Performance Optimization

Overview

This guide outlines strategies and implementations for optimizing the L4VA API's performance across multiple layers: API, Database, Caching, and Blockchain interactions.

API Layer Optimization

1. Request-Response Optimization

```
interface ResponseOptimization {
  // Field selection
  fields?: string[];          // Selected fields to return
  expand?: string[];          // Related data to include
  version?: string;           // Response format version
}

// Implementation
const optimizeResponse = (data: any, options: ResponseOptimization) => {
  const optimized = options.fields
    ? pickFields(data, options.fields)
    : data;

  if (options.expand) {
    await expandRelations(optimized, options.expand);
  }

  return optimized;
};

// Usage example
app.get('/api/v1/vaults/:id', async (req, res) => {
  const vault = await VaultService.findById(req.params.id);
  const optimized = await optimizeResponse(vault, {
    fields: ['id', 'status', 'assets'],
    expand: ['activeProposals']
  });
  res.json(optimized);
});
```

2. Request Batching

```
interface BatchRequest {
  id: string;
  method: string;
  path: string;
  body?: any;
}

const batchHandler = async (requests: BatchRequest[]) => {
  return Promise.all(requests.map(async (request) => {
    try {
      const result = await router.handle(request);
      return {
        id: request.id,
        status: 'success',
        data: result
      };
    } catch (error) {
      return {
        id: request.id,
        status: 'error',
        error: error.message
      };
    }
  }));
};
```

3. Rate Limiting with Redis

```
class RateLimiter {
  private redis: Redis;

  async checkLimit(key: string, limit: number, window: number): Promise<boolean> {
    const multi = this.redis.multi();
    const now = Date.now();

    multi.zremrangebyscore(key, 0, now - window);
    multi.zadd(key, now, `${now}`);
    multi.zcard(key);
```

```
    const [,, count] = await multi.exec();
    return count < limit;
  }
}
```

Database Optimization

1. Query Optimization

```
// Optimized query builder
class QueryBuilder {
  private query: any = {};
  private options: QueryOptions = {};

  // Index-aware filtering
  addFilter(field: string, value: any) {
    if (this.hasIndex(field)) {
      this.query[field] = value;
    } else {
      this.options.postProcess = true;
    }
  }

  // Efficient pagination
  setPagination(page: number, limit: number) {
    this.options.skip = (page - 1) * limit;
    this.options.limit = limit;
    this.options.sort = { _id: 1 }; // Index-based sorting
  }

  // Selective field projection
  selectFields(fields: string[]) {
    this.options.projection = fields.reduce((acc, field) => {
      acc[field] = 1;
      return acc;
    }, {});
  }
}
```

2. Aggregation Pipeline Optimization

```
const optimizedAggregation = [
  // Early filtering
  {
    $match: {
      status: 'ACTIVE',
      'assetWindow.endTime': { $gt: new Date() }
    }
  },

  // Limit fields early
  {
    $project: {
      id: 1,
      status: 1,
      assets: 1
    }
  },

  // Use index for sorting
  {
    $sort: {
      'assetWindow.endTime': 1
    }
  },

  // Paginate results
  {
    $skip: skip
  },
  {
    $limit: limit
  }
];
```

3. Indexing Strategy

```
interface IndexStrategy {
  // Compound indexes for common queries
```

```
  compoundIndexes: {
    vault_status_type: { status: 1, type: 1 },
    proposal_vault_status: { vaultId: 1, status: 1 },
    asset_contract_token: { contractAddress: 1, tokenId: 1 }
  };

  // Text indexes for search
  textIndexes: {
    vault_search: { name: 'text', description: 'text' }
  };

  // Partial indexes for active records
  partialIndexes: {
    active_vaults: {
      index: { status: 1 },
      filter: { status: 'ACTIVE' }
    }
  };
}
```

Caching Layer

1. Multi-Level Caching

```
class CacheManager {
  private memoryCache: Map<string, any>;
  private redis: Redis;

  async get(key: string, fetchFn: () => Promise<any>) {
    // Check memory cache
    if (this.memoryCache.has(key)) {
      return this.memoryCache.get(key);
    }

    // Check Redis cache
    const redisValue = await this.redis.get(key);
    if (redisValue) {
      this.memoryCache.set(key, redisValue);
      return redisValue;
    }
```

```
    // Fetch and cache
    const value = await fetchFn();
    await this.set(key, value);
    return value;
  }

  async set(key: string, value: any) {
    this.memoryCache.set(key, value);
    await this.redis.set(key, value);
  }
}
```

