vespa

Big data ◦ Real time

The open big data serving engine; store, search,
rank and organize big data at user serving time.

By Vespa architect @jonbratseth

This deck

What’s big data serving?

Vespa - the big data serving engine

Vespa architecture and capabilities

Using Vespa

Big data maturity levels

Latent	Data is produced but not systematically leveraged	
	Example	**Logging**: Movie streaming events are logged.
Analysis	Data is used to inform decisions made by humans	
	Example	**Analytics**: Lists of popular movies are compiled to create curated recommendations for user segments.
Learning	Data is used to take decisions offline	
	Example	**Machine learning**: Lists of movie recommendations per user segment are automatically generated.
Acting	Automated data-driven decisions in real time	
	Examples	**Stream processing:** Each movie is assigned a quality score as they are added **Big data serving**: Personalized movie recommendations are computed when needed by that user.

Big data serving: Advantages

Decisions use up to date information

No wasted computation

Fine-grained decision making

Architecturally simple

Big data serving: What is required?

state x scatter-gather x consistent low latency x high availability

Real-time actions:	Find data and make inferences in tens of milliseconds.
Realtime knowledge:	Handle data updates at a high continuous rate.
Scalable:	Handle large requests rates over big data sets.
Always available:	Recover from hardware failures without human intervention.
Online evolvable:	Change schemas, logic, models, hardware while online.
Integrated:	Data feeds from Hadoop, learned models from TensorFlow etc.

Introducing ...

vespa

Make big data serving universally available

Open source, available on https://vespa.ai (Apache 2.0 license)

Provenance:

- Web search: The canonical big data serving use case
- Yahoo search made Hadoop and Vespa to solve it
- Core idea of both: Move computation to data

Vespa at Verizon Media (ex Yahoo)

Tumblr, TechCrunch, Huffington Post, Aol, Engadget, Gemini, Yahoo News, Yahoo Sports, Yahoo Finance, Yahoo Mail, Flickr, etc.

Hundreds of Vespa applications,

… serving **over a billion** users

… **over 250.000** queries per second

… **over billions** of content items

including the worlds 3rd largest ad network

Big data serving: Use cases

Search

Query: *Keywords*
Model(s) evaluated: *Relevance*
Selected items: *By relevance*

Recommendation

Query: *Filters + user model*

Model: *Recommendation*

Selected items: *By recommendation score*

What else?

Big data serving: Beyond search, recommendation

A different use case:

Items: *Priced assets (e.g stocks)*
Model: *Price predictor*
Query: *A representation of an event*
Selected items: *By largest predicted price difference*

Result:

- Find the new predicted prices of assets changing most in response to an event
- … using completely up-to-date information
- … faster than anybody else

Analytics vs big data serving

Analytics (e.g Elastic Search)		Big data serving (Vespa)
Response time in low seconds		Response time in low milliseconds
Low query rate		High query rate
Time series, append only	VS	Random writes
Down time, data loss acceptable		HA, no data loss, online redistribution
Massive data sets (trillions of docs) are cheap		Massive data sets are more expensive
Analytics GUI integration		Machine learning integration

Where are we?

What’s big data serving?

