What's New in MarkLogic 12

MarkLogic 12

Publication date 2024-08-21
Copyright © 2024 Progress Software Corporation

All Rights Reserved

Table of Contents

1. What's New in MarkLogic 12 3
2. Release Notes 4
    2.1. MarkLogic 12.0 EA1 4
        2.1.1. Features 4
        2.1.2. Installation 4
        2.1.3. Upgrade 4
3. New Features in MarkLogic 12.0 EA1 5
    3.1. Native Vector Support 5
    3.2. BM25 Relevance Ranking 9
    3.3. Shortest Path Graph Algorithm 11
4. Technical support 15
5. Copyright 16

1. What's New in MarkLogic 12

This guide includes these topics:

1. Release Notes
2. New Features in MarkLogic 12.0 EA1

2. Release Notes

2.1. MarkLogic 12.0 EA1

MarkLogic 12.0 EA1 expands the already comprehensive query capabilities of MarkLogic Server by providing early access to the first step towards native vector search support, Best Match 25 (BM25) relevance ranking, and a shortest path graph algorithm. Explore these features to develop secure, AI-enhanced applications or other search-based transactional systems with MarkLogic Server.

NOTICE

This software is made available under the following Early Access Agreement. It may be used only for evaluation purposes. It may not be used in production.

2.1.1. Features

Detailed descriptions of each new feature in MarkLogic 12.0 EA1 can be found in the new features sections:

- Native Vector Support
- BM25 Relevance Ranking
- Shortest Path Graph Algorithm

2.1.2. Installation

MarkLogic 12.0 EA1 is available only for Red Hat Linux 8 (or compatible Linux distributions).

To install MarkLogic 12.0 EA1, follow the MarkLogic 11 instructions from Installing MarkLogic in *Installing MarkLogic Server*.

2.1.3. Upgrade

NOTICE

Upgrading to or from early access releases of MarkLogic Server is not supported. Therefore, upgrading from any earlier version to MarkLogic 12.0 EA1 is not supported, nor will upgrading from MarkLogic 12.0 EA1 to any later version be supported.

3. New Features in MarkLogic 12.0 EA1

3.1. Native Vector Support

With the recent developments in Generative AI (GenAI) and Retrieval Augmented Generation (RAG), vector search has become a valuable tool in a search developer's toolbox to further improve information retrieval through vector search or vector reranking. Combining keyword-based scoring methods like BM25 with vector similarity operations to perform a "hybrid search" boosts textually and semantically similar documents to the top of your search result.

In a typical RAG architecture, the text classifier models map chunks of text into representative vectors of floating-point numbers. Separate but semantically similar chunks of text would map to vectors that are close to one another in the high-dimensional vector space.

These vectors can then be used to compute, for example, cosine similarity, dot product, or basic vector arithmetic.

MarkLogic 12.0 EA1 introduces operators on vectors. This is the first step towards the full-scale vector search support planned for MarkLogic 12 GA. While this support requires a full implementation of an Approximate Nearest Neighbor (ANN) index, the operations available in MarkLogic 12.0 EA1 give an API preview and allow exploration of hybrid search and vector-based reranking. The vector operations available in MarkLogic 12.0 EA1 include creation, basic arithmetic, score helpers, and cosine and euclidean distance functions that allow linear scans for nearest neighbors.

Integration with text classifier models like those provided by OpenAI is beyond the scope of this article. Please refer to the documentation of your model of choice regarding mapping of text to vector equivalents. The rest of this article assumes that this integration has already happened and that these vectors are now available.

Vectors output by text classifier models are often represented as a JSON array:

JSON (URI: `/sample.json`)

```
{
  "envelope": {
    "headers": [
      {
        "textEmbedding" : {
          "lang": "zxx",
          "model": "text-embedding-ada-002",
          "source": "OpenAI",
          "dimension": 1536,
          "vector": [
            0.435647279024124, 0.167360082268715, 0.577132880687714, 0.0405717119574547,
-0.345730692148209,
            ...
            0.413799345493317, 0.339704662561417, -0.259793192148209, 0.118780590593815,
0.649678707122803
          ]
        }
      }
    ],
    "instance": {
      "url": "https://simple.wikipedia.org/wiki?curid=8126",
      "text": "The Trojan War was one of the most important ... in the 12th century BC."
    }
  }
}
```

They can also be represented as a serialized array of numbers in XML:

