KURSE: Knowledge Utilization, Retreival and Summarization Engine

with Application to BeeKurse E-Commerce Platform

Varshith Kada Vishnu Teja Himesh Anant Siripuru Abhiram
Abhinav Goyal Bhuvan Bhedika Kunjan Manoj

Abstract

KURSE is a neuro-symbolic inference engine combining neural networks with symbolic reasoning through a tri-store architecture (SQL, Vector DB, Knowledge Graph). We present the KG ingestion pipeline with N-ary event representation and demonstrate two applications: *Alibi Breaker* for constraint satisfaction and *BeeKurse* for conversational e-commerce. BeeKurse features a 5-type query classifier, multi-strategy search orchestration (HQ/RQ/SQ paths), and intelligent scoring with property weights (0.5-2.0 scale).

1 Introduction

Modern applications require systems that can both store and reason over unstructured data. KURSE addresses this through a **neuro-symbolic approach**:

- **Neural**: LLMs for entity extraction and semantic parsing
- **Symbolic**: Knowledge graphs with ontologies for structured reasoning

1.1 Tri-Store Architecture

KURSE integrates three storage paradigms:

1. **SQL**: Structured metadata with ACID guarantees
2. **Vector DB (Qdrant)**: Semantic similarity via 1024-dim NVCLIP embeddings
3. **Knowledge Graph (Memgraph)**: Entity relationships with Cypher queries

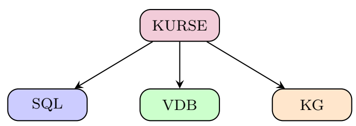


Figure 1: Tri-Store Architecture

2 KURSE Core Pipeline

2.1 Input Processing

Documents undergo OCR (olmOCR/Qwen 7B), then chunking with parameters:

Parameter	Value
CHUNK_SIZE	1000 chars
CHUNK_OVERLAP	200 chars
NVCLIP_DIM	1024

2.2 KG Ingestion Pipeline

The pipeline operates in two phases:

Phase 1 - N-ary Event Identification: Identifies complex events involving 3+ participants or critical metadata (time, location). Events are stored with embeddings for semantic retrieval.

Phase 2 - Relation Extraction: Extracts triplets (subject-predicate-object), verifies against source text, and standardizes to ontology using VDB similarity:

Threshold	Value
TYPE_THRESHOLD	0.25
RELATION_THRESHOLD	0.20

A streaming bridge connects phases via async queues, enabling parallel processing before Memgraph storage.

2.3 Ontology & Reasoning

KURSE supports formal axioms: **transitivity** ($R(x, y) \land R(y, z) \Rightarrow R(x, z)$), **symmetry**, and **domain/range constraints** for validation. See Appendix D for details.

2.4 Applications

KURSE is a at it's core an intelligent data storage hence it's ubiquitously applicable. Let's see some applications to see the potential of what we can do with the KURSE Engine.

Alibi Breaker: Constraint satisfaction to detect contradictions in statements with verified facts or evidence.

Risk Contagion: Identify Non-Obvious risk to the CCC, cash flow operations or stock price with global news integration to identify indirect impacts and system risk.

BeeKurse: A Conversational E commerce Agent specializing in product recommendation

3 BeeKurse: E-Commerce Application

BeeKurse adapts KURSE for conversational product search, dropping strict ontology for flexible VDB-based relations.

3.1 Query Classification

The **Strontium** parser (269-line system prompt) classifies into 5 types:

3.2 SEARCH Query Model

Parsed into `ProductRequest` with:

- **Properties**: (name, value, weight) where weight $\in [0.5, 2.0]$

Type	Purpose
SEARCH	Product search with properties/literals
DETAIL	Info about specific product
CHAT	General conversation
CART_ACTION	Add/remove from cart
CART_VIEW	View cart contents

- **Literals**: Numeric constraints with buffers
- **is_hq**: Hurry Query flag for fast path
- **sort_literal**: For superlatives ("cheapest")

3.3 Search Orchestration

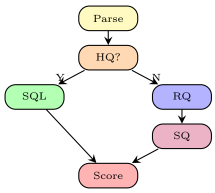


Figure 2: Search Pipeline

HQ Fast Path: Direct SQL for urgent/generic queries.

