Table of Contents

Introduction

The ArangoDB query language (AQL) can be used to retrieve and modify data that are stored in ArangoDB. The general workflow when executing a query is as follows:

- A client application ships an AQL query to the ArangoDB server. The query text contains everything ArangoDB needs to compile the result set
- ArangoDB will parse the query, execute it and compile the results. If the query is invalid or cannot be executed, the server will return an error that the client can process and react to. If the query can be executed successfully, the server will return the query results (if any) to the client

AQL is mainly a declarative language, meaning that a query expresses what result should be achieved but not how it should be achieved. AQL aims to be human-readable and therefore uses keywords from the English language. Another design goal of AQL was client independency, meaning that the language and syntax are the same for all clients, no matter what programming language the clients may use. Further design goals of AQL were the support of complex query patterns and the different data models ArangoDB offers.

In its purpose, AQL is similar to the Structured Query Language (SQL). AQL supports reading and modifying collection data, but it doesn't support data-definition operations such as creating and dropping databases, collections and indexes. It is a pure data manipulation language (DML), not a data definition language (DDL) or a data control language (DCL).

The syntax of AQL queries is different to SQL, even if some keywords overlap. Nevertheless, AQL should be easy to understand for anyone with an SQL background.

For some example queries, please refer to the pages Data Queries and Usual query patterns.

How to invoke AQL

AQL queries can be executed using:

- the web interface,
- the `db` object (either in arangosh or in a Foxx service)
- or the raw HTTP API.

There are always calls to the server's API under the hood, but the web interface and the `db` object abstract away the low-level communication details and are thus easier to use.

The ArangoDB Web Interface has a specific tab for AQL queries execution.

You can run AQL queries from the ArangoDB Shell with the _query and _createStatement methods of the `db` object. This chapter also describes how to use bind parameters, statistics, counting and cursors with arangosh.

If you are using Foxx, see how to write database queries for examples including tagged template strings.

If you want to run AQL queries from your application via the HTTP REST API, see the full API description at HTTP Interface for AQL Query Cursors.

Executing queries from Arangosh

Within the ArangoDB shell, the *_query* and *_createStatement* methods of the *db* object can be used to execute AQL queries. This chapter also describes how to use bind parameters, counting, statistics and cursors.

with db._query

One can execute queries with the *_query* method of the *db* object. This will run the specified query in the context of the currently selected database and return the query results in a cursor. The results of the cursor can be printed using its *toArray* method:

```
arangosh> db._create("mycollection")
arangosh> db.mycollection.save({ _key: "testKey", Hello : "World" })
arangosh> db._query('FOR my IN mycollection RETURN my._key').toArray()
```

show execution results

db._query Bind parameters

To pass bind parameters into a query, they can be specified as second argument to the *_query* method:

```
arangosh> db._query(
........> 'FOR c IN @@collection FILTER c._key == @key RETURN c._key', {
........>   '@collection': 'mycollection',
........>   'key': 'testKey'
........> }).toArray();
[
  "testKey"
]
```

ES6 template strings

It is also possible to use ES6 template strings for generating AQL queries. There is a template string generator function named *aql*; we call it once to demonstrate its result, and once putting it directly into the query:

```
var key = 'testKey';
aql`FOR c IN mycollection FILTER c._key == ${key} RETURN c._key`;
{
  "query" : "FOR c IN mycollection FILTER c._key == @value0 RETURN c._key",
  "bindVars" : {
    "value0" : "testKey"
  }
}
```

```
arangosh> var key = 'testKey';
arangosh> db._query(
........> aql`FOR c IN mycollection FILTER c._key == ${key} RETURN c._key`
........> ).toArray();
[
  "testKey"
]
```

Arbitrary JavaScript expressions can be used in queries that are generated with the *aql* template string generator. Collection objects are handled automatically:

```
arangosh> var key = 'testKey';
arangosh> db._query(aql`FOR doc IN ${ db.mycollection } RETURN doc`
........> ).toArray();
```

show execution results
Note: data-modification AQL queries normally do not return a result (unless the AQL query contains an extra *RETURN* statement). When not using a *RETURN* statement in the query, the *toArray* method will return an empty array.

Statistics and extra Information

It is always possible to retrieve statistics for a query with the *getExtra* method:

```
arangosh> db._query(`FOR i IN 1..100
........>             INSERT { _key: CONCAT('test', TO_STRING(i)) }
........>                INTO mycollection`
........> ).getExtra();
```

show execution results
The meaning of the statistics values is described in Execution statistics. You also will find warnings in here; If you're designing queries on the shell be sure to also look at it.

Setting a memory limit

To set a memory limit for the query, pass *options* to the *_query* method. The memory limit specifies the maximum number of bytes that the query is allowed to use. When a single AQL query reaches the specified limit value, the query will be aborted with a *resource limit exceeded* exception. In a cluster, the memory accounting is done per shard, so the limit value is effectively a memory limit per query per shard.

```
arangosh> db._query(
........> 'FOR i IN 1..100000 SORT i RETURN i', {}, {
........>   memoryLimit: 100000
........> }).toArray();
[ArangoError 32: AQL: query would use more memory than allowed (while executing)]
```

If no memory limit is specified, then the server default value (controlled by startup option *--query.memory-limit* will be used for restricting the maximum amount of memory the query can use. A memory limit value of *0* means that the maximum amount of memory for the query is not restricted.

Setting options

There are further options that can be passed in the *options* attribute of the *_query* method:

- *failOnWarning*: when set to *true*, this will make the query throw an exception and abort in case a warning occurs. This option should be used in development to catch errors early. If set to *false*, warnings will not be propagated to exceptions and will be returned with the query results. There is also a server configuration option `--query.fail-on-warning` for setting the default value for *failOnWarning* so it does not need to be set on a per-query level.
- *cache*: if set to *true*, this will put the query result into the query result cache if the query result is eligible for caching and the query cache is running in demand mode. If set to *false*, the query result will not be inserted into the query result cache. Note that query results will never be inserted into the query result cache if the query result cache is disabled, and that they will be automatically inserted into the query result cache when it is active in non-demand mode.
- *profile*: if set to *true*, returns extra timing information for the query. The timing information is accessible via the *getExtra* method of the query result.
- *maxWarningCount*: limits the number of warnings that are returned by the query if *failOnWarning* is not set to *true*. The default value is *10*.

- *maxNumberOfPlans*: limits the number of query execution plans the optimizer will create at most. Reducing the number of query execution plans may speed up query plan creation and optimization for complex queries, but normally there is no need to adjust this value.

The following additional attributes can be passed to queries in the RocksDB storage engine:

- *maxTransactionSize*: transaction size limit in bytes
- *intermediateCommitSize*: maximum total size of operations after which an intermediate commit is performed automatically
- *intermediateCommitCount*: maximum number of operations after which an intermediate commit is performed automatically

In the ArangoDB Enterprise Edition there is an additional parameter:

- *skipInaccessibleCollections* AQL queries (especially graph traversals) will treat collection to which a user has **no access** rights as if these collections were empty. Instead of returning a *forbidden access* error, your queries will execute normally. This is intended to help with certain use-cases: A graph contains several collections and different users execute AQL queries on that graph. You can now naturally limit the accessible results by changing the access rights of users on collections.

with _createStatement (ArangoStatement)

The *_query* method is a shorthand for creating an ArangoStatement object, executing it and iterating over the resulting cursor. If more control over the result set iteration is needed, it is recommended to first create an ArangoStatement object as follows:

```
arangosh> stmt = db._createStatement( {
........> "query": "FOR i IN [ 1, 2 ] RETURN i * 2" } );
[object ArangoStatement]
```

To execute the query, use the *execute* method of the statement:

```
arangosh> c = stmt.execute();
```

show execution results

Cursors

Once the query executed the query results are available in a cursor. The cursor can return all its results at once using the *toArray* method. This is a short-cut that you can use if you want to access the full result set without iterating over it yourself.

```
arangosh> c.toArray();
[
  2,
  4
]
```

Cursors can also be used to iterate over the result set document-by-document. To do so, use the *hasNext* and *next* methods of the cursor:

```
arangosh> while (c.hasNext()) { require("@arangodb").print(c.next()); }
2
4
```

Please note that you can iterate over the results of a cursor only once, and that the cursor will be empty when you have fully iterated over it. To iterate over the results again, the query needs to be re-executed.

Additionally, the iteration can be done in a forward-only fashion. There is no backwards iteration or random access to elements in a cursor.

ArangoStatement parameters binding

To execute an AQL query using bind parameters, you need to create a statement first and then bind the parameters to it before execution:

```
arangosh> var stmt = db._createStatement( {
........> "query": "FOR i IN [ @one, @two ] RETURN i * 2" } );
arangosh> stmt.bind("one", 1);
arangosh> stmt.bind("two", 2);
arangosh> c = stmt.execute();
```

show execution results
The cursor results can then be dumped or iterated over as usual, e.g.:

```
arangosh> c.toArray();
[
  2,
  4
]
```

or

```
arangosh> while (c.hasNext()) { require("@arangodb").print(c.next()); }
2
4
```

Please note that bind parameters can also be passed into the *_createStatement* method directly, making it a bit more convenient:

```
arangosh> stmt = db._createStatement( {
........>  "query": "FOR i IN [ @one, @two ] RETURN i * 2",
........>  "bindVars": {
........>    "one": 1,
........>    "two": 2
........>  }
........> } );
[object ArangoStatement]
```

Counting with a cursor

Cursors also optionally provide the total number of results. By default, they do not. To make the server return the total number of results, you may set the *count* attribute to *true* when creating a statement:

```
arangosh> stmt = db._createStatement( {
........> "query": "FOR i IN [ 1, 2, 3, 4 ] RETURN i",
........> "count": true } );
[object ArangoStatement]
```

After executing this query, you can use the *count* method of the cursor to get the number of total results from the result set:

```
arangosh> var c = stmt.execute();
arangosh> c.count();
4
```

Please note that the *count* method returns nothing if you did not specify the *count* attribute when creating the query.

This is intentional so that the server may apply optimizations when executing the query and construct the result set incrementally. Incremental creation of the result sets is no possible if all of the results need to be shipped to the client anyway. Therefore, the client has the choice to specify *count* and retrieve the total number of results for a query (and disable potential incremental result set creation on the server), or to not retrieve the total number of results and allow the server to apply optimizations.

Please note that at the moment the server will always create the full result set for each query so specifying or omitting the *count* attribute currently does not have any impact on query execution. This may change in the future. Future versions of ArangoDB may create result sets incrementally on the server-side and may be able to apply optimizations if a result set is not fully fetched by a client.

Using cursors to obtain additional information on internal timings

Cursors can also optionally provide statistics of the internal execution phases. By default, they do not. To get to know how long parsing, otpimisation, instanciation and execution took, make the server return that by setting the *profile* attribute to *true* when creating a statement:

```
arangosh> stmt = db._createStatement( {
........> "query": "FOR i IN [ 1, 2, 3, 4 ] RETURN i",
........> options: {"profile": true}} );
[object ArangoStatement]
```

After executing this query, you can use the *getExtra()* method of the cursor to get the produced statistics:

```
arangosh> var c = stmt.execute();
arangosh> c.getExtra();
```

show execution results

AQL with ArangoDB Web Interface

In the ArangoDB Web Interface the AQL Editor tab allows to execute ad-hoc AQL queries.

Type in a query in the main box and execute it by pressing the *Execute* button. The query result will be shown in another tab. The editor provides a few example queries that can be used as templates.

It also provides a feature to explain a query and inspect its execution plan (with the *Explain* button).

Bind parameters can be defined in the right-hand side pane. The format is the same as used for bind parameters in the HTTP REST API and in (JavaScript) application code.

Here is an example:

```
FOR doc IN @@collection
  FILTER CONTAINS(LOWER(doc.author), @search, false)
  RETURN { "name": doc.name, "descr": doc.description, "author": doc.author }
```

Bind parameters (table view mode):

Key	Value
@collection	_apps
search	arango

Bind parameters (JSON view mode):

```
{
    "@collection": "_apps",
    "search": "arango"
}
```

How bind parameters work can be found in AQL Fundamentals.

Queries can also be saved in the AQL editor along with their bind parameter values for later reuse. This data is stored in the user profile in the current database (in the *_users* system table).

Also see the detailed description of the Web Interface.

AQL Fundamentals

- AQL Syntax explains the structure of the AQL language.
- Data Types describes the primitive and compound data types supported by AQL.
- Bind Parameters: AQL supports the usage of bind parameters. This allows to separate the query text from literal values used in the query.
- Type and value order: AQL uses a set of rules (using values and types) for equality checks and comparisons.
- Accessing Data from Collections: describes the impact of non-existent or null attributes for selection queries.
- Query Results: the result of an AQL query is an array of values.
- Query Errors: errors may arise from the AQL parsing or execution.

AQL Syntax

Query types

An AQL query must either return a result (indicated by usage of the *RETURN* keyword) or execute a data-modification operation (indicated by usage of one of the keywords *INSERT*, *UPDATE*, *REPLACE*, *REMOVE* or *UPSERT*). The AQL parser will return an error if it detects more than one data-modification operation in the same query or if it cannot figure out if the query is meant to be a data retrieval or a modification operation.

AQL only allows *one* query in a single query string; thus semicolons to indicate the end of one query and separate multiple queries (as seen in SQL) are not allowed.

Whitespace

Whitespaces (blanks, carriage returns, line feeds, and tab stops) can be used in the query text to increase its readability. Tokens have to be separated by any number of whitespaces. Whitespace within strings or names must be enclosed in quotes in order to be preserved.

Comments

Comments can be embedded at any position in a query. The text contained in the comment is ignored by the AQL parser.

Multi-line comments cannot be nested, which means subsequent comment starts within comments are ignored, comment ends will end the comment.

AQL supports two types of comments:

- Single line comments: These start with a double forward slash and end at the end of the line, or the end of the query string (whichever is first).
- Multi line comments: These start with a forward slash and asterisk, and end with an asterisk and a following forward slash. They can span as many lines as necessary.

```
/* this is a comment */ RETURN 1
/* these */ RETURN /* are */ 1 /* multiple */ + /* comments */ 1
/* this is
   a multi line
   comment */
// a single line comment
```

Keywords

On the top level, AQL offers the following operations:

- `FOR` : array iteration
- `RETURN` : results projection
- `FILTER` : results filtering
- `SORT` : result sorting
- `LIMIT` : result slicing
- `LET` : variable assignment
- `COLLECT` : result grouping
- `INSERT` : insertion of new documents
- `UPDATE` : (partial) update of existing documents
- `REPLACE` : replacement of existing documents
- `REMOVE` : removal of existing documents
- `UPSERT` : insertion or update of existing documents

Each of the above operations can be initiated in a query by using a keyword of the same name. An AQL query can (and typically does) consist of multiple of the above operations.

An example AQL query may look like this:

```
FOR u IN users
  FILTER u.type == "newbie" && u.active == true
  RETURN u.name
```

In this example query, the terms *FOR*, *FILTER*, and *RETURN* initiate the higher-level operation according to their name. These terms are also keywords, meaning that they have a special meaning in the language.

For example, the query parser will use the keywords to find out which high-level operations to execute. That also means keywords can only be used at certain locations in a query. This also makes all keywords reserved words that must not be used for other purposes than they are intended for.

For example, it is not possible to use a keyword as a collection or attribute name. If a collection or attribute need to have the same name as a keyword, the collection or attribute name needs to be quoted.

Keywords are case-insensitive, meaning they can be specified in lower, upper, or mixed case in queries. In this documentation, all keywords are written in upper case to make them distinguishable from other query parts.

There are a few more keywords in addition to the higher-level operation keywords. Additional keywords may be added in future versions of ArangoDB. The complete list of keywords is currently:

- AGGREGATE
- ALL
- AND
- ANY
- ASC
- COLLECT
- DESC
- DISTINCT
- FALSE
- FILTER
- FOR
- GRAPH
- IN
- INBOUND
- INSERT
- INTO
- LET
- LIMIT
- NONE
- NOT
- NULL
- OR
- OUTBOUND
- REMOVE
- REPLACE
- RETURN
- SHORTEST_PATH
- SORT
- TRUE
- UPDATE
- UPSERT
- WITH

Names

In general, names are used to identify objects (collections, attributes, variables, and functions) in AQL queries.

The maximum supported length of any name is 64 bytes. Names in AQL are always case-sensitive.

