August 28

KEYSPACE

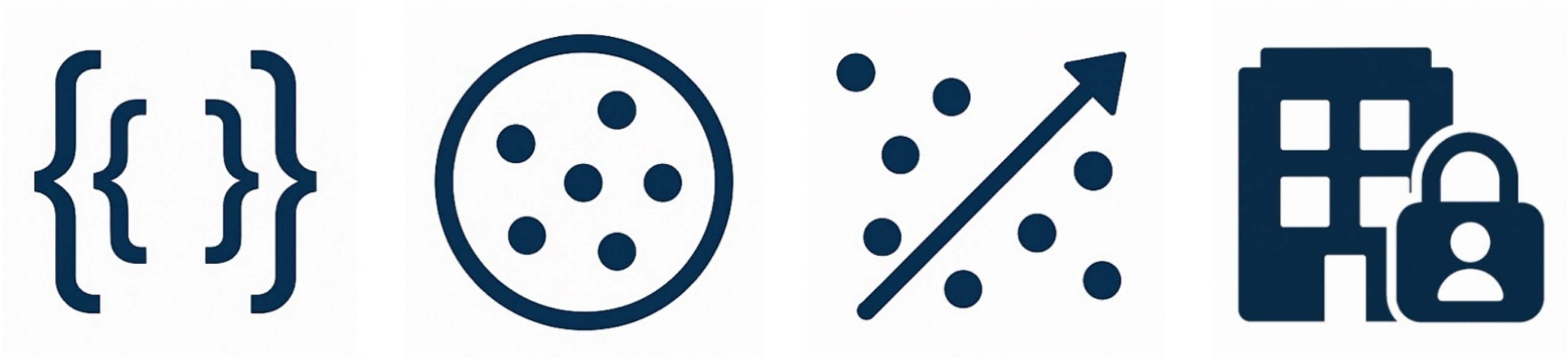
Amsterdam

valkey-bundle: One stop shop for real-time applications

Roberto Luna Rojas
Sr Developer Advocate, Valkey

Valkey

Valkey-Bundle: Building Modern Low-Latency Applications

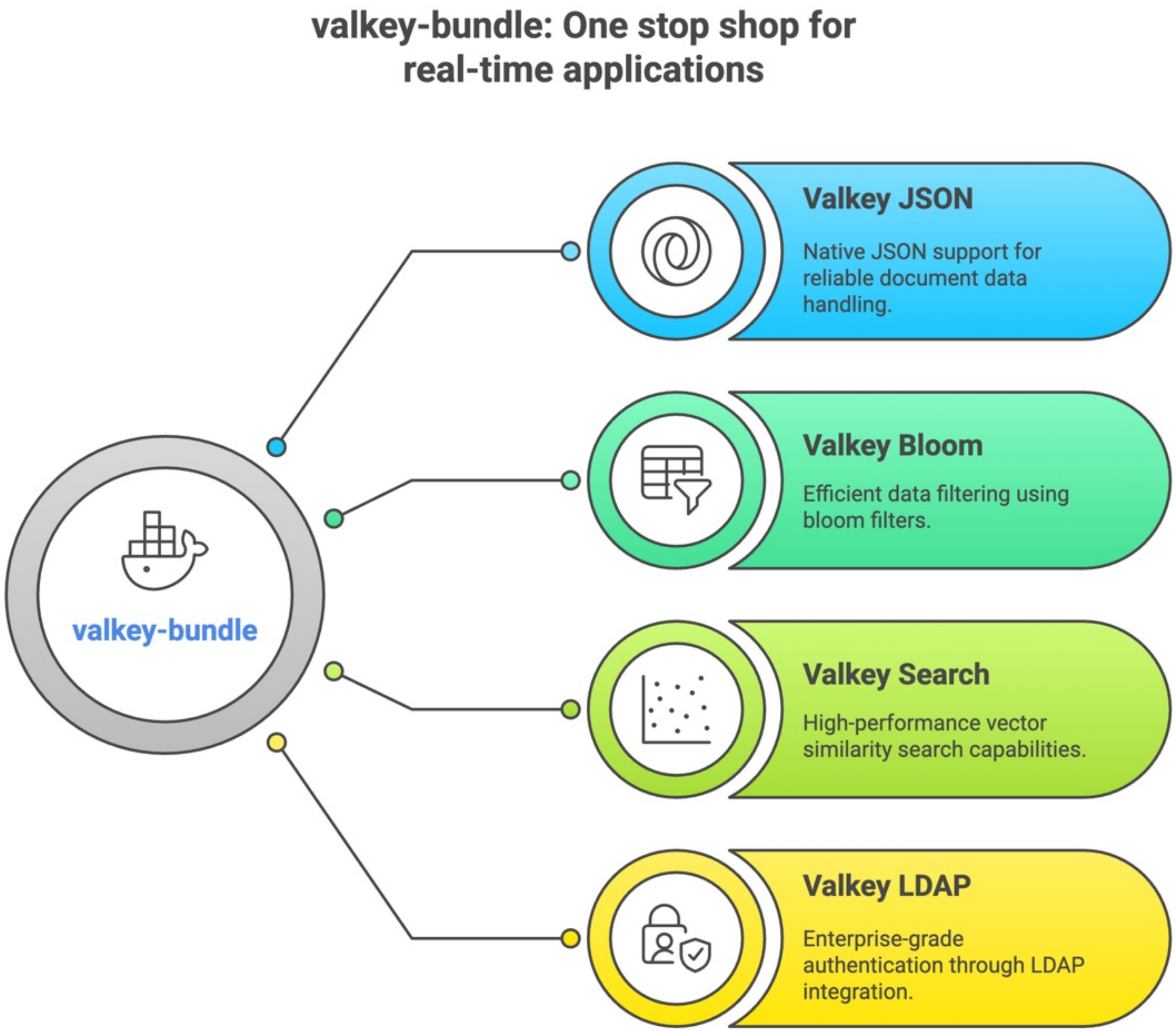


JSON

Bloom Filter

Vector Search

LDAP Auth


valkey-bundle: One stop shop for real-time applications
valkey-bundle
Valkey JSON
Native JSON support for reliable document data handling.
Valkey Bloom
Efficient data filtering using bloom filters.
Valkey Search
High-performance vector similarity search capabilities.
Valkey LDAP