Property Search (RQ): Dual VDB search (main + property collections) with semantic matching.

Connected Search (SQ): KG traversal via ALSO_BOUGHT, SIMILAR_TO, COMPLEMENTS relations.

3.4 Scoring System

$$\text{score} = \sum_i (sim_i \times w_i) + B_{conn} + B_{subcat}$$

Where $B_{conn} = 0.5$ (connected bonus), $B_{subcat} = 0.4$ (subcategory bonus).

For superlatives: 70% literal ranking + 30% semantic ranking.

4 Conclusion

KURSE demonstrates effective neuro-symbolic integration through tri-store architecture, N-ary event representation, and ontology-driven reasoning. BeeKurse showcases adaptability with sophisticated query classification, multi-strategy search, and intelligent scoring.

References

[1] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, Jose Emilio Labra-Gayo, Roberto Navigli, Sebastian Neumaier, A. Ngonga Moses Ngomo, Axel Polleres, Shanmugalingam Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37, 2021.

Appendices

A Input Processing Details

A.1 OCR with olmOCR

The OCR module uses **olmOCR**, a vision-language model based on Qwen 7B, fine-tuned for:

- Dense text recognition in documents
- Table structure understanding
- Handwritten text interpretation
- Multi-language support

A.2 Document ID Generation

Each document receives a unique identifier combining content hash and timestamp:

```
import hashlib
from datetime import datetime

def generate_document_id(content: bytes) -> str:
    content_hash = hashlib.md5(content).hexdigest()[:8]
    timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
    return f"doc_{content_hash}_{timestamp}"
```

A.3 Chunking Algorithm

The chunking algorithm respects sentence boundaries:

```
def chunk_document(text, chunk_size=1000, overlap=200):
    chunks = []
    start = 0
    while start < len(text):
        end = min(start + chunk_size, len(text))
        if end < len(text):
            # Find sentence boundary
            boundary = find_sentence_boundary(text[start:end])
            if boundary > chunk_size * 0.7:
                end = start + boundary
        chunks.append(text[start:end])
        start = end - overlap
    return chunks
```

B Vector Database Schema

B.1 NVCLIP Embeddings

KURSE uses NVIDIA's NVCLIP model:

- **Model**: nvidia/nvclip
- **Dimension**: 1024
- **Normalization**: L2-normalized for cosine similarity

B.2 Qdrant Point Structure

```
{
  "id": "chunk_abc123_001",
  "vector": [0.123, -0.456, ...],  # 1024 dimensions
  "payload": {
    "document_id": "doc_a1b2c3d4_20240101120000",
    "chunk_id": "chunk_001",
    "chunk_index": 0,
    "page_no": 1,
    "text": "Original␣chunk␣text...",
    "created_at": "2024-01-01T12:00:00Z"
  }
}
```

B.3 Collection Configuration

```
from qdrant_client import QdrantClient
from qdrant_client.models import VectorParams, Distance

client.create_collection(
    collection_name="document_chunks",
    vectors_config=VectorParams(
        size=1024,
        distance=Distance.COSINE
    )
)
```

C Knowledge Graph Fundamentals

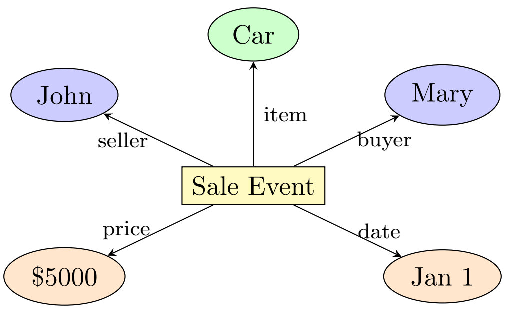
C.1 Node Types

The KURSE knowledge graph has three node categories:

