



*One Engine, one Query Language.
Multiple Data Models.*



+



+



Hej, trevligt att träffas!

My name is Jan!

I am from Cologne, Germany

I work for a database company named ArangoDB...

...on an open-source database also named ArangoDB

Multi-model databases (with emphasis on ArangoDB)



Databases market trends



Database trends

Relational
197x - now

relational storage, fixed schema, SQL, transactions

"NoSQL"
200x - now

horizontal scaling, high availability, low-latency,
non-relational storage
but: no complex querying, no transactions!

"NewSQL"
201x - now

fusing scalability and resilience with
complex querying and transactions

Now the database market is fragmented

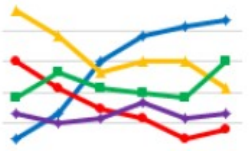
db-engines.com lists 343 database systems as of now:

Ranking > Complete Ranking RSS RSS Feed

DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the [method](#) of calculating the scores.



WTF?!?

343 systems in ranking, January 2019

We now got: relational databases, key-value stores, document stores, time-series databases, graph databases, search engines, object databases, XML databases, RDF stores, ...

What database
to use for a
new project?



Solution: keep using relational databases

- one obvious solution is to keep on using relational databases for all tasks
- this will work if the database schema is relatively fixed
- for dynamically structured data, or varyingly connected data, this approach will likely run into problems
- it will very likely be a problematic approach if horizontal scaling and availability are required

Alternative: "Polyglot persistence"

- "use the right tool(s) for the job"
- assumption: specialized products are better suited than generic products
- for example, why not use (all at the same time):
 - an in-memory key-value store for caching,
 - a persistent key-value store for user management,
 - a relational database for order processing,
 - a graph database for recommendations,
 - a search engine for fulltext search?

Issues with polyglot persistence

- Requires learning, administering and maintaining multiple technologies
- Needs custom scripts and / or application logic for shipping data from one system to the other, or for syncing systems
- Potential atomicity and consistency issues across the different database systems (i.e. no transactions)

Multi-model databases to the rescue?



What are multi-model databases?

- The main idea behind multi-model databases is to support different data models in the same database product
- So that the different data models in a multi-model database can easily be combined in queries and even transactions (compasability)

Multi-model database key benefits

- Having to master and administer only a single technology
- Being less locked-in to one specific data model and its limitations
- More flexible in case the requirements change
- Removing atomicity and consistency issues for syncing different datastores, and not having to deal with these issues in the application layer

Multi-model databases (from Wikipedia)

Databases [\[edit \]](#)

Main article: [Comparison of multi-model databases](#)

Multi-model databases include (in alphabetic order):

- [ArangoDB](#) – document (JSON), graph, key-value
- [Cosmos DB](#) – document (JSON), key-value, SQL
- [Couchbase](#) – document (JSON), key-value, [N1QL](#)
- [Datastax](#) – key-value, tabular, graph
- [EnterpriseDB](#) – document (XML and JSON), key-value
- [MarkLogic](#) – document (XML and JSON), graph triplestore, binary, SQL
- [Oracle Database](#) – relational, document (JSON and XML), graph triplestore, property graph, key-value, objects
- [OrientDB](#) – document (JSON), graph, key-value, reactive, SQL
- [SAP HANA](#) – relational, document (JSON), graph, streaming

"Layered multi-model" is also a trend

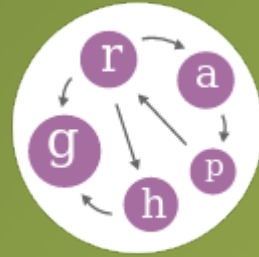
- Some traditional relational databases recently got extensions for key-value handling and document storage
- Strictly speaking, all of them could be called "multi-model"
- Often the extra data models have been "bolted on top" of these products, leading to poor support or inefficient implementations

ArangoDB, the native multi-model database



At a glance

ArangoDB is a native multi-model database, with support for key-values, documents, graphs and (recently) search functionality



These data models are composable and can be combined via AQL, ArangoDB's unified query language

Databases, collections, documents

- On the highest level, data in ArangoDB is organized in "databases" and "collections" (think "schemas" and "tables")
- Collections are used to store "documents" of similar types
- "Documents" are just JSON objects with arbitrary attributes, with optional nesting (sub-objects, sub-arrays)
- There is no fixed schema for documents (any JSON object is valid)

Homogeneous documents

In the easiest case, documents in a collection are homogeneous (i.e. same attributes and types)

Example use case: product categories

```
{ "_key" : "books",    "title" : "Books" }  
{ "_key" : "cam",      "title" : "Camera products" }  
{ "_key" : "kitchen",  "title" : "Kitchen Appliances" }  
{ "_key" : "toys",     "title" : "Toys & Games" }
```