Keywords must not be used as names. If a reserved keyword should be used as a name, the name must be enclosed in backticks or forward ticks. Enclosing a name in backticks or forward ticks makes it possible to use otherwise reserved keywords as names. An example for this is:

```
FOR f IN `filter`
  RETURN f.`sort`
```

Due to the backticks, *filter* and *sort* are interpreted as names and not as keywords here.

The example can alternatively written as:

```
FOR f IN ´filter´
  RETURN f.´sort´
```

Collection names

Collection names can be used in queries as they are. If a collection happens to have the same name as a keyword, the name must be enclosed in backticks.

Please refer to the Naming Conventions in ArangoDB about collection naming conventions.

AQL currently has a limit of up to 256 collections used in one AQL query. This limit applies to the sum of all involved document and edge collections.

Attribute names

When referring to attributes of documents from a collection, the fully qualified attribute name must be used. This is because multiple collections with ambiguous attribute names may be used in a query. To avoid any ambiguity, it is not allowed to refer to an unqualified attribute name.

Please refer to the Naming Conventions in ArangoDB for more information about the attribute naming conventions.

```
FOR u IN users
  FOR f IN friends
    FILTER u.active == true && f.active == true && u.id == f.userId
    RETURN u.name
```

In the above example, the attribute names *active*, *name*, *id*, and *userId* are qualified using the collection names they belong to (*u* and *f* respectively).

Variable names

AQL allows the user to assign values to additional variables in a query. All variables that are assigned a value must have a name that is unique within the context of the query. Variable names must be different from the names of any collection name used in the same query.

```
FOR u IN users
  LET friends = u.friends
  RETURN { "name" : u.name, "friends" : friends }
```

In the above query, *users* is a collection name, and both *u* and *friends* are variable names. This is because the *FOR* and *LET* operations need target variables to store their intermediate results.

Allowed characters in variable names are the letters *a* to *z* (both in lower and upper case), the numbers *0* to *9*, the underscore (_) symbol and the dollar (*$*) sign. A variable name must not start with a number. If a variable name starts with the underscore character, the underscore must be followed by least one letter (a-z or A-Z) or digit (0-9).

The dollar sign can be used only as the very first character in a variable name.

Data types

AQL supports both primitive and compound data types. The following types are available:

- Primitive types: Consisting of exactly one value
  - null: An empty value, also: The absence of a value
  - bool: Boolean truth value with possible values *false* and *true*
  - number: Signed (real) number
  - string: UTF-8 encoded text value
- Compound types: Consisting of multiple values
  - array: Sequence of values, referred to by their positions
  - object / document: Sequence of values, referred to by their names

Primitive types

Numeric literals

Numeric literals can be integers or real values. They can optionally be signed using the + or - symbols. The scientific notation is also supported.

```
1
42
-1
-42
1.23
-99.99
0.1
-4.87e103
```

All numeric values are treated as 64-bit double-precision values internally. The internal format used is IEEE 754.

String literals

String literals must be enclosed in single or double quotes. If the used quote character is to be used itself within the string literal, it must be escaped using the backslash symbol. Backslash literals themselves also be escaped using a backslash.

```
"yikes!"
"don't know"
"this is a \"quoted\" word"
"this is a longer string."
"the path separator on Windows is \\"

'yikes!'
'don\'t know'
'this is a longer string."
'the path separator on Windows is \\'
```

All string literals must be UTF-8 encoded. It is currently not possible to use arbitrary binary data if it is not UTF-8 encoded. A workaround to use binary data is to encode the data using base64 or other algorithms on the application side before storing, and decoding it on application side after retrieval.

Compound types

AQL supports two compound types:

- arrays: A composition of unnamed values, each accessible by their positions
- objects / documents: A composition of named values, each accessible by their names

Arrays / Lists

The first supported compound type is the array type. Arrays are effectively sequences of (unnamed / anonymous) values. Individual array elements can be accessed by their positions. The order of elements in an array is important.

An *array-declaration* starts with the *[* symbol and ends with the *]* symbol. An *array-declaration* contains zero or many *expression*s, separated from each other with the , symbol.

In the easiest case, an array is empty and thus looks like:

```
[ ]
```

Array elements can be any legal *expression* values. Nesting of arrays is supported.

```
[ 1, 2, 3 ]
[ -99, "yikes!", [ true, [ "no"], [ ] ], 1 ]
[ [ "fox", "marshal" ] ]
```

Individual array values can later be accessed by their positions using the *[]* accessor. The position of the accessed element must be a numeric value. Positions start at 0. It is also possible to use negative index values to access array values starting from the end of the array. This is convenient if the length of the array is unknown and access to elements at the end of the array is required.

```
// access 1st array element (elements start at index 0)
u.friends[0]

// access 3rd array element
u.friends[2]

// access last array element
u.friends[-1]

// access second to last array element
u.friends[-2]
```

Objects / Documents

The other supported compound type is the object (or document) type. Objects are a composition of zero to many attributes. Each attribute is a name/value pair. Object attributes can be accessed individually by their names.

Object declarations start with the *{* symbol and end with the *}* symbol. An object contains zero to many attribute declarations, separated from each other with the , symbol. In the simplest case, an object is empty. Its declaration would then be:

```
{ }
```

Each attribute in an object is a name / value pair. Name and value of an attribute are separated using the *:* symbol.

The attribute name is mandatory and must be specified as a quoted or unquoted string. If a keyword is used as an attribute name, the attribute name must be quoted:

```
{ return : 1 }     /* won't work */
{ "return" : 1 }   /* works ! */
{ `return` : 1 }   /* works, too! */
```

Since ArangoDB 2.6, object attribute names can be computed using dynamic expressions, too. To disambiguate regular attribute names from attribute name expressions, computed attribute names must be enclosed in *[* and *]*:

```
{ [ CONCAT("test/", "bar") ] : "someValue" }
```

Since ArangoDB 2.7, there is also shorthand notation for attributes which is handy for returning existing variables easily:

```
LET name = "Peter"
LET age = 42
```

```
RETURN { name, age }
```

The above is the shorthand equivalent for the generic form:

```
LET name = "Peter"
LET age = 42
RETURN { name : name, age : age }
```

Any valid expression can be used as an attribute value. That also means nested objects can be used as attribute values:

```
{ name : "Peter" }
{ "name" : "Vanessa", "age" : 15 }
{ "name" : "John", likes : [ "Swimming", "Skiing" ], "address" : { "street" : "Cucumber lane", "zip" : "94242" } }
```

Individual object attributes can later be accessed by their names using the . accessor:

```
u.address.city.name
u.friends[0].name.first
```

Attributes can also be accessed using the *[]* accessor:

```
u["address"]["city"]["name"]
u["friends"][0]["name"]["first"]
```

In contrast to the dot accessor, the square brackets allow for expressions:

```
LET attr1 = "friends"
LET attr2 = "name"
u[attr1][0][attr2][ CONCAT("fir", "st") ]
```

Note that if a non-existing attribute is accessed in one or the other way, the result will be *null*, without error or warning.

Bind parameters

AQL supports the usage of bind parameters, thus allowing to separate the query text from literal values used in the query. It is good practice to separate the query text from the literal values because this will prevent (malicious) injection of keywords and other collection names into an existing query. This injection would be dangerous because it may change the meaning of an existing query.

Using bind parameters, the meaning of an existing query cannot be changed. Bind parameters can be used everywhere in a query where literals can be used.

The syntax for bind parameters is *@name* where *@* signifies that this is a bind parameter and *name* is the actual parameter name. Parameter names must start with any of the letters *a* to *z* (upper or lower case) or a digit (*0* to *9*), and can be followed by any letter, digit or the underscore symbol.

```
FOR u IN users
  FILTER u.id == @id && u.name == @name
  RETURN u
```

The bind parameter values need to be passed along with the query when it is executed, but not as part of the query text itself. In the web interface, there is a pane next to the query editor where the bind parameters can be entered. When using `db._query()` (in arangosh for instance), then an object of key-value pairs can be passed for the parameters. Such an object can also be passed to the HTTP API endpoint `_api/cursor`, as attribute value for the key *bindVars*:

```
{
  "query": "FOR u IN users FILTER u.id == @id && u.name == @name RETURN u",
  "bindVars": {
    "id": 123,
    "name": "John Smith"
  }
}
```

Bind parameters that are declared in the query must also be passed a parameter value, or the query will fail. Specifying parameters that are not declared in the query will result in an error too.

Bind variables represent a value like a string, and must not be put in quotes in the AQL code:

```
FILTER u.name == "@name" // wrong
FILTER u.name == @name   // correct
```

If you need to do string processing (concatenation, etc.) in the query, you need to use string functions to do so:

```
FOR u IN users
  FILTER u.id == CONCAT('prefix', @id, 'suffix') && u.name == @name
  RETURN u
```

Bind paramers can be used for both, the dot notation as well as the square bracket notation for sub-attribute access. They can also be chained:

```
LET doc = { foo: { bar: "baz" } }

RETURN doc.@attr.@subattr
// or
RETURN doc[@attr][@subattr]
```

```
{
  "attr": "foo",
  "subattr": "bar"
}
```

Both variants in above example return `[ "baz" ]` as query result.

The whole attribute path, for highly nested data in particular, can also be specified using the dot notation and a single bind parameter, by passing an array of strings as parameter value. The elements of the array represent the attribute keys of the path:

```
LET doc = { a: { b: { c: 1 } } }
RETURN doc.@attr
```

```
{ "attr": [ "a", "b", "c" ] }
```

The example query returns `[ 1 ]` as result. Note that `{ "attr": "a.b.c" }` would return the value of an attribute called *a.b.c*, not the value of attribute *c* with the parents *a* and *b* as `[ "a", "b", "c" ]` would.

A special type of bind parameter exists for injecting collection names. This type of bind parameter has a name prefixed with an additional @ symbol (thus when using the bind parameter in a query, two @ symbols must be used).

```
FOR u IN @@collection
  FILTER u.active == true
  RETURN u
```

```
{ "@collection": "myCollection" }
```

Keywords can't be replaced by bind-values; i.e. `FOR`, `FILTER`, `IN`, `INBOUND` or function calls.

Specific information about parameters binding can also be found in:

- AQL with Web Interface
- AQL with Arangosh
- HTTP Interface for AQL Queries

Type and value order

When checking for equality or inequality or when determining the sort order of values, AQL uses a deterministic algorithm that takes both the data types and the actual values into account.

The compared operands are first compared by their data types, and only by their data values if the operands have the same data types.

The following type order is used when comparing data types:

```
null < bool < number < string < array/list < object/document
```

This means *null* is the smallest type in AQL and *document* is the type with the highest order. If the compared operands have a different type, then the comparison result is determined and the comparison is finished.

For example, the boolean *true* value will always be less than any numeric or string value, any array (even an empty array) or any object / document. Additionally, any string value (even an empty string) will always be greater than any numeric value, a boolean value, *true* or *false*.

```
null < false
null < true
null < 0
null < ''
null < ' '
null < '0'
null < 'abc'
null < [ ]
null < { }

false < true
false < 0
false < ''
false < ' '
false < '0'
false < 'abc'
false < [ ]
false < { }

true < 0
true < ''
true < ' '
true < '0'
true < 'abc'
true < [ ]
true < { }

0 < ''
0 < ' '
0 < '0'
0 < 'abc'
0 < [ ]
0 < { }

'' < ' '
'' < '0'
'' < 'abc'
'' < [ ]
'' < { }

[ ] < { }
```

If the two compared operands have the same data types, then the operands values are compared. For the primitive types (null, boolean, number, and string), the result is defined as follows:

- null: *null* is equal to *null*
- boolean: *false* is less than *true*
- number: numeric values are ordered by their cardinal value

- string: string values are ordered using a localized comparison, using the configured server language for sorting according to the alphabetical order rules of that language

Note: unlike in SQL, *null* can be compared to any value, including *null* itself, without the result being converted into *null* automatically.

For compound, types the following special rules are applied:

Two array values are compared by comparing their individual elements position by position, starting at the first element. For each position, the element types are compared first. If the types are not equal, the comparison result is determined, and the comparison is finished. If the types are equal, then the values of the two elements are compared. If one of the arrays is finished and the other array still has an element at a compared position, then *null* will be used as the element value of the fully traversed array.

If an array element is itself a compound value (an array or an object / document), then the comparison algorithm will check the element's sub values recursively. The element's sub-elements are compared recursively.

```
[ ] < [ 0 ]
[ 1 ] < [ 2 ]
[ 1, 2 ] < [ 2 ]
[ 99, 99 ] < [ 100 ]
[ false ] < [ true ]
[ false, 1 ] < [ false, '' ]
```

Two object / documents operands are compared by checking attribute names and value. The attribute names are compared first. Before attribute names are compared, a combined array of all attribute names from both operands is created and sorted lexicographically. This means that the order in which attributes are declared in an object / document is not relevant when comparing two objects / documents.

The combined and sorted array of attribute names is then traversed, and the respective attributes from the two compared operands are then looked up. If one of the objects / documents does not have an attribute with the sought name, its attribute value is considered to be *null*. Finally, the attribute value of both objects / documents is compared using the before mentioned data type and value comparison. The comparisons are performed for all object / document attributes until there is an unambiguous comparison result. If an unambiguous comparison result is found, the comparison is finished. If there is no unambiguous comparison result, the two compared objects / documents are considered equal.

```
{ } < { "a" : 1 }
{ } < { "a" : null }
{ "a" : 1 } < { "a" : 2 }
{ "b" : 1 } < { "a" : 0 }
{ "a" : { "c" : true } } < { "a" : { "c" : 0 } }
{ "a" : { "c" : true, "a" : 0 } } < { "a" : { "c" : false, "a" : 1 } }

{ "a" : 1, "b" : 2 } == { "b" : 2, "a" : 1 }
```

Accessing data from collections

Collection data can be accessed by specifying a collection name in a query. A collection can be understood as an array of documents, and that is how they are treated in AQL. Documents from collections are normally accessed using the *FOR* keyword. Note that when iterating over documents from a collection, the order of documents is undefined. To traverse documents in an explicit and deterministic order, the *SORT* keyword should be used in addition.

Data in collections is stored in documents, with each document potentially having different attributes than other documents. This is true even for documents of the same collection.

It is therefore quite normal to encounter documents that do not have some or all of the attributes that are queried in an AQL query. In this case, the non-existing attributes in the document will be treated as if they would exist with a value of *null*. That means that comparing a document attribute to *null* will return true if the document has the particular attribute and the attribute has a value of *null*, or that the document does not have the particular attribute at all.

For example, the following query will return all documents from the collection *users* that have a value of *null* in the attribute *name*, plus all documents from *users* that do not have the *name* attribute at all:

```
FOR u IN users
  FILTER u.name == null
  RETURN u
```

Furthermore, *null* is less than any other value (excluding *null* itself). That means documents with non-existing attributes may be included in the result when comparing attribute values with the less than or less equal operators.

For example, the following query will return all documents from the collection *users* that have an attribute *age* with a value less than *39*, but also all documents from the collection that do not have the attribute *age* at all.

```
FOR u IN users
  FILTER u.age < 39
  RETURN u
```

This behavior should always be taken into account when writing queries.

Query results

Result sets

The result of an AQL query is an array of values. The individual values in the result array may or may not have a homogeneous structure, depending on what is actually queried.

For example, when returning data from a collection with inhomogeneous documents (the individual documents in the collection have different attribute names) without modification, the result values will as well have an inhomogeneous structure. Each result value itself is a document:

```
FOR u IN users
    RETURN u
```

```
[ { "id": 1, "name": "John", "active": false },
  { "age": 32, "id": 2, "name": "Vanessa" },
  { "friends": [ "John", "Vanessa" ], "id": 3, "name": "Amy" } ]
```

However, if a fixed set of attributes from the collection is queried, then the query result values will have a homogeneous structure. Each result value is still a document:

```
FOR u IN users
    RETURN { "id": u.id, "name": u.name }
```

```
[ { "id": 1, "name": "John" },
  { "id": 2, "name": "Vanessa" },
  { "id": 3, "name": "Amy" } ]
```

It is also possible to query just scalar values. In this case, the result set is an array of scalars, and each result value is a scalar value:

```
FOR u IN users
    RETURN u.id
```

```
[ 1, 2, 3 ]
```

If a query does not produce any results because no matching data can be found, it will produce an empty result array:

```
[ ]
```

Errors

Issuing an invalid query to the server will result in a parse error if the query is syntactically invalid. ArangoDB will detect such errors during query inspection and abort further processing. Instead, the error number and an error message are returned so that the errors can be fixed.