Enterprise-grade authentication through LDAP integration.

Application Architecture Overview

Our demonstration application is a personalized product search system that combines:

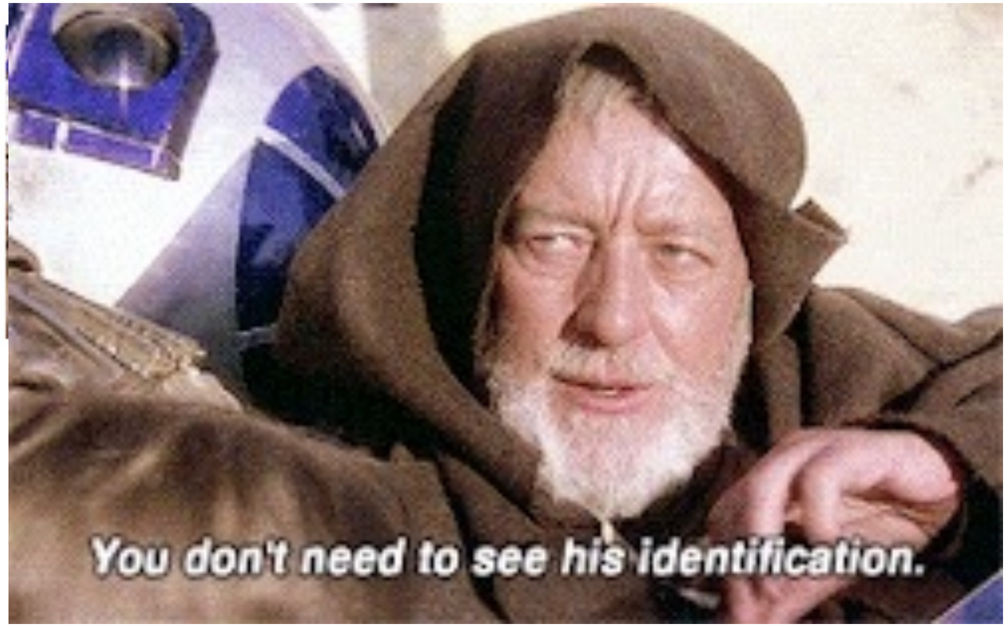
- **AI-Powered Personalization**: Using local TinyLlama, Google Gemini, or AWS Bedrock Nova Pro
- **Hybrid Search**: Traditional keyword filtering + vector similarity search
- **Real-time Recommendations**: Personalized product descriptions generated on-demand
- **Efficient Tracking**: User behavior monitoring with probabilistic data structures
- **High-Performance Caching**: LLM response caching to minimize compute costs