Processing such data in AQL queries is as straightforward as with a SQL query on a relational, fixed-schema table

AQL queries – hello world examples

```
// SELECT c.* FROM categories c WHERE c._key IN ...  
FOR c IN categories  
  FILTER c._key IN [ 'books', 'kitchen' ]  
  RETURN c
```

```
// SELECT c._key, c.title FROM categories c ORDER BY c.title  
FOR c IN categories  
  SORT c.title  
  RETURN {  
    _key: c._key,  
    title: c.title  
  }
```

Heterogeneous documents

In more complex cases, documents in the same collection are heterogeneous (i.e. attribute names or types differ)

This is often the case when objects have some common attributes, but there are different special attributes for sub-types

Example use case: product catalogs

Heterogeneous documents example

```
{
  "_key" : "A053720452",
  "category" : "books",
  "name" : "Harry Potter and the Cursed Child",
  "author" : "Joanne K. Rowling",
  "isbn" : "978-0-7515-6535-5",
  "published" : 2016
}

{
  "_key" : "ZB4061305X34",
  "category" : "toys",
  "name" : "Nerf N-Strike Elite Mega CycloneShock Blaster",
  "upc" : "630509278862",
  "colors" : [ "black", "red" ]
}
```

Querying heterogeneous documents

ArangoDB is schema-less, meaning optional attributes can be queried the same way as common attributes:

```
// no SQL equivalent, because no optional attributes allowed
FOR p IN products
  FILTER p.name == @search OR
         p.isbn == @search OR
         p.upc  == @search
RETURN p
```

The value of non-existing attributes evaluates to `null` at runtime

AQL – joins

Like SQL, AQL allows joining data from multiple sources:

```
// SELECT c.* AS category, p.* AS product FROM categories c,  
//   products p WHERE p.category = c._key ORDER BY c.title, p.name  
FOR c IN categories  
  FOR p IN products  
    FILTER p.category == c._key  // "category" should be indexed!  
    SORT c.title, p.name  
  RETURN {  
    category: c  
    product: p  
  }
```


Indexes for more efficient querying

- Documents can be accessed efficiently by their primary key, which is automatically indexed in ArangoDB ("_key" attribute)
- Extra secondary indexes can be created as needed to support efficient querying by different attributes
- Index types: hash, skiplist, geo, fulltext

AQL – subqueries

```
// this will return an array of product details per category
FOR c IN categories
  RETURN {
    category: c.title,
    products: (
      FOR p IN products
        FILTER p.category == c._key
        SORT p.name
        RETURN { _key: p._key, name: p.name }
    )
  }
```

AQL – aggregation

```
// SELECT s.year, s.month, s.product, SUM(c.count) AS count,  
//      SUM(s.amount) AS amount FROM sales s WHERE s.year  
//      BETWEEN 2017 AND 2019 GROUP BY s.year, s.month, s.product  
FOR s IN sales  
  FILTER s.year >= 2017 AND s.year <= 2019  
  COLLECT    year      = s.year,  
             month     = s.month,  
             product   = s.product  
  AGGREGATE count      = SUM(s.count),  
            amount     = SUM(s.amount)  
  
RETURN {  
  year, month, product, count, amount  
}
```

AQL – data modification

```
// update all documents which don't have a "lastModified"
// attribute, store the current date in it, and return the _keys
// of the documents updated
FOR p IN products
  FILTER !HAS(p, "lastModified")
  UPDATE p WITH {
    lastModified: DATE_ISO8601(DATE_NOW())
  } IN products
RETURN OLD._key
```

AQL – data modification

```
// delete documents of the specified categories, and return
// all data of the deleted documents
FOR p IN products
  FILTER p.category IN [ "cam", "video" ]
  REMOVE p IN products RETURN OLD
```

Other AQL features (not shown here)

- Geo queries
- Fulltext indexing and searching

The graph data model

- ArangoDB also supports the graph data model
- Graph queries can reveal which documents are directly or indirectly connected to which other documents, and via what connections
- Graphs are often used for data exploration, and to understand connections in the data

Edges

- In graphs, connections between documents are called "edges"
- In ArangoDB edges are stored in "edge collections"
- Edges have "`_from`" and "`_to`" attributes, which reference the connected vertices
- Edges are always directed (`_from -> _to`), but can also be queried in opposite order