If a query passes the parsing stage, all collections referenced in the query will be opened. If any of the referenced collections is not present, query execution will again be aborted and an appropriate error message will be returned.

Under some circumstances, executing a query may also produce run-time errors that cannot be predicted from inspecting the query text alone. This is because queries may use data from collections that may also be inhomogeneous. Some examples that will cause run-time errors are:

- Division by zero: Will be triggered when an attempt is made to use the value *0* as the divisor in an arithmetic division or modulus operation
- Invalid operands for arithmetic operations: Will be triggered when an attempt is made to use any non-numeric values as operands in arithmetic operations. This includes unary (unary minus, unary plus) and binary operations (plus, minus, multiplication, division, and modulus)
- Invalid operands for logical operations: Will be triggered when an attempt is made to use any non-boolean values as operand(s) in logical operations. This includes unary (logical not/negation), binary (logical and, logical or), and the ternary operators

Please refer to the Arango Errors page for a list of error codes and meanings.

Operators

AQL supports a number of operators that can be used in expressions. There are comparison, logical, arithmetic, and the ternary operator.

Comparison operators

Comparison (or relational) operators compare two operands. They can be used with any input data types, and will return a boolean result value.

The following comparison operators are supported:

- == equality
- *!*= inequality
- < less than
- <= less or equal
- > greater than
- >= greater or equal
- *IN* test if a value is contained in an array
- *NOT IN* test if a value is not contained in an array
- *LIKE* tests if a string value matches a pattern
- =~ tests if a string value matches a regular expression
- *!*~ tests if a string value does not match a regular expression

Each of the comparison operators returns a boolean value if the comparison can be evaluated and returns *true* if the comparison evaluates to true, and *false* otherwise.

The comparison operators accept any data types for the first and second operands. However, *IN* and *NOT IN* will only return a meaningful result if their right-hand operand is a string, and *LIKE* will only execute if both operands are string values. The comparison operators will not perform any implicit type casts if the compared operands have different or non-sensible types.

Some examples for comparison operations in AQL:

```
0 == null                // false
1 > 0                    // true
true != null             // true
45 <= "yikes!"           // true
65 != "65"               // true
65 == 65                 // true
1.23 > 1.32              // false
1.5 IN [ 2, 3, 1.5 ]     // true
"foo" IN null            // false
42 NOT IN [ 17, 40, 50 ] // true
"abc" == "abc"           // true
"abc" == "ABC"           // false
"foo" LIKE "f%"          // true
"foo" =~ "^f[o].$"       // true
"foo" !~ "[a-z]+bar$"    // true
```

The *LIKE* operator checks whether its left operand matches the pattern specified in its right operand. The pattern can consist of regular characters and wildcards. The supported wildcards are _ to match a single arbitrary character, and % to match any number of arbitrary characters. Literal % and _ need to be escaped with a backslash. Backslashes need to be escaped themselves, which effectively means that two reverse solidus characters need to preceed a literal percent sign or underscore. In arangosh, additional escaping is required, making it four backslashes in total preceeding the to-be-escaped character.

```
"abc" LIKE "a%"             // true
"abc" LIKE "_bc"            // true
"a_b_foo" LIKE "a\\_b\\_foo" // true
```

The pattern matching performed by the *LIKE* operator is case-sensitive.

The regular expression operators =~ and *!*~ expect their left-hand operands to be strings, and their right-hand operands to be strings containing valid regular expressions as specified in the documentation for the AQL function REGEX_TEST().

Array comparison operators

The comparison operators also exist as *array variant*. In the array variant, the operator is prefixed with one of the keywords *ALL*, *ANY* or *NONE*. Using one of these keywords changes the operator behavior to execute the comparison operation for all, any, or none of its left hand argument values. It is therefore expected that the left hand argument of an array operator is an array.

Examples:

```
[ 1, 2, 3 ] ALL IN [ 2, 3, 4 ]   // false
[ 1, 2, 3 ] ALL IN [ 1, 2, 3 ]   // true
[ 1, 2, 3 ] NONE IN [ 3 ]        // false
[ 1, 2, 3 ] NONE IN [ 23, 42 ]   // true
[ 1, 2, 3 ] ANY IN [ 4, 5, 6 ]   // false
[ 1, 2, 3 ] ANY IN [ 1, 42 ]     // true
[ 1, 2, 3 ] ANY == 2             // true
[ 1, 2, 3 ] ANY == 4             // false
[ 1, 2, 3 ] ANY > 0              // true
[ 1, 2, 3 ] ANY <= 1             // true
[ 1, 2, 3 ] NONE < 99            // false
[ 1, 2, 3 ] NONE > 10            // true
[ 1, 2, 3 ] ALL > 2              // false
[ 1, 2, 3 ] ALL > 0              // true
[ 1, 2, 3 ] ALL >= 3             // false
["foo", "bar"] ALL != "moo"      // true
["foo", "bar"] NONE == "bar"     // false
["foo", "bar"] ANY == "foo"      // true
```

Note that these operators are not optimized yet. Indexes will not be utilized.

Logical operators

The following logical operators are supported in AQL:

- && logical and operator
- || logical or operator
- *!* logical not/negation operator

AQL also supports the following alternative forms for the logical operators:

- *AND* logical and operator
- *OR* logical or operator
- *NOT* logical not/negation operator

The alternative forms are aliases and functionally equivalent to the regular operators.

The two-operand logical operators in AQL will be executed with short-circuit evaluation (except if one of the operands is or includes a subquery. In this case the subquery will be pulled out an evaluated before the logical operator).

The result of the logical operators in AQL is defined as follows:

- `lhs && rhs` will return `lhs` if it is `false` or would be `false` when converted into a boolean. If `lhs` is `true` or would be `true` when converted to a boolean, `rhs` will be returned.
- `lhs || rhs` will return `lhs` if it is `true` or would be `true` when converted into a boolean. If `lhs` is `false` or would be `false` when converted to a boolean, `rhs` will be returned.
- `! value` will return the negated value of `value` converted into a boolean

Some examples for logical operations in AQL:

```
u.age > 15 && u.address.city != ""
true || false
NOT u.isInvalid
1 || ! 0
```

Passing non-boolean values to a logical operator is allowed. Any non-boolean operands will be casted to boolean implicitly by the operator, without making the query abort.

The *conversion to a boolean value* works as follows:

- `null` will be converted to `false`
- boolean values remain unchanged
- all numbers unequal to zero are `true`, zero is `false`
- an empty string is `false`, all other strings are `true`
- arrays (`[ ]`) and objects / documents (`{ }`) are `true`, regardless of their contents

The result of *logical and* and *logical or* operations can now have any data type and is not necessarily a boolean value.

For example, the following logical operations will return boolean values:

```
25 > 1 && 42 != 7                       // true
22 IN [ 23, 42 ] || 23 NOT IN [ 22, 7 ]  // true
25 != 25                                // false
```

whereas the following logical operations will not return boolean values:

```
1 || 7                                  // 1
null || "foo"                           // "foo"
null && true                            // null
true && 23                              // 23
```

Arithmetic operators

Arithmetic operators perform an arithmetic operation on two numeric operands. The result of an arithmetic operation is again a numeric value.

AQL supports the following arithmetic operators:

- + addition
- - subtraction
- * multiplication
- / division
- % modulus

Unary plus and unary minus are supported as well:

```
LET x = -5
LET y = 1
RETURN [-x, +y]
// [5, 1]
```

For exponentiation, there is a numeric function *POW()*. The syntax `base ** exp` is not supported.

For string concatenation, you must use the string function *CONCAT()*. Combining two strings with a plus operator (`"foo" + "bar"`) will not work! Also see Common Errors.

Some example arithmetic operations:

```
1 + 1
33 - 99
12.4 * 4.5
13.0 / 0.1
23 % 7
-15
+9.99
```

The arithmetic operators accept operands of any type. Passing non-numeric values to an arithmetic operator will cast the operands to numbers using the type casting rules applied by the TO_NUMBER() function:

- `null` will be converted to `0`
- `false` will be converted to `0`, true will be converted to `1`
- a valid numeric value remains unchanged, but NaN and Infinity will be converted to `0`
- string values are converted to a number if they contain a valid string representation of a number. Any whitespace at the start or the end of the string is ignored. Strings with any other contents are converted to the number `0`
- an empty array is converted to `0`, an array with one member is converted to the numeric representation of its sole member. Arrays with more members are converted to the number `0`.
- objects / documents are converted to the number `0`.

An arithmetic operation that produces an invalid value, such as `1 / 0` (division by zero) will also produce a result value of `null`. The query is not aborted, but you may see a warning.

Here are a few examples:

```
1 + "a"                 // 1
1 + "99"                // 100
1 + null                // 1
null + 1                // 1
3 + [ ]                 // 3
24 + [ 2 ]              // 26
24 + [ 2, 4 ]           // 0
25 - null               // 25
17 - true               // 16
23 * { }                // 0
5 * [ 7 ]               // 35
24 / "12"               // 2
1 / 0                   // 0
```

Ternary operator

AQL also supports a ternary operator that can be used for conditional evaluation. The ternary operator expects a boolean condition as its first operand, and it returns the result of the second operand if the condition evaluates to true, and the third operand otherwise.

Examples

```
u.age > 15 || u.active == true ? u.userId : null
```

There is also a shortcut variant of the ternary operator with just two operands. This variant can be used when the expression for the boolean condition and the return value should be the same:

Examples

```
u.value ? : 'value is null, 0 or not present'
```

Range operator

AQL supports expressing simple numeric ranges with the .. operator. This operator can be used to easily iterate over a sequence of numeric values.

The .. operator will produce an array of values in the defined range, with both bounding values included.

Examples

```
2010..2013
```

will produce the following result:

```
[ 2010, 2011, 2012, 2013 ]
```

There is also a RANGE() function.

Array operators

AQL provides array operators *[*]* for array variable expansion and *[**]* for array contraction.

Operator precedence

The operator precedence in AQL is similar as in other familiar languages (lowest precedence first):

- *? :* ternary operator
- || logical or
- && logical and
- ==, *!=* equality and inequality
- *IN* in operator
- <, <=, >=, > less than, less equal, greater equal, greater than
- +, - addition, subtraction
- *, /, %* multiplication, division, modulus
- *!*, +, - logical negation, unary plus, unary minus
- *[*]* expansion
- *()* function call
- . member access
- *[]* indexed value access

The parentheses *(* and *)* can be used to enforce a different operator evaluation order.

Data Queries

Data Access Queries

Retrieving data from the database with AQL does always include a **RETURN** operation. It can be used to return a static value, such as a string:

```
RETURN "Hello ArangoDB!"
```

The query result is always an array of elements, even if a single element was returned and contains a single element in that case: `["Hello ArangoDB!"]`

The function `DOCUMENT()` can be called to retrieve a single document via its document handle, for instance:

```
RETURN DOCUMENT("users/phil")
```

RETURN is usually accompanied by a **FOR** loop to iterate over the documents of a collection. The following query executes the loop body for all documents of a collection called *users*. Each document is returned unchanged in this example:

```
FOR doc IN users
    RETURN doc
```

Instead of returning the raw `doc`, one can easily create a projection:

```
FOR doc IN users
    RETURN { user: doc, newAttribute: true }
```

For every user document, an object with two attributes is returned. The value of the attribute *user* is set to the content of the user document, and *newAttribute* is a static attribute with the boolean value *true*.

Operations like **FILTER**, **SORT** and **LIMIT** can be added to the loop body to narrow and order the result. Instead of above shown call to `DOCUMENT()`, one can also retrieve the document that describes user *phil* like so:

```
FOR doc IN users
    FILTER doc._key == "phil"
    RETURN doc
```

The document key is used in this example, but any other attribute could equally be used for filtering. Since the document key is guaranteed to be unique, no more than a single document will match this filter. For other attributes this may not be the case. To return a subset of active users (determined by an attribute called *status*), sorted by name in ascending order, you can do:

```
FOR doc IN users
    FILTER doc.status == "active"
    SORT doc.name
    LIMIT 10
```

Note that operations do not have to occur in a fixed order and that their order can influence the result significantly. Limiting the number of documents before a filter is usually not what you want, because it easily misses a lot of documents that would fulfill the filter criterion, but are ignored because of a premature *LIMIT* clause. Because of the aforementioned reasons, *LIMIT* is usually put at the very end, after *FILTER*, *SORT* and other operations.

See the High Level Operations chapter for more details.

Data Modification Queries

AQL supports the following data-modification operations:

- **INSERT**: insert new documents into a collection
- **UPDATE**: partially update existing documents in a collection
- **REPLACE**: completely replace existing documents in a collection
- **REMOVE**: remove existing documents from a collection
- **UPSERT**: conditionally insert or update documents in a collection

Below you find some simple example queries that use these operations. The operations are detailed in the chapter High Level Operations.

Modifying a single document

Let's start with the basics: `INSERT`, `UPDATE` and `REMOVE` operations on single documents. Here is an example that insert a document in an existing collection *users*:

```
INSERT {
    firstName: "Anna",
    name: "Pavlova",
    profession: "artist"
} IN users
```

You may provide a key for the new document; if not provided, ArangoDB will create one for you.

```
INSERT {
    _key: "GilbertoGil",
    firstName: "Gilberto",
    name: "Gil",
    city: "Fortalezza"
} IN users
```

As ArangoDB is schema-free, attributes of the documents may vary:

```
INSERT {
    _key: "PhilCarpenter",
    firstName: "Phil",
    name: "Carpenter",
    middleName: "G.",
    status: "inactive"
} IN users
```

```
INSERT {
    _key: "NatachaDeclerck",
    firstName: "Natacha",
    name: "Declerck",
    location: "Antwerp"
} IN users
```

Update is quite simple. The following AQL statement will add or change the attributes status and location

```
UPDATE "PhilCarpenter" WITH {
    status: "active",
    location: "Beijing"
} IN users
```

Replace is an alternative to update where all attributes of the document are replaced.

```
REPLACE {
    _key: "NatachaDeclerck",
    firstName: "Natacha",
    name: "Leclerc",
    status: "active",
    level: "premium"
} IN users
```

Removing a document if you know its key is simple as well :

```
REMOVE "GilbertoGil" IN users
```

or

```
REMOVE { _key: "GilbertoGil" } IN users
```

Modifying multiple documents

Data-modification operations are normally combined with *FOR* loops to iterate over a given list of documents. They can optionally be combined with *FILTER* statements and the like.

Let's start with an example that modifies existing documents in a collection *users* that match some condition:

```
FOR u IN users
    FILTER u.status == "not active"
    UPDATE u WITH { status: "inactive" } IN users
```

Now, let's copy the contents of the collection *users* into the collection *backup*:

```
FOR u IN users
    INSERT u IN backup
```

As a final example, let's find some documents in collection *users* and remove them from collection *backup*. The link between the documents in both collections is established via the documents' keys:

```
FOR u IN users
    FILTER u.status == "deleted"
    REMOVE u IN backup
```

Returning documents

Data-modification queries can optionally return documents. In order to reference the inserted, removed or modified documents in a `RETURN` statement, data-modification statements introduce the `OLD` and/or `NEW` pseudo-values:

```
FOR i IN 1..100
    INSERT { value: i } IN test
    RETURN NEW
```

```
FOR u IN users
    FILTER u.status == "deleted"
    REMOVE u IN users
    RETURN OLD
```

```
FOR u IN users
    FILTER u.status == "not active"
    UPDATE u WITH { status: "inactive" } IN users
    RETURN NEW
```

`NEW` refers to the inserted or modified document revision, and `OLD` refers to the document revision before update or removal. `INSERT` statements can only refer to the `NEW` pseudo-value, and `REMOVE` operations only to `OLD`. `UPDATE`, `REPLACE` and `UPSERT` can refer to either.

In all cases the full documents will be returned with all their attributes, including the potentially auto-generated attributes such as `_id`, `_key`, or `_rev` and the attributes not specified in the update expression of a partial update.

Projections

It is possible to return a projection of the documents in `OLD` or `NEW` instead of returning the entire documents. This can be used to reduce the amount of data returned by queries.