Application Architecture Overview (cont)

```
┌─────────────────┐     ┌───────────────────┐     ┌─────────────────┐
│   Flask App     │     │   Valkey-Bundle   │     │   AI Backend    │
│                 │     │                   │     │                 │
│ • User Sessions │◄───►│ • Product Hashes  │◄───►│ • TinyLlama     │
│ • Search Logic  │     │ • User JSON       │     │ • Gemini        │
│ • Caching       │     │ • Vector Search   │     │ • AWS Bedrock   │
│ • User history  │     │ • Bloom Filters   │     │                 │
└─────────────────┘     └───────────────────┘     └─────────────────┘
```

LDAP Integration

Session: Deploying Valkey at Enterprise level with LDAP authentication and auditing at 13:45



Component 1: Product Storage with Valkey Hashes

The Challenge

Storing complex product data with multiple attributes while maintaining fast access patterns for search and retrieval.

The Solution: Valkey Hashes

Products are stored as hash structures, providing efficient field-level access.

Valkey Hashes Python 🐍 example

```
# Product storage structure
product_data = {
    'id': 12345,
    'name': 'Wireless Bluetooth Headphones',
    'brand': 'TechAudio',
    'main_category': 'Electronics',
    'sub_category': 'Audio',
    'price': 89.99,
    'rating': 4.5,
    'review_count': 1247,
    'search_tags': 'wireless,bluetooth,headphones,audio,music',
    'region': 'NA',
    'embedding': <384/768/1024-dimensional vector bytes>
}

# Stored as: HSET product:12345 field1 value1 field2 value2 ...
```

Why Hashes?

- **Memory Efficient**: Optimized storage for objects with multiple fields
- **Atomic Operations**: Update individual fields without affecting others
- **Fast Access**: O(1) field retrieval and updates
- **Structured Data**: Natural mapping to application objects

Embedding Integration

Each product includes a vector embedding generated by:

- **Local Mode**: sentence-transformers (384 dimensions)
- **Google Cloud**: Vertex AI text-embedding-004 (768 dimensions)
- **AWS Bedrock**: Titan Text Embeddings v2 (1024 dimensions)

The embedding captures semantic meaning of the product for similarity search.

Demo: Valkey Hash for Products

Storing the value for product 123456789

```
HSET product:123456789 'id' 123456789 'name' 'Wireless Bluetooth Headphones' 'brand' 'TechAudio' 'main_category' 'Electronics' 'sub_category' 'Audio' 'price' 89.99 'rating' 4.5 'review_count' 1247 'search_tags' 'wireless,bluetooth,headphones,audio,music' 'region' 'NA'
```

Response:

```
(integer) 10
```

Retrieving the value for product 123456789

```
HGETALL product:123456789
```

Response:

```
 1# "id" => "123456789"
 2# "name" => "Wireless Bluetooth Headphones"
 3# "brand" => "TechAudio"
 4# "main_category" => "Electronics"
 5# "sub_category" => "Audio"
 6# "price" => "89.99"
 7# "rating" => "4.5"
 8# "review_count" => "1247"
 9# "search_tags" => "wireless,bluetooth,headphones,audio,music"
10# "region" => "NA"
```

Or retrieving specific field to reduce latency and network transfer

```
HGET product:123456789 name
```

Response:

```
"Wireless Bluetooth Headphones"
```

Or retrieving multiple cherry picked fields

```
HMGET product:123456789 name price rating
```

Response:

```
1) "Wireless Bluetooth Headphones"
2) "89.99"
3) "4.5"
```

Component 2: User Profiles with Valkey-JSON

The Challenge

Storing complex user profiles with nested data structures, purchase history, and dynamic attributes that may evolve over time.

