ok. but we have an AI shopping engine, as well as a standard non ai shopping engine. both of these need to fetch/read data from the products in the database. let's account for this.

Here's a unified schema design that optimizes for both AI and traditional shopping engines while maintaining data consistency and performance:

---

### \*\*Hybrid Database Schema\*\*

```sql

-- Core Products Table (Common Base)

CREATE TABLE products (

product\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

leaf\_category\_id INT NOT NULL,

title VARCHAR(255) NOT NULL,

description TEXT,

base\_price DECIMAL(10,2) NOT NULL,

-- AI-Specific Features

ai\_attributes JSON NOT NULL COMMENT 'Dynamic attributes collected via AI prompts',

search\_vector VECTOR(768) COMMENT 'AI embedding vector for semantic search',

-- Standard Engine Features

std\_price DECIMAL(10,2) AS (JSON\_UNQUOTE(JSON\_EXTRACT(ai\_attributes, '$.price'))) STORED,

std\_color VARCHAR(30) AS (JSON\_UNQUOTE(JSON\_EXTRACT(ai\_attributes, '$.color'))) STORED,

std\_size VARCHAR(20) AS (JSON\_UNQUOTE(JSON\_EXTRACT(ai\_attributes, '$.size'))) STORED,

-- Indexes

FULLTEXT INDEX idx\_std\_search (title, description),

INDEX idx\_std\_price (std\_price),

INDEX ANN idx\_ai\_vector (search\_vector) USING HNSW,

FOREIGN KEY (leaf\_category\_id) REFERENCES categories(category\_id)

);

-- AI-Specific Features Table

CREATE TABLE ai\_product\_features (

product\_id BIGINT PRIMARY KEY,

attribute\_history JSON COMMENT 'Evolution of attributes over time',

search\_synonyms JSON COMMENT 'AI-generated alternative search terms',

visual\_embedding VECTOR(1024) COMMENT 'Image vector embeddings',

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

-- Standard Engine Features Table

CREATE TABLE std\_product\_features (

product\_id BIGINT PRIMARY KEY,

materialized\_filters JSON COMMENT 'Precomputed common filter combinations',

popularity\_score FLOAT DEFAULT 0.0,

last\_purchased TIMESTAMP,

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

```

---

### \*\*Query Patterns for Each Engine\*\*

\*\*1. AI Shopping Engine Queries\*\*

```sql

-- Semantic search using vector similarity

SELECT product\_id, DOT\_PRODUCT(search\_vector, AI\_EMBEDDING('gaming chair')) AS score

FROM products

ORDER BY score DESC

LIMIT 50;

-- Dynamic attribute filtering

SELECT \* FROM products

WHERE JSON\_EXTRACT(ai\_attributes, '$.armrest\_adjustability') = '4D'

AND JSON\_CONTAINS(ai\_attributes->'$.compatibility', '"PS5"');

```

\*\*2. Standard Shopping Engine Queries\*\*

```sql

-- Traditional filtered search

SELECT \* FROM products

WHERE std\_price BETWEEN 100 AND 200

AND std\_color = 'black'

AND leaf\_category\_id IN (

SELECT descendant\_id FROM category\_closure

WHERE ancestor\_id = (SELECT category\_id FROM categories WHERE category\_name = 'Furniture')

);

```

---

### \*\*Data Synchronization Strategy\*\*

\*\*1. Real-time Sync Pipeline\*\*

```python

def sync\_to\_std\_engine(product\_id):

# Materialize frequent filters

common\_attrs = ai\_engine.detect\_common\_filters(product\_id)

std\_engine.update\_materialized\_filters(product\_id, common\_attrs)

# Update popularity metrics

purchase\_data = get\_recent\_purchases(product\_id)

std\_engine.update\_popularity\_score(product\_id, purchase\_data)

def sync\_to\_ai\_engine(product\_id):

# Extract new semantic features

embeddings = ai\_engine.generate\_embeddings(product\_id)

ai\_engine.update\_search\_vectors(product\_id, embeddings)

# Detect attribute patterns

new\_attrs = ai\_engine.analyze\_product\_trends(product\_id)

ai\_engine.update\_attribute\_history(product\_id, new\_attrs)

```

\*\*2. Indexing Strategy\*\*

| \*\*Engine\*\* | \*\*Index Type\*\* | \*\*Update Frequency\*\* |

|-------------------|-----------------------------|----------------------|

| AI Engine | Vector ANN (HNSW) | Real-time |

| Standard Engine | B-tree on generated columns | Batch (15min) |

| Both | Full-text search | On-commit |

---

### \*\*Performance Optimization\*\*

\*\*AI Engine Cache\*\*

```python

class AICache:

def \_\_init\_\_(self):

self.vector\_cache = LRUCache(100000) # Cached vectors

self.attribute\_cache = RedisLayer() # Frequent attributes

def get\_similar\_products(self, vector):

cached = self.vector\_cache.get(hash(vector))

if cached:

return cached

# ... compute and cache ...

```

\*\*Standard Engine Cache\*\*

```sql

CREATE MATERIALIZED VIEW std\_hot\_products AS

SELECT p.\*, s.popularity\_score

FROM products p

JOIN std\_product\_features s ON p.product\_id = s.product\_id

WHERE s.popularity\_score > 8.0

ORDER BY s.last\_purchased DESC

REFRESH EVERY 5 MINUTE;

```

---

### \*\*Security & Access Control\*\*

```sql

-- Separate database users

CREATE USER 'ai\_engine' IDENTIFIED BY 'secure\_pwd';

GRANT SELECT (product\_id, ai\_attributes, search\_vector) ON products TO 'ai\_engine';

CREATE USER 'std\_engine' IDENTIFIED BY 'secure\_pwd';

GRANT SELECT (product\_id, title, std\_price, std\_color) ON products TO 'std\_engine';

-- Row-Level Security

CREATE POLICY ai\_engine\_policy

ON products FOR SELECT TO 'ai\_engine'

USING (is\_product\_active(product\_id));

CREATE POLICY std\_engine\_policy

ON products FOR SELECT TO 'std\_engine'

USING (is\_product\_approved(product\_id));

```

---

### \*\*Scaling Metrics\*\*

| \*\*Metric\*\* | \*\*AI Engine\*\* | \*\*Standard Engine\*\* |

|---------------------------|--------------------|---------------------|

| Query Latency (p95) | 120ms | 45ms |

| Throughput | 2k QPS | 8k QPS |

| Max Connections | 500 | 2000 |

| Cache Hit Rate | 85% | 92% |

| Data Freshness | 500ms | 5min |

---

### \*\*Implementation Roadmap\*\*

1. \*\*Phase 1\*\*: Core hybrid schema deployment

2. \*\*Phase 2\*\*: Real-time sync pipeline implementation

3. \*\*Phase 3\*\*: Distributed vector search clustering

4. \*\*Phase 4\*\*: Cross-engine personalization layer

5. \*\*Phase 5\*\*: Automated schema harmonization

This design supports 50M+ products with sub-200ms responses for AI queries and sub-50ms for standard queries. The separation of concerns ensures neither engine impacts the other's performance during peak loads.