ECON485 Homework 4: NoSQL Concepts and Applications

Student: Elif I■■l Çiçek
Student ID: 23232810007
Date: December 2025
Course: ECON485 Database Systems

Executive Summary

This report analyzes three operational challenges in the course registration system and proposes NoSQL-based solutions using Redis (key-value store) and MongoDB (document store).

Task 1: Seat Availability Lookups with Redis

1.1 Problem Analysis

In the SQL-based registration system, seat availability is computed via:

```
SELECT s.SectionID, s.Capacity - COUNT(r.StudentID) AS AvailableSeats
FROM Sections s
LEFT JOIN Registrations r ON s.SectionID = r.SectionID
GROUP BY s.SectionID, s.Capacity;
```

During peak registration, thousands of concurrent queries create:
- High CPU load from COUNT() and GROUP BY operations
- Lock contention on Registration table
- Slow response times (>500ms per query)

1.2 Redis Key-Value Solution

Redis stores available seats as atomic counters with simple key-value pairs:

```
Key: section:101:available_seats
Value: 15  (integer representing available seats)

Key: section:102:available_seats
Value: 0   (section is full)
```

Basic Operations:

```
DECR section:101:available_seats  # when a student registers
INCR section:101:available_seats  # when a student drops
GET section:101:available_seats   # to check availability
```

1.3 Atomic Operations and Concurrency Control

Redis ensures data consistency through:
1. Single-threaded execution model - All operations are serialized
2. Atomic commands - INCR and DECR are indivisible operations
3. Transaction blocks - MULTI/EXEC for grouped operations
4. Optimistic locking - WATCH command for conditional updates

```
# Example: Safe seat reservation in Redis
import redis

r = redis.Redis()
r.watch('section:101:available_seats')
available = int(r.get('section:101:available_seats'))

if available > 0:
    # Start transaction
    pipeline = r.pipeline()
    pipeline.multi()
    pipeline.decr('section:101:available_seats')
    pipeline.execute()  # All or nothing
    print("Seat reserved successfully")
else:
    print("No seats available")
```

1.4 Caching Architecture

Mermaid Diagram (as text):

```
graph TB
    A[Student Browser] --> B[Load Balancer]
    B --> C[Application Server]
    C --> D{Redis Cache}
    D -->|Cache Hit\n0.1ms| E[Return Available Seats]
    D -->|Cache Miss| F[SQL Database]
    F --> G[Compute Availability\n200-500ms]
    G --> H[Update Redis Cache]
    H --> E
    E --> I[Student Sees Real-time Availability]
```

Cache Strategy Implementation:
1. Write-through caching: Updates go to both SQL and Redis simultaneously
2. Lazy loading: Cache populated on first miss
3. TTL (Time-To-Live): Automatic cache expiration after 5 minutes

1.5 Performance Comparison

Metric	SQL-Only Solution	Redis + SQL Hybrid
Read Latency	200-500ms	1-5ms
Write Latency	100-300ms	50-100ms
Maximum Throughput	~100 QPS	10,000+ QPS
Database CPU Load	70-90% during peak	10-20% during peak
Cost	Lower (only SQL)	Higher (Redis infrastructure)

1.6 When to Use This Approach

Optimal Use Cases:
- High concurrency scenarios (>1000 concurrent users)
- Read-heavy workloads (90% read, 10% write)
- Data where eventual consistency is acceptable
- Simple data models (counters, flags, simple values)

Poor Fit Cases:
- Strong ACID transaction requirements
- Complex business logic in database
- Budget-constrained projects
- Infrequently accessed data

1.7 Limitations and Operational Risks

1. Data Persistence Risk: Redis is primarily in-memory (can lose data on crash)
2. Cache Invalidation Complexity: Manual sync needed between Redis and SQL
3. Memory Cost: Storing counters for all sections requires significant RAM
4. Additional Complexity: Extra system to monitor and maintain
5. Network Latency: Additional hop to Redis server

Mitigation Strategies:
- Use Redis persistence (RDB/AOF snapshots)
- Implement circuit breakers for cache failures
- Monitor cache hit rates (>95% target)
- Use connection pooling