Vespa - the big data serving engine

Vespa architecture and capabilities

Using Vespa

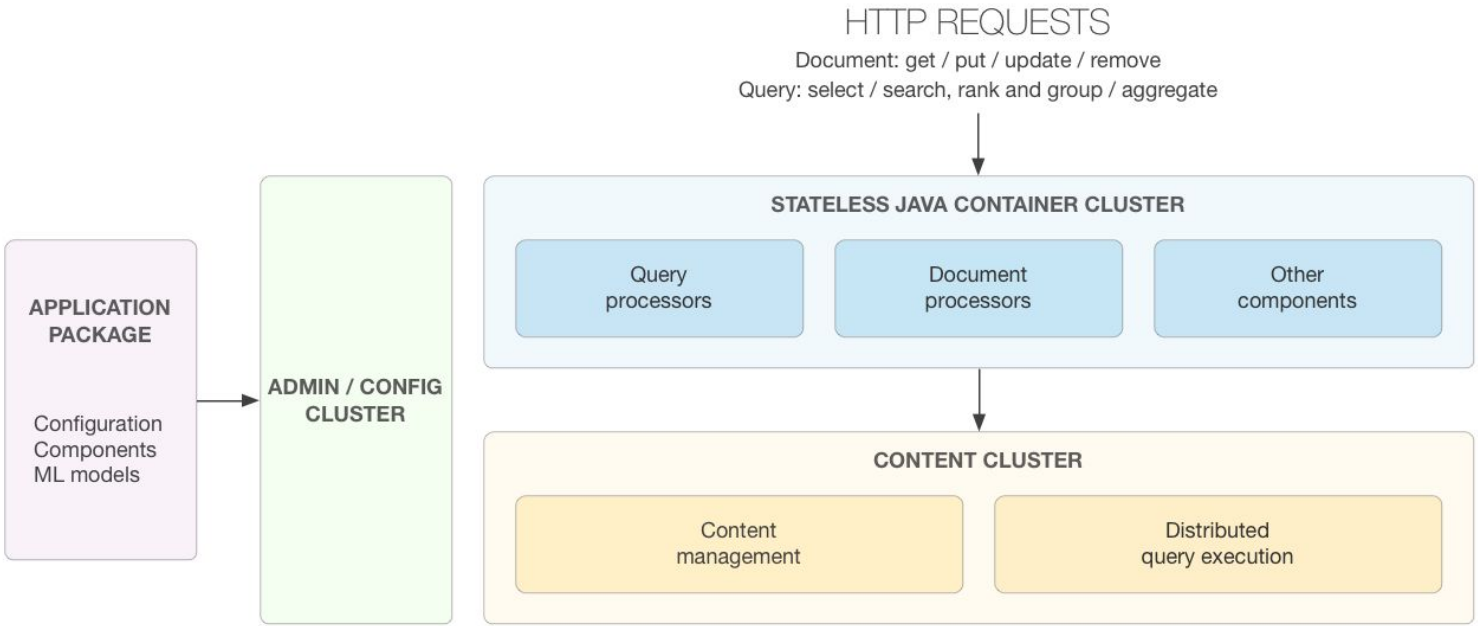
Vespa is

A platform for low latency computations over large, evolving data sets

- **Search** and **selection** over structured and unstructured data
- Scoring/relevance/inference: NL features, advanced ML models, TensorFlow etc.
- Query time **organization** and **aggregation** of matching data
- Real-time writes at a high sustained rate
- Live **elastic** and **auto-recovering** stateful content clusters
- Processing logic container (Java)
- Managed clusters: One to hundreds of nodes

Typical use cases: text search, personalization / recommendation / targeting, real-time data display, ++