For example, the following query will return only the keys of the inserted documents:

```
FOR i IN 1..100
    INSERT { value: i } IN test
    RETURN NEW._key
```

Using OLD and NEW in the same query

For `UPDATE`, `REPLACE` and `UPSERT` statements, both `OLD` and `NEW` can be used to return the previous revision of a document together with the updated revision:

```
FOR u IN users
    FILTER u.status == "not active"
    UPDATE u WITH { status: "inactive" } IN users
    RETURN { old: OLD, new: NEW }
```

Calculations with OLD or NEW

It is also possible to run additional calculations with `LET` statements between the data-modification part and the final `RETURN` of an AQL query. For example, the following query performs an upsert operation and returns whether an existing document was updated, or a new document was inserted. It does so by checking the `OLD` variable after the `UPSERT` and using a `LET` statement to store a temporary string for the operation type:

```
UPSERT { name: "test" }
    INSERT { name: "test" }
    UPDATE { } IN users
LET opType = IS_NULL(OLD) ? "insert" : "update"
RETURN { _key: NEW._key, type: opType }
```

Restrictions

The name of the modified collection (*users* and *backup* in the above cases) must be known to the AQL executor at query-compile time and cannot change at runtime. Using a bind parameter to specify the collection name is allowed.

Data-modification queries are restricted to a single modify operation per collection. That means you may not place several `REMOVE` or `UPDATE` statements for one collection in one query. In case you have several places in a query providing you lists of documents to delete, collect them in an array so you can remove them all at once.

Only a single data-modification operation can be used per AQL query. Data-modification queries cannot be used inside subqueries. Data-modification operations can optionally be followed by `LET` operations and a single `RETURN` operation to return data. If expressions are used within these operations, they cannot contain subqueries or access data in collections using AQL functions.

Finally, data-modification operations can optionally be followed by `LET` and `RETURN`, but not by other statements such as `SORT`, `COLLECT` etc.

Transactional Execution

On a single server, data-modification operations are executed transactionally. If a data-modification operation fails, any changes made by it will be rolled back automatically as if they never happened.

In a cluster, AQL data-modification queries are currently not executed transactionally. Additionally, *update*, *replace*, *upsert* and *remove* AQL queries currently require the *_key* attribute to be specified for all documents that should be modified or removed, even if a shared key attribute other than *_key* was chosen for the collection. This restriction may be overcome in a future release of ArangoDB.

High-level operations

The following high-level operations are described here after:

- FOR: Iterate over all elements of an array.
- RETURN: Produce the result of a query.
- FILTER: Restrict the results to elements that match arbitrary logical conditions.
- SORT: Force a sort of the array of already produced intermediate results.
- LIMIT: Reduce the number of elements in the result to at most the specified number, optionally skip elements (pagination).
- LET: Assign an arbitrary value to a variable.
- COLLECT: Group an array by one or multiple group criteria. Can also count and aggregate.
- REMOVE: Remove documents from a collection.
- UPDATE: Partially update documents in a collection.
- REPLACE: Completely replace documents in a collection.
- INSERT: Insert new documents into a collection.
- UPSERT: Update/replace an existing document, or create it in the case it does not exist.
- WITH: Specify collections used in a query (at query begin only).

FOR

The *FOR* keyword can be to iterate over all elements of an array. The general syntax is:

```
FOR variableName IN expression
```

There is also a special variant for graph traversals:

```
FOR vertexVariableName, edgeVariableName, pathVariableName IN traversalExpression
```

For this special case see the graph traversals chapter. For all other cases read on:

Each array element returned by *expression* is visited exactly once. It is required that *expression* returns an array in all cases. The empty array is allowed, too. The current array element is made available for further processing in the variable specified by *variableName*.

```
FOR u IN users
  RETURN u
```

This will iterate over all elements from the array *users* (note: this array consists of all documents from the collection named "users" in this case) and make the current array element available in variable *u*. *u* is not modified in this example but simply pushed into the result using the *RETURN* keyword.

Note: When iterating over collection-based arrays as shown here, the order of documents is undefined unless an explicit sort order is defined using a *SORT* statement.

The variable introduced by *FOR* is available until the scope the *FOR* is placed in is closed.

Another example that uses a statically declared array of values to iterate over:

```
FOR year IN [ 2011, 2012, 2013 ]
  RETURN { "year" : year, "isLeapYear" : year % 4 == 0 && (year % 100 != 0 || year % 400 == 0) }
```

Nesting of multiple *FOR* statements is allowed, too. When *FOR* statements are nested, a cross product of the array elements returned by the individual *FOR* statements will be created.

```
FOR u IN users
  FOR l IN locations
    RETURN { "user" : u, "location" : l }
```

In this example, there are two array iterations: an outer iteration over the array *users* plus an inner iteration over the array *locations*. The inner array is traversed as many times as there are elements in the outer array. For each iteration, the current values of *users* and *locations* are made available for further processing in the variable *u* and *l*.

RETURN

The *RETURN* statement can be used to produce the result of a query. It is mandatory to specify a *RETURN* statement at the end of each block in a data-selection query, otherwise the query result would be undefined. Using *RETURN* on the main level in data-modification queries is optional.

The general syntax for *RETURN* is:

```
RETURN expression
```

The *expression* returned by *RETURN* is produced for each iteration in the block the *RETURN* statement is placed in. That means the result of a *RETURN* statement is **always an array**. This includes an empty array if no documents matched the query and a single return value returned as array with one element.

To return all elements from the currently iterated array without modification, the following simple form can be used:

```
FOR variableName IN expression
  RETURN variableName
```

As *RETURN* allows specifying an expression, arbitrary computations can be performed to calculate the result elements. Any of the variables valid in the scope the *RETURN* is placed in can be used for the computations.

To iterate over all documents of a collection called *users* and return the full documents, you can write:

```
FOR u IN users
  RETURN u
```

In each iteration of the for-loop, a document of the *users* collection is assigned to a variable *u* and returned unmodified in this example. To return only one attribute of each document, you could use a different return expression:

```
FOR u IN users
  RETURN u.name
```

Or to return multiple attributes, an object can be constructed like this:

```
FOR u IN users
  RETURN { name: u.name, age: u.age }
```

Note: *RETURN* will close the current scope and eliminate all local variables in it. This is important to remember when working with subqueries.

Dynamic attribute names are supported as well:

```
FOR u IN users
  RETURN { [ u._id ]: u.age }
```

The document *_id* of every user is used as expression to compute the attribute key in this example:

```
[
  {
    "users/9883": 32
  },
  {
    "users/9915": 27
  },
  {
    "users/10074": 69
  }
]
```

The result contains one object per user with a single key/value pair each. This is usually not desired. For a single object, that maps user IDs to ages, the individual results need to be merged and returned with another `RETURN`:

```
RETURN MERGE(
  FOR u IN users
    RETURN { [ u._id ]: u.age }
)
```

```
[
  {
    "users/10074": 69,
    "users/9883": 32,
    "users/9915": 27
  }
]
```

Keep in mind that if the key expression evaluates to the same value multiple times, only one of the key/value pairs with the duplicate name will survive MERGE(). To avoid this, you can go without dynamic attribute names, use static names instead and return all document properties as attribute values:

```
FOR u IN users
  RETURN { name: u.name, age: u.age }
```

```
[
  {
    "name": "John Smith",
    "age": 32
  },
  {
    "name": "James Hendrix",
    "age": 69
  },
  {
    "name": "Katie Foster",
    "age": 27
  }
]
```

RETURN DISTINCT

Since ArangoDB 2.7, *RETURN* can optionally be followed by the *DISTINCT* keyword. The *DISTINCT* keyword will ensure uniqueness of the values returned by the *RETURN* statement:

```
FOR variableName IN expression
  RETURN DISTINCT expression
```

If the *DISTINCT* is applied on an expression that itself is an array or a subquery, the *DISTINCT* will not make the values in each array or subquery result unique, but instead ensure that the result contains only distinct arrays or subquery results. To make the result of an array or a subquery unique, simply apply the *DISTINCT* for the array or the subquery.

For example, the following query will apply *DISTINCT* on its subquery results, but not inside the subquery:

```
FOR what IN 1..2
  RETURN DISTINCT (
    FOR i IN [ 1, 2, 3, 4, 1, 3 ]
      RETURN i
  )
```

Here we'll have a *FOR* loop with two iterations that each execute a subquery. The *DISTINCT* here is applied on the two subquery results. Both subqueries return the same result value (that is [1, 2, 3, 4, 1, 3]), so after *DISTINCT* there will only be one occurrence of the value [1, 2, 3, 4, 1, 3] left:

```
[
```

```
  [ 1, 2, 3, 4, 1, 3 ]
]
```

If the goal is to apply the *DISTINCT* inside the subquery, it needs to be moved there:

```
FOR what IN 1..2
  LET sub = (
    FOR i IN [ 1, 2, 3, 4, 1, 3 ]
      RETURN DISTINCT i
  )
  RETURN sub
```

In the above case, the *DISTINCT* will make the subquery results unique, so that each subquery will return a unique array of values ([1, 2, 3, 4]). As the subquery is executed twice and there is no *DISTINCT* on the top-level, that array will be returned twice:

```
[
  [ 1, 2, 3, 4 ],
  [ 1, 2, 3, 4 ]
]
```

Note: the order of results is undefined for *RETURN DISTINCT*.

Note: *RETURN DISTINCT* is not allowed on the top-level of a query if there is no *FOR* loop preceding it.

FILTER

The *FILTER* statement can be used to restrict the results to elements that match an arbitrary logical condition.

General syntax

```
FILTER condition
```

condition must be a condition that evaluates to either *false* or *true*. If the condition result is false, the current element is skipped, so it will not be processed further and not be part of the result. If the condition is true, the current element is not skipped and can be further processed. See Operators for a list of comparison operators, logical operators etc. that you can use in conditions.

```
FOR u IN users
  FILTER u.active == true && u.age < 39
  RETURN u
```

It is allowed to specify multiple *FILTER* statements in a query, even in the same block. If multiple *FILTER* statements are used, their results will be combined with a logical AND, meaning all filter conditions must be true to include an element.

```
FOR u IN users
  FILTER u.active == true
  FILTER u.age < 39
  RETURN u
```

In the above example, all array elements of *users* that have an attribute *active* with value *true* and that have an attribute *age* with a value less than *39* (including *null* ones) will be included in the result. All other elements of *users* will be skipped and not be included in the result produced by *RETURN*. You may refer to the chapter Accessing Data from Collections for a description of the impact of non-existent or null attributes.

Order of operations

Note that the positions of *FILTER* statements can influence the result of a query. There are 16 active users in the test data for instance:

```
FOR u IN users
  FILTER u.active == true
  RETURN u
```

We can limit the result set to 5 users at most:

```
FOR u IN users
  FILTER u.active == true
  LIMIT 5
  RETURN u
```

This may return the user documents of Jim, Diego, Anthony, Michael and Chloe for instance. Which ones are returned is undefined, since there is no *SORT* statement to ensure a particular order. If we add a second *FILTER* statement to only return women...

```
FOR u IN users
  FILTER u.active == true
  LIMIT 5
  FILTER u.gender == "f"
  RETURN u
```

... it might just return the Chloe document, because the *LIMIT* is applied before the second *FILTER*. No more than 5 documents arrive at the second *FILTER* block, and not all of them fulfill the gender criterion, eventhough there are more than 5 active female users in the collection. A more deterministic result can be achieved by adding a *SORT* block:

```
FOR u IN users
  FILTER u.active == true
  SORT u.age ASC
  LIMIT 5
  FILTER u.gender == "f"
  RETURN u
```

This will return the users Mariah and Mary. If sorted by age in *DESC* order, then the Sophia, Emma and Madison documents are returned. A *FILTER* after a *LIMIT* is not very common however, and you probably want such a query instead:

```
FOR u IN users
  FILTER u.active == true AND u.gender == "f"
  SORT u.age ASC
  LIMIT 5
  RETURN u
```

The significance of where *FILTER* blocks are placed allows that this single keyword can assume the roles of two SQL keywords, *WHERE* as well as *HAVING*. AQL's *FILTER* thus works with *COLLECT* aggregates the same as with any other intermediate result, document attribute etc.

SORT

The *SORT* statement will force a sort of the array of already produced intermediate results in the current block. *SORT* allows specifying one or multiple sort criteria and directions. The general syntax is:

```
SORT expression direction
```

Example query that is sorting by lastName (in ascending order), then firstName (in ascending order), then by id (in descending order):

```
FOR u IN users
  SORT u.lastName, u.firstName, u.id DESC
  RETURN u
```

Specifying the *direction* is optional. The default (implicit) direction for a sort expression is the ascending order. To explicitly specify the sort direction, the keywords *ASC* (ascending) and *DESC* can be used. Multiple sort criteria can be separated using commas. In this case the direction is specified for each expression sperately. For example

```
SORT doc.lastName, doc.firstName
```

will first sort documents by lastName in ascending order and then by firstName in ascending order.

```
SORT doc.lastName DESC, doc.firstName
```

will first sort documents by lastName in descending order and then by firstName in ascending order.

```
SORT doc.lastName, doc.firstName DESC
```

will first sort documents by lastName in ascending order and then by firstName in descending order.

Note: when iterating over collection-based arrays, the order of documents is always undefined unless an explicit sort order is defined using *SORT*.

Note that constant *SORT* expressions can be used to indicate that no particular sort order is desired. Constant *SORT* expressions will be optimized away by the AQL optimizer during optimization, but specifying them explicitly may enable further optimizations if the optimizer does not need to take into account any particular sort order. This is especially the case after a *COLLECT* statement, which is supposed to produce a sorted result. Specifying an extra *SORT null* after the *COLLECT* statement allows to AQL optimizer to remove the post-sorting of the collect results altogether.

LIMIT

The *LIMIT* statement allows slicing the result array using an offset and a count. It reduces the number of elements in the result to at most the specified number. Two general forms of *LIMIT* are followed:

```
LIMIT count
LIMIT offset, count
```

The first form allows specifying only the *count* value whereas the second form allows specifying both *offset* and *count*. The first form is identical using the second form with an *offset* value of *0*.

```
FOR u IN users
  LIMIT 5
  RETURN u
```

Above query returns the first five documents of the *users* collection. It could also be written as `LIMIT 0, 5` for the same result. Which documents it actually returns is rather arbitrary, because no explicit sorting order is specified however. Therefore, a limit should be usually accompanied by a `SORT` operation.

The *offset* value specifies how many elements from the result shall be skipped. It must be 0 or greater. The *count* value specifies how many elements should be at most included in the result.

```
FOR u IN users
  SORT u.firstName, u.lastName, u.id DESC
  LIMIT 2, 5
  RETURN u
```

In above example, the documents of *users* are sorted, the first two results get skipped and it returns the next five user documents.

Note that variables and expressions can not be used for *offset* and *count*. Their values must be known at query compile time, which means that you can use number literals and bind parameters only.

Where a *LIMIT* is used in relation to other operations in a query has meaning. *LIMIT* operations before *FILTER*s in particular can change the result significantly, because the operations are executed in the order in which they are written in the query. See FILTER for a detailed example.

LET

The *LET* statement can be used to assign an arbitrary value to a variable. The variable is then introduced in the scope the *LET* statement is placed in.

The general syntax is:

```
LET variableName = expression
```

Variables are immutable in AQL, which means they can not be re-assigned:

```
LET a = [1, 2, 3]  // initial assignment

a = PUSH(a, 4)     // syntax error, unexpected identifier
LET a = PUSH(a, 4) // parsing error, variable 'a' is assigned multiple times
LET b = PUSH(a, 4) // allowed, result: [1, 2, 3, 4]
```

LET statements are mostly used to declare complex computations and to avoid repeated computations of the same value at multiple parts of a query.

```
FOR u IN users
  LET numRecommendations = LENGTH(u.recommendations)
  RETURN {
    "user" : u,
    "numRecommendations" : numRecommendations,
    "isPowerUser" : numRecommendations >= 10
  }
```

In the above example, the computation of the number of recommendations is factored out using a *LET* statement, thus avoiding computing the value twice in the *RETURN* statement.

Another use case for *LET* is to declare a complex computation in a subquery, making the whole query more readable.

```
FOR u IN users
  LET friends = (
  FOR f IN friends
    FILTER u.id == f.userId
    RETURN f
  )
  LET memberships = (
  FOR m IN memberships
    FILTER u.id == m.userId
      RETURN m
  )
  RETURN {
    "user" : u,
    "friends" : friends,
    "numFriends" : LENGTH(friends),
    "memberShips" : memberships
  }
```

COLLECT

The *COLLECT* keyword can be used to group an array by one or multiple group criteria.