1. **Entities**: Real-world objects (Person, Organization, Location, Event)
2. **Properties**: Attributes (color, size, material, date)
3. **Literals**: Concrete values ("red", 42, "2024-01-01")

C.2 N-ary Relations

Traditional binary triplets are insufficient for complex events. N-ary relations create an **Event Node** connecting all participants:



C.3 Cypher Schema

```
// Create Event Node
CREATE (e:Event {
    id: "event_001",
```

```
    type: "Sale",
    source_chunk: "chunk_abc123_001",
    confidence: 0.95
})

// Connect Participants
MATCH (e:Event {id: "event_001"})
MATCH (seller:Person {name: "John"})
MATCH (buyer:Person {name: "Mary"})
CREATE (e)-[:HAS_SELLER]->(seller)
CREATE (e)-[:HAS_BUYER]->(buyer)
```

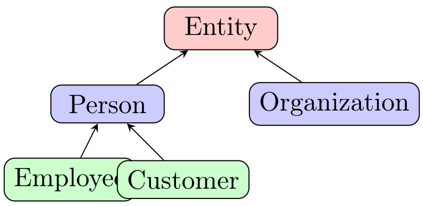
D Ontology & Reasoning

D.1 What is an Ontology?

An **ontology** is a formal, explicit specification of a shared conceptualization:

- **Formal**: Machine-readable with precise semantics
- **Explicit**: Clearly defined concepts and constraints
- **Shared**: Common understanding across components

D.2 Class Hierarchies (subClassOf)



D.3 Ontology Axioms

Transitivity:

$$\forall x, y, z : R(x, y) \land R(y, z) \Rightarrow R(x, z)$$

Example: LOCATED_IN is transitive.

Symmetry:

$$\forall x, y : R(x, y) \Rightarrow R(y, x)$$

Example: SIBLING_OF is symmetric.

Equivalence:

$$A \equiv B \Leftrightarrow (A \sqsubseteq B) \land (B \sqsubseteq A)$$

D.4 First-Order Logic Mappings

D.5 Domain and Range Constraints

```
def validate_relation(subject, predicate, object, ontology):
    """Validate␣relation␣against␣domain/range␣constraints."""
    domain = ontology.get_domain(predicate)
    range_ = ontology.get_range(predicate)

    if not is_instance_of(subject, domain):
        raise ValidationError(f"Domain␣violation")
    if not is_instance_of(object, range_):
```

Ontology Construct	FOL Representation
subClassOf(A, B)	$\forall x : A(x) \Rightarrow B(x)$
domain(R, C)	$\forall x, y : R(x, y) \Rightarrow C(x)$
range(R, C)	$\forall x, y : R(x, y) \Rightarrow C(y)$
transitive(R)	$\forall x, y, z : R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$
symmetric(R)	$\forall x, y : R(x, y) \Rightarrow R(y, x)$

```
        raise ValidationError(f"Range violation")
    return True
```

E KG Ingestion Pipeline (Full)

E.1 Phase 1: N-ary Event Identification

Event Criteria:

1. **Three or more participants**: Multiple entities in single action
2. **Critical metadata**: Temporal, spatial, or quantitative data
3. **Semantic unity**: Participants connected by coherent action

Event Data Structure:

```
@dataclass
class NaryEvent:
    event_id: str
    event_type: str
    participants: List[Participant]
    metadata: Dict[str, Any]
    source_chunk_id: str
    source_document_id: str
    confidence: float
    embedding: List[float]  # 1024-dim

@dataclass
class Participant:
    entity_name: str
    entity_type: str
    role: str  # seller, buyer, witness, etc.
```

Manifest Structure:

```
{
  "document_id": "doc_abc123",
  "chunk_id": "chunk_001",
  "page_no": 1,
  "events_identified": [
    {"event_id": "evt_001", "event_type": "Transaction"}
  ],
  "processing_status": "phase1_complete"
}
```