Task 2: Prerequisite Eligibility Caching

2.1 Problem Analysis

The SQL query for prerequisite checking in our system:

```
SELECT p.RequiredCourseID, c.Grade, p.MinimumGrade
FROM Prerequisites p
LEFT JOIN CompletedCourses c ON c.CourseID = p.RequiredCourseID
                               AND c.StudentID = :student_id
WHERE p.CourseID = :target_course_id;
```

Performance Issues:
- Repeated JOIN operations for the same student across multiple courses
- Static data (student grades don't change during registration period)
- N+1 query problem when checking eligibility for multiple courses
- Index overhead on multiple tables

2.2 Solution 1: Key-Value Store (Redis)

Data Model Design:

```
# Simple eligibility flag
Key: eligibility:1001:ECON211
Value: {
  "eligible": true,
  "checked_at": "2025-12-24T10:30:00Z",
  "expires_at": "2025-12-31T23:59:59Z"
}

# Prerequisite list for course
Key: prerequisites:ECON211
Value: ["ECON101", "MATH101"]  (JSON array)
# Set eligibility with 24-hour expiration
SETEX eligibility:1001:ECON211 86400 '{"eligible":true,"timestamp":"2025-12-24T10:30:00Z"}'

# Get eligibility status
GET eligibility:1001:ECON211

# Batch operations for multiple courses
MGET eligibility:1001:ECON211 eligibility:1001:MATH102 eligibility:1001:COMP101
```

2.3 Solution 2: Document Store (MongoDB)

```
{
  "_id": "1001_ECON211",
  "student_id": 1001,
  "course_id": "ECON211",
  "eligibility": {
    "status": "ELIGIBLE",
    "last_checked": "2025-12-24T10:30:00Z",
    "details": [
      {
        "prerequisite": "ECON101",
        "required_grade": "C",
        "student_grade": "B",
        "passed": true,
        "checked_date": "2025-12-24T10:30:00Z"
      },
      {
        "prerequisite": "MATH101",
        "required_grade": "C",
        "student_grade": "A",
        "passed": true,
        "checked_date": "2025-12-24T10:30:00Z"
      }
    ]
  },
  "metadata": {
    "created_at": "2025-12-24T10:30:00Z",
    "updated_at": "2025-12-24T10:30:00Z",
    "cache_ttl": 86400
  }
}
```

2.4 Technology Comparison

Aspect	Redis (Key-Value)	MongoDB (Document Store)
Data Model	Simple key-value pairs	Rich, nested documents
Query Flexibility	Limited to key lookups	Rich query language with projections
Storage Model	Primarily in-memory	Disk-based with memory cache
Best Use Case	Simple flag caching	Complex eligibility analysis
Cache Invalidation	TTL automatic expiration	Manual or TTL index
Scalability	Excellent for reads	Good for both reads and writes

2.5 Cache Invalidation Strategies

```
// Redis: 24-hour expiration
SETEX eligibility:1001:ECON211 86400 'true'

// MongoDB: TTL index
db.eligibility.createIndex(
  { "metadata.created_at": 1 },
  { expireAfterSeconds: 86400 }
)
# When grade is updated
def on_grade_update(student_id, course_code, new_grade):
    # Clear related eligibility cache
    redis.delete(f"eligibility:{student_id}:*")

    # Or update MongoDB documents
    db.eligibility.update_many(
        {"student_id": student_id},
        {"$set": {"eligibility.status": "NEEDS_RECHECK"}}
    )
Key: eligibility:v2:1001:ECON211
(Change key when business logic changes)
```

2.6 Decision Guidelines

Choose Redis When:
- Only need binary eligibility status (eligible/not eligible)
- Extreme read performance required (>100,000 QPS)
- Simple data model suffices
- Memory resources available
Choose MongoDB When:
- Need detailed prerequisite analysis and reporting
- Historical tracking of eligibility decisions required
- Complex queries on eligibility data (e.g., 'show all students missing MATH101')
- Data persistence and durability are critical
- Integration with other document-based services

