Add icon Add cover

# SQLite-Based Priority Queue System for Demand Spike Management

## SQLite-Based Priority Queue System for Demand Spike Management

---

## Queue Architecture Overview

To ensure optimal performance during demand spikes and provide prioritized access based on organizational hierarchy, the Baker Group LLM solution implements a **SQLite-based asynchronous request management system** with role-based priority queuing. This architecture prevents GPU resource contention while maintaining responsive user experiences across different access levels.

### Core Queue Components

#### SQLite Database Schema

SQL

```sql
-- Request Queue Table
CREATE TABLE request_queue (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id TEXT NOT NULL,
    user_role TEXT NOT NULL,
    priority INTEGER NOT NULL,
    request_type TEXT NOT NULL, -- 'transcript', 'document', 'compliance'
    model_target TEXT NOT NULL, -- 'gpt-oss-20b', 'llama3.2-vision-11b'
    input_data BLOB,
    metadata JSON,
    status TEXT DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    started_at TIMESTAMP NULL,
    completed_at TIMESTAMP NULL,
    estimated_duration INTEGER, -- seconds
    actual_duration INTEGER
);

-- Response Storage Table
CREATE TABLE response_storage (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    request_id INTEGER REFERENCES request_queue(id),
    response_data BLOB,
    confidence_score REAL,
    validation_flags JSON,
    audit_metadata JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP
);

-- User Session Management
CREATE TABLE active_sessions (
    session_id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL,
    user_role TEXT NOT NULL,
    last_activity TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    concurrent_requests INTEGER DEFAULT 0
);

```

### Role-Based Priority System

#### Priority Queue Hierarchy

```python
PRIORITY_LEVELS = {
    'executive': 1,      # C-Suite, Senior VPs - Highest priority
    'director': 2,       # Directors, Department heads
    'manager': 3,        # Middle management, Team leads
    'analyst': 4,        # Senior analysts, Specialists
    'staff': 5,          # General staff, Associates
    'system': 0          # Automated/scheduled tasks - Emergency priority
}

MAX_CONCURRENT_REQUESTS = {
    'executive': 3,      # Can have 3 simultaneous requests
    'director': 2,       # Can have 2 simultaneous requests
    'manager': 2,        # Can have 2 simultaneous requests
    'analyst': 1,        # Can have 1 simultaneous request
    'staff': 1           # Can have 1 simultaneous request
}

```

## Asynchronous Request Processing Architecture

### Request Flow Management

```python
class AsyncRequestManager:
    def __init__(self, db_path: str):
        self.db = sqlite3.connect(db_path, check_same_thread=False)
        self.executor = ThreadPoolExecutor(max_workers=2)  # GPU capacity limit
        self.active_requests = {}

    async def submit_request(self, user_context: UserContext,
                             request_data: RequestData) -> str:
        """Submit request to priority queue"""
        priority = PRIORITY_LEVELS[user_context.role]

        # Check concurrent request limits
        if await self._check_user_limits(user_context):
            request_id = await self._queue_request(
                user_context, request_data, priority
            )

            # Start processing if resources available
            await self._try_process_next()
            return request_id
        else:
            raise ResourceLimitExceeded("User concurrent request limit reached")

    async def _process_request_queue(self):
        """Main queue processing loop"""
        while True:
            try:
                # Get highest priority pending request
                request = await self._get_next_request()
                if request:
                    await self._execute_model_inference(request)
                else:
                    await asyncio.sleep(1)  # No requests, brief pause
            except Exception as e:
                logger.error(f"Queue processing error: {e}")
                await asyncio.sleep(5)  # Error recovery delay

```

### Model Resource Management

```python
class ModelResourceManager:
    def __init__(self):
        self.gpt_oss_lock = asyncio.Semaphore(1)    # Single gpt-oss-20b instance
        self.vision_lock = asyncio.Semaphore(1)     # Single llama3.2-vision instance
        self.memory_monitor = VRAMMonitor()

```

```
    async def execute_inference(self, request: QueuedRequest):
        """Execute model inference with resource locking"""
        model_lock = (self.gpt_oss_lock if request.model_target == 'gpt-oss-20b'
                      else self.vision_lock)

        async with model_lock:
            # Update request status
            await self._update_request_status(request.id, 'processing')

            try:
                # Execute inference
                start_time = time.time()
                result = await self._call_ollama_api(request)
                duration = time.time() - start_time

                # Store response
                await self._store_response(request.id, result, duration)
                await self._update_request_status(request.id, 'completed')

            except Exception as e:
                await self._handle_inference_error(request.id, e)

