Task B: Enhancement Plan - Multi-Document QA System

Team 70 - Contributors

Name	Email Address	Contributions %
NEERAJ BHATT	2024aa05020@wilp.bits-pilani.ac.in	100%
V. S. BALAKRISHNAN	2024aa05017@wilp.bits-pilani.ac.in	100%
KURUVELLA VENKATA SAI UPENDRA	2024aa05016@wilp.bits-pilani.ac.in	100%
SAJAL CHAUDHARY	2024aa05026@wilp.bits-pilani.ac.in	100%
SACHIN KUMAR	2024aa05024@wilp.bits-pilani.ac.in	100%

Summary

This document outlines the roadmap for enhancing the Document-Based Question Answering System to handle:

1. Multi-document querying with large document collections

2. Real-time document updates and indexing

3. Complex multi-hop questions requiring inference across multiple documents

---

1. Multi-Document Querying with Large Collections

Current Limitations

- In-memory storage limited to ~500 documents
- Linear search through all passages
- No document filtering or pagination
- All documents searched for every query

Enhancement Strategy

1.1 Database Integration

Technology: PostgreSQL + SQLAlchemy

Schema:

```python
class Document(Base):
    __tablename__ = "documents"

    id = Column(String(36), primary_key=True)
    filename = Column(String(255), nullable=False)
    upload_time = Column(DateTime, default=datetime.utcnow)
    text = Column(Text, nullable=False)
    text_length = Column(Integer)
    num_sentences = Column(Integer)
    metadata = Column(JSON)  # Tags, categories, etc.

class Passage(Base):
    __tablename__ = "passages"

    id = Column(String(36), primary_key=True)
    doc_id = Column(String(36), ForeignKey("documents.id"))
    text = Column(Text, nullable=False)
    start_pos = Column(Integer)
    end_pos = Column(Integer)
    embedding = Column(ARRAY(Float))  # For vector search

class QueryLog(Base):
    __tablename__ = "query_logs"

    id = Column(String(36), primary_key=True)
    question = Column(Text)
    documents_searched = Column(JSON)
    answers_returned = Column(Integer)
    processing_time = Column(Float)
    timestamp = Column(DateTime, default=datetime.utcnow)
```

Implementation:

```python
# Async database access
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy import select

async def get_documents(session: AsyncSession, limit: int = 100, offset: int = 0):
    stmt = select(Document).limit(limit).offset(offset)
    result = await session.execute(stmt)
    return result.scalars().all()

async def search_documents(session: AsyncSession, keyword: str):
    stmt = select(Document).where(
        Document.text.icontains(keyword)
    )
    result = await session.execute(stmt)
    return result.scalars().all()
```

1.2 Vector Search Integration

Technology: FAISS (Facebook AI Similarity Search)

Implementation:

```
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

class VectorSearcher:
    def __init__(self, model_name: str = "all-MiniLM-L6-v2"):
        self.encoder = SentenceTransformer(model_name)
        self.index = None
        self.passage_map = {}

    def build_index(self, passages: List[str]):
        """Create FAISS index from passages"""
        embeddings = self.encoder.encode(passages, convert_to_numpy=True)

        # Create FAISS index
        dimension = embeddings.shape[1]
        self.index = faiss.IndexFlatL2(dimension)
        self.index.add(embeddings.astype('float32'))

        # Map index positions to passages
        for i, passage in enumerate(passages):
            self.passage_map[i] = passage

    def search(self, query: str, top_k: int = 5):
        """Search for similar passages"""
        query_embedding = self.encoder.encode([query], convert_to_numpy=True)
        distances, indices = self.index.search(query_embedding.astype('float32'), top_k)

        results = [self.passage_map[idx] for idx in indices[0]]
        return results
```

Advantages:

- Fast similarity search in high-dimensional space
- Semantic understanding (not just lexical)
- Scales to millions of passages
- GPU acceleration available

1.3 Document Filtering & Metadata

Implementation:

```
# API endpoint with filtering
@router.get("/documents/search")
async def search_documents(
    session: AsyncSession,
    keyword: str = None,
    category: str = None,
    date_from: datetime = None,
    date_to: datetime = None,
    limit: int = 10,
    offset: int = 0
):
    """Search documents with filters"""
    query = select(Document)

    if keyword:
        query = query.where(Document.text.icontains(keyword))
    if category:
        query = query.where(Document.metadata['category'].astext == category)
    if date_from:
        query = query.where(Document.upload_time >= date_from)
    if date_to:
        query = query.where(Document.upload_time <= date_to)

    query = query.limit(limit).offset(offset)
    result = await session.execute(query)
    return result.scalars().all()
```