The Solution: Valkey-JSON

Users are stored as native JSON documents, enabling rich data structures:

Valkey-JSON Python 🐍 example

```
# User profile structure
user_profile = {
    "id": "6379",
    "name": "Roberto Luna-Rojas",
    "country": "Mexico 🇲"
    "bio": "Tech enthusiast and early adopter who loves cutting-edge gadgets...",
    "avatar": "data:image/svg+xml;base64,PHN2ZyB2aWV3Qm94PSIwIDAgODAgODAi...",
    "purchase_history": [
        {"product_id": 456, "date": "2024-12-15", "rating": 5, "price": 123.45},
        {"product_id": 789, "date": "2024-11-20", "rating": 4, "price": 234.56}
    ],
    "preferences": {
        "categories": ["electronics", "gaming"],
        "price_range": {"min": 50, "max": 500},
        "brands": ["Apple", "Samsung", "Sony"]
    },
    "embedding": [0.1234, -0.5678, 0.9012, ...] // User preference vector
}

# Stored as: JSON.SET user:6379 $ '{"id":"101","name":"Roberto Luna-Rojas",...}'
```

Why JSON?

- **Flexible Schema**: Easy to add new fields without migration
- **Nested Structures**: Natural representation of complex data
- **Atomic Updates**: Modify specific paths within the document
- **Query Capabilities**: JSONPath queries for complex data retrieval
- **Type Preservation**: Maintains data types (numbers, booleans, arrays)

Demo: Valkey JSON

Store the user JSON document

```
JSON.SET user:6379 $ '{"id": 6379, "name": "Roberto Luna-Rojas", "country": "Mexico 🇲🇽", "bio": "Tech enthusiast and early adopter who loves cutting-edge gadgets...", "avatar": "data:image/svg+xml;base64,PHN2ZyB2aWV3Qm94PSIwIDAgODAgODAi...", "purchase_history": [ {"product_id": 456, "date": "2024-12-15", "rating": 5, "price": 123.45}, {"product_id": 789, "date": "2024-11-20", "rating": 4, "price": 234.56} ], "preferences": { "categories": ["electronics", "gaming"], "price_range": {"min": 50, "max": 500}, "brands": ["Apple", "Samsung", "Sony"]}}'
```

Response:

```
OK
```

Retrieve the whole JSON document

```
valkey-cli -h localhost -p 6379 -3 JSON.GET user:6379 $ | jq -C '.'
```

Response:


```
[
  {
    "id": 6379,
    "name": "Roberto Luna-Rojas",
    "country": "Mexico Ⓜ️",
    "bio": "Tech enthusiast and early adopter who loves cutting-edge gadgets...",
    "avatar": "data:image/svg+xml;base64,PHN2ZyB2aWV3Qm94PSIwIDAgODAgODAi...",
    "purchase_history": [
      {"product_id": 456, "date": "2024-12-15", "rating": 5, "price": 123.45},
      {"product_id": 789, "date": "2024-11-20", "rating": 4, "price": 234.56}
    ],
    "preferences": {
      "categories": ["electronics", "gaming" ],
      "price_range": { "min": 50, "max": 500 },
      "brands": ["Apple", "Samsung", "Sony"]
    }
  }
```

What if I want to only find products over $100 and bellow $200? Let's use JSONPath

```
valkey-cli -h localhost -p 6379 -3 \
JSON.GET user:6379 \
'$.purchase_history[?(@.price > 100 && @.price < 200)]' \
| jq -C '.'
```

Response:

```
[
  {
    "product_id": 456,
    "date": "2024-12-15",
    "rating": 5,
    "price": 123.45
  }
]
```

If I want to update the rating for product 789, I can do so:

```
JSON.SET user:6379 \
$.purchase_history[?(@.product_id==789)].rating 4.5
```