Edge collection example

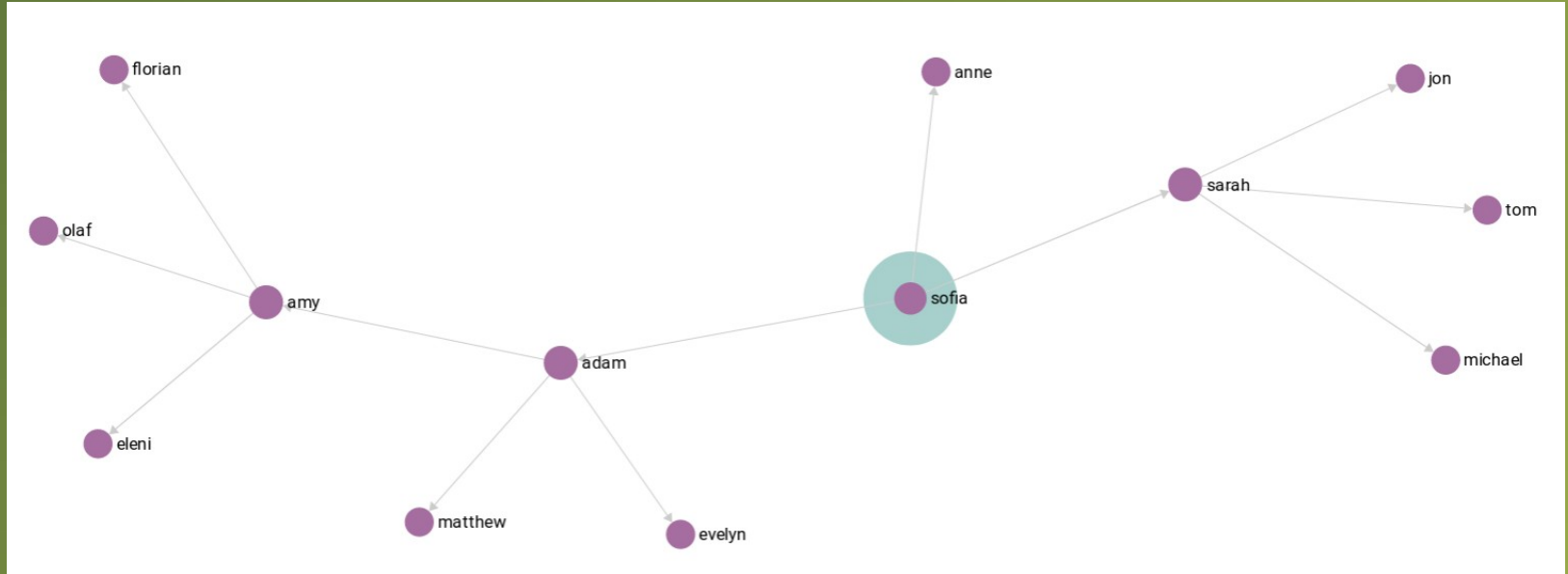
Let's assume there are some "employees" documents like this:

```
{ "_key" : "sofia",   "_id" : "employees/sofia" }  
{ "_key" : "adam",   "_id" : "employees/adam" }  
{ "_key" : "sarah",  "_id" : "employees/sarah" }  
{ "_key" : "jon",    "_id" : "employees/jon" }
```

And there is an "isManagerOf" edge collection connecting them:

```
{ "_from" : "employees/sofia",  "_to" : "employees/adam" }  
{ "_from" : "employees/sofia",  "_to" : "employees/sarah" }  
{ "_from" : "employees/sarah",  "_to" : "employees/jon" }  
...
```

Graph example, employees graph



AQL graph traversals, examples

```
// find all employees that "sofia" is a direct manager of
FOR emp IN 1..1 OUTBOUND 'employees/sofia' isManagerOf
  RETURN emp._key
```

```
// find direct and indirect subordinates of "sofia":
FOR emp IN 1..5 OUTBOUND 'employees/sofia' isManagerOf
  RETURN emp._key
```

AQL graph traversals, more examples

```
// find employees that "olaf" is indirectly connected to (level 2)
FOR emp IN 2..2 ANY 'employees/olaf' isManagerOf
  RETURN emp._key
```

```
// find all line managers of employee "olaf", calculate distance
FOR emp, edge, path IN 1..5 INBOUND 'employees/olaf'
  isManagerOf
  RETURN {
    who: emp._key,
    level: LENGTH(path.edges)
  }
```

Some graph use cases

- Organizational structures
- Social networks (friends, friends of friends)
- Recommendation and prediction engines
- Product component planning and assessment
- Network and infrastructure assessment
- Fraud detection

Advanced graph querying

- Graphs can also consist of many different collections
- While traversing the graph, it is also possible to filter on edge attributes and exclude unwanted edges or paths
- Graphs can contain cycles, so it is possible to stop searching already-visited documents and edges
- Shortest-path queries are also supported