1.4 Pagination

Implementation:

```
class PaginationResponse(BaseModel):
    items: List[DocumentMetadata]
    total: int
    page: int
    page_size: int
    total_pages: int

@router.get("/documents", response_model=PaginationResponse)
async def list_documents(
    session: AsyncSession,
    page: int = 1,
    page_size: int = 20
):
    """Get paginated document list"""
    offset = (page - 1) * page_size

    # Get total count
    total = await session.scalar(select(func.count(Document.id)))

    # Get page data
    stmt = select(Document).offset(offset).limit(page_size)
    documents = await session.execute(stmt)

    return PaginationResponse(
        items=[DocumentMetadata.from_orm(d) for d in documents.scalars()],
        total=total,
        page=page,
        page_size=page_size,
        total_pages=(total + page_size - 1) // page_size
    )
```

1.5 Document Batching

Implementation:

```
class BatchProcessor:
    async def process_batch(self, documents: List[str], chunk_size: int = 10):
        """Process large document set in batches"""
        for i in range(0, len(documents), chunk_size):
            batch = documents[i:i+chunk_size]
            results = await asyncio.gather(*[
                self.process_document(doc) for doc in batch
            ])
            yield results
```

Expected Improvements

- **Scalability**: From 500 to 1M+ documents
- **Speed**: Vector search ~10ms vs TF-IDF 100ms
- **Flexibility**: Filter by date, category, source
- **Pagination**: Handle large result sets
- **Analytics**: Query logging and statistics

---

2. Real-Time Document Updates and Indexing

Current Limitations

- Batch processing only (synchronous)
- No incremental updates
- Full re-indexing on changes
- Blocks on large documents

Enhancement Strategy

2.1 Asynchronous Task Queue

Technology: Celery + Redis

Setup:

```
from celery import Celery

# Initialize Celery
celery_app = Celery(
    'doc-qa',
    broker='redis://localhost:6379/0',
    backend='redis://localhost:6379/0'
)

# Define async tasks
@celery_app.task
async def process_document_async(file_path: str, filename: str):
    """Process document asynchronously"""
    try:
        doc_id, text_length, num_passages = indexer.add_document(file_path, filename)

        # Update vector index
        passages = indexer.get_document_passages(doc_id)
        vector_index.add_passages(passages)

        # Log success
        await log_processing(doc_id, "success")
        return {"doc_id": doc_id, "status": "completed"}
    except Exception as e:
        await log_processing(file_path, "error", str(e))
        raise
```

API Integration:

```
@router.post("/documents/upload-async")
async def upload_document_async(file: UploadFile):
    """Upload document asynchronously"""
    # Save file
    file_path = save_temporary_file(file)

    # Queue processing task
    task = process_document_async.delay(file_path, file.filename)

    return {
        "message": "Document queued for processing",
        "task_id": task.id,
        "filename": file.filename
    }

@router.get("/documents/upload-status/{task_id}")
async def get_upload_status(task_id: str):
    """Check document upload status"""
    from celery.result import AsyncResult

    task = AsyncResult(task_id)
    return {
        "task_id": task_id,
        "status": task.status,
        "result": task.result if task.ready() else None
    }
```

2.2 Incremental Indexing

Implementation:

```
class IncrementalIndexer:
    def __init__(self, vector_index, db_session):
        self.index = vector_index
        self.db = db_session

    async def add_passages_to_index(self, doc_id: str, passages: List[str]):
        """Add new passages without re-indexing entire collection"""
        # Get embeddings for new passages only
        embeddings = self.encoder.encode(passages)

        # Add to FAISS index
        self.index.add(embeddings)

        # Update database
        for passage in passages:
            await self.db.add(Passage(
                doc_id=doc_id,
                text=passage,
                embedding=embeddings[i].tolist()
            ))

        await self.db.commit()

    async def update_document(self, doc_id: str, new_text: str):
        """Update document with new text"""
        # Get old passages to remove
        old_passages = await self.db.execute(
            select(Passage).where(Passage.doc_id == doc_id)
        )

        # Remove old passages from index
        for old_passage in old_passages.scalars():
            self.index.remove_ids([old_passage.id])

        # Add new passages
        new_passages = split_into_passages(new_text)
        await self.add_passages_to_index(doc_id, new_passages)
```

2.3 Cache Invalidation

Implementation:

```
from redis import Redis

class CacheManager:
    def __init__(self, redis_client: Redis):
        self.cache = redis_client

    async def invalidate_document_cache(self, doc_id: str):
        """Invalidate cache for updated document"""
        # Clear document-specific cache
        await self.cache.delete(f"doc:{doc_id}")
        await self.cache.delete(f"passages:{doc_id}")

        # Clear query cache for this document
        pattern = f"query:*:{doc_id}"
        keys = await self.cache.keys(pattern)
        if keys:
            await self.cache.delete(*keys)

    async def cache_query_result(self, question: str, doc_ids: List[str], result):
        """Cache query results"""
        key = f"query:{hash(question)}:{':'.join(doc_ids)}"
        await self.cache.setex(
            key,
            3600,  # 1 hour TTL
            json.dumps(result)
        )
```