E.2 Phase 2: Relation Extraction

RawTriplet Structure:

```
class RawTriplet(BaseModel):
    subject: str           # Entity name
    subject_type: str      # Raw type from LLM
```

```
    predicate: str          # Relation name
    object: str             # Entity or literal
    object_type: str        # Raw type from LLM
    confidence: float       # Extraction confidence
    source_span: Optional[str]
```

Verification Against Source:

```
def verify_triplet(triplet: RawTriplet, source_text: str) -> bool:
    """Verify triplet is supported by source text."""
    verification_prompt = f"""
    Source text: {source_text}
    Claimed fact: {triplet.subject} {triplet.predicate} {triplet.object}
    Is this fact directly supported by the source text?
    """
    response = llm.generate(verification_prompt)
    return response.startswith("YES")
```

Standardization to Ontology:

```
TYPE_THRESHOLD = 0.25
RELATION_THRESHOLD = 0.20

def standardize_type(raw_type: str, types_vdb) -> str:
    """Match raw type to ontology using VDB."""
    embedding = encoder.encode(raw_type)
    results = types_vdb.search(
        collection_name="entity_types",
        query_vector=embedding,
        limit=1
    )
    if results[0].score >= TYPE_THRESHOLD:
        return results[0].payload["type_name"]
    return "Unknown"
```

E.3 Streaming Bridge Architecture

```
class StreamingBridge:
    def __init__(self, num_workers: int = 4):
        self.queue = Queue()
        self.num_workers = num_workers

    async def phase1_producer(self, chunks):
        for chunk in chunks:
            events = await identify_nary_events(chunk)
            manifest = create_manifest(chunk, events)
            await self.queue.put((chunk, manifest))
        for _ in range(self.num_workers):
            await self.queue.put(None)

    async def phase2_worker(self, worker_id):
        while True:
            item = await self.queue.get()
            if item is None: break
            chunk, manifest = item
            triplets = await extract_relations(chunk, manifest)
            await store_to_memgraph(triplets)
```

E.4 Memgraph Storage

```
// Store Entity (idempotent)
MERGE (e:Entity {name: $name})
ON CREATE SET
    e.type = $type,
    e.created_at = timestamp(),
    e.source_documents = [$document_id]
ON MATCH SET
    e.source_documents = e.source_documents + $document_id
```

```
// Store Relation
MATCH (s:Entity {name: $subject})
MATCH (o:Entity {name: $object})
MERGE (s)-[r:$predicate]->(o)
ON CREATE SET
    r.source_chunk = $chunk_id,
    r.confidence = $confidence
```

F Verification System

F.1 Chunk Verification Pipeline

```
async def verify_chunk_extractions(chunk, triplets):
    """Verify␣extracted␣triplets␣against␣source␣chunk."""
    verified = []
    for triplet in triplets:
        prompt = f"""
␣␣␣␣␣␣␣␣Source␣text:␣{chunk}
␣␣␣␣␣␣␣␣Verify␣this␣extracted␣fact:
␣␣␣␣␣␣␣␣Subject:␣{triplet.subject}␣(type:␣{triplet.subject_type})
␣␣␣␣␣␣␣␣Relation:␣{triplet.predicate}
␣␣␣␣␣␣␣␣Object:␣{triplet.object}␣(type:␣{triplet.object_type})

␣␣␣␣␣␣␣␣Is␣this␣fact:
␣␣␣␣␣␣␣␣1.␣Explicitly␣stated␣in␣the␣text?
␣␣␣␣␣␣␣␣2.␣Reasonably␣inferable␣from␣the␣text?
␣␣␣␣␣␣␣␣3.␣Not␣supported␣by␣the␣text?
␣␣␣␣␣␣␣␣"""
        response = await llm.generate(prompt)
        if response.startswith("1") or response.startswith("2"):
            verified.append(triplet)
    return verified
```