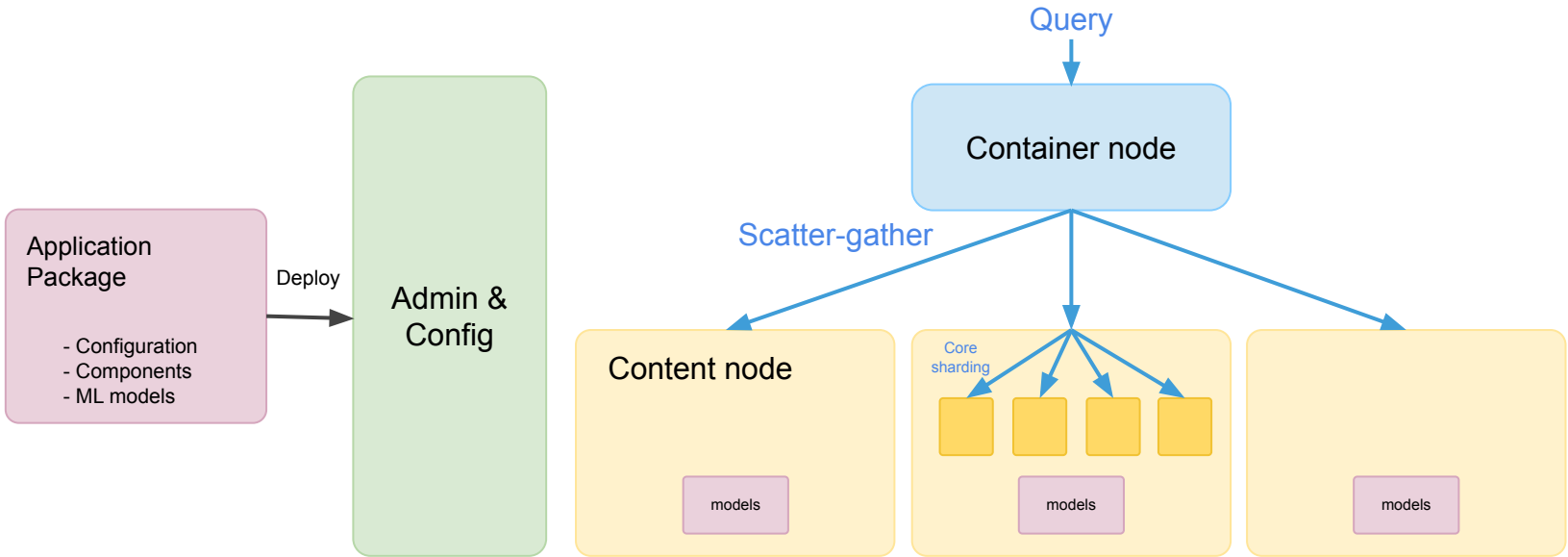
Vespa architecture



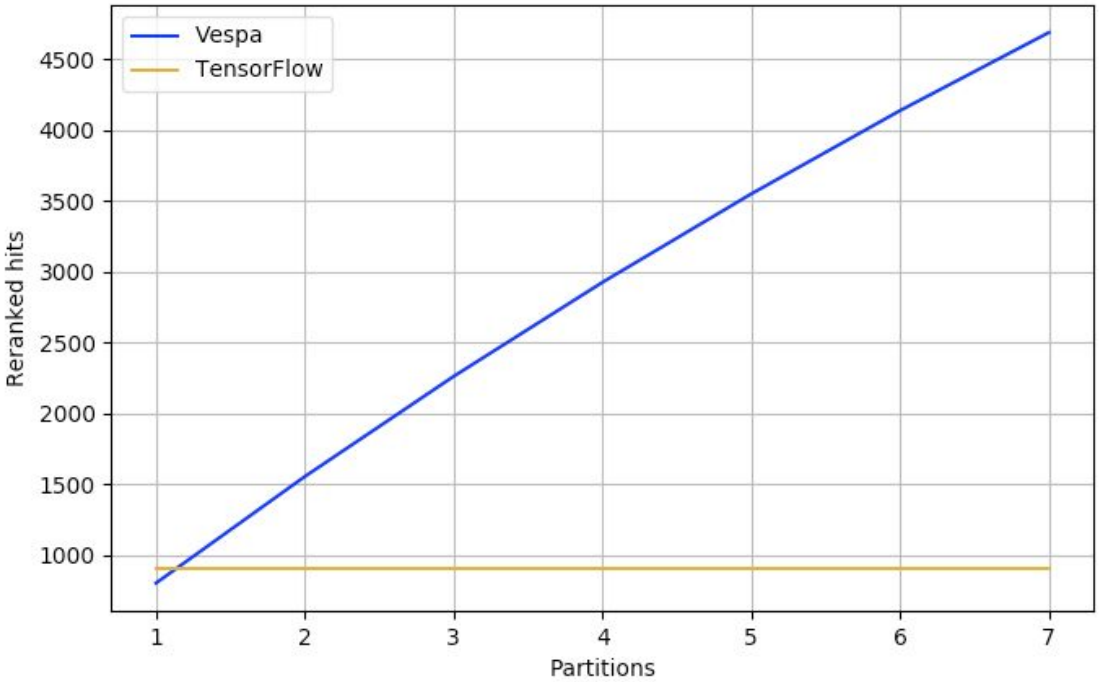
Scalable low latency execution:

How to bound latency in three easy steps

1) Parallelization
2) Prepare data structures at write time and in the background
3) Move execution to data nodes



Model evaluation: Increasing number of evaluated items



Latency: 100ms @ 95%
Throughput: 500 qps

Query execution and data storage

- Document-at-a-time evaluation over all query operators
- *index* fields:
  - positional text indexes (dictionaries + posting lists), and
  - B-trees in memory containing recent changes
- *attribute* fields:
  - In-memory forward dense data, optionally with B-trees
  - For search, grouping and ranking
- Transaction log for persistence+replay
- Separate store of raw data for serving+recovery+redistribution
- One instance of all of this per doc schema

Data distribution

Vespa auto-distributes data over

- A set of nodes
- With a certain replication factor
- *Optionally*: In multiple node groups
- *Optionally*: With locality (e.g personal search)

Changes to nodes/configuration -> Online data redistribution

No need to manually partition data

Distribution based on CRUSH algorithm: Minimal data movement without registry

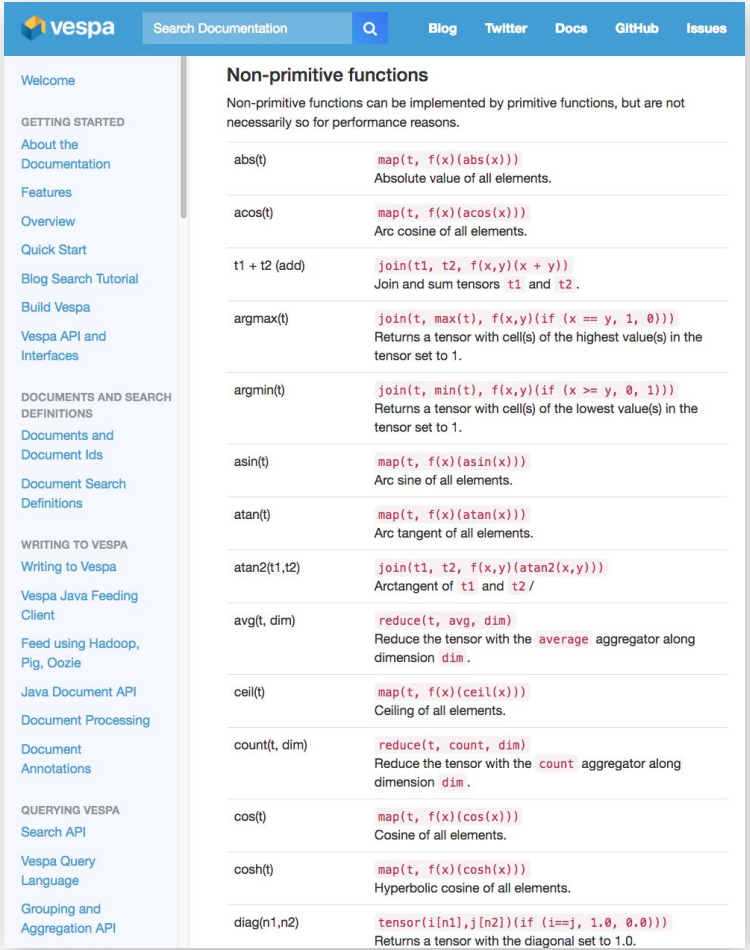
Inference in Vespa

Tensor data model: Multidimensional collections of numbers. In queries, documents, models