XML (URI: `/sample.xml`)

```
<envelope>
  <headers>
    <text-embedding>
      <model>text-embedding-ada-002</model>
      <source>OpenAI</source>
      <dimension>1536</dimension>
      <vector
xml:lang="zxx">[0.435647279024124,0.167360082268715,0.577132880687714,0.0405717119574547,-0.345730692148209,
        ...,0.413799345493317,0.339704662561417,-0.259793192148209,0.118780590593815,0.649678707122803]</vector>
    </text-embedding>
  </headers>
  <instance>
    <url>https://simple.wikipedia.org/wiki?curid=8126</url>
    <text>The Trojan War was one of the most important ... in the 12th century BC.</text>
  </instance>
</envelope>
```

Ingestion

SQL-aware Retrieval-Augmented Generation (RAG) systems and RAG-enabled Business Intelligence (BI) tools are commonly configured to interact with SQL interfaces. MarkLogic Server supports this through Template Driven Extraction (TDE) -based views, which can ingest document data into table-like structures with columns that can be declared as scalar type `vector`.

Assuming the context of the template is `/envelope`, this is what your vector column would look like:

JSON column declaration within a TDE view

```
"columns": [{
    "name": "textEmbedding",
    "scalarType": "vector",
    "val": "vec:vector(headers/textEmbedding/array-node('vector'))",
    "dimension": "1536"
  },
  …
]
```

XML column declaration within a TDE view

```
<columns>
  <column>
    <name>textEmbedding</name>
    <scalar-type>vector</scalar-type>
    <val>vec:vector(headers/text-embedding/vector)</val>
    <dimension>1536</dimension>
  </column>
 …
</columns>
```

These generated columns can then be accessed through Optic queries.

Reference: "Template Dialect and Data Transformation Functions" in the "Template Driven Extraction (TDE)" chapter of the *Application Developer's Guide*

Query

The following example query focuses on documents that contain the term `trojan`. Rows from a view called `article` are joined on `fragment ID`, constraining the results to documents that match

the search term. `queryVector` contains the vector generated by sending a chunk of one of these documents to a third-party or internal model. `queryVector` can be compared with the vector column of each row to generate a vector rating--in this case, a cosine similarity. This rating can either be used to sort the results directly or be combined with the `cts.score` of a document search to compute a hybrid score:

SJS

```
const op = require('/MarkLogic/optic');

const documentQuery = cts.wordQuery("trojan")
const queryVector = vec.vector([
    -0.05992345495992422,-0.1234123430928, ... ,-4.549399422136e-05,-0.012034502243
  ])

const documents = op.fromSearch(
    documentQuery,
    ['fragmentId', 'score'],
    'docs_view',
    {
      'scoreMethod': 'bm25',
      'bm25LengthWeight': 0.5
    }
  ).joinDoc('doc',op.fragmentIdCol('fragmentId'));
const rows = op.fromView(
    'examples',
    'article',
    null,
    op.fragmentIdCol('$$viewFragment')
  )

const result =
  documents
    .joinInner(
      rows,
      op.on(
        op.fragmentIdCol('fragmentId'),
        op.fragmentIdCol('$$viewFragment')
      )
    )
    .orderBy(op.desc(op.col('score')))
    .limit(30)
    .bind(op.as('queryvector', queryVector))
    .bind(op.as('cosineSim',
      op.vec.cosineSimilarity(
        op.col('textEmbedding'),
        op.col('queryvector')
      )
    ))
    .bind(op.as('hybridScore',
      op.vec.vectorScore(op.col('score'), op.col('cosineSim'), 0.1)
    ))
    .select([
      op.col('doc'),
      op.col('cosineSim'),
      op.col('score'),
      op.col('hybridScore')
    ])
    .orderBy(op.desc(op.col('hybridScore')))
    .limit(20)
    .result();
result;
```

- `op.fromSearch()` retrieves the `cts.score`.