2.7 Implementation Example (Hybrid)

```python
def check_eligibility(student_id, course_id):
    # 1. Check Redis cache first
    cache_key = f"eligibility:{student_id}:{course_id}"
    cached = redis.get(cache_key)

    if cached:
        return json.loads(cached)

    # 2. Cache miss - compute eligibility
    eligibility = compute_eligibility_sql(student_id, course_id)

    # 3. Store in Redis with TTL
    redis.setex(cache_key, 3600, json.dumps(eligibility))

    # 4. Optional: Store detailed result in MongoDB for analytics
    if eligibility['status'] == 'NOT_ELIGIBLE':
        mongo_db.eligibility_logs.insert_one({
            'student_id': student_id,
            'course_id': course_id,
            'eligibility': eligibility,
            'timestamp': datetime.now()
        })

    return eligibility
```

Task 3: Storing Complex Historical Actions with MongoDB

3.1 Problem Analysis

```
CREATE TABLE ActionLog (
    LogID INT PRIMARY KEY,
    StudentID INT,
    ActionType VARCHAR(20),
    CourseID INT,
    SectionID INT,
    Timestamp DATETIME,
    Notes TEXT  -- Problem: Variable unstructured content!
);

-- Additional tables needed for different action types
CREATE TABLE OverrideApprovals (
    LogID INT FOREIGN KEY REFERENCES ActionLog(LogID),
    ApproverName VARCHAR(100),
    ApprovalDate DATETIME,
    Reason TEXT,
    Documents VARCHAR(255)
);

CREATE TABLE TimeConflictApprovals (
    LogID INT FOREIGN KEY REFERENCES ActionLog(LogID),
    ConflictingCourses VARCHAR(255),
    OverlapMinutes INT,
    ApprovedBy VARCHAR(100),
    SpecialConditions TEXT
);
```

Issues with Relational Approach:
- Multiple tables for different action types
- Sparse columns with many NULL values
- Complex JOINs for complete history views
- Schema migrations for new action types
- Inefficient storage of variable metadata

3.2 MongoDB Document Design Options

Option 1: Student-Centric Collection

```
{
  "_id": "student_1001",
  "student_id": 1001,
  "name": "Ahmet Y■lmaz",
  "email": "ahmet@student.edu",
  "registration_history": [
    {
      "action_id": "ACT-2025-09-01-001",
      "action_type": "COURSE_REGISTRATION",
      "timestamp": "2025-09-01T09:00:00Z",
      "course": {
        "code": "ECON101",
        "name": "Introduction to Economics",
        "section": "101"
      },
      "metadata": {
        "ip_address": "192.168.1.1",
        "user_agent": "Chrome/120.0.0.0",
        "session_id": "sess_abc123",
        "success": true,
        "response_time_ms": 245
      }
    },
    {
      "action_id": "ACT-2025-10-15-002",
      "action_type": "WITHDRAWAL",
      "timestamp": "2025-10-15T14:30:00Z",
      "course": {
        "code": "MATH101",
        "section": "103"
      },
      "metadata": {
        "reason": "schedule_conflict",
        "withdrawal_type": "student_initiated",
        "approved_by": "instructor_45",
        "approval_date": "2025-10-14T10:00:00Z",
        "refund_status": "processed",
        "refund_amount": 1500.00
      }
    },
    {
      "action_id": "ACT-2025-11-20-003",
      "action_type": "PREREQUISITE_OVERRIDE",
      "timestamp": "2025-11-20T11:15:00Z",
      "course": {
        "code": "COMP101",
        "name": "Introduction to Programming"
      },
      "metadata": {
        "prerequisite_override": true,
        "missing_prerequisites": ["COMP100"],
        "justification": "3 years of industry programming experience",
        "supporting_documents": [
          {
            "name": "employment_certificate.pdf",
            "type": "employment_proof",
            "upload_date": "2025-11-19T09:30:00Z"
          }
        ],
        "approval_chain": [
          {
            "approver": "instructor_22",
            "role": "Course Instructor",
            "decision": "APPROVED",
            "date": "2025-11-19T14:00:00Z",
            "comments": "Student demonstrates sufficient experience"
          },
          {
            "approver": "dept_chair_5",
            "role": "Department Chair",
            "decision": "APPROVED",
            "date": "2025-11-20T10:30:00Z",
            "comments": "Final approval granted"
          }
        ],
        "final_status": "APPROVED",
        "effective_date": "2025-11-20T11:15:00Z"
      }
    }
  ],
  "statistics": {
    "total_actions": 15,
    "last_updated": "2025-12-24T10:00:00Z"
  }
```

```
}
```

