;
}
```

Performance Tips □□

1. **Size your tiers appropriately** - Set memory limits based on your hot data size
2. **Use meaningful cache keys** - Include version/hash in keys for cache invalidation
3. **Set realistic TTLs** - Balance freshness with performance
4. **Monitor cache hit rates** - Track `checkKeyPresence()` to optimize tier sizes

5. **Batch operations** - Group related cache operations when possible
6. **Use compression** - Compress large values before caching to maximize tier utilization

Migration & Compatibility

Coming from other caching solutions? Here's how LevelCache compares:

- **Redis** → LevelCache provides similar speed with added persistence and S3 backup
- **Memcached** → LevelCache adds persistence and automatic tier management
- **Local storage** → LevelCache adds memory tier and S3 backup capabilities
- **S3 only** → LevelCache adds memory and disk tiers for dramatic speed improvements

API Reference

LevelCache Class

Constructor Options

Option	Type	Default	Description
`cacheId`	string	required	Unique identifier for the cache instance
`maxMemoryStorageInMB`	number	0.5	Maximum memory storage in megabytes
`maxDiskStorageInMB`	number	10	Maximum disk storage in megabytes
`maxS3StorageInMB`	number	50	Maximum S3 storage in megabytes
`diskStoragePath`	string	'.nogit'	Path for disk cache storage
`s3Config`	object	undefined	AWS S3 configuration object
`s3BucketName`	string	undefined	S3 bucket name for cache storage
`forceLevel`	string	undefined	Force storage to specific tier
`immutableCache`	boolean	false	Prevent cache entry modifications
`persistentCache`	boolean	false	Persist cache across restarts

Methods

Method	Returns	Description
`ready`	Promise	Resolves when cache is initialized
`storeCacheEntryByKey(key, entry)`	Promise	Store a cache entry
`retrieveCacheEntryByKey(key)`	Promise<CacheEntry|null>	Retrieve a cache entry
`checkKeyPresence(key)`	Promise	Check if key exists
`deleteCacheEntryByKey(key)`	Promise	Delete a cache entry
`cleanOutdated()`	Promise	Remove expired entries
`cleanAll()`	Promise	Clear entire cache

CacheEntry Class

Constructor Options

Option	Type	Required	Description
`contents`	Buffer	yes	The data to cache
`ttl`	number	yes	Time-to-live in milliseconds
`typeInfo`	string	no	Optional metadata about content type

License and Legal Information

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

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

Trademarks

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

herein. Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines, and any usage must be approved in writing by Task Venture Capital GmbH.

Company Information

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

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

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

Revision #3
Created 2026-03-28 11:09:54 UTC by foss.global Team
Updated 2026-03-28 12:16:30 UTC by foss.global Team