- `op.joinDoc()` joins the document content to pass to the RAG pipeline.
- `op.limit()` reduces the number of documents to be returned for vector computation.
- `op.fromView()` retrieves the vector column for processing.
- `op.bind(op.as('cosineSim', ... ))` binds a new column that is the result of the similarity calculation between each vector value in the `examples` view and the query vector.
- `op.vec.cosineSimilarity()` computes the cosine similarity between the vector in the `textEmbedding` column and `queryVector`.
- `op.vec.vectorScore()` is a convenience function. It uses a formula that takes `cts.score` as a base in the first argument then adjusts it according to the vector similarity in the second argument. In this example, the higher the cosine similarity of the vector, the more significant the boost to that base `cts.score`. This pushes the more semantically similar documents higher in the result set.
  This formula is used instead of Reciprocal Rank Fusion (RRF), a common method to fuse search results from different sources into one final score.
  Add `op.vec.vectorScore()`'s third argument, `similarityWeight`, to tweak the lift that cosine similarity has on the hybrid score:
  - Default: `0.1` (no `similarityWeight` argument)
  - Lowest: `0.0` (`cts.score` remains unchanged)
  - Highest: `1.0` (boosts `cts.score` significantly as the vector similarity (second argument) approaches `1.0`)
- `op.select()` renders these columns in its result:
  - `doc`: The document content.
  - `cosineSim`: The `cosineSimilarity` between each value in the `vector` column and `queryVector`.
  - `score`: The `cts.score`.
  - `hybridScore`: The hybrid score.
- `op.orderBy()` with `op.desc()` on `op.col()` orders the resulting rows from highest to lowest value in the `hybridScore` column.

Here is the XQuery version:

XQuery

```
xquery version "1.0-ml";

import module namespace op = "http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";
import module namespace opvec = "http://marklogic.com/optic/expression/vec"
     at "/MarkLogic/optic/optic-vec.xqy";

let $document-query := cts:word-query("trojan")
let $query-vector := vec:vector((
    0.435647279024124, 0.167360082268715, 0.577132880687714, 0.0405717119574547,
-0.345730692148209,
        ...
    0.413799345493317, 0.339704662561417, -0.259793192148209, 0.118780590593815,
0.649678707122803
))

let $documents := op:from-search(
    $document-query,
    ("fragmentId","score"),
    "docs_view",
    map:map()
      => map:with("scoreMethod", "bm25")
      => map:with("bm25LengthWeight", 0.5)
  )
  =>op:join-doc("doc",op:fragment-id-col("fragmentId"))
let $view := op:from-view(
    "examples",
    "article",
    (),
    op:fragment-id-col("$$view-fragment")
  )
return $documents
  => op:join-inner(
    $view,
    op:on(
      op:fragment-id-col("fragmentId"),
      op:fragment-id-col("$$view-fragment")
    )
  )
  => op:order-by(op:desc(op:col("score")))
  => op:limit(30)
  => op:bind(op:as("queryvector", $query-vector))
  => op:bind(op:as("cosineSim",
    opvec:cosine-similarity(op:col("textEmbedding"), op:col("queryvector"))
  ))
  => op:bind(
    op:as("hybridScore",
      opvec:vector-score(op:col("score"), op:col("cosineSim"), 0.1)
    )
  )
  => op:select((
    op:col("doc"),
    op:col("cosineSim"),
    op:col("score"),
    op:col("hybridScore")
  ))
  => op:order-by(op:desc(op:col("hybridScore")))
  => op:limit(20)
  => op:result()
```

Be sure to explore the other new vector operators.

3.2. BM25 Relevance Ranking

MarkLogic 12.0 EA1 supports Best Match 25 (BM25) for relevance scoring of search results.

The BM25 method of scoring documents is widely used because of its effectiveness in ranking documents based on relevance to a query. With the rise of Generative AI (GenAI) and Retrieval Augmented Generation (RAG) workflows, BM25 has come into play for retrieving relevant documents to supply as context to a Large Language Model (LLM). LLM-generated answers see a significant quality boost when ranked documents become the sources of knowledge.

The MarkLogic Server core text search has always returned results based on relevance. The default MarkLogic Server relevance scoring method, logTF-IDF, implicitly weights document length by counting unique terms. The new BM25 scoring method explicitly does this by adding a tunable parameter to directly increase or decrease the weight of a document's length on the score.

Consider the following documents' term frequencies and lengths retrieved by `cts.wordQuery("trojan")`:

URI	Term Frequency	Length (Average: 1396 characters)
/doc1.json	3	1628
/doc2.json	2	976
/doc3.json	2	916
/doc4.json	10	2592
/doc5.json	2	868

Compare the ranking differences between traditional logTF-IDF, which uses term frequency alone, and BM25, which heavily penalizes `/doc4.json` for being significantly longer than the average document length of 1396:

Ranking	logTF-IDF	BM25
1st	/doc4.json	/doc5.json
2nd	/doc1.json	/doc2.json
3rd	/doc2.json	/doc3.json
4th	/doc3.json	/doc1.json
5th	/doc5.json	/doc4.json

The BM25 scoring method accounts for the fact that a short document with a high term frequency is more likely to be about that term than a long document with the same term frequency, which is more likely to simply be using that term in passing.