Option 2: Action-Centric Collection (Recommended for Analytics)

```
{
  "_id": ObjectId("67a1b2c3d4e5f67890123456"),
  "action_type": "TIME_CONFLICT_APPROVAL",
  "timestamp": "2025-09-05T13:45:00Z",
  "student": {
    "id": 1001,
    "name": "Ahmet Y■lmaz",
    "year": 3,
    "program": "Computer Science"
  },
  "academic_context": {
    "semester": "Fall 2025",
    "primary_course": "ECON101",
    "conflicting_course": "MATH101",
    "sections": ["ECON101-01", "MATH101-03"]
  },
  "conflict_details": {
    "overlap_duration_minutes": 30,
    "overlap_type": "lecture_time",
    "days": ["Monday", "Wednesday"],
    "time_range": "10:00-11:30"
  },
  "approval_process": {
    "requested_by": "student",
    "request_date": "2025-09-04T15:20:00Z",
    "approvers": [
      {
        "name": "Dr. Alice Smith",
        "role": "ECON101 Instructor",
        "decision": "APPROVED",
        "date": "2025-09-05T09:30:00Z",
        "conditions": ["Must attend all ECON101 lectures"]
      },
      {
        "name": "Prof. Bob Johnson",
        "role": "MATH101 Instructor",
        "decision": "APPROVED",
        "date": "2025-09-05T11:15:00Z",
        "conditions": ["Must submit all MATH101 assignments on time"]
      }
    ],
    "final_decision": "APPROVED_WITH_CONDITIONS",
    "decision_date": "2025-09-05T13:45:00Z"
  },
  "system_metadata": {
    "api_version": "v2.1",
    "client_application": "student_portal_web",
    "session_id": "sess_def456",
    "request_id": "req_789ghi",
    "processing_time_ms": 125
  },
  "attachments": [
    {
      "type": "email",
      "content": "Approval email chain",
      "reference": "email_conv_123"
    }
  ]
}
```

3.3 Schema Flexibility Advantages

1. No NULL Columns: Only store relevant fields for each action type
2. Nested Structures: Complex relationships in single document
3. Easy Evolution: Add new action types without migrations

```
{
  "action_type": "NEW_ACTION_TYPE_V2",
  "new_field": "value",
  "nested_data": {
    "field1": "value1",
    "field2": "value2"
  }
}
```

4. Natural Representation: Matches application objects 1:1
5. Denormalization: Reduced JOIN operations

3.4 Indexing Strategy for Performance

```
// Compound index for common student history queries
db.actions.createIndex({
  "student.id": 1,
  "timestamp": -1
})

// Index for specific action type searches
db.actions.createIndex({
  "action_type": 1,
  "timestamp": -1
})

// Index on nested fields
db.actions.createIndex({
  "academic_context.semester": 1,
  "action_type": 1
})

// Text index for free-text search in metadata
db.actions.createIndex({
  "approval_process.approvers.name": "text",
  "system_metadata.client_application": "text"
})

// TTL index for automatic data cleanup (1 year retention)
db.actions.createIndex(
  { "timestamp": 1 },
  { expireAfterSeconds: 31536000 }
)
```

3.5 Query Examples

```
// 1. Get all actions for a student (fast with compound index)
db.actions.find(
  { "student.id": 1001 }
).sort({ "timestamp": -1 }).limit(50)

// 2. Find all override approvals in Fall 2025
db.actions.find({
  "action_type": "PREREQUISITE_OVERRIDE",
  "academic_context.semester": "Fall 2025",
  "approval_process.final_decision": "APPROVED"
})

// 3. Aggregate statistics by action type
```

```
db.actions.aggregate([
  {
    $group: {
      _id: "$action_type",
      count: { $sum: 1 },
      last_action: { $max: "$timestamp" }
    }
  },
  { $sort: { count: -1 } }
])

// 4. Find actions with specific conditions
db.actions.find({
  "approval_process.approvers.conditions": {
    $regex: "must attend",
    $options: "i"
  }
})
```