2.4 Webhook-Based Updates

Implementation:

```
@router.post("/webhooks/document-updated")
async def webhook_document_updated(payload: WebhookPayload):
    """Handle external document updates via webhook"""
    # Queue update task
    update_task = update_document_async.delay(
        doc_id=payload.doc_id,
        new_content=payload.content
    )

    return {"task_id": update_task.id, "status": "queued"}
```

Expected Improvements

- **Responsiveness**: Non-blocking uploads
- **Efficiency**: Incremental indexing saves time

- **Scalability**: Multiple workers process documents in parallel
- **Real-time**: Documents searchable immediately after upload
- **Monitoring**: Task status tracking and monitoring

---

3. Complex Multi-Hop Questions

Current Limitations

- Single-hop retrieval only
- No reasoning across documents
- No entity linking
- No multi-stage reasoning

Enhancement Strategy

3.1 Multi-Hop Retrieval

Implementation:

```
class MultiHopQAEngine:
    def __init__(self, qa_model, vector_index):
        self.qa = qa_model
        self.retriever = vector_index
        self.entity_linker = EntityLinker()

    async def process_multihop_question(self, question: str, max_hops: int = 3):
        """Process questions requiring multiple retrieval steps"""
        # Step 1: Initial retrieval
        initial_context = self.retriever.search(question, top_k=3)
        initial_answer = self.qa.answer_question(question, initial_context[0])

        # Extract entities from answer
        entities = self.entity_linker.extract_entities(initial_answer['answer'])

        # Step 2: Follow-up retrieval
        follow_up_questions = self._generate_follow_up_questions(question, entities)

        all_contexts = []
        for follow_up_q in follow_up_questions:
            contexts = self.retriever.search(follow_up_q, top_k=2)
            all_contexts.extend(contexts)

        # Step 3: Final answer synthesis
        combined_context = "\n".join(all_contexts)
        final_answer = self.qa.answer_question(question, combined_context)

        return {
            "answer": final_answer['answer'],
            "hops": len(follow_up_questions) + 1,
            "reasoning_path": {
                "initial_answer": initial_answer['answer'],
                "follow_ups": follow_up_questions,
                "final_answer": final_answer['answer']
            }
        }

    def _generate_follow_up_questions(self, original_q: str, entities: List[str]):
        """Generate follow-up questions based on entities"""
        follow_ups = [
            f"What is {entity}?" for entity in entities
        ]
        return follow_ups
```

3.2 Entity Linking

Technology: spaCy + Entity Linking models

Implementation:

```
import spacy
from spacy import displacy

class EntityLinker:
    def __init__(self):
        self.nlp = spacy.load("en_core_web_md")

    def extract_entities(self, text: str):
        """Extract named entities from text"""
        doc = self.nlp(text)
        entities = []

        for ent in doc.ents:
            entities.append({
                "text": ent.text,
                "type": ent.label_,
                "start": ent.start_char,
                "end": ent.end_char
            })

        return entities

    def link_entities_to_documents(self, entities: List[str], documents: List[str]):
        """Find which documents contain entity mentions"""
        entity_doc_mapping = {}

        for entity in entities:
            matching_docs = []
            for doc in documents:
                if entity.lower() in doc.lower():
                    matching_docs.append(doc)
            entity_doc_mapping[entity] = matching_docs

        return entity_doc_mapping
```

3.3 Knowledge Graph Construction

Implementation:

```python
from networkx import DiGraph
import networkx as nx

class KnowledgeGraph:
    def __init__(self):
        self.graph = DiGraph()

    def build_from_passages(self, passages: List[str]):
        """Build knowledge graph from document passages"""
        for passage in passages:
            # Extract entities and relations
            entities = self.extract_entities(passage)
            relations = self.extract_relations(passage)

            # Add to graph
            for entity in entities:
                self.graph.add_node(entity['text'], type=entity['type'])

            for rel in relations:
                self.graph.add_edge(rel['subject'], rel['object'],
                                    relation=rel['type'])

    def find_paths(self, source: str, target: str, max_length: int = 3):
        """Find paths between entities"""
        try:
            paths = list(nx.all_simple_paths(
                self.graph, source, target, cutoff=max_length
            ))
            return paths
        except nx.NetworkXNoPath:
            return []

    def extract_relations(self, text: str):
        """Extract subject-predicate-object relations"""
        # Use dependency parsing
        doc = self.nlp(text)
        relations = []

        for token in doc:
            if token.dep_ == "ROOT":
                subject = token.head
                for child in token.children:
                    relations.append({
                        "subject": subject.text,
                        "type": token.lemma_,
                        "object": child.text
                    })

        return relations
```