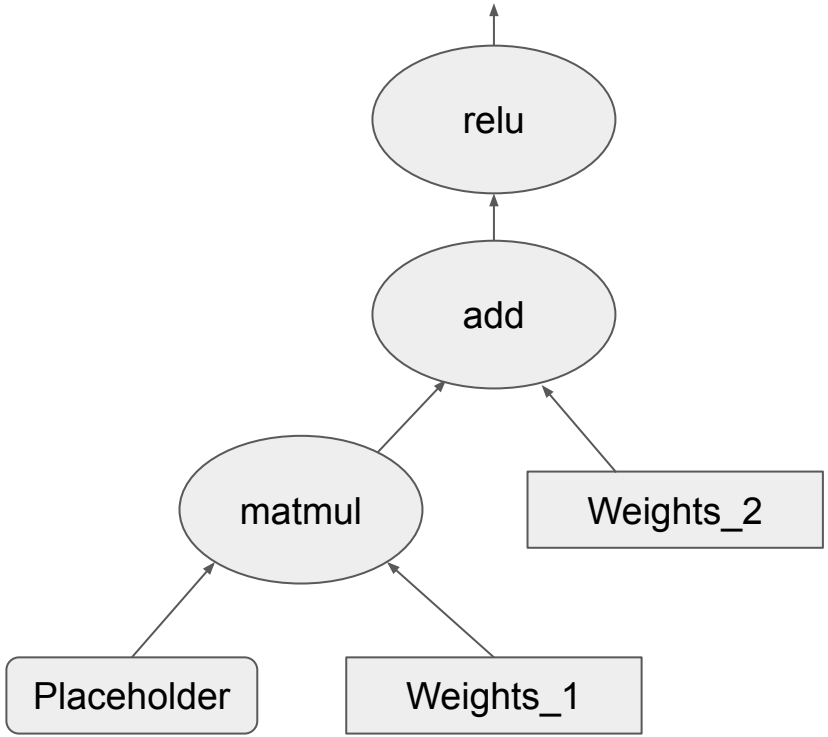
Tensor math express all common machine-learned models with join, map, reduce

TensorFlow, ONNX and XGBoost integration: Deploy TensorFlow, ONNX (SciKit, Caffe2, PyTorch etc.) and XGBoost models directly on Vespa

Vespa execution engine optimized for repeated execution of models over many data items and running many inferences in parallel



Converting computational graphs to Vespa tensors




```
map(
    join(
        reduce(
            join(
                Placeholder,
                Weights_1,
                f(x,y)(x * y)
            ),
            sum,
            d1
        ),
        Weights_2,
        f(x,y)(x + y)
    ),
    f(x)(max(0,x))
)
```

Releases

New production releases of Vespa are made Monday to Thursday each week

Releases:

- Have passed our suite of ~1100 functional tests and ~75 performance tests
- Are already running the ~150 production applications in our cloud service

All development is in the open: https://github.com/vespa-engine/vespa

Big Data Serving and Vespa intro summary

Making the best use of big data often means **making decisions in real time**

Vespa is the **only open source** platform optimized for such big data serving

Available on **https://vespa.ai**

Quick start: Run a **complete application** (on a laptop or AWS) **in 10 minutes**
http://docs.vespa.ai/documentation/vespa-quick-start.html

Tutorial: Make a scalable blog search and recommendation engine from scratch
http://docs.vespa.ai/documentation/tutorials/blog-search.html

Where are we?

What’s big data serving?

Vespa - the big data serving engine

Vespa architecture and capabilities

Using Vespa

Installing Vespa

Rpm packages or **Docker images**

All nodes have the same packages/image

CentOS (On Mac and Win inside Docker or VirtualBox)

1 config variable:

```
echo "override VESPA_CONFIGSERVERS [config-server-hostnames]" >> $VESPA_HOME/conf/vespa/default-env.txt
```

https://docs.vespa.ai/documentation/vespa-quick-start.html

https://docs.vespa.ai/documentation/vespa-quick-start-centos.html

https://docs.vespa.ai/documentation/vespa-quick-start-multinode-aws.html

Configuring Vespa: Application packages

Manifest-based configuration

All of the application: system config, schemas, jars, ML models

deployed to Vespa:

- vespa-deploy prepare [application-package-path]
- vespa-deploy activate

Deploying again carries out changes made

Most changes happen live (including Java code changes)