Deployment options



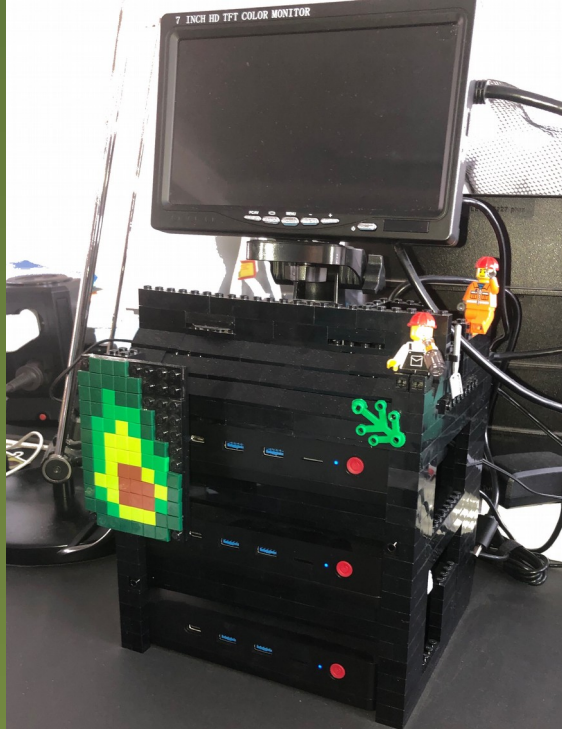
Deployment modes

- Single server
- Active failover: 2+ single servers with asynchronous replication and automatic failover
- Cluster: multiple query processing and storage nodes, horizontal write scaling and resilience

Supported platforms

- Linux, with 64 bit OS
- Windows 10, Windows 7 SP1, Windows server, with 64 bit OS
- Mac OS, Sierra and newer
- Kubernetes (including Google GKE, Amazon EKS, Pivotal PKS)
- Docker

ArangoDB clusters running on ARM64



Bonus features



Administration via web interface

The screenshot displays the ArangoDB Enterprise Edition web interface. The top header bar is dark blue and contains the ArangoDB logo, the text "ENTERPRISE EDITION", and system status information: "USER: ROOT", "DB: _SYSTEM", and "HEALTH: GOOD". A sidebar on the left lists navigation options: DASHBOARD, COLLECTIONS (selected), VIEWS, QUERIES, GRAPHS, SERVICES, USERS, DATABASES, REPLICATION, LOGS, SUPPORT, and HELP US. The main content area shows a grid of collection tiles. Each tile includes an icon (a plus sign for 'Add Collection', a document icon for single collections, or a share icon for graph collections) and the collection name with a green 'loaded' status indicator. The collections shown are: acl, acl, pageViews, productRatings, products, purchases, ratings, relations, tags, and users. A search bar is located in the top right of the main area.

ArangoDB
ENTERPRISE EDITION

USER: ROOT DB: _SYSTEM HEALTH: GOOD

DASHBOARD

COLLECTIONS

VIEWS

QUERIES

GRAPHS

SERVICES

USERS

DATABASES

REPLICATION

LOGS

SUPPORT

HELP US

Search...