Response:

```
OK
```

Verify the change by getting the product details:

```
valkey-cli -h localhost -p 6379 -3 \
JSON.GET user:6379 \
'$.purchase_history[?(@.product_id==789)]' \
| jq -C '.'
```

Response:

```
[
  {
    "product_id": 789,
    "date": "2024-11-20",
    "rating": 4.5,
    "price": 234.56
  }
]
```

User Embedding Generation

User embeddings are created by combining:

- Bio text semantic analysis
- Purchase history patterns
- Preference indicators
- Behavioral signals

This creates a vector representation of user preferences for personalized search.

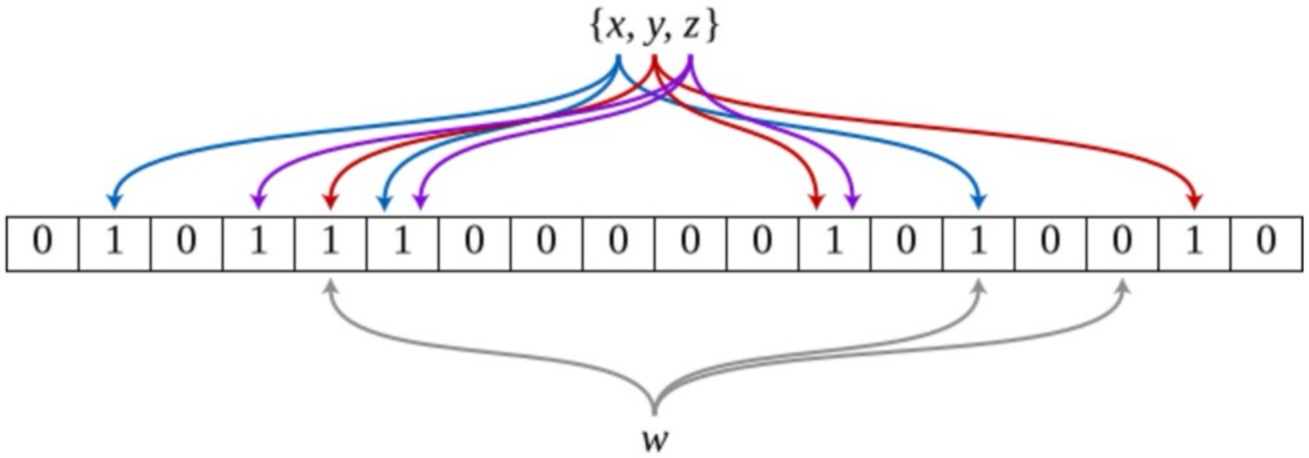
Component 3: Viewed Products Tracking with Valkey-Bloom

The Challenge

Efficiently tracking which products each user has viewed without storing massive sets that consume memory and slow down queries.

The Solution: Bloom Filters

Probabilistic data structure that provides memory-efficient membership testing.



Bloom Filters 🐍 example

```python
# Initialize Bloom filter for each user
bloom_key = f"viewed:{user_id}"
client.bf().reserve(bloom_key, 0.01, 1000)  # 1% error rate, 1000 items capacity

# Mark product as viewed
def mark_product_viewed(user_id, product_id):
    bloom_key = f"viewed:{user_id}"
    valkey_client.bf().add(bloom_key, product_id)

# Check if product was viewed
def is_product_viewed(user_id, product_id):
    bloom_key = f"viewed:{user_id}"
    return valkey_client.bf().exists(bloom_key, product_id)

# UI Integration - show 👀 emoji for viewed products
for product in products:
    product['viewed'] = is_product_viewed(user_id, product['id'])
```

Why Bloom Filters?

- **Memory Efficient**: Uses minimal memory regardless of item count
- **Fast Operations**: O(1) add and lookup operations
- **Scalable**: Handles millions of items with consistent performance
- **Probabilistic**: No false negatives, controlled false positive rate
- **Perfect for UX**: "Have I seen this before?" is ideal for Bloom filters

Real-World Benefits

- **Reduced Cognitive Load**: Users can quickly identify new vs. seen products
- **Improved Engagement**: Focus attention on unseen items
- **Memory Savings**: 99% less memory than storing actual sets
- **Performance**: No impact on search speed

Component 4: Vector Similarity Search with Valkey-Search

The Challenge

Finding products similar to user preferences and enabling semantic search beyond keyword matching.