If actions needed: List of actions needed are returned by deploy prepare

https://docs.vespa.ai/documentation/cloudconfig/application-packages.html

A complete application package, 1: Services/clusters

./services.xml

```
<services version='1.0'>

  <container id='default' version='1.0'>
    <search/>
    <document-api/>
    <nodes>
      <node hostalias=”node1”/>
    </nodes>
  </container>

  <content id='music' version='1.0'>
    <redundancy>2</redundancy>
    <documents>
      <document mode='index' type='music'/>
    </documents>
    <nodes>
      <node hostalias=”node2” distribution-key=”1”/>
      <node hostalias=”node3” distribution-key=”2”/>
    </nodes>
  </content>

</services>
```

./hosts.xml

```
<hosts>

  <host name="host1.domain.name">
    <alias>node1</alias>
  </host>

  <host name="host2.domain.name">
    <alias>node2</alias>
  </host>

  <host name="host3.domain.name">
    <alias>node3</alias>
  </host>

</hosts>
```

A complete application package, 2: Schema(s)

./searchdefinitions/music.sd:

```
search music {

    document music {
        field artist type string {
            indexing: summary | index
        }
        field album type string {
            indexing: summary | index
        }
        field track type string {
            indexing: summary | index
        }
        field popularity type int {
            indexing: summary | attribute
            attribute: fast-search
        }
    }

    rank-profile song inherits default {
        first-phase {
            expression {
                0.7 * nativeRank(artist,album,track) +
                0.3 * attribute(popularity)
            }
        }
    }
}
```

https://docs.vespa.ai/documentation/search-definitions.html

Calling Vespa: HTTP(S) interfaces

POST docs/individual fields:

```
{
    "fields": {
        "artist": "War on Drugs",
        "album": "A Deeper Understanding",
        "track": "Thinking of a Place",
        "popularity": 0.97
    }
}
```

to http://host1.domain.name:8080/document/v1/music/music/docid/1

(or use the Vespa Java HTTP client for high throughput)

GET single doc: http://host1.domain.name:8080/document/v1/music/music/docid/1

GET query result: http://host1.domain.name:8080/search/?query=track:place

Operations in production

No single point of failure

Automatic failover + data recovery -> no time-critical ops needed

Log collection to config server

Metrics integration

- Prometheus integration in https://github.com/vespa-engine/vespa_exporter
- Or, access metrics from a web service on each node

Matching

Matching finds all the documents matching a query

Query = Tree of operators:

- TERM, AND, OR, PHRASE, NEAR, RANK, WeightedSet, …
- RANGE, WAND

Goal of matching: a) Selecting a subset of documents, or b) Skipping for perf.

Queries are evaluated in parallel:

over all clusters, document types, partitions, and N cores

Queries are passed in HTTP requests (YQL), or constructed in Searchers

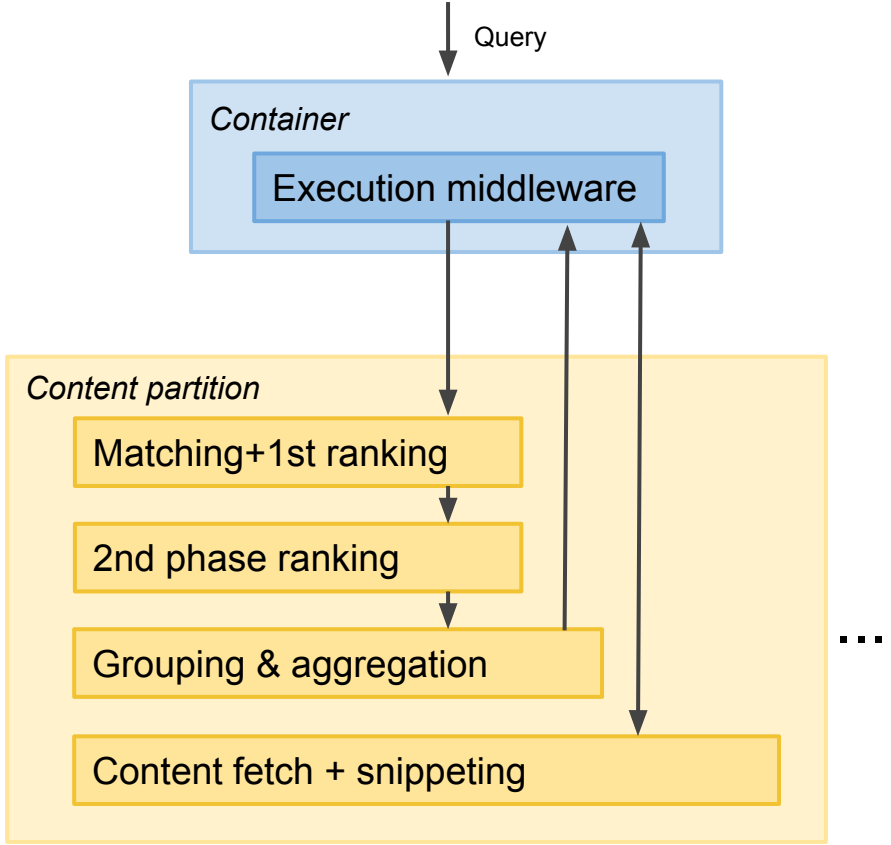
Execution

Low latency computation over large data sets

… by parallelization over nodes and cores

... pushing execution to the data

... and preparing data structures at write time



Ranking/inference

It's just math

Ranking expressions: Compute a score from *features*

a + b * log(c) - if(e > f, g, h)