2. Cache Invalidation Strategy

```
interface CacheInvalidation {
  patterns: {
    vault: 'vault:*',
    proposal: 'proposal:*',
    asset: 'asset:*'
  };

  dependencies: {
    vault: ['asset', 'proposal'],
    proposal: ['vote'],
    asset: ['valuation']
  };
}

class CacheInvalidator {
  async invalidate(type: string, id: string) {
    const pattern = this.patterns[type];
    const keys = await this.redis.keys(pattern);

    // Invalidate direct cache
    await this.redis.del(keys);

    // Invalidate dependencies
    for (const depType of this.dependencies[type]) {
      await this.invalidate(depType, id);
```

```
    }
  }
}
```

Blockchain Optimization

1. Transaction Batching

```
class TransactionBatcher {
  private queue: Transaction[] = [];
  private batchSize: number = 10;
  private batchTimeout: number = 5000;

  async addTransaction(tx: Transaction) {
    this.queue.push(tx);

    if (this.queue.length >= this.batchSize) {
      await this.processBatch();
    }
  }

  private async processBatch() {
    const batch = this.queue.splice(0, this.batchSize);
    const multicall = await this.createMulticall(batch);
    return await this.sendTransaction(multicall);
  }
}
```

2. Event Processing Optimization

```
class EventProcessor {
  private lastProcessedBlock: number;
  private batchSize: number = 1000;

  async processEvents(startBlock: number, endBlock: number) {
    for (let block = startBlock; block <= endBlock; block += this.batchSize) {
      const events = await this.fetchEvents(block, block + this.batchSize);
      await this.processEventBatch(events);
    }
  }
```

```
  private async processEventBatch(events: Event[]) {
    // Group events by type
    const grouped = groupBy(events, 'eventType');

    // Process each type in parallel
    await Promise.all(
      Object.entries(grouped).map(([type, events]) =>
        this.processEventType(type, events)
      )
    );
  }
}
```

Load Testing and Monitoring

1. Load Testing Configuration

```
interface LoadTest {
  scenarios: {
    name: string;
    weight: number;
    flow: RequestFlow[];
  }[];

  thresholds: {
    http_req_duration: ['p(95)<500'],
    http_reqs: ['rate>100'],
    errors: ['rate<0.1']
  };

  stages: {
    duration: string;
    target: number;
  }[];
}
```

2. Performance Monitoring

```
interface PerformanceMetrics {
  // API metrics
  api: {
    responseTime: Histogram;
    requestRate: Counter;
    errorRate: Counter;
  };

  // Database metrics
  db: {
    queryTime: Histogram;
    connectionPool: Gauge;
    activeQueries: Gauge;
  };

  // Cache metrics
  cache: {
    hitRate: Gauge;
    missRate: Gauge;
    evictionRate: Counter;
  };

  // Blockchain metrics
  blockchain: {
    transactionTime: Histogram;
    gasUsage: Histogram;
    nodeLatency: Gauge;
  };
}
```

Best Practices

1. **API Optimization**
   - Implement field selection
   - Use request batching
   - Enable compression
   - Optimize payload size

2. **Database Optimization**
   - Create efficient indexes
   - Optimize query patterns

- Use connection pooling
- Implement sharding strategy

3. **Caching Strategy**
   - Implement multi-level caching
   - Use appropriate TTLs
   - Handle cache invalidation
   - Monitor cache hit rates

4. **Blockchain Optimization**
   - Batch transactions
   - Implement retry strategies
   - Optimize gas usage
   - Cache blockchain data

5. **Monitoring and Alerting**
   - Track key metrics
   - Set up alerts
   - Monitor resource usage
   - Analyze performance trends

Implementation Checklist

- ☐ Configure response optimization
- ☐ Implement query optimization
- ☐ Set up caching layer
- ☐ Configure blockchain batching
- ☐ Implement monitoring
- ☐ Set up load testing
- ☐ Document optimization strategies
- ☐ Train team on best practices

Revision #1
Created 26 November 2024 01:26:52 by Aric Fedida
Updated 26 November 2024 01:27:13 by Aric Fedida