The *COLLECT* statement will eliminate all local variables in the current scope. After *COLLECT* only the variables introduced by *COLLECT* itself are available.

The general syntaxes for *COLLECT* are:

```
COLLECT variableName = expression options
COLLECT variableName = expression INTO groupsVariable options
COLLECT variableName = expression INTO groupsVariable = projectionExpression options
COLLECT variableName = expression INTO groupsVariable KEEP keepVariable options
COLLECT variableName = expression WITH COUNT INTO countVariable options
COLLECT variableName = expression AGGREGATE variableName = aggregateExpression options
COLLECT AGGREGATE variableName = aggregateExpression options
COLLECT WITH COUNT INTO countVariable options
```

`options` is optional in all variants.

Grouping syntaxes

The first syntax form of *COLLECT* only groups the result by the defined group criteria specified in *expression*. In order to further process the results produced by *COLLECT*, a new variable (specified by *variableName*) is introduced. This variable contains the group value.

Here's an example query that find the distinct values in *u.city* and makes them available in variable *city*:

```
FOR u IN users
  COLLECT city = u.city
  RETURN {
    "city" : city
  }
```

The second form does the same as the first form, but additionally introduces a variable (specified by *groupsVariable*) that contains all elements that fell into the group. This works as follows: The *groupsVariable* variable is an array containing as many elements as there are in the group. Each member of that array is a JSON object in which the value of every variable that is defined in the AQL query is bound to the corresponding attribute. Note that this considers all variables that are defined before the *COLLECT* statement, but not those on the top level (outside of any *FOR*), unless the *COLLECT* statement is itself on the top level, in which case all variables are taken. Furthermore note that it is possible that the optimizer moves *LET* statements out of *FOR* statements to improve performance.

```
FOR u IN users
  COLLECT city = u.city INTO groups
  RETURN {
    "city" : city,
    "usersInCity" : groups
  }
```

In the above example, the array *users* will be grouped by the attribute *city*. The result is a new array of documents, with one element per distinct *u.city* value. The elements from the original array (here: *users*) per city are made available in the variable *groups*. This is due to the *INTO* clause.

COLLECT also allows specifying multiple group criteria. Individual group criteria can be separated by commas:

```
FOR u IN users
  COLLECT country = u.country, city = u.city INTO groups
  RETURN {
    "country" : country,
    "city" : city,
    "usersInCity" : groups
  }
```

In the above example, the array *users* is grouped by country first and then by city, and for each distinct combination of country and city, the users will be returned.

Discarding obsolete variables

The third form of *COLLECT* allows rewriting the contents of the *groupsVariable* using an arbitrary *projectionExpression*:

```
FOR u IN users
  COLLECT country = u.country, city = u.city INTO groups = u.name
  RETURN {
    "country" : country,
    "city" : city,
    "userNames" : groups
  }
```

In the above example, only the *projectionExpression* is *u.name*. Therefore, only this attribute is copied into the *groupsVariable* for each document. This is probably much more efficient than copying all variables from the scope into the *groupsVariable* as it would happen without a *projectionExpression*.

The expression following *INTO* can also be used for arbitrary computations:

```
FOR u IN users
  COLLECT country = u.country, city = u.city INTO groups = {
    "name" : u.name,
    "isActive" : u.status == "active"
  }
  RETURN {
    "country" : country,
    "city" : city,
    "usersInCity" : groups
  }
```

COLLECT also provides an optional *KEEP* clause that can be used to control which variables will be copied into the variable created by `INTO`. If no *KEEP* clause is specified, all variables from the scope will be copied as sub-attributes into the *groupsVariable*. This is safe but can have a negative impact on performance if there are many variables in scope or the variables contain massive amounts of data.

The following example limits the variables that are copied into the *groupsVariable* to just *name*. The variables *u* and *someCalculation* also present in the scope will not be copied into *groupsVariable* because they are not listed in the *KEEP* clause:

```
FOR u IN users
  LET name = u.name
  LET someCalculation = u.value1 + u.value2
  COLLECT city = u.city INTO groups KEEP name
  RETURN {
    "city" : city,
    "userNames" : groups[*].name
  }
```

KEEP is only valid in combination with *INTO*. Only valid variable names can be used in the *KEEP* clause. *KEEP* supports the specification of multiple variable names.

Group length calculation

COLLECT also provides a special *WITH COUNT* clause that can be used to determine the number of group members efficiently.

The simplest form just returns the number of items that made it into the *COLLECT*:

```
FOR u IN users
  COLLECT WITH COUNT INTO length
  RETURN length
```

The above is equivalent to, but more efficient than:

```
RETURN LENGTH(
```

```
  FOR u IN users
    RETURN length
)
```

The *WITH COUNT* clause can also be used to efficiently count the number of items in each group:

```
FOR u IN users
  COLLECT age = u.age WITH COUNT INTO length
  RETURN {
    "age" : age,
    "count" : length
  }
```

Note: the *WITH COUNT* clause can only be used together with an *INTO* clause.

Aggregation

A `COLLECT` statement can be used to perform aggregation of data per group. To only determine group lengths, the `WITH COUNT INTO` variant of `COLLECT` can be used as described before.

For other aggregations, it is possible to run aggregate functions on the `COLLECT` results:

```
FOR u IN users
  COLLECT ageGroup = FLOOR(u.age / 5) * 5 INTO g
  RETURN {
    "ageGroup" : ageGroup,
    "minAge" : MIN(g[*].u.age),
    "maxAge" : MAX(g[*].u.age)
  }
```

The above however requires storing all group values during the collect operation for all groups, which can be inefficient.

The special `AGGREGATE` variant of `COLLECT` allows building the aggregate values incrementally during the collect operation, and is therefore often more efficient.

With the `AGGREGATE` variant the above query becomes:

```
FOR u IN users
  COLLECT ageGroup = FLOOR(u.age / 5) * 5
  AGGREGATE minAge = MIN(u.age), maxAge = MAX(u.age)
  RETURN {
    ageGroup,
    minAge,
    maxAge
  }
```

The `AGGREGATE` keyword can only be used after the `COLLECT` keyword. If used, it must directly follow the declaration of the grouping keys. If no grouping keys are used, it must follow the `COLLECT` keyword directly:

```
FOR u IN users
  COLLECT AGGREGATE minAge = MIN(u.age), maxAge = MAX(u.age)
  RETURN {
    minAge,
    maxAge
  }
```

Only specific expressions are allowed on the right-hand side of each `AGGREGATE` assignment:

- on the top level, an aggregate expression must be a call to one of the supported aggregation functions `LENGTH`, `MIN`, `MAX`, `SUM`, `AVERAGE`, `STDDEV_POPULATION`, `STDDEV_SAMPLE`, `VARIANCE_POPULATION`, or `VARIANCE_SAMPLE`
- an aggregate expression must not refer to variables introduced by the `COLLECT` itself

COLLECT variants

Since ArangoDB 2.6, there are two variants of *COLLECT* that the optimizer can choose from: the *sorted* variant and the *hash* variant. The *hash* variant only becomes a candidate for *COLLECT* statements that do not use an *INTO* clause.

The optimizer will always generate a plan that employs the *sorted* method. The *sorted* method requires its input to be sorted by the group criteria specified in the *COLLECT* clause. To ensure correctness of the result, the AQL optimizer will automatically insert a *SORT* statement into the query in front of the *COLLECT* statement. The optimizer may be able to optimize away that *SORT* statement later if a sorted index is present on the group criteria.

In case a *COLLECT* qualifies for using the *hash* variant, the optimizer will create an extra plan for it at the beginning of the planning phase. In this plan, no extra *SORT* statement will be added in front of the *COLLECT*. This is because the *hash* variant of *COLLECT* does not require sorted input. Instead, a *SORT* statement will be added after the *COLLECT* to sort its output. This *SORT* statement may be optimized away again in later stages. If the sort order of the *COLLECT* is irrelevant to the user, adding the extra instruction *SORT null* after the *COLLECT* will allow the optimizer to remove the sorts altogether:

```
FOR u IN users
  COLLECT age = u.age
  SORT null  /* note: will be optimized away */
  RETURN age
```

Which *COLLECT* variant is used by the optimizer depends on the optimizer's cost estimations. The created plans with the different *COLLECT* variants will be shipped through the regular optimization pipeline. In the end, the optimizer will pick the plan with the lowest estimated total cost as usual.

In general, the *sorted* variant of *COLLECT* should be preferred in cases when there is a sorted index present on the group criteria. In this case the optimizer can eliminate the *SORT* statement in front of the *COLLECT*, so that no *SORT* will be left.

If there is no sorted index available on the group criteria, the up-front sort required by the *sorted* variant can be expensive. In this case it is likely that the optimizer will prefer the *hash* variant of *COLLECT*, which does not require its input to be sorted.

Which variant of *COLLECT* was actually used can be figured out by looking into the execution plan of a query, specifically the *AggregateNode* and its *aggregationOptions* attribute.

Setting COLLECT options

options can be used in a *COLLECT* statement to inform the optimizer about the preferred *COLLECT* method. When specifying the following appendix to a *COLLECT* statement, the optimizer will always use the *sorted* variant of *COLLECT* and not even create a plan using the *hash* variant:

```
OPTIONS { method: "sorted" }
```

Note that specifying *hash* as method will not make the optimizer use the *hash* variant. This is because the *hash* variant is not eligible for all queries. Instead, if no options or any other method than *sorted* are specified in *OPTIONS*, the optimizer will use its regular cost estimations.

COLLECT vs. RETURN DISTINCT

In order to make a result set unique, one can either use *COLLECT* or *RETURN DISTINCT*. Behind the scenes, both variants will work by creating an *AggregateNode*. For both variants, the optimizer may try the sorted and the hashed variant of *COLLECT*. The difference is therefore mainly syntactical, with *RETURN DISTINCT* saving a bit of typing when compared to an equivalent *COLLECT*:

```
FOR u IN users
  RETURN DISTINCT u.age
```

```
FOR u IN users
  COLLECT age = u.age
  RETURN age
```

However, *COLLECT* is vastly more flexible than *RETURN DISTINCT*. Additionally, the order of results is undefined for a *RETURN DISTINCT*, whereas for a *COLLECT* the results will be sorted.

REMOVE

The *REMOVE* keyword can be used to remove documents from a collection. On a single server, the document removal is executed transactionally in an all-or-nothing fashion. For sharded collections, the entire remove operation is not transactional.

Each *REMOVE* operation is restricted to a single collection, and the collection name must not be dynamic. Only a single *REMOVE* statement per collection is allowed per AQL query, and it cannot be followed by read operations that access the same collection, by traversal operations, or AQL functions that can read documents.

The syntax for a remove operation is:

```
REMOVE keyExpression IN collection options
```

collection must contain the name of the collection to remove the documents from. *keyExpression* must be an expression that contains the document identification. This can either be a string (which must then contain the document key) or a document, which must contain a *_key* attribute.

The following queries are thus equivalent:

```
FOR u IN users
  REMOVE { _key: u._key } IN users

FOR u IN users
  REMOVE u._key IN users

FOR u IN users
  REMOVE u IN users
```

Note: A remove operation can remove arbitrary documents, and the documents do not need to be identical to the ones produced by a preceding *FOR* statement:

```
FOR i IN 1..1000
  REMOVE { _key: CONCAT('test', i) } IN users

FOR u IN users
  FILTER u.active == false
  REMOVE { _key: u._key } IN backup
```

Setting query options

options can be used to suppress query errors that may occur when trying to remove non-existing documents. For example, the following query will fail if one of the to-be-deleted documents does not exist:

```
FOR i IN 1..1000
  REMOVE { _key: CONCAT('test', i) } IN users
```

By specifying the *ignoreErrors* query option, these errors can be suppressed so the query completes:

```
FOR i IN 1..1000
  REMOVE { _key: CONCAT('test', i) } IN users OPTIONS { ignoreErrors: true }
```

To make sure data has been written to disk when a query returns, there is the *waitForSync* query option:

```
FOR i IN 1..1000
  REMOVE { _key: CONCAT('test', i) } IN users OPTIONS { waitForSync: true }
```

Returning the removed documents

The removed documents can also be returned by the query. In this case, the `REMOVE` statement must be followed by a `RETURN` statement (intermediate `LET` statements are allowed, too). `REMOVE` introduces the pseudo-value `OLD` to refer to the removed documents:

```
REMOVE keyExpression IN collection options RETURN OLD
```

Following is an example using a variable named `removed` for capturing the removed documents. For each removed document, the document key will be returned.

```
FOR u IN users
  REMOVE u IN users
  LET removed = OLD
  RETURN removed._key
```

UPDATE

The *UPDATE* keyword can be used to partially update documents in a collection. On a single server, updates are executed transactionally in an all-or-nothing fashion. For sharded collections, the entire update operation is not transactional.

Each *UPDATE* operation is restricted to a single collection, and the collection name must not be dynamic. Only a single *UPDATE* statement per collection is allowed per AQL query, and it cannot be followed by read operations that access the same collection, by traversal operations, or AQL functions that can read documents. The system attributes *_id*, *_key* and *_rev* cannot be updated, *_from* and *_to* can.

The two syntaxes for an update operation are:

```
UPDATE document IN collection options
UPDATE keyExpression WITH document IN collection options
```

collection must contain the name of the collection in which the documents should be updated. *document* must be a document that contains the attributes and values to be updated. When using the first syntax, *document* must also contain the *_key* attribute to identify the document to be updated.

```
FOR u IN users
  UPDATE { _key: u._key, name: CONCAT(u.firstName, " ", u.lastName) } IN users
```

The following query is invalid because it does not contain a *_key* attribute and thus it is not possible to determine the documents to be updated:

```
FOR u IN users
  UPDATE { name: CONCAT(u.firstName, " ", u.lastName) } IN users
```

When using the second syntax, *keyExpression* provides the document identification. This can either be a string (which must then contain the document key) or a document, which must contain a *_key* attribute.

The following queries are equivalent:

```
FOR u IN users
  UPDATE u._key WITH { name: CONCAT(u.firstName, " ", u.lastName) } IN users

FOR u IN users
  UPDATE { _key: u._key } WITH { name: CONCAT(u.firstName, " ", u.lastName) } IN users

FOR u IN users
  UPDATE u WITH { name: CONCAT(u.firstName, " ", u.lastName) } IN users
```

An update operation may update arbitrary documents which do not need to be identical to the ones produced by a preceding *FOR* statement:

```
FOR i IN 1..1000
  UPDATE CONCAT('test', i) WITH { foobar: true } IN users

FOR u IN users
  FILTER u.active == false
  UPDATE u WITH { status: 'inactive' } IN backup
```

Using the current value of a document attribute

The pseudo-variable `OLD` is not supported inside of `WITH` clauses (it is available after `UPDATE`). To access the current attribute value, you can usually refer to a document via the variable of the `FOR` loop, which is used to iterate over a collection:

```
FOR doc IN users
  UPDATE doc WITH {
    fullName: CONCAT(doc.firstName, " ", doc.lastName)
```

```
  } IN users
```

If there is no loop, because a single document is updated only, then there might not be a variable like above (`doc`), which would let you refer to the document which is being updated:

```
UPDATE "users/john" WITH { ... } IN users
```

To access the current value in this situation, the document has to be retrieved and stored in a variable first:

```
LET doc = DOCUMENT("users/john")
UPDATE doc WITH {
  fullName: CONCAT(doc.firstName, " ", doc.lastName)
} IN users
```

An existing attribute can be modified based on its current value this way, to increment a counter for instance:

```
UPDATE doc WITH {
  karma: doc.karma + 1
} IN users
```

If the attribute `karma` doesn't exist yet, `doc.karma` is evaluated to *null*. The expression `null + 1` results in the new attribute `karma` being set to *1*. If the attribute does exist, then it is increased by *1*.

Arrays can be mutated too of course:

```
UPDATE doc WITH {
  hobbies: PUSH(doc.hobbies, "swimming")
} IN users
```

If the attribute `hobbies` doesn't exist yet, it is conveniently initialized as `[ "swimming" ]` and otherwise extended.

Setting query options

options can be used to suppress query errors that may occur when trying to update non-existing documents or violating unique key constraints:

```
FOR i IN 1..1000
  UPDATE {
    _key: CONCAT('test', i)
  } WITH {
    foobar: true
  } IN users OPTIONS { ignoreErrors: true }
```

An update operation will only update the attributes specified in *document* and leave other attributes untouched. Internal attributes (such as *_id*, *_key*, *_rev*, *_from* and *_to*) cannot be updated and are ignored when specified in *document*. Updating a document will modify the document's revision number with a server-generated value.