- Constant features (in application package)
- Document features
- Query features
- Match features: Computed from doc+query data at matching time

First-phase ranking: Computed during matching, on each match

Second-phase ranking: Optional re-ranking of top n on each partition

https://docs.vespa.ai/documentation/ranking.html

Match feature examples

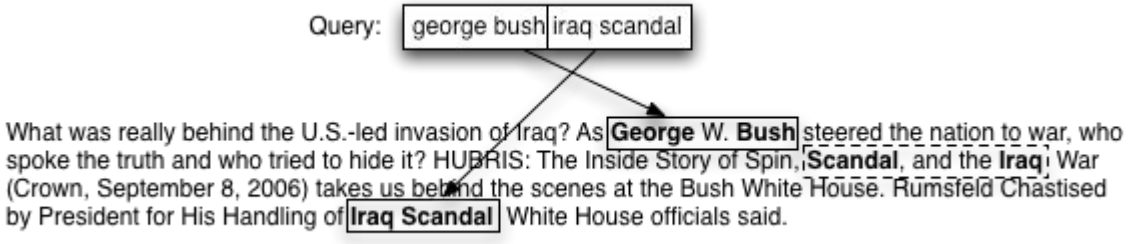
- *nativeRank* feature: Pretty good text ranking out of the box
- Text ranking: *fieldMatch* feature set
  - Positional info
  - Text segmentation
- Multivalue text field signal aggregation:
  - *elementCompleteness*
  - *elementSimilarity*
- Geo distance
  - *closeness*
  - *distance*
  - *distanceToPath*
- Time ranking:
  - *freshness*
  - *age*

https://docs.vespa.ai/documentation/reference/rank-features.html

fieldMatch text ranking feature set

Accurate proximity based text matching features

Highest on the quality-cost tradeoff curve: Usually for second-phase ranking



fieldMatch feature: Aggregate text relevance score

Fine-grained fieldMatch sub-features: Useful for ML ranking

Machine learned scoring

Example: Text search

- Supervised machine-learned ranking of matches to a user query

Example: Recommendation/personalization

- Query is a user+context in some vector/tensor space
- Document belongs to same space
- Evaluate machine-learned model on *all* documents
  - ...ideally - optimizations to reduce cost: 2nd phase, WAND, match-phase, clustering, …
- Reinforcement learning

“Search 2.0”

Defining Search 2.0

Search 1.0	**Search 2.0**
Documents & Terms	Embeddings
Lookup, Matching	Nearest Neighbor Retrieval
Index optimization	Approximate nearest neighbors/quantization
Hand-engineered ranking	Learned and encoded into embedding
Faceting	Conditioning, Disentangled representations
Query refinement	Manifold traversal

Gradient boosted decision trees

- Commonly used for supervised learning of text search ranking
- Defer most “Natural language intelligence” to ranking instead of matching -> better result at higher cpu cost … but modern hardware has sufficient power
- Ranking function: Sum of decision trees
- A few hundreds/thousand trees
- Written as a sum of nested *if* expressions on scalars
- Vespa can read XGBoost models
- Special optimizations for GBDT-shaped ranking expressions
- Training: Issue queries which requests ranking features in the response

… however

Universal Approximation Functions for Fast Learning to Rank

Replacing Expensive Regression Forests with Simple Feed-Forward Networks

Daniel Cohen*, John Foley*, Hamed Zamani, James Allan and W. Bruce Croft
Center for Intelligent Information Retrieval
University of Massachusetts Amherst
{dcohen,jfoley,zamani,allan,croft}@cs.umass.edu

We observe CPU document scoring speed improvements of up to 400x over traditional algorithms and up to 10x over state-of-the-art algorithms with no measurable loss in mean average precision.

Tensors

A data type in ranking expressions (in addition to scalars)

Makes it possible to deploy large and complex ML models to Vespa

- Deep neural nets
- FTRL (regression models with millions of parameters)
- Word2vec models
- etc.

https://docs.vespa.ai/documentation/tensor-intro.html

What is a tensor?

Tensor: A multidimensional array which can be used for computation

Textual form: { {address}:double, .. } where address is {identifier:value},...

Examples

- 0-dimensional: A scalar {{}:0.1}
- 1-dimensional: A vector {{x:0}:0.1, {x:1}:0.2}
- 2-dimensional: A matrix {{x:0,y:0}:0.1, {x:0,y:1}:0.2}

Indexed tensor dimensions: Values addressed by *numbers*, continuous from 0

Mapped tensor dimensions: Values addressed by *identifiers*, sparse

Tensor sources

Tensors may be added to **documents**

```
field my_tensor type tensor(x{},y[10]) { ... }
```

… **queries**

```
query.getRanking().getFeatures()

.put("my_tensor_feature", Tensor.from("{{x:foo,y:0}:1.3}"));
```

… and **application packages**

```
constant tensor_constant {

    file: constants/constant_tensor_file.json.lz4

    type: tensor(x{})

}
```

… or be created on the fly from other doc fields

From document **weighted sets**

```
tensorFromWeightedSet(source, dimension)
```

From document **vectors**

```
tensorFromLabels(source, dimension)
```

From single attributes

```
concat(attribute(attr1), attribute(attr2), dimension)
```

Tensor computation

6 primitive operations

map(tensor, f(x)(expr))

reduce(tensor, aggregator, dim1, dim2, ...)