The Solution: Vector Search with HNSW

Valkey-Search provides high-performance vector similarity using Hierarchical Navigable Small World (HNSW) algorithm:

Valkey-Search index creation example:

```
# Index creation with vector field
FT.CREATE products ON HASH PREFIX 1 product: SCHEMA
  brand_tags TAG SEPARATOR ,
  search_tags TAG SEPARATOR ,
  region TAG
  price NUMERIC
  rating NUMERIC
  embedding VECTOR HNSW 6 TYPE FLOAT32 DIM 1024 DISTANCE_METRIC COSINE
```

Hybrid search 🐍

```
# Hybrid search query combining filters + vector similarity
def search_products(user_embedding, tags, region=None):
    # Build tag filter
    tag_filter = " ".join(f"@search_tags:{{{tag}}}" for tag in tags)
    # Add region filter if specified
    if region:
        tag_filter += f" @region:{{{region}}}"
    # Combine with vector search
    query = f"({tag_filter})=>[KNN 25 @embedding $user_vec]"
    return valkey_client.ft("products").search(
        Query(query).return_fields("id", "name", "price", "rating")
                   .sort_by("_score", asc=False)
                   .dialect(2),
        query_params={"user_vec": user_embedding}
    )
```

Vector Similarity Concepts

What is a Vector?

A vector is a quantity that has both magnitude (size) and direction. It's a fundamental concept in mathematics and physics, used to describe quantities that can't be fully represented by a single number alone.

Vector Similarity Concepts (cont)

1. **Embedding Spaces**
   - Products and users exist in high-dimensional vector space
   - Similar items cluster together
   - Distance metrics measure similarity
2. **Cosine Similarity**
   - Measures angle between vectors, not magnitude
   - Perfect for semantic similarity (0 = identical, 1 = opposite)
   - Robust to vector normalization

Vector Similarity Concepts (cont)

3. **HNSW Algorithm**
    - Hierarchical graph structure for approximate nearest neighbor search
    - Logarithmic search complexity: $O(\log N)$
    - Tunable accuracy vs. speed tradeoffs
4. **Hybrid Search Benefits**
    - **Precision**: Keyword filters ensure relevance
    - **Discovery**: Vector search finds unexpected matches
    - **Personalization**: User embedding biases results toward preferences

Maximal Marginal Relevance (MMR)

To avoid showing too many similar products, we use MMR for result diversification: Balance relevance VS Diversity 🐍

```python
def mmr_rerank(query_embedding, candidate_embeddings, lambda_param=0.7, top_n=5):
    """
    lambda_param: 1.0 = pure relevance, 0.0 = pure diversity
    """
    selected_indices = []
    # Start with most relevant item
    relevance_scores = cosine_similarity(candidates, query_embedding)
    selected_indices.append(np.argmax(relevance_scores))
    # Iteratively select items that are relevant but diverse
    while len(selected_indices) < top_n:
        mmr_scores = {}
        for candidate_idx in remaining_candidates:
            relevance = relevance_scores[candidate_idx]
            # Measure similarity to already selected items
            diversity = max_similarity_to_selected(candidate_idx, selected_indices)
            # Balance relevance and diversity
            mmr_scores[candidate_idx] = lambda_param * relevance - (1 - lambda_param) * diversity
        # Select item with highest MMR score
        best_candidate = max(mmr_scores, key=mmr_scores.get)
        selected_indices.append(best_candidate)
    return selected_indices
```

Component 5: Personalized Recommendations

User Persona-Based Results

The system uses detailed user personas to tailor search results 🐍

```
# Example personas
personas = {
    "101": {
        "name": "Roberto Luna-Rojas",
        "bio": "Tech enthusiast and geek by nature..",
        "interests": ["technology", "innovation", "smart_home", "gadgets"]
    },
    "102": {
        "name": "Tay Tay",
        "bio": "Best female pop artist ever...",
        "interests": ["music", "cats"]
    }
}
```