When updating an attribute with a null value, ArangoDB will not remove the attribute from the document but store a null value for it. To get rid of attributes in an update operation, set them to null and provide the *keepNull* option:

```
FOR u IN users
  UPDATE u WITH {
    foobar: true,
    notNeeded: null
  } IN users OPTIONS { keepNull: false }
```

The above query will remove the *notNeeded* attribute from the documents and update the *foobar* attribute normally.

There is also the option *mergeObjects* that controls whether object contents will be merged if an object attribute is present in both the *UPDATE* query and in the to-be-updated document.

The following query will set the updated document's *name* attribute to the exact same value that is specified in the query. This is due to the *mergeObjects* option being set to *false*:

```
FOR u IN users
  UPDATE u WITH {
    name: { first: "foo", middle: "b.", last: "baz" }
  } IN users OPTIONS { mergeObjects: false }
```

Contrary, the following query will merge the contents of the *name* attribute in the original document with the value specified in the query:

```
FOR u IN users
  UPDATE u WITH {
    name: { first: "foo", middle: "b.", last: "baz" }
  } IN users OPTIONS { mergeObjects: true }
```

Attributes in *name* that are present in the to-be-updated document but not in the query will now be preserved. Attributes that are present in both will be overwritten with the values specified in the query.

Note: the default value for *mergeObjects* is *true*, so there is no need to specify it explicitly.

To make sure data are durable when an update query returns, there is the *waitForSync* query option:

```
FOR u IN users
  UPDATE u WITH {
    foobar: true
  } IN users OPTIONS { waitForSync: true }
```

Returning the modified documents

The modified documents can also be returned by the query. In this case, the `UPDATE` statement needs to be followed a `RETURN` statement (intermediate `LET` statements are allowed, too). These statements can refer to the pseudo-values `OLD` and `NEW`. The `OLD` pseudo-value refers to the document revisions before the update, and `NEW` refers to document revisions after the update.

Both `OLD` and `NEW` will contain all document attributes, even those not specified in the update expression.

```
UPDATE document IN collection options RETURN OLD
UPDATE document IN collection options RETURN NEW
UPDATE keyExpression WITH document IN collection options RETURN OLD
UPDATE keyExpression WITH document IN collection options RETURN NEW
```

Following is an example using a variable named `previous` to capture the original documents before modification. For each modified document, the document key is returned.

```
FOR u IN users
  UPDATE u WITH { value: "test" }
  LET previous = OLD
  RETURN previous._key
```

The following query uses the `NEW` pseudo-value to return the updated documents, without some of the system attributes:

```
FOR u IN users
  UPDATE u WITH { value: "test" }
  LET updated = NEW
  RETURN UNSET(updated, "_key", "_id", "_rev")
```

It is also possible to return both `OLD` and `NEW`:

```
FOR u IN users
  UPDATE u WITH { value: "test" }
  RETURN { before: OLD, after: NEW }
```

REPLACE

The *REPLACE* keyword can be used to completely replace documents in a collection. On a single server, the replace operation is executed transactionally in an all-or-nothing fashion. For sharded collections, the entire replace operation is not transactional.

Each *REPLACE* operation is restricted to a single collection, and the collection name must not be dynamic. Only a single *REPLACE* statement per collection is allowed per AQL query, and it cannot be followed by read operations that access the same collection, by traversal operations, or AQL functions that can read documents. The system attributes *_id*, *_key* and *_rev* cannot be replaced, *_from* and *_to* can.

The two syntaxes for a replace operation are:

```
REPLACE document IN collection options
REPLACE keyExpression WITH document IN collection options
```

collection must contain the name of the collection in which the documents should be replaced. *document* is the replacement document. When using the first syntax, *document* must also contain the *_key* attribute to identify the document to be replaced.

```
FOR u IN users
  REPLACE { _key: u._key, name: CONCAT(u.firstName, u.lastName), status: u.status } IN users
```

The following query is invalid because it does not contain a *_key* attribute and thus it is not possible to determine the documents to be replaced:

```
FOR u IN users
  REPLACE { name: CONCAT(u.firstName, u.lastName, status: u.status) } IN users
```

When using the second syntax, *keyExpression* provides the document identification. This can either be a string (which must then contain the document key) or a document, which must contain a *_key* attribute.

The following queries are equivalent:

```
FOR u IN users
  REPLACE { _key: u._key, name: CONCAT(u.firstName, u.lastName) } IN users

FOR u IN users
  REPLACE u._key WITH { name: CONCAT(u.firstName, u.lastName) } IN users

FOR u IN users
  REPLACE { _key: u._key } WITH { name: CONCAT(u.firstName, u.lastName) } IN users

FOR u IN users
  REPLACE u WITH { name: CONCAT(u.firstName, u.lastName) } IN users
```

A replace will fully replace an existing document, but it will not modify the values of internal attributes (such as *_id*, *_key*, *_from* and *_to*). Replacing a document will modify a document's revision number with a server-generated value.

A replace operation may update arbitrary documents which do not need to be identical to the ones produced by a preceding *FOR* statement:

```
FOR i IN 1..1000
  REPLACE CONCAT('test', i) WITH { foobar: true } IN users

FOR u IN users
  FILTER u.active == false
  REPLACE u WITH { status: 'inactive', name: u.name } IN backup
```

Setting query options

options can be used to suppress query errors that may occur when trying to replace non-existing documents or when violating unique key constraints:

```
FOR i IN 1..1000
  REPLACE { _key: CONCAT('test', i) } WITH { foobar: true } IN users OPTIONS { ignoreErrors: true }
```

To make sure data are durable when a replace query returns, there is the *waitForSync* query option:

```
FOR i IN 1..1000
  REPLACE { _key: CONCAT('test', i) } WITH { foobar: true } IN users OPTIONS { waitForSync: true }
```

Returning the modified documents

The modified documents can also be returned by the query. In this case, the `REPLACE` statement must be followed by a `RETURN` statement (intermediate `LET` statements are allowed, too). The `OLD` pseudo-value can be used to refer to document revisions before the replace, and `NEW` refers to document revisions after the replace.

Both `OLD` and `NEW` will contain all document attributes, even those not specified in the replace expression.

```
REPLACE document IN collection options RETURN OLD
REPLACE document IN collection options RETURN NEW
REPLACE keyExpression WITH document IN collection options RETURN OLD
REPLACE keyExpression WITH document IN collection options RETURN NEW
```

Following is an example using a variable named `previous` to return the original documents before modification. For each replaced document, the document key will be returned:

```
FOR u IN users
  REPLACE u WITH { value: "test" }
  LET previous = OLD
  RETURN previous._key
```

The following query uses the `NEW` pseudo-value to return the replaced documents (without some of their system attributes):

```
FOR u IN users
  REPLACE u WITH { value: "test" }
  LET replaced = NEW
  RETURN UNSET(replaced, '_key', '_id', '_rev')
```

INSERT

The *INSERT* keyword can be used to insert new documents into a collection. On a single server, an insert operation is executed transactionally in an all-or-nothing fashion. For sharded collections, the entire insert operation is not transactional.

Each *INSERT* operation is restricted to a single collection, and the collection name must not be dynamic. Only a single *INSERT* statement per collection is allowed per AQL query, and it cannot be followed by read operations that access the same collection, by traversal operations, or AQL functions that can read documents.

The syntax for an insert operation is:

```
INSERT document IN collection options
```

Note: The *INTO* keyword is also allowed in the place of *IN*.

collection must contain the name of the collection into which the documents should be inserted. *document* is the document to be inserted, and it may or may not contain a *_key* attribute. If no *_key* attribute is provided, ArangoDB will auto-generate a value for *_key* value. Inserting a document will also auto-generate a document revision number for the document.

```
FOR i IN 1..100
  INSERT { value: i } IN numbers
```

When inserting into an edge collection, it is mandatory to specify the attributes *_from* and *_to* in document:

```
FOR u IN users
  FOR p IN products
    FILTER u._key == p.recommendedBy
    INSERT { _from: u._id, _to: p._id } IN recommendations
```

Setting query options

options can be used to suppress query errors that may occur when violating unique key constraints:

```
FOR i IN 1..1000
  INSERT {
    _key: CONCAT('test', i),
    name: "test",
    foobar: true
  } INTO users OPTIONS { ignoreErrors: true }
```

To make sure data are durable when an insert query returns, there is the *waitForSync* query option:

```
FOR i IN 1..1000
  INSERT {
    _key: CONCAT('test', i),
    name: "test",
    foobar: true
  } INTO users OPTIONS { waitForSync: true }
```

Returning the inserted documents

The inserted documents can also be returned by the query. In this case, the `INSERT` statement can be a `RETURN` statement (intermediate `LET` statements are allowed, too). To refer to the inserted documents, the `INSERT` statement introduces a pseudo-value named `NEW`.

The documents contained in `NEW` will contain all attributes, even those auto-generated by the database (e.g. `_id`, `_key`, `_rev`).

```
INSERT document IN collection options RETURN NEW
```

Following is an example using a variable named `inserted` to return the inserted documents. For each inserted document, the document key is returned:

```
FOR i IN 1..100
  INSERT { value: i }
  LET inserted = NEW
  RETURN inserted._key
```

UPSERT

The *UPSERT* keyword can be used for checking whether certain documents exist, and to update/replace them in case they exist, or create them in case they do not exist. On a single server, upserts are executed transactionally in an all-or-nothing fashion. For sharded collections, the entire update operation is not transactional.

Each *UPSERT* operation is restricted to a single collection, and the collection name must not be dynamic. Only a single *UPSERT* statement per collection is allowed per AQL query, and it cannot be followed by read operations that access the same collection, by traversal operations, or AQL functions that can read documents.

The syntax for an upsert operation is:

```
UPSERT searchExpression INSERT insertExpression UPDATE updateExpression IN collection options
UPSERT searchExpression INSERT insertExpression REPLACE updateExpression IN collection options
```

When using the *UPDATE* variant of the upsert operation, the found document will be partially updated, meaning only the attributes specified in *updateExpression* will be updated or added. When using the *REPLACE* variant of upsert, existing documents will be replaced with the contexts of *updateExpression*.

Updating a document will modify the document's revision number with a server-generated value. The system attributes *_id*, *_key* and *_rev* cannot be updated, *_from* and *_to* can.

The *searchExpression* contains the document to be looked for. It must be an object literal without dynamic attribute names. In case no such document can be found in *collection*, a new document will be inserted into the collection as specified in the *insertExpression*.

In case at least one document in *collection* matches the *searchExpression*, it will be updated using the *updateExpression*. When more than one document in the collection matches the *searchExpression*, it is undefined which of the matching documents will be updated. It is therefore often sensible to make sure by other means (such as unique indexes, application logic etc.) that at most one document matches *searchExpression*.

The following query will look in the *users* collection for a document with a specific *name* attribute value. If the document exists, its *logins* attribute will be increased by one. If it does not exist, a new document will be inserted, consisting of the attributes *name*, *logins*, and *dateCreated*:

```
UPSERT { name: 'superuser' }
INSERT { name: 'superuser', logins: 1, dateCreated: DATE_NOW() }
UPDATE { logins: OLD.logins + 1 } IN users
```

Note that in the *UPDATE* case it is possible to refer to the previous version of the document using the *OLD* pseudo-value.

Setting query options

As in several above examples, the *ignoreErrors* option can be used to suppress query errors that may occur when trying to violate unique key constraints.

When updating or replacing an attribute with a null value, ArangoDB will not remove the attribute from the document but store a null value for it. To get rid of attributes in an upsert operation, set them to null and provide the *keepNull* option.

There is also the option *mergeObjects* that controls whether object contents will be merged if an object attribute is present in both the *UPDATE* query and in the to-be-updated document.

Note: the default value for *mergeObjects* is *true*, so there is no need to specify it explicitly.

To make sure data are durable when an update query returns, there is the *waitForSync* query option.

Returning documents

`UPSERT` statements can optionally return data. To do so, they need to be followed by a `RETURN` statement (intermediate `LET` statements are allowed, too). These statements can optionally perform calculations and refer to the pseudo-values `OLD` and `NEW`. In case the upsert performed an insert operation, `OLD` will have a value of *null*. In case the upsert performed an update or replace

operation, `OLD` will contain the previous version of the document, before update/replace.

`NEW` will always be populated. It will contain the inserted document in case the upsert performed an insert, or the updated/replaced document in case it performed an update/replace.

This can also be used to check whether the upsert has performed an insert or an update internally:

```
UPSERT { name: 'superuser' }
INSERT { name: 'superuser', logins: 1, dateCreated: DATE_NOW() }
UPDATE { logins: OLD.logins + 1 } IN users
RETURN { doc: NEW, type: OLD ? 'update' : 'insert' }
```

WITH

An AQL query can optionally start with a *WITH* statement and the list of collections used by the query. All collections specified in *WITH* will be read-locked at query start, in addition to the other collections the query uses and that are detected by the AQL query parser.

Specifying further collections in *WITH* can be useful for queries that dynamically access collections (e.g. via traversals or via dynamic document access functions such as `DOCUMENT()`). Such collections may be invisible to the AQL query parser at query compile time, and thus will not be read-locked automatically at query start. In this case, the AQL execution engine will lazily lock these collections whenever they are used, which can lead to deadlock with other queries. In case such deadlock is detected, the query will automatically be aborted and changes will be rolled back. In this case the client application can try sending the query again. However, if client applications specify the list of used collections for all their queries using *WITH*, then no deadlocks will happen and no queries will be aborted due to deadlock situations.

From ArangoDB 3.1 onwards `WITH` is required for traversals in a clustered environment in order to avoid deadlocks.

Note that for queries that access only a single collection or that have all collection names specified somewhere else in the query string, there is no need to use *WITH*. *WITH* is only useful when the AQL query parser cannot automatically figure out which collections are going to be used by the query. *WITH* is only useful for queries that dynamically access collections, e.g. via traversals, shortest path operations or the *DOCUMENT()* function.

```
WITH managers, usersHaveManagers
FOR v, e, p IN OUTBOUND 'users/1' GRAPH 'userGraph'
  RETURN { v, e, p }
```

Note that constant *WITH* is also a keyword that is used in other contexts, for example in *UPDATE* statements. If *WITH* is used to specify the extra list of collections, then it must be placed at the very start of the query string.

Functions

AQL supports functions to allow more complex computations. Functions can be called at any query position where an expression is allowed. The general function call syntax is:

```
FUNCTIONNAME(arguments)
```

where *FUNCTIONNAME* is the name of the function to be called, and *arguments* is a comma-separated list of function arguments. If a function does not need any arguments, the argument list can be left empty. However, even if the argument list is empty the parentheses around it are still mandatory to make function calls distinguishable from variable names.

Some example function calls:

```
HAS(user, "name")
LENGTH(friends)
COLLECTIONS()
```

In contrast to collection and variable names, function names are case-insensitive, i.e. *LENGTH(foo)* and *length(foo)* are equivalent.

Extending AQL

It is possible to extend AQL with user-defined functions. These functions need to be written in JavaScript, and have to be registered before they can be used in a query. Please refer to Extending AQL for more details.

Type cast functions

Some operators expect their operands to have a certain data type. For example, logical operators expect their operands to be boolean values, and the arithmetic operators expect their operands to be numeric values. If an operation is performed with operands of other types, an automatic conversion to the expected types is tried. This is called implicit type casting. It helps to avoid query aborts.

Type casts can also be performed upon request by invoking a type cast function. This is called explicit type casting. AQL offers several functions for this. Each of the these functions takes an operand of any data type and returns a result value with the type corresponding to the function name. For example, *TO_NUMBER()* will return a numeric value.

TO_BOOL()

`TO_BOOL(value) → bool`

Take an input *value* of any type and convert it into the appropriate boolean value.

- **value** (any): input of arbitrary type
- returns **bool** (boolean):
  - *null* is converted to *false*
  - Numbers are converted to *true*, except for 0, which is converted to *false*
  - Strings are converted to *true* if they are non-empty, and to *false* otherwise
  - Arrays are always converted to *true* (even if empty)
  - Objects / documents are always converted to *true*

It's also possible to use double negation to cast to boolean:

```
!!1 // true
!!0 // false
!!-0.0 // false
not not 1 // true
!!"non-empty string" // true
!!"" // false
```

`TO_BOOL()` is preferred however, because it states the intention clearer.