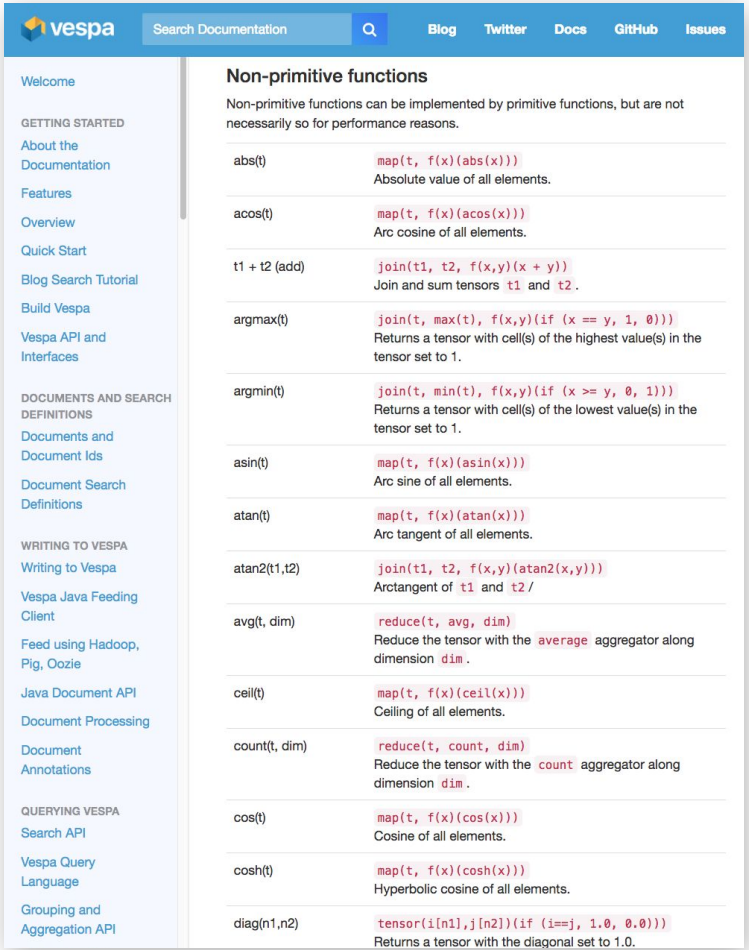
join(tensor1, tensor2, f(x,y)(expr))

tensor(tensor-type-spec)(expr)

rename(tensor, from-dims, to-dims)

concat(tensor1, tensor2, dim)

https://docs.vespa.ai/documentation/reference/tensor.html



The tensor join operator

Naming is awesome, or computer science strikes again!

Generalization of other tensor products:

Hadamard, tensor product, inner, outer matrix product

Like the regular tensor product, it is associative:

a * (b * c) = (a * b) * c

Unlike the tensor product, it is also commutative:

a * b = b * a

Use case: FTRL

```
sum(                                    // model computation:
    tensor0 * tensor1 * tensor2 // feature combinations
    * tensor3                           // model weights application
)
```

Where tensors 0, 1, 2 come from the document or query:

```
tensor(userlocation{}), tensor(userinterests{}), tensor(articletopics{})
```

and tensor 3 comes from the application package:

```
tensor(userlocation{}, userinterests{}, articletopics{})
```

Use case: Neural net

```
rank-profile nn_tensor {

    function nn_input() {
        expression: concat(attribute(user_item_cf), query(user_item_cf), input)
    }

    function hidden_layer() {
        expression: relu(sum(nn_input * constant(W_hidden), input) + constant(b_hidden))
    }

    function final_layer() {
        expression: sigmoid(sum(hidden_layer * constant(W_final), hidden) + constant(b_final))
    }

    first-phase {
        expression: sum(final_layer)
    }

}
```

TensorFlow, ONNX and XGBoost integration

1) **Save** models directly to <application package>/models/
2) **Reference** model outputs in ranking expressions:

Faster than native TensorFlow evaluation

More scalable as evaluation happens at content partitions

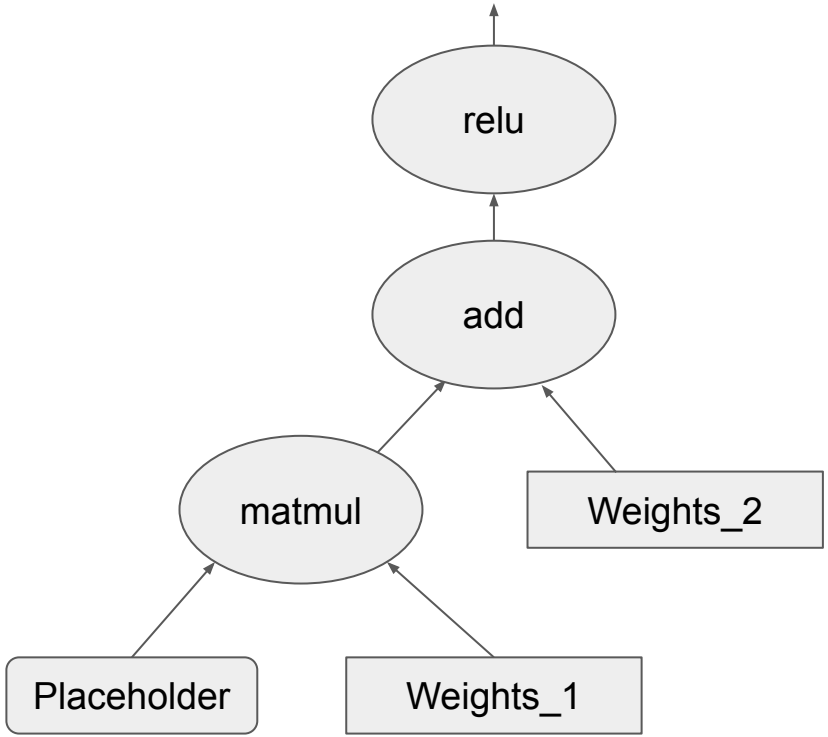
```
search music {

    ...

    rank-profile song inherits default {
        first-phase {
            expression {
                0.7 * nativeRank(artist,album,track) +
                0.1 * tensorflow(tf-model-dir) +
                0.1 * onnx(onnx-model-file, output) +
                0.1 * xgboost(xgboost-model-file)
            }
        }
    }
}
```

https://docs.vespa.ai/documentation/tensorflow.html
https://docs.vespa.ai/documentation/onnx.html
https://docs.vespa.ai/documentation/xgboost.html

Converting computational graphs to Vespa tensors




```
map(
    join(
        reduce(
            join(
                Placeholder,
                Weights_1,
                f(x,y)(x * y)
            ),
            sum,
            d1
        ),
        Weights_2,
        f(x,y)(x + y)
    ),
    f(x)(max(0,x))
)
```

Grouping and aggregation

Organizing data at request time

```
…(yql query)... | all(group(advertiser) each( output(count()) max(3) each(output(summary()))))
```

For navigational views, visualization, grouping, diversity etc.

Evaluated over **all** matches

… distributed over all partitions

Any number of levels and parallel groupings (may become expensive)

https://docs.vespa.ai/documentation/grouping.html

Grouping operations

`all`: Perform an operation on a list

`each`: Perform an operation on each item in a list

`group`: Create a new list level

`max`: Limit the number of elements in a list

`order`: Order a list

`output`: Add some data to the output produced by the current list/element

Grouping aggregators and expressions

Aggregators: `count`, `sum`, `avg`, `max`, `min`, `xor`, `stddev`, `summary`

(summary: Output data from a document)

Expressions:

- Standard math
- Static and dynamic bucketing
- Time
- Geo (zcurve)
- Access attributes + relevance score of documents

Grouping examples

Group hits and output the count in each group : `all(group(a) each(output(count())))`

Group hits and output the best in each group: `all(group(a) each(max(1) each(output(summary()))))`

Group into fixed buckets, then on attribute “a”, and count hits in leafs:

```
all(group(fixedwidth(n, 3)) each(group(a) max(2) each(output(count()))))
```

Group into today, yesterday, last week and month, group each into separate days:

```
all(group(predefined((now() - a) / (60 * 60 * 24),
                     bucket(0,1), bucket(1,2),
                     bucket(3,7), bucket(8,31))) each(output(count())
  all(max(2) each(output(summary())))
    all(group((now() - a) / (60 * 60 * 24)) each(output(count())
      all(max(2) each(output(summary()))))));
```

https://docs.vespa.ai/documentation/reference/grouping-syntax.html

Container for Java components

- Query and result processing, federation, etc.: Searchers
- Document processors
- General request handlers
- Any Java component (no Vespa interface/class needed)
- Dependency injection, component config
- Hotswap of code, without disrupting traffic
- Query profiles
- HTTP serving through embedding Jetty

https://docs.vespa.ai/documentation/jdisc/

Summary

Making the best use of big data often implies **making decisions in real time**

Vespa is the **only open source** platform optimized for such big data serving

Available on **https://vespa.ai**

Quick start: Run a **complete application** (on a laptop or AWS) **in 10 minutes**
http://docs.vespa.ai/documentation/vespa-quick-start.html

Tutorial: Make a scalable blog search and recommendation engine from scratch
http://docs.vespa.ai/documentation/tutorials/blog-search.html

Questions?

By Vespa architect @jonbratseth

vespa