3.4 Reasoning Chain Generation

Implementation:

```python
class ReasoningEngine:
    def __init__(self, qa_engine, knowledge_graph):
        self.qa = qa_engine
        self.kg = knowledge_graph

    async def reason_with_chain_of_thought(self, question: str):
        """Generate chain of thought reasoning"""
        reasoning_steps = []

        # Step 1: Question decomposition
        sub_questions = self._decompose_question(question)

        # Step 2: Solve each sub-question
        intermediate_answers = {}
        for i, sub_q in enumerate(sub_questions):
            answer = await self.qa.answer_question(sub_q)
            intermediate_answers[sub_q] = answer

            reasoning_steps.append({
                "step": i + 1,
                "question": sub_q,
                "answer": answer
            })

        # Step 3: Synthesize final answer
        synthesis_context = "\n".join(
            f"Q: {q}\nA: {ans}"
            for q, ans in intermediate_answers.items()
        )

        final_answer = await self.qa.answer_question(
            question,
            synthesis_context
        )

        return {
            "question": question,
            "reasoning_chain": reasoning_steps,
            "final_answer": final_answer,
            "confidence": sum(
                r["answer"]["score"] for r in reasoning_steps
            ) / len(reasoning_steps)
        }

    def _decompose_question(self, question: str) -> List[str]:
        """Decompose complex question into sub-questions"""
        # Use question word patterns
        patterns = {
            "how_many": "What is the count of",
            "why": "What is the reason for",
            "when": "What is the date of"
        }

        sub_questions = []
        # Add decomposition logic
        return sub_questions
```

Expected Improvements

- **Reasoning**: Multi-step reasoning across documents
- **Completeness**: Answers synthesized from multiple sources
- **Explainability**: Reasoning chain shown to user
- **Accuracy**: Better answers for complex questions
- **Coverage**: Handle 50-70% more question types

---

4. Implementation Roadmap

Phase 1: Foundation (Weeks 1-2)

- [ ] Set up PostgreSQL database
- [ ] Implement database models
- [ ] Add async/await support
- [ ] Migrate in-memory storage to DB

Phase 2: Vector Search (Weeks 3-4)

- [ ] Integrate FAISS
- [ ] Add sentence transformers
- [ ] Build vector index
- [ ] Replace TF-IDF retrieval

Phase 3: Asynchronous Processing (Weeks 5-6)

- [ ] Set up Celery + Redis
- [ ] Implement async document processing

- [ ] Add cache management
- [ ] Build task monitoring UI

Phase 4: Advanced QA (Weeks 7-8)

- [ ] Implement entity linking
- [ ] Build knowledge graph
- [ ] Add multi-hop retrieval
- [ ] Implement reasoning chain

Phase 5: Polish & Testing (Week 9)

- [ ] Performance optimization
- [ ] Load testing
- [ ] Documentation
- [ ] Deployment preparation

---

5. Performance Metrics

Current System

- Documents: 100-500
- Latency: 1-3 seconds
- Accuracy: 85-90%
- Throughput: 1-2 req/sec

Enhanced System (Target)

- Documents: 1M+
- Latency: 500-1500ms
- Accuracy: 90-95%
- Throughput: 10-20 req/sec

6. Testing Strategy

Load Testing

```
# Locust load test
from locust import HttpUser, task, between

class QASystemUser(HttpUser):
    wait_time = between(1, 3)

    @task
    def ask_question(self):
        self.client.post("/api/qa/ask", json={
            "question": "What is AI?",
            "top_k": 3
        })
```

Benchmarking

```
# Performance benchmarks
import time

def benchmark_retrieval():
    start = time.time()
    results = vector_index.search("test query", k=10)
    elapsed = time.time() - start
    assert elapsed < 0.1  # 100ms target
```

7. Conclusion

This enhancement plan provides a clear path to scale the Document-Based QA System from handling hundreds of documents to millions, with real-time updates and complex reasoning capabilities. The phased approach allows for incremental improvement while maintaining system stability.

Expected Timeline: 8-9 weeks for full implementation

Resource Requirements: 2-3 developers

Infrastructure: PostgreSQL, Redis, additional compute for async workers

---

Contact

For questions or clarifications, contact: Balakrishnan V S (2024aa05017@wilp.bits-pilani.ac.in)

Last Updated: December 2025