TO_NUMBER()

`TO_NUMBER(value) → number`

Take an input *value* of any type and convert it into a numeric value.

- **value** (any): input of arbitrary type
- returns **number** (number):
  - *null* and *false* are converted to the value *0*
  - *true* is converted to *1*
  - Numbers keep their original value
  - Strings are converted to their numeric equivalent if the string contains a valid representation of a number. Whitespace at the start and end of the string is allowed. String values that do not contain any valid representation of a number will be converted to the number *0*.
  - An empty array is converted to *0*, an array with one member is converted into the result of `TO_NUMBER()` for its sole member. An array with two or more members is converted to the number *0*.
  - An object / document is converted to the number *0*.

    A unary plus will also cast to a number, but `TO_NUMBER()` is the preferred way:

    ```
    +'5' // 5
    +[8] // 8
    +[8,9] // 0
    +{} // 0
    ```

A unary minus works likewise, except that a numeric value is also negated:

```
-'5' // -5
-[8] // -8
-[8,9] // 0
-{} // 0
```

TO_STRING()

`TO_STRING(value) → str`

Take an input *value* of any type and convert it into a string value.

- **value** (any): input of arbitrary type
- returns **str** (string):
  - *null* is converted to an empty string *""*
  - *false* is converted to the string *"false"*, *true* to the string *"true"*
  - Numbers are converted to their string representations. This can also be a scientific notation (e.g. "2e-7")
  - Arrays and objects / documents are converted to string representations, which means JSON-encoded strings with no additional whitespace

```
TO_STRING(null) // ""
TO_STRING(true) // "true"
TO_STRING(false) // "false"
TO_STRING(123) // "123"
TO_STRING(+1.23) // "1.23"
TO_STRING(-1.23) // "-1.23"
TO_STRING(0.0000002) // "2e-7"
TO_STRING( [1, 2, 3] ) // "[1,2,3]"
TO_STRING( { foo: "bar", baz: null } ) // "{\"foo\":\"bar\",\"baz\":null}"
```

TO_ARRAY()

`TO_ARRAY(value) → array`

Take an input *value* of any type and convert it into an array value.

- **value** (any): input of arbitrary type
- returns **array** (array):
  - *null* is converted to an empty array
  - Boolean values, numbers and strings are converted to an array containing the original value as its single element
  - Arrays keep their original value
  - Objects / documents are converted to an array containing their attribute **values** as array elements, just like VALUES()

```
TO_ARRAY(null) // []
TO_ARRAY(false) // [false]
TO_ARRAY(true) // [true]
TO_ARRAY(5) // [5]
TO_ARRAY("foo") // ["foo"]
TO_ARRAY([1, 2, "foo"]) // [1, 2, "foo"]
TO_ARRAY({foo: 1, bar: 2, baz: [3, 4, 5]}) // [1, 2, [3, 4, 5]]
```

TO_LIST()

`TO_LIST(value) → array`

This is an alias for TO_ARRAY().

Type check functions

AQL also offers functions to check the data type of a value at runtime. The following type check functions are available. Each of these functions takes an argument of any data type and returns true if the value has the type that is checked for, and false otherwise.

The following type check functions are available:

- `IS_NULL(value) → bool` : Check whether *value* is a *null* value, also see HAS()
- `IS_BOOL(value) → bool` : Check whether *value* is a *boolean* value
- `IS_NUMBER(value) → bool` : Check whether *value* is a *numeric* value
- `IS_STRING(value) → bool` : Check whether *value* is a *string* value
- `IS_ARRAY(value) → bool` : Check whether *value* is an *array* value
- `IS_LIST(value) → bool` : This is an alias for *IS_ARRAY()*
- `IS_OBJECT(value) → bool` : Check whether *value* is an *object* / *document* value
- `IS_DOCUMENT(value) → bool` : This is an alias for *IS_OBJECT()*
- `IS_DATESTRING(value) → bool` : Check whether *value* is a string that can be used in a date function. This includes partial dates such as *"2015"* or *"2015-10"* and strings containing invalid dates such as *"2015-02-31"*. The function will return false for all non-string values, even if some of them may be usable in date functions.
- `TYPENAME(value) → typeName` : Return the data type name of *value*. The data type name can be either *"null"*, *"bool"*, *"number"*, *"string"*, *"array"* or *"object"*.

String functions

For string processing, AQL offers the following functions:

CHAR_LENGTH()

`CHAR_LENGTH(value) → length`

Return the number of characters in *value* (not byte length).

input	length
String	number of unicode characters
Number	number of unicode characters that represent the number
Array / Object	number of unicode characters from the resulting stringification
true	4
false	5
null	0

CONCAT()

`CONCAT(value1, value2, ... valueN) → str`

Concatenate the values passed as *value1* to *valueN*.

- **values** (any, *repeatable*): elements of arbitrary type (at least 1)
- returns **str** (string): a concatenation of the elements. *null* values are ignored.

```
CONCAT("foo", "bar", "baz") // "foobarbaz"
CONCAT(1, 2, 3) // "123"
CONCAT("foo", [5, 6], {bar: "baz"}) // "foo[5,6]{\"bar\":\"baz\"}"
```

`CONCAT(anyArray) → str`

If a single array is passed to *CONCAT()*, its members are concatenated.

- **anyArray** (array): array with elements of arbitrary type
- returns **str** (string): a concatenation of the array elements. *null* values are ignored.

```
CONCAT( [ "foo", "bar", "baz" ] ) // "foobarbaz"
CONCAT( [1, 2, 3] ) // "123"
```

CONCAT_SEPARATOR()

`CONCAT_SEPARATOR(separator, value1, value2, ... valueN) → joinedString`

Concatenate the strings passed as arguments *value1* to *valueN* using the *separator* string.

- **separator** (string): an arbitrary separator string
- **values** (string|array, *repeatable*): strings or arrays of strings as multiple arguments (at least 1)
- returns **joinedString** (string): a concatenated string of the elements, using *separator* as separator string. *null* values are ignored. Array value arguments are expanded automatically, and their individual members will be concatenated. Nested arrays will be expanded too, but with their elements separated by commas if they have more than a single element.

```
CONCAT_SEPARATOR(", ", "foo", "bar", "baz")
// "foo, bar, baz"

CONCAT_SEPARATOR(", ", [ "foo", "bar", "baz" ])
// "foo, bar, baz"
```

```
CONCAT_SEPARATOR(", ", [ "foo", [ "b", "a", "r" ], "baz" ])
// [ "foo, b,a,r, baz" ]

CONCAT_SEPARATOR("-", [1, 2, 3, null], [4, null, 5])
// "1-2-3-4-5"
```

CONTAINS()

`CONTAINS(text, search, returnIndex) → match`

Check whether the string *search* is contained in the string *text*. The string matching performed by *CONTAINS* is case-sensitive.

- **text** (string): the haystack
- **search** (string): the needle
- **returnIndex** (bool, *optional*): if set to *true*, the character position of the match is returned instead of a boolean. The default is *false*. The default is *false*.
- returns **match** (bool|number): by default, *true* is returned if *search* is contained in *text*, and *false* otherwise. With *returnIndex* set to *true*, the position of the first occurrence of *search* within *text* is returned (starting at offset 0), or *-1* if *search* is not contained in *text*.

```
CONTAINS("foobarbaz", "bar") // true
CONTAINS("foobarbaz", "horse") // false
CONTAINS("foobarbaz", "ba", true) // 3
CONTAINS("foobarbaz", "horse", true) // -1
```

COUNT()

This is an alias for LENGTH().

FIND_FIRST()

`FIND_FIRST(text, search, start, end) → position`

Return the position of the first occurrence of the string *search* inside the string *text*. Positions start at 0.

- **text** (string): the haystack
- **search** (string): the needle
- **start** (number, *optional*): limit the search to a subset of the text, beginning at *start*
- **end** (number, *optional*): limit the search to a subset of the text, ending at *end*
- returns **position** (number): the character position of the match. If *search* is not contained in *text*, -1 is returned.

```
FIND_FIRST("foobarbaz", "ba") // 3
FIND_FIRST("foobarbaz", "ba", 4) // 6
FIND_FIRST("foobarbaz", "ba", 0, 3) // -1
```

FIND_LAST()

`FIND_LAST(text, search, start, end) → position`

Return the position of the last occurrence of the string *search* inside the string *text*. Positions start at 0.

- **text** (string): the haystack
- **search** (string): the needle
- **start** (number, *optional*): limit the search to a subset of the text, beginning at *start*
- **end** (number, *optional*): limit the search to a subset of the text, ending at *end*
- returns **position** (number): the character position of the match. If *search* is not contained in *text*, -1 is returned.

```
FIND_LAST("foobarbaz", "ba") // 6
FIND_LAST("foobarbaz", "ba", 7) // -1
FIND_LAST("foobarbaz", "ba", 0, 4) // 3
```

JSON_PARSE()

`JSON_PARSE(text) → value`

Return an AQL value described by the JSON-encoded input string.

- **text** (string): the string to parse as JSON
- returns **value** (mixed): the value corresponding to the given JSON text. For input values that are no valid JSON strings, the function will return *null*.

```
JSON_PARSE("123") // 123
JSON_PARSE("[ true, false, 2 ]") // [ true, false, 2 ]
JSON_PARSE("\\\"abc\\\"") // "abc"
JSON_PARSE("{\\\"a\\\": 1}") // { a : 1 }
JSON_PARSE("abc") // null
```

JSON_STRINGIFY()

`JSON_STRINGIFY(value) → text`

Return a JSON string representation of the input value.

- **value** (mixed): the value to convert to a JSON string
- returns **text** (string): the JSON string representing *value*. For input values that cannot be converted to JSON, the function will return *null*.

```
JSON_STRINGIFY("1") // "1"
JSON_STRINGIFY("abc") // "\"abc\""
JSON_STRINGIFY("[1, 2, 3]") // "[1,2,3]"
```

LEFT()

`LEFT(value, length) → substring`

Return the *length* leftmost characters of the string *value*.

- **value** (string): a string
- **length** (number): how many characters to return
- returns **substring** (string): at most *length* characters of *value*, starting on the left-hand side of the string

```
LEFT("foobar", 3) // "foo"
LEFT("foobar", 10) // "foobar"
```

LENGTH()

`LENGTH(str) → length`

Determine the character length of a string.

- **str** (string): a string. If a number is passed, it will be casted to string first.
- returns **length** (number): the character length of *str* (not byte length)

```
LENGTH("foobar") // 6
LENGTH("    ") // 4
```

LENGTH() can also determine the number of elements in an array, the number of attribute keys of an object / document and the amount of documents in a collection.

LIKE()

`LIKE(text, search, caseInsensitive) → bool`

Check whether the pattern *search* is contained in the string *text*, using wildcard matching.

- **text** (string): the string to search in
- **search** (string): a search pattern that can contain the wildcard characters `%` (meaning any sequence of characters, including none)

and `_` (any single character). Literal *%* and *:* must be escaped with two backslashes (four in arangosh). *search* cannot be a variable or a document attribute. The actual value must be present at query parse time already.

- **caseInsensitive** (bool, *optional*): if set to *true*, the matching will be case-insensitive. The default is *false*.
- returns **bool** (bool): *true* if the pattern is contained in *text*, and *false* otherwise

```
LIKE("cart", "ca_t")   // true
LIKE("carrot", "ca_t") // false
LIKE("carrot", "ca%t") // true

LIKE("foo bar baz", "bar")   // false
LIKE("foo bar baz", "%bar%") // true
LIKE("bar", "%bar%")         // true

LIKE("FoO bAr BaZ", "fOo%bAz")       // false
LIKE("FoO bAr BaZ", "fOo%bAz", true) // true
```

LOWER()

`LOWER(value) → lowerCaseString`

Convert upper-case letters in *value* to their lower-case counterparts. All other characters are returned unchanged.

- **value** (string): a string
- returns **lowerCaseString** (string): *value* with upper-case characters converted to lower-case characters

LTRIM()

`LTRIM(value, chars) → strippedString`

Return the string *value* with whitespace stripped from the start only.

- **value** (string): a string
- **chars** (string, *optional*): override the characters that should be removed from the string. It defaults to `\r\n \t` (i.e. `0x0d` , `0x0a` , `0x20` and `0x09`).
- returns **strippedString** (string): *value* without *chars* at the left-hand side

```
LTRIM("foo bar") // "foo bar"
LTRIM("  foo bar  ") // "foo bar  "
LTRIM("--==[foo-bar]==--", "-=[]") // "foo-bar]==--"
```

MD5()

`MD5(text) → hash`

Calculate the MD5 checksum for *text* and return it in a hexadecimal string representation.

- **text** (string): a string
- returns **hash** (string): MD5 checksum as hex string

```
MD5("foobar") // "3858f62230ac3c915f300c664312c63f"
```

RANDOM_TOKEN()

`RANDOM_TOKEN(length) → randomString`

Generate a pseudo-random token string with the specified length. The algorithm for token generation should be treated as opaque.

- **length** (number): desired string length for the token. It must be greater than 0 and at most 65536.
- returns **randomString** (string): a generated token consisting of lowercase letters, uppercase letters and numbers

```
RANDOM_TOKEN(8) // "zGl09z42"
RANDOM_TOKEN(8) // "m9w50Ft9"
```

REGEX_TEST()

`REGEX_TEST(text, search, caseInsensitive) → bool`

Check whether the pattern *search* is contained in the string *text*, using regular expression matching.

- **text** (string): the string to search in
- **search** (string): a regular expression search pattern
- returns **bool** (bool): *true* if the pattern is contained in *text*, and *false* otherwise

The regular expression may consist of literal characters and the following characters and sequences:

- `.` – the dot matches any single character except line terminators. To include line terminators, use `[\s\S]` instead to simulate `.` with *DOTALL* flag.
- `\d` – matches a single digit, equivalent to `[0-9]`
- `\s` – matches a single whitespace character
- `\S` – matches a single non-whitespace character
- `\t` – matches a tab character
- `\r` – matches a carriage return
- `\n` – matches a line-feed character
- `[xyz]` – set of characters. matches any of the enclosed characters (i.e. *x*, *y* or *z* in this case
- `[^xyz]` – negated set of characters. matches any other character than the enclosed ones (i.e. anything but *x*, *y* or *z* in this case)
- `[x-z]` – range of characters. Matches any of the characters in the specified range, e.g. `[0-9A-F]` to match any character in *0123456789ABCDEF*
- `[^x-z]` – negated range of characters. Matches any other character than the ones specified in the range
- `(xyz)` – defines and matches a pattern group
- `(x|y)` – matches either *x* or *y*
- `^` – matches the beginning of the string (e.g. `^xyz`)
- `$` – matches the end of the string (e.g. `xyz$`)

Note that the characters `.`, `*`, `?`, `[`, `]`, `(`, `)`, `{`, `}`, `^`, and `$` have a special meaning in regular expressions and may need to be escaped using a backslash, which requires escaping itself (`\\`). A literal backslash needs to be escaped using another escaped backslash, i.e. `\\\\`. In arangosh, the amount of backslashes needs to be doubled.

Characters and sequences may optionally be repeated using the following quantifiers:

- `x*` – matches zero or more occurrences of *x*
- `x+` – matches one or more occurrences of *x*
- `x?` – matches one or zero occurrences of *x*
- `x{y}` – matches exactly *y* occurrences of *x*
- `x{y,z}` – matches between *y* and *z* occurrences of *x*
- `x{y,}` – matches at least *y* occurences of *x*

Note that `xyz+` matches *xyzzz*, but if you want to match *xyzxyz* instead, you need to define a pattern group by wrapping the subexpression in parentheses and place the quantifier right behind it: `(xyz)+`.

If the regular expression in *search* is invalid, a warning will be raised and the function will return *null*.

```
REGEX_TEST("the quick brown fox", "the.*fox") // true
REGEX_TEST("the quick brown fox", "^(a|the)\s+(quick|slow).*f.x$") // true
REGEX_TEST("the\nquick\nbrown\nfox", "^the(\n[a-w]+)+\nfox$") // true
```

REGEX_REPLACE()

`REGEX_REPLACE(text, search, replacement, caseInsensitive) → string`

Replace the pattern *search* with the string *replacement* in the string *text*, using regular expression matching.