Personalization Pipeline

1. **User Embedding**: Convert bio and preferences to vector
2. **Product Matching**: Find products with similar embeddings
3. **Context Filtering**: Apply user's category and price preferences
4. **Relevance Scoring**: Combine similarity + user history + ratings
5. **Diversification**: Use MMR to avoid redundant recommendations

Component 6: Session Management with Valkey

The Challenge

Managing user sessions, shopping carts, and temporary state across requests while maintaining performance.

The Solution: Valkey Strings and Hashes 🐍

```
# Session storage
session_key = f"session:{session_id}"
```

```
# Store session data as hash
HSET session:abc123
  user_id 101
  cart_total 299.99
  last_activity 1704067200
  preferences '{"theme":"dark","language":"en"}'

# Shopping cart as list
LPUSH cart:abc123 "product:456" "product:789"

# Recent activity tracking
LPUSH activity:abc123 "viewed:product:456" "added_to_cart:product:789"
LTRIM activity:abc123 0 99  # Keep last 100 activities

# Session expiration
EXPIRE session:abc123 3600  # 1 hour TTL
```

Session Benefits

- **Fast Access**: O(1) session retrieval
- **Automatic Cleanup**: TTL-based session expiration
- **Atomic Updates**: Update cart without race conditions
- **Scalability**: Shared sessions across multiple app instances

Component 7: LLM Response Caching

The Challenge

AI-generated personalized product descriptions are expensive to compute and can have high latency, especially when using cloud APIs.

The Solution: Intelligent Caching Strategy 🐍

```
# Cache key structure
cache_key = f"llm_cache:user:{user_id}:product:{product_id}"
```

```python
# Cache lookup before AI generation
def get_personalized_description(user_profile, product):
    cache_key = f"llm_cache:user:{user_profile['id']}:product:{product['id']}"
    # Try cache first
    cached_desc = valkey_client.get(cache_key)
    if cached_desc:
        return cached_desc.decode()
    # Generate new description
    prompt = f"""
    You are a helpful sales assistant. A user named {user_profile['name']}
    is considering the product: '{product['name']}'.
    Their bio is: '{user_profile['bio']}'.
    Write a personalized paragraph that addresses their interests.
    """
    # Call AI backend (AWS Bedrock, Google Gemini, or local Ollama)
    description = generate_with_ai(prompt)
    # Cache for 2 hours
    valkey_client.set(cache_key, description, ex=7200)
    return description
```

Caching Strategy Benefits

- **Cost Reduction**: Avoid repeated expensive AI API calls
- **Latency Improvement**: Cached responses return in <1ms vs 500-2000ms for AI generation
- **Scalability**: Handle more concurrent users with same AI quota
- **Reliability**: Cached responses available even if AI service is down

Cache Performance Metrics 🐍

```python
# Cache hit rate monitoring
def track_cache_performance():
    total_requests = valkey_client.get("cache:total_requests") or 0
    cache_hits = valkey_client.get("cache:hits") or 0
    hit_rate = (cache_hits / total_requests) * 100 if total_requests > 0 else 0
    print(f"Cache Hit Rate: {hit_rate:.1f}%")
    print(f"Total Requests: {total_requests}")
    print(f"Cache Hits: {cache_hits}")
```

Asynchronous Cache Warming 🐍

```python
def warm_cache_async(user_profile, products):
    """
    Background thread generates descriptions for products
    without blocking the user interface
    """
    def background_task():
        for product in products:
            cache_key = f"llm_cache:user:{user_profile['id']}:product:{product['id']}"
            if not valkey_client.exists(cache_key):
                # Generate and cache description
                description = generate_personalized_description(user_profile, product)
                valkey_client.set(cache_key, description, ex=7200)
    # Start background thread
    threading.Thread(target=background_task).start()
```

Performance Characteristics

Valkey-Bundle Performance Profile

Operation	Data Structure	Complexity	Typical Latency
Product Lookup	Hash	O(1)	<1ms
User Profile	JSON	O(1)	<1ms
Viewed Check	Bloom Filter	O(1)	<0.1ms
Vector Search	HNSW Index	O(log N)	2.5-10ms
Cache Lookup	String	O(1)	<0.5ms
Session Access	Hash	O(1)	<1ms

Memory Efficiency

Memory usage comparison for 1M users tracking 10K products each

Traditional Set Storage:

- 1M users × 10K products × 8 bytes = 80GB

Bloom Filter Storage:

- 1M users × 1KB per filter = 1GB (98.75% memory reduction!)