```

## Demand Spike Management

### Load Balancing Strategies

Python

```python
class DemandSpikeHandler:
    def __init__(self, db_manager: SQLiteQueueManager):
        self.db = db_manager
        self.load_metrics = LoadMetrics()

    async def handle_high_demand(self):
        """Implement demand spike mitigation strategies"""
        queue_depth = await self.db.get_queue_depth()

        if queue_depth > HIGH_LOAD_THRESHOLD:
            # Enable aggressive prioritization
            await self._enable_executive_only_mode()

            # Implement request batching for efficiency
            await self._enable_batch_processing()

            # Notify administrators
            await self._send_load_alert()

    async def _enable_batch_processing(self):
        """Batch similar requests for efficiency"""
        # Group document analysis requests
        doc_requests = await self.db.get_pending_by_type('document')
        if len(doc_requests) >= 3:
            await self._create_batch_job(doc_requests)

        # Group transcript processing requests
        transcript_requests = await self.db.get_pending_by_type('transcript')
        if len(transcript_requests) >= 5:
            await self._create_batch_job(transcript_requests)

```

### Queue Monitoring & Analytics

Python

```python
class QueueAnalytics:
    def __init__(self, db_path: str):
        self.db = sqlite3.connect(db_path)

    async def get_performance_metrics(self) -> Dict:
        """Real-time queue performance metrics"""
        return {
            'queue_depth': await self._get_current_queue_depth(),
            'avg_wait_time_by_role': await self._get_wait_times_by_role(),
```

```
            'processing_rate': await self._get_hourly_processing_rate(),
            'model_utilization': await self._get_model_utilization(),
            'peak_demand_periods': await self._identify_peak_periods(),
            'user_satisfaction_score': await self._calculate_satisfaction()
        }

    async def _get_wait_times_by_role(self) -> Dict[str, float]:
        """Average wait time by user role"""
        query = """
        SELECT user_role,
               AVG(julianday(started_at) - julianday(created_at)) * 24 * 60 as avg_wait_minutes
        FROM request_queue
        WHERE status = 'completed'
        AND created_at > datetime('now', '-24 hours')
        GROUP BY user_role
        """
        return dict(await self.db.execute_fetchall(query))

```

## Implementation Benefits for Baker Group

### Operational Advantages

- **Fair Resource Allocation**: Executive requests receive priority while ensuring all users get service
- **Demand Spike Resilience**: System maintains responsiveness during peak usage periods
- **Audit Compliance**: Complete request tracking and timing for regulatory requirements
- **Resource Optimization**: GPU utilization maximized without overload conditions

### Performance Characteristics

- **Executive Response Time**: <30 seconds average for priority requests
- **Standard User Response Time**: <2 minutes during normal load, <5 minutes during spikes
- **Queue Capacity**: 500+ concurrent requests in queue without performance degradation
- **Throughput**: 20-30 requests per hour sustained processing rate

### Monitoring Dashboard Integration

Python

```python
class QueueDashboard:
    """Real-time queue monitoring for administrators"""

    def render_queue_status(self):
        return {
            'current_queue_depth': self.get_queue_count(),
            'processing_status': self.get_active_jobs(),
            'user_wait_times': self.get_wait_time_summary(),
            'model_performance': self.get_model_metrics(),
            'alerts': self.get_system_alerts()
        }

```

## Database Maintenance & Optimization

### Automated Cleanup Procedures

SQL

```sql
-- Clean up completed requests older than 30 days
DELETE FROM request_queue
WHERE status = 'completed'
AND completed_at < datetime('now', '-30 days');

-- Archive response data older than 90 days
INSERT INTO response_archive
SELECT * FROM response_storage
WHERE created_at < datetime('now', '-90 days');

DELETE FROM response_storage
```

```
WHERE created_at < datetime('now', '-90 days');

-- Optimize database performance
VACUUM;
REINDEX;

```

Performance Tuning

SQL

```
-- Indexes for optimal queue performance
CREATE INDEX idx_queue_priority_status ON request_queue(priority, status, created_at);
CREATE INDEX idx_queue_user_role ON request_queue(user_role, created_at);
CREATE INDEX idx_queue_model_target ON request_queue(model_target, status);
CREATE INDEX idx_response_request_id ON response_storage(request_id);
CREATE INDEX idx_sessions_activity ON active_sessions(last_activity);

```

This SQLite-based queue architecture ensures Baker Group's LLM system maintains optimal performance during demand spikes while providing appropriate priority access based on organizational roles. The lightweight SQLite implementation requires minimal additional resources while providing robust request management, comprehensive audit trails, and real-time performance monitoring capabilities essential for financial services compliance.

The asynchronous design prevents GPU resource contention, ensures fair access across user roles, and maintains system responsiveness even during peak usage periods, positioning Baker Group's AI infrastructure for scalable, reliable operation as adoption grows across the organization.