F.2 Ontology Feasibility Checking

```
def check_ontology_feasibility(triplet, ontology):
    """Check␣if␣triplet␣is␣feasible␣under␣ontology."""
    if triplet.predicate in ontology.relations:
        expected_domain = ontology.get_domain(triplet.predicate)
        if not ontology.is_subclass(triplet.subject_type, expected_domain):
            return False, f"Domain␣violation"
        expected_range = ontology.get_range(triplet.predicate)
        if not ontology.is_subclass(triplet.object_type, expected_range):
            return False, f"Range␣violation"
    return True, "Valid"
```

G Query Classification Details

G.1 Query Type Output Formats

SEARCH Query Output:

```
{
  "query_type": "SEARCH",
  "product_request": {
    "product_category": "dress",
    "properties": [
      {"name": "color", "value": "red", "weight": 1.5}
    ],
    "literals": [
      {"name": "price", "value": 50, "operator": "<=", "buffer": 7.5}
    ],
```

```
    "is_hq": false,
    "sort_literal": null
  }
}
```

DETAIL Query Output:

```
{
  "query_type": "DETAIL",
  "product_id": "123",
  "question": "What material is this made of?"
}
```

CART_ACTION Output:

```
{
  "query_type": "CART_ACTION",
  "action": "add",
  "product_id": "123",
  "quantity": 2
}
```

G.2 ProductRequest Model

```
class Property(BaseModel):
    name: str
    value: str
    weight: float = 1.0  # 0.5 to 2.0 scale

class Literal(BaseModel):
    name: str
    value: float
    operator: str  # =, <, <=, >, >=, between
    buffer: float = 0.0

class ProductRequest(BaseModel):
    product_category: str
    product_subcategory: Optional[str] = None
    properties: List[Property] = []
    literals: List[Literal] = []
    is_hq: bool = False
    sort_literal: Optional[str] = None
```

G.3 Property Weight Scale

Weight	Level	Indicators
2.0	Critical	"must be", "definitely", "nothing else"
1.5	Important	Direct mention with emphasis
1.0	Standard	Simple mention
0.5	Nice-to-have	"preferably", "if possible"

H Search Orchestration Details

H.1 HQ Fast Path

What: Direct SQL lookup bypassing semantic search.

When:

- User indicates urgency ("show me any", "quick")

- Generic queries without specific requirements
- Re-queries after failed detailed search

Why: Sub-100ms response, no embedding computation needed.

```
async def hq_fast_path(request):
    query = """
␣␣␣␣SELECT␣*␣FROM␣products
␣␣␣␣WHERE␣category␣=␣:category␣AND␣status␣=␣'active'
␣␣␣␣ORDER␣BY␣popularity␣DESC␣LIMIT␣20
␣␣␣␣"""
    return await sql_client.fetch_all(
        query, {"category": request.product_category}
    )
```

H.2 Property Search (RQ)

What: Dual-path semantic search using Main VDB and Property VDB.

When: User specifies properties (color, material, style).

Why: Handles synonyms ("crimson" → "red"), semantic similarity.

```
async def property_search(request):
    results = []
    # Path 1: Main VDB search
    query_embedding = encoder.encode(request.to_search_string())
    main_results = await main_vdb.search(
        collection_name="products",
        query_vector=query_embedding, limit=50
    )
    # Path 2: Property VDB search
    for prop in request.properties:
        prop_embedding = encoder.encode(f"{prop.name}:␣{prop.value}")
        prop_results = await property_vdb.search(
            collection_name="product_properties",
            query_vector=prop_embedding, limit=30
        )
        for r in prop_results:
            r.score *= prop.weight
        results.extend(prop_results)
    return merge_results(main_results, results)
```

H.3 Connected Search (SQ)

What: Knowledge graph traversal for related products.

When: Recommendations, "similar to" queries, complementary products.