False positive rate: 1% (configurable)

False negative rate: 0% (guaranteed)

Scalability Patterns

Horizontal Scaling with Valkey Cluster 🐍

```
# Cluster configuration
startup_nodes = [
    ClusterNode(host="valkey-node-1", port=6379),
    ClusterNode(host="valkey-node-2", port=6380),
    ClusterNode(host="valkey-node-3", port=6381)
]
client = ValkeyCluster(startup_nodes=startup_nodes)
# Data automatically sharded across nodes
# Hash tags ensure related data stays together
HSET {user:101}:profile name "Roberto Luna-Rojas"
HSET {user:101}:session cart_total 299.99
BF.ADD {user:101}:viewed product:456
```

Development and Deployment

Local Development Setup

```
# Start Valkey-bundle
docker run -d --rm --name valkey-demo -p 6379:6379 valkey/valkey-bundle

# Setup Python environment
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt

# Initialize data
python3 load_data.py
python3 init_bloom_filters.py

# Run application
flask run --host=0.0.0.0 --port=5001
```

Production Considerations

- **Memory Planning**: Size Valkey instances based on dataset and cache requirements
- **Backup Strategy**: Regular RDB snapshots + AOF for durability
- **Monitoring**: Track cache hit rates, search latency, and memory usage
- **Security**: Network isolation, authentication, and encryption in transit

Key Takeaways

Why Valkey-Bundle Excels for Modern Applications

1. **Unified Platform**: Single solution for diverse data needs
2. **Performance**: Sub-millisecond operations for most use cases
3. **Scalability**: Horizontal scaling with cluster mode
4. **Flexibility**: Multiple data structures for different patterns
5. **AI Integration**: Native vector search for ML applications
6. **Developer Experience**: Rich ecosystem and tooling

When to Choose Valkey-Bundle

✅ **Perfect For:**

- Real-time applications requiring low latency
- AI/ML applications with vector similarity search
- Applications with diverse data access patterns
- High-performance caching layers
- Session management and user state
- Analytics and recommendation engines

When to Choose Valkey-Bundle (cont)

⚠️ Consider Alternatives For:

- Applications requiring strong consistency guarantees
- Complex relational queries with joins
- Long-term analytical data warehousing
- Applications with minimal performance requirements

The Future of Low-Latency Applications

Valkey-bundle represents the evolution toward:

- **Unified Data Platforms**: Reducing operational complexity
- **AI-Native Infrastructure**: Built-in support for vector operations
- **Edge Computing**: Fast, local data processing
- **Real-Time Personalization**: Instant, context-aware experiences

Conclusion

This demonstration showcases how Valkey-bundle's integrated approach solves real-world challenges in modern application development. By combining traditional data structures with advanced capabilities like vector search and probabilistic filters, developers can build sophisticated, high-performance applications with a single, unified platform.

The key insight is that modern applications require diverse data access patterns - from simple key-value lookups to complex vector similarity searches. Valkey-bundle provides all these capabilities in a cohesive, high-performance package that scales from prototype to production.

Conclusion (cont)

Whether you're building recommendation engines, real-time analytics, or AI-powered applications, Valkey-bundle offers the performance, flexibility, and developer experience needed for success in today's demanding application landscape.

For more information about Valkey-bundle and to explore the complete source code of this demonstration, visit: https://valkey.io/blog/valkey-bundle-one-stop-shop-for-low-latency-modern-applications/ by Roberto Luna-Rojas

Demo enhanced from original Valkey Search Demo by Ping Xie PingXie