Use one of these code samples to enable the BM25 scoring method for core text search queries:

- Each specifies BM25 as the score method.
  Default: logTF-IDF
- Each provides the optional parameter BM25 length weight of `0.25`:
  - Default: `0.33` (no BM25 length weight parameter)
  - Lowest: Just above `0.0` (most similar to logTF-IDF results)
  - Highest: `1.0` (weight document length as heavily as possible)

CTS

```
cts.search(
  cts.wordQuery("trojan"),
  [
    "score-bm25",
    "bm25-length-weight=0.25"
  ]
)
```

Optic

```
const op = require('/MarkLogic/optic');
op.fromSearch(
  cts.wordQuery("trojan"),
  null,
  null,
  {
    "scoreMethod" : "bm25",
    "bm25LengthWeight": 0.25
  }
)
```

NOTE

`op.fromSearchDocs()` also takes BM25 scoring parameters.

Search/REST Search API

```
import module namespace search = "http://marklogic.com/appservices/search"
    at "/MarkLogic/appservices/search/search.xqy";
search:search("trojan",
  <options
      xmlns="http://marklogic.com/appservices/search">
    <search-option>score-bm25</search-option>
    <search-option>bm25-length-weight=0.25</search-option>
  </options>
)
```

JSearch

```
import jsearch from '/MarkLogic/jsearch.mjs';
jsearch.documents()
  .where(cts.wordQuery("trojan"))
  .withOptions(
    {
      search: [
        'score-bm25',
        'bm25-length-weight=0.25'
      ]
    }
  )
  .result()
```

Using core text search within MarkLogic Server has always been the key to finding the data that you need. BM25 is an extra knob to further tune your search results.

3.3. Shortest Path Graph Algorithm

Many real-world problems--such as ones within navigational systems, network routing, and circuit design--can be represented as shortest path problems.

In MarkLogic Server, triples and the Semantics capability can model and resolve such shortest path problems. Each subject and object can be considered as nodes separated by predicate edges. Crossing these edges comes at a cost, so finding the path with the fewest edges is crucial.

Likewise, the ways that recommendation systems and natural language processing applications use knowledge graphs to model relationships between people, words, or concepts can be looked at as shortest path problems.

Shortest path queries can help discover relevant facts in knowledge graphs and piece together information that may have been missed before.

To facilitate such queries, MarkLogic 12.0 EA1 introduces the SPARQL magic property, `xdmp:shortestPath`.

`xdmp:shortestPath` makes use of these named arguments:

- `xdmp:start`
- `xdmp:end`
- `xdmp:pattern`
- `xdmp:path`
- `xdmp:length`

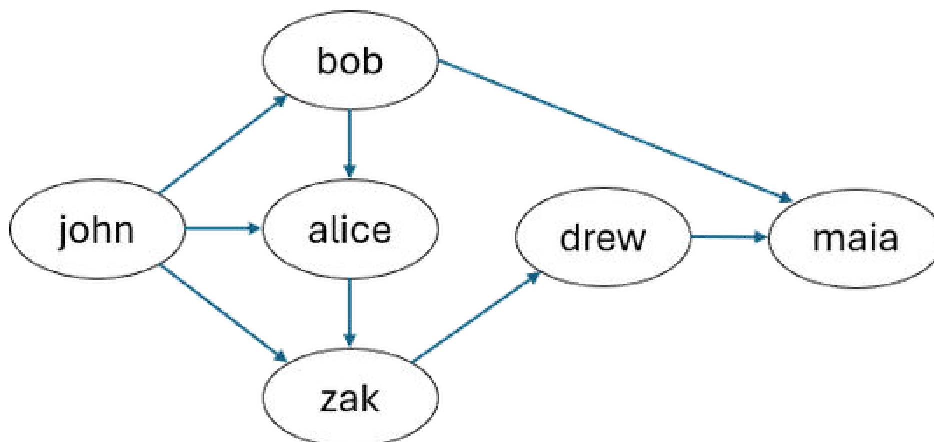
The framework to support `xdmp:shortestPath` can easily allow the addition of future SPARQL magic properties like `xdmp:unnest`.