Why: Discovers non-obvious connections via co-purchase patterns.

```
async def connected_search(seed_products, relation_types=None):
    if relation_types is None:
        relation_types = ["ALSO_BOUGHT", "SIMILAR_TO", "COMPLEMENTS"]
    query = """
␣␣␣␣MATCH␣(seed:Product)-[r]->(connected:Product)
␣␣␣␣WHERE␣seed.id␣IN␣$seed_ids␣AND␣type(r)␣IN␣$relations
␣␣␣␣RETURN␣connected,␣type(r)␣as␣relation,␣r.strength
␣␣␣␣ORDER␣BY␣r.strength␣DESC␣LIMIT␣30
␣␣␣␣"""
    return await memgraph.execute(query, {
        "seed_ids": seed_products, "relations": relation_types
    })
```

H.4 Subcategory Scoring

```python
def apply_subcategory_bonus(products, target_subcategory):
    target_embedding = encoder.encode(target_subcategory)
    for product in products:
        subcat_embedding = encoder.encode(product.subcategory)
        similarity = cosine_similarity(target_embedding, subcat_embedding)
        bonus = similarity * SUBCATEGORY_MAX_BONUS  # 0.4
        product.score += bonus
    return products
```

I Weighting System

I.1 Property Weight Formula

$$score_{property} = similarity \times weight$$

Where similarity is cosine similarity (0-1) and weight is user importance (0.5-2.0).

I.2 Score Combination Formula

$$final\_score = \sum_i (property\_scores_i) + connected\_bonus + subcategory\_bonus$$

```python
def calculate_final_score(product, property_scores,
                          connected_bonus, subcategory_bonus,
                          literal_results):
    base_score = sum(property_scores.values())
    total_score = base_score + connected_bonus + subcategory_bonus

    for literal_name, passed in literal_results.items():
        if not passed:
            return -1  # Filter out
    return total_score
```

I.3 Bonus Values

Factor	Value	Description
CONNECTED_BONUS	+0.5	Product found via KG traversal
SUBCATEGORY_MAX_BONUS	+0.4	Max subcategory similarity bonus
EXACT_MATCH_BONUS	+0.25	Exact property value match
LITERAL_FAIL	Filter out	Product doesn't meet constraint

I.4 Superlative Re-ranking (70/30 Split)

```python
def rerank_for_superlative(products, sort_literal, ascending):
    # Sort by literal value
    sorted_by_literal = sorted(
        products, key=lambda p: p.product[sort_literal],
        reverse=not ascending
    )
    literal_ranks = {p.product.id: i for i, p in enumerate(sorted_by_literal)}
    semantic_ranks = {p.product.id: i for i, p in enumerate(products)}

    # Combine: 70% literal, 30% semantic
```

```
    for product in products:
        pid = product.product.id
        combined_rank = 0.7 * literal_ranks[pid] + 0.3 * semantic_ranks[pid]
        product.final_rank = combined_rank
    return sorted(products, key=lambda p: p.final_rank)
```

J User Context Enrichment

J.1 Three-Stage Pipeline

1. **Parse**: Extract user preferences from profile
2. **Filter**: Remove irrelevant preferences for current query
3. **Enrich**: Merge relevant preferences into query with low weight

J.2 User Context Model

```
class UserContext(BaseModel):
    user_id: str
    gender: Optional[str] = None
    size_preferences: Dict[str, str] = {}  # category -> size
    age_group: Optional[str] = None
    style_preferences: List[str] = []
    price_range: Optional[Tuple[float, float]] = None
    brand_preferences: List[str] = []
    recent_searches: List[str] = []
    viewed_products: List[str] = []
    purchased_products: List[str] = []
```

J.3 Property Merging Strategy

```
def merge_user_context(request, context):
    enriched = request.copy()

    # Add gender if relevant and not specified
    if context.gender and not has_property(request, "gender"):
        if is_gendered_category(request.product_category):
            enriched.properties.append(
                Property(name="gender", value=context.gender, weight=0.5)
            )

    # Add size preference
    if request.product_category in context.size_preferences:
        if not has_property(request, "size"):
            enriched.properties.append(Property(
                name="size",
                value=context.size_preferences[request.product_category],
                weight=0.5
            ))

    # Add style preferences with low weight
    for style in context.style_preferences[:2]:
        if not has_property(request, "style"):
            enriched.properties.append(
                Property(name="style", value=style, weight=0.3)
            )

    return enriched
```