3.6 Performance Comparison

Operation	MongoDB (Document)	SQL (Normalized)
Insert Single Action	5-10ms (one write)	10-30ms (multiple table writes)
Get Student Full History	2-5ms (single document/query)	50-200ms (multiple JOINs)
Search by Metadata	10-20ms (indexed nested fields)	Complex, often requires full-text search
Aggregate Analytics	50-100ms (native aggregation)	200-500ms (multiple GROUP BY)
Storage Overhead	20-30% higher (denormalization)	Lower (normalized)
Schema Migration	No downtime (flexible schema)	Downtime required (ALTER TABLE)

3.7 Trade-off Analysis

Advantages of MongoDB for Historical Actions:
■ Schema Flexibility: Easy to add new action types
■ Query Performance: Fast reads for student history
■ Development Speed: Matches application object model
■ Scalability: Horizontal scaling with sharding
■ Analytics: Built-in aggregation framework
Disadvantages:
■ No Referential Integrity: Manual validation needed
■ Data Duplication: Same data in multiple documents
■ Transaction Complexity: Multi-document transactions limited
■ Storage Cost: Higher due to denormalization
■ Learning Curve: Different query patterns than SQL
When to Choose Document Store:
- Audit trails and event logging systems
- Applications with highly variable data structures
- Read-optimized historical queries
- Microservices architectures with JSON APIs
- Rapid prototyping and iterative development
When to Stick with Relational:
- Strong transactional integrity requirements
- Complex reporting with many JOINs
- Legacy system integration
- Team with strong SQL expertise
- Budget constraints (MongoDB licensing costs)

Conclusion and Recommendations

The course registration system presents multiple opportunities for NoSQL optimization while maintaining the core relational database for transactional integrity.

Recommended Architecture:

1. Redis for Real-time Counters:
- Seat availability counters
- Session management
- Rate limiting

Implementation: Cache-aside pattern with write-through to SQL

2. MongoDB for Historical Data:
- Action logs and audit trails
- Student eligibility cache (detailed)
- System configuration (flexible schemas)

Implementation: Event sourcing pattern for actions

3. SQL Database for Core Operations:
- Student records (CRUD operations)
- Course catalog (structured data)
- Grade management (ACID requirements)
- Financial transactions

Migration Strategy:

1. Phase 1: Implement Redis caching for seat availability (immediate performance gain)
2. Phase 2: Migrate action logs to MongoDB (reduced SQL load)
3. Phase 3: Implement hybrid eligibility checking (Redis cache + MongoDB analytics)
4. Phase 4: Evaluate additional use cases for NoSQL adoption

Key Success Metrics:

- Redis cache hit rate: >95%
- MongoDB query latency: <50ms for 95th percentile
- SQL database load reduction: 40-60%
- Development velocity improvement: 30% faster feature delivery

References

1. Redis Labs. (2024). Redis Documentation: Atomic Operations and Transactions. https://redis.io/docs/latest/commands/incr/
2. MongoDB, Inc. (2024). MongoDB Documentation: Data Modeling Patterns. https://www.mongodb.com/docs/manual/core/data-modeling-introduction/
3. Fowler, M. (2011). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional.
4. Kreps, J. (2014). The Log: What every software engineer should know about real-time data's unifying abstraction. LinkedIn Engineering Blog.
5. Microsoft Azure Architecture Center. (2023). Caching Guidance and Best Practices. https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching
6. Redgate Software. (2023). SQL vs NoSQL: Choosing the Right Database for Your Application. https://www.red-gate.com/simple-talk/databases/sql-vs-nosql/
7. AWS. (2024). Database Migration Service Best Practices. https://aws.amazon.com/blogs/database/