Reference: W3C's SPARQL/Extensions/Computed Properties

To better understand the shortest path problem, consider this simple data set:

```
@prefix p: <http://example.org/kennedy/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
p:john foaf:knows p:bob .
p:john foaf:knows p:alice .
p:john foaf:knows p:zak .
p:bob foaf:knows p:alice .
p:bob foaf:knows p:maia .
p:alice foaf:knows p:zak .
p:zak foaf:knows p:drew .
p:drew foaf:knows p:maia .
```

This diagram represents the data set:



The shortest path from `john` to `maia` would be the one with the fewest edges (blue arrows) between them: through `bob`.

The Shortest Path from John to Maia

This SPARQL query uses the new `xdmp:shortestPath` property to find the shortest path from `john` to `maia`:

```
PREFIX xdmp: <http://marklogic.com/xdmp#>
PREFIX p: <http://example.org/kennedy/>
SELECT *
where {
  (
    [xdmp:start ?s ]
    [xdmp:end ?o]
  ) xdmp:shortestPath (
    [xdmp:path ?path]
    [xdmp:length ?length]
  )
  FILTER (?s = p:john && ?o = p:maia )
}
```

Here is the response:

```
[{
  "s": "<http://example.org/kennedy/john>",
  "o": "<http://example.org/kennedy/maia>",
  "path": [{
      "s": "http://example.org/kennedy/john",
      "_:ANON9088488689022242345": "http://xmlns.com/foaf/0.1/knows",
      "o": "http://example.org/kennedy/bob"
    }, {
      "s": "http://example.org/kennedy/bob",
      "_:ANON9088488689022242345": "http://xmlns.com/foaf/0.1/knows",
      "o": "http://example.org/kennedy/maia"
    }
  ],
  "length": "\"2\"^^xs:unsignedLong"
}]
```

By default, `xdmp:shortestPath` finds the shortest path from the subject (`john`) to the object (`maia`).

The FILTERs in this query are optional:

- `?s`: Constrains the start of the shortest path search on the subject. If left off, the query would display all the shortest paths that lead to the end object of a triple (in this case, to `maia`).
- `?o`: Constrains the end of the shortest path search on the object. If left off, the query would display all the shortest paths that start from the subject of a triple (in this case, from `john`).

For large graphs, apply FILTER on `?length` to stop the shortest path search after reaching a specified number of hops.

The Shortest Path from Maia to John

What about getting from `maia` (object) to `john` (subject) instead? In this case, use `xdmp:pattern` to reverse the direction of the edges:

```
## query
PREFIX xdmp: <http://marklogic.com/xdmp#>
PREFIX p: <http://example.org/kennedy/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/knows>
SELECT *
where {
  (
    [xdmp:start ?s]
    [xdmp:end ?o]
    [xdmp:pattern "?s foaf:|^foaf: ?o"]
  ) xdmp:shortestPath (
    [xdmp:path ?path]
    [xdmp:length ?length]
  )
  FILTER (?s = p:maia && ?o = p:john)
}
```

`xdmp:pattern` takes a triple pattern lookup as a string. This string uses the OR operator (`|`) along with the inverse operator (`^`) to implement an inverse property path. This inverse property path reverses the relationship between `subject` and `object`, forcing the shortest path computation to start with `object` instead.

That query renders this result:

```
[{
  "s": "<http://example.org/kennedy/maia>",
  "o": "<http://example.org/kennedy/john>",
  "path": [{
      "o": "http://example.org/kennedy/bob",
      "s": "http://example.org/kennedy/maia"
    }, {
      "o": "http://example.org/kennedy/john",
      "s": "http://example.org/kennedy/bob"
    }
  ],
  "length": "\"2\"^^xs:unsignedLong"
}]
```

Reference: SPARQL 1.1 Property Paths

This new shortest path magic property can help discover links in a knowledge graph and supplies a method to retrieve the path from start to finish--or from finish to start.

4. Technical support

Progress Software provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at `http://help.marklogic.com` to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the *Support Handbook* for instructions on registering support contacts and on working with the MarkLogic Server Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at `http://developer.marklogic.com`. For technical questions, we encourage you to ask your question on the Progress Community.

5. Copyright

For copyright information, see Product Documentation and Copyright Notice.