+ Add Collection

acl loaded

pageViews loaded

productRatings loaded

products loaded

purchases loaded

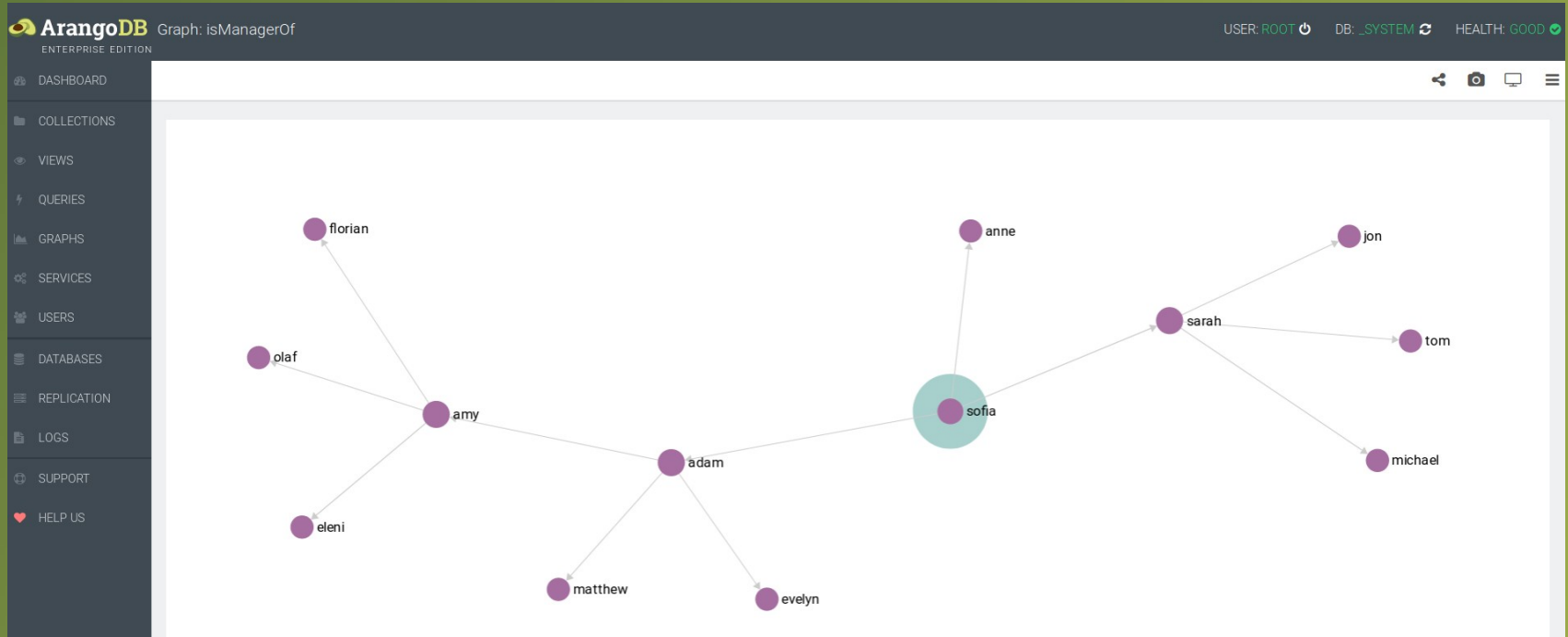
ratings loaded

relations loaded

tags loaded

users loaded

Built-in graph viewer



Additional enterprise edition features

- "Smart graphs" (colocating of documents with all their outgoing edges on the same physical server)
- Datacenter-to-datacenter (DC2DC) replication
- Encryption at rest, encrypted backups
- LDAP integration for user management
- Support subscription with SLAs

ArangoDB Foxx, for exposing data via REST APIs



Foxx – built-in microservice framework

- Combine your data with an API and you have a microservice!
- ArangoDB comes with Foxx, a built-in framework for easily exposing dedicated database data via REST APIs
- The REST API code is written using JavaScript, which runs inside the ArangoDB server, close to the data

Foxx example, custom API code

```
// endpoint: HTTP GET /managersOf/:employee
router.get("/managersOf/:employee", function (req, res) {
  let query = `FOR emp, edge, path IN 1..5 INBOUND
    CONCAT('employees/', @employee) isManagerOf
    RETURN { who: emp._key, level: LENGTH(path.edges) }' `;

  let result = db._query(query, {
    employee: req.param("employee") // use path parameter
  }).toArray();

  res.json(result); // return JSON result to caller
});
```

Foxx – built-in microservice framework

- It is easy to add input parameter validation and sanitation
- API documentation facility is already included in the framework
- API documentation can be made available for humans (via Swagger UI) and client applications as a service description JSON file

Example with input validation and docs

Adding the below code to the existing route adds simple validation for the input parameter and also API documentation:

```
router.get("/managersOf/:employee", function (req, res) {  
  ... // previous code here  
})  
.pathParam("employee", joi.string().required(), "employee name")  
.summary("returns managers of the employee")  
.response(200, 'will return a list of the managers')  
.error(404, 'employee has no managers');
```

Foxx – use cases

- Data validation, sanitation and transformation
- Keeping things private by filtering data and running projections before returning it
- Complex permissioning based on database data or queries
- Eliminating network hops by bundling multiple database queries in a single server-side REST API method

Summary

- ArangoDB is a free and open-source native multi-model database
- Provides support for key-values, documents, graphs
- Search functionality has been recently added and will be extended
- Different data models can be used flexibly
- AQL query language can be used for rich querying, combining all data models
- Various deployment modes, including cluster setups

Tack alla er!

Any questions?
Or t-shirts?



Getting in touch

- Website: <https://www.arangodb.com/>
- Slack: <https://slack.arangodb.com/>
- Github: [arangodb/arangodb](https://github.com/arangodb/arangodb)
- Twitter: [@arangodb](https://twitter.com/arangodb)
- Stack Overflow: [#arangodb](https://stackoverflow.com/questions/tagged/arangodb)