- **text** (string): the string to search in
- **search** (string): a regular expression search pattern
- **replacement** (string): the string to replace the *search* pattern with
- returns **string** (string): the string *text* with the *search* regex pattern replaced with the *replacement* string wherever the pattern exists in *text*

For more details about the rules for characters and sequences refer REGEX_TEST().

If the regular expression in *search* is invalid, a warning will be raised and the function will return *null*.

```
REGEX_REPLACE("the quick brown fox", "the.*fox", "jumped over") // jumped over
REGEX_REPLACE("the quick brown fox", "o", "i") // the quick briwn fix
```

REVERSE()

`REVERSE(value) → reversedString`

Return the reverse of the string *str*.

- **value** (string): a string
- returns **reversedString** (string): a new string with the characters in reverse order

```
REVERSE("foobar") // "raboof"
REVERSE(""    ) // ""
```

RIGHT()

`RIGHT(value, length) → substring`

Return the *length* rightmost characters of the string *value*.

- **value** (string): a string
- **length** (number): how many characters to return
- returns **substring** (string): at most *length* characters of *value*, starting on the right-hand side of the string

```
RIGHT("foobar", 3) // "bar"
RIGHT("foobar", 10) // "foobar"
```

RTRIM()

`RTRIM(value, chars) → strippedString`

Return the string *value* with whitespace stripped from the end only.

- **value** (string): a string
- **chars** (string, *optional*): override the characters that should be removed from the string. It defaults to `\r\n \t` (i.e. `0x0d`, `0x0a`, `0x20` and `0x09`).
- returns **strippedString** (string): *value* without *chars* at the right-hand side

```
RTRIM("foo bar") // "foo bar"
RTRIM("  foo bar  ") // "  foo bar"
RTRIM("--==[foo-bar]==--", "-=[]") // "--==[foo-bar"
```

SHA1()

`SHA1(text) → hash`

Calculate the SHA1 checksum for *text* and returns it in a hexadecimal string representation.

- **text** (string): a string
- returns **hash** (string): SHA1 checksum as hex string

```
SHA1("foobar") // "8843d7f92416211de9ebb963ff4ce28125932878"
```

SPLIT()

`SPLIT(value, separator, limit) → strArray`

Split the given string *value* into a list of strings, using the *separator*.

- **value** (string): a string
- **separator** (string): either a string or a list of strings. If *separator* is an empty string, *value* will be split into a list of characters. If no *separator* is specified, *value* will be returned as array.
- **limit** (number, *optional*): limit the number of split values in the result. If no *limit* is given, the number of splits returned is not bounded.
- returns **strArray** (array): an array of strings

```
SPLIT( "foo-bar-baz", "-" ) // [ "foo", "bar", "baz" ]
SPLIT( "foo-bar-baz", "-", 1 ) // [ "foo", "bar-baz" ]
SPLIT( "foo, bar & baz", [ ", ", " & " ] ) // [ "foo", "bar", "baz" ]
```

SUBSTITUTE()

`SUBSTITUTE(value, search, replace, limit) → substitutedString`

Replace search values in the string *value*.

- **value** (string): a string
- **search** (string|array): if *search* is a string, all occurrences of *search* will be replaced in *value*. If *search* is an array of strings, each occurrence of a value contained in *search* will be replaced by the corresponding array element in *replace*. If *replace* has less list items than *search*, occurrences of unmapped *search* items will be replaced by an empty string.
- **replace** (string|array, *optional*): a replacement string, or an array of strings to replace the corresponding elements of *search* with. Can have less elements than *search* or be left out to remove matches. If *search* is an array but *replace* is a string, then all matches will be replaced with *replace*.
- **limit** (number, *optional*): cap the number of replacements to this value
- returns **substitutedString** (string): a new string with matches replaced (or removed)

```
SUBSTITUTE( "the quick brown foxx", "quick", "lazy" )
// "the lazy brown foxx"

SUBSTITUTE( "the quick brown foxx", [ "quick", "foxx" ], [ "slow", "dog" ] )
// "the slow brown dog"

SUBSTITUTE( "the quick brown foxx", [ "the", "foxx" ], [ "that", "dog" ], 1 )
//  "that quick brown foxx"

SUBSTITUTE( "the quick brown foxx", [ "the", "quick", "foxx" ], [ "A", "VOID!" ] )
// "A VOID! brown "

SUBSTITUTE( "the quick brown foxx", [ "quick", "foxx" ], "xx" )
// "the xx brown xx"
```

`SUBSTITUTE(value, mapping, limit) → substitutedString`

Alternatively, *search* and *replace* can be specified in a combined value.

- **value** (string): a string
- **mapping** (object): a lookup map with search strings as keys and replacement strings as values. Empty strings and *null* as values remove matches.
- **limit** (number, *optional*): cap the number of replacements to this value
- returns **substitutedString** (string): a new string with matches replaced (or removed)

```
SUBSTITUTE("the quick brown foxx", {
  "quick": "small",
  "brown": "slow",
  "foxx": "ant"
})
// "the small slow ant"

SUBSTITUTE("the quick brown foxx", {
  "quick": "",
  "brown": null,
  "foxx": "ant"
})
// "the   ant"
```

```
SUBSTITUTE("the quick brown foxx", {
  "quick": "small",
  "brown": "slow",
  "foxx": "ant"
}, 2)
// "the small slow foxx"
```

SUBSTRING()

`SUBSTRING(value, offset, length) → substring`

Return a substring of *value*.

- **value** (string): a string
- **offset** (number): start at *offset*, offsets start at position 0
- **length** (number, *optional*): at most *length* characters, omit to get the substring from *offset* to the end of the string
- returns **substring** (string): a substring of *value*

TRIM()

`TRIM(value, type) → strippedString`

Return the string *value* with whitespace stripped from the start and/or end.

The optional *type* parameter specifies from which parts of the string the whitespace is stripped. LTRIM() and RTRIM() are preferred however.

- **value** (string): a string
- **type** (number, *optional*): strip whitespace from the
  - 0 – start and end of the string
  - 1 – start of the string only
  - 2 – end of the string only The default is 0.

`TRIM(value, chars) → strippedString`

Return the string *value* with whitespace stripped from the start and end.

- **value** (string): a string
- **chars** (string, *optional*): override the characters that should be removed from the string. It defaults to `\r\n \t` (i.e. `0x0d`, `0x0a`, `0x20` and `0x09`).
- returns **strippedString** (string): *value* without *chars* on both sides

```
TRIM("foo bar") // "foo bar"
TRIM("  foo bar  ") // "foo bar"
TRIM("--==[foo-bar]==--", "-=[]") // "foo-bar"
TRIM("  foobar\t \r\n ") // "foobar"
TRIM(";foo;bar;baz, ", ",; ") // "foo;bar;baz"
```

UPPER()

`UPPER(value) → upperCaseString`

Convert lower-case letters in *value* to their upper-case counterparts. All other characters are returned unchanged.

- **value** (string): a string
- returns **upperCaseString** (string): *value* with lower-case characters converted to upper-case characters

Numeric functions

AQL offers some numeric functions for calculations. The following functions are supported:

ABS()

`ABS(value) → unsignedValue`

Return the absolute part of *value*.

- **value** (number): any number, positive or negative
- returns **unsignedValue** (number): the number without + or - sign

```
ABS(-5) // 5
ABS(+5) // 5
ABS(3.5) // 3.5
```

ACOS()

`ACOS(value) → num`

Return the arccosine of *value*.

- **value** (number): the input value
- returns **num** (number|null): the arccosine of *value*, or *null* if *value* is outside the valid range -1 and 1 (inclusive)

```
ACOS(-1) // 3.141592653589793
ACOS(0) // 1.5707963267948966
ACOS(1) // 0
ACOS(2) // null
```

ASIN()

`ASIN(value) → num`

Return the arcsine of *value*.

- **value** (number): the input value
- returns **num** (number|null): the arcsine of *value*, or *null* if *value* is outside the valid range -1 and 1 (inclusive)

```
ASIN(1) // 1.5707963267948966
ASIN(0) // 0
ASIN(-1) // -1.5707963267948966
ASIN(2) // null
```

ATAN()

`ATAN(value) → num`

Return the arctangent of *value*.

- **value** (number): the input value
- returns **num** (number): the arctangent of *value*

```
ATAN(-1) // -0.7853981633974483
ATAN(0) // 0
ATAN(10) // 1.4711276743037347
```

ATAN2()

`ATAN2(y, x) → num`

Return the arctangent of the quotient of *y* and *x*.

```
ATAN2(0, 0) // 0
ATAN2(1, 0) // 1.5707963267948966
ATAN2(1, 1) // 0.7853981633974483
ATAN2(-10, 20) // -0.4636476090008061
```

AVERAGE()

`AVERAGE(numArray) → mean`

Return the average (arithmetic mean) of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **mean** (number|null): the average value of *numArray*. If the array is empty or contains *null* values only, *null* will be returned.

```
AVERAGE( [5, 2, 9, 2] ) // 4.5
AVERAGE( [ -3, -5, 2 ] ) // -2
AVERAGE( [ 999, 80, 4, 4, 4, 3, 3, 3 ] ) // 137.5
```

CEIL()

`CEIL(value) → roundedValue`

Return the integer closest but not less than *value*.

- **value** (number): any number
- returns **roundedValue** (number): the value rounded to the ceiling

```
CEIL(2.49) // 3
CEIL(2.50) // 3
CEIL(-2.50) // -2
CEIL(-2.51) // -2
```

COS()

`COS(value) → num`

Return the cosine of *value*.

- **value** (number): the input value
- returns **num** (number): the cosine of *value*

```
COS(1) // 0.5403023058681398
COS(0) // 1
COS(-3.141592653589783) // -1
COS(RADIANS(45)) // 0.7071067811865476
```

DEGREES()

`DEGREES(rad) → num`

Return the angle converted from radians to degrees.

- **rad** (number): the input value
- returns **num** (number): the angle in degrees

```
DEGREES(0.7853981633974483) // 45
DEGREES(0) // 0
DEGREES(3.141592653589793) // 180
```

EXP()

`EXP(value) → num`

Return Euler's constant (2.71828...) raised to the power of *value*.

- **value** (number): the input value
- returns **num** (number): Euler's constant raised to the power of *value*

```
EXP(1) // 2.718281828459045
EXP(10) // 22026.46579480671
EXP(0) // 1
```

EXP2()

`EXP2(value) → num`

Return 2 raised to the power of *value*.

- **value** (number): the input value
- returns **num** (number): 2 raised to the power of *value*

```
EXP2(16) // 65536
EXP2(1) // 2
EXP2(0) // 1
```

FLOOR()

`FLOOR(value) → roundedValue`

Return the integer closest but not greater than *value*.

- **value** (number): any number
- returns **roundedValue** (number): the value rounded to the floor

```
FLOOR(2.49) // 2
FLOOR(2.50) // 2
FLOOR(-2.50) // -3
FLOOR(-2.51) // -3
```

LOG()

`LOG(value) → num`

Return the natural logarithm of *value*. The base is Euler's constant (2.71828...).

- **value** (number): the input value
- returns **num** (number|null): the natural logarithm of *value*, or *null* if *value* is equal or less than 0

```
LOG(2.718281828459045) // 1
LOG(10) // 2.302585092994046
LOG(0) // null
```

LOG2()

`LOG2(value) → num`

Return the base 2 logarithm of *value*.

- **value** (number): the input value
- returns **num** (number|null): the base 2 logarithm of *value*, or *null* if *value* is equal or less than 0

```
LOG2(1024) // 10
LOG2(8) // 3
LOG2(0) // null
```

LOG10()

`LOG10(value) → num`

Return the base 10 logarithm of *value*.

- **value** (number): the input value
- returns **num** (number): the base 10 logarithm of *value*, or *null* if *value* is equal or less than 0

```
LOG10(10000) // 10
LOG10(10) // 1
LOG10(0) // null
```

MAX()

`MAX(anyArray) → max`

Return the greatest element of *anyArray*. The array is not limited to numbers. Also see type and value order.

- **anyArray** (array): an array of numbers, *null* values are ignored
- returns **max** (any|null): the element with the greatest value. If the array is empty or contains *null* values only, the function will return *null*.

```
MAX( [5, 9, -2, null, 1] ) // 9
MAX( [ null, null ] ) // null
```

MEDIAN()

`MEDIAN(numArray) → median`

Return the median value of the values in *array*.

The array is sorted and the element in the middle is returned. If the array has an even length of elements, the two center-most elements are interpolated by calculating the average value (arithmetic mean).

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **median** (number|null): the median of *numArray*. If the array is empty or contains *null* values only, the function will return *null*.

```
MEDIAN( [ 1, 2, 3] ) // 2
MEDIAN( [ 1, 2, 3, 4 ] ) // 2.5
MEDIAN( [ 4, 2, 3, 1 ] ) // 2.5
MEDIAN( [ 999, 80, 4, 4, 4, 3, 3, 3 ] ) // 4
```

MIN()

`MIN(anyArray) → min`

Return the smallest element of *anyArray*. The array is not limited to numbers. Also see type and value order.

- **anyArray** (array): an array of numbers, *null* values are ignored
- returns **min** (any|null): the element with the smallest value. If the array is empty or contains *null* values only, the function will return *null*.

```
MIN( [5, 9, -2, null, 1] ) // -2
MIN( [ null, null ] ) // null
```

PERCENTILE()

`PERCENTILE(numArray, n, method) → percentile`

Return the *n*th percentile of the values in *numArray*.

- **numArray** (array): an array of numbers, *null* values are ignored
- **n** (number): must be between 0 (excluded) and 100 (included)
- **method** (string, *optional*): "rank" (default) or "interpolation"

- returns **percentile** (number|null): the *n*th percentile, or *null* if the array is empty or only *null* values are contained in it or the percentile cannot be calculated

```
PERCENTILE( [1, 2, 3, 4], 50 ) // 2
PERCENTILE( [1, 2, 3, 4], 50, "rank" ) // 2
PERCENTILE( [1, 2, 3, 4], 50, "interpolation" ) // 2.5
```

PI()

`PI() → pi`

Return pi.

- returns **pi** (number): the first few significant digits of pi (3.141592653589793)

```
PI() // 3.141592653589793
```

POW()

`POW(base, exp) → num`

Return the *base* to the exponent *exp*.

- **base** (number): the base value
- **exp** (number): the exponent value
- returns **num** (number): the exponentiated value

```
POW( 2, 4 ) // 16
POW( 5, -1 ) // 0.2
POW( 5, 0 ) // 1
```

RADIANS()

`RADIANS(deg) → num`

Return the angle converted from degrees to radians.

- **deg** (number): the input value
- returns **num** (number): the angle in radians

```
RADIANS(180) // 3.141592653589793
RADIANS(90) // 1.5707963267948966
RADIANS(0) // 0
```

RAND()

`RAND() → randomNumber`

Return a pseudo-random number between 0 and 1.

- returns **randomNumber** (number): a number greater than 0 and less than 1

```
RAND() // 0.3503170117504508
RAND() // 0.6138226173882478
```

Complex example:

```
LET coinFlips = (
    FOR i IN 1..100000
    RETURN RAND() > 0.5 ? "heads" : "tails"
)
RETURN MERGE(
    FOR flip IN coinFlips
        COLLECT f = flip WITH COUNT INTO count
```

```
        RETURN { [f]: count }
)
```

Result:

```
[
  {
    "heads": 49902,
    "tails": 50098
  }
]
```

RANGE()

`RANGE(start, stop, step) → numArray`

Return an array of numbers in the specified range, optionally with increments other than 1.

For integer ranges, use the range operator instead for better performance.

- **start** (number): the value to start the range at (inclusive)
- **stop** (number): the value to end the range with (inclusive)
- **step** (number, *optional*): how much to increment in every step, the default is *1.0*
- returns **numArray** (array): all numbers in the range as array

```
RANGE(1, 4) // [ 1, 2, 3, 4 ]
RANGE(1, 4, 2) // [ 1, 3 ]
RANGE(1, 4, 3) // [ 1, 4 ]
RANGE(1.5, 2.5) // [ 1.5, 2.5 ]
RANGE(1.5, 2.5, 0.5) // [ 1.5, 2, 2.5 ]
RANGE(-0.75, 1.1, 0.5) // [ -0.75, -0.25, 0.25, 0.75 ]
```

ROUND()

`ROUND(value) → roundedValue`

Return the integer closest to *value*.

- **value** (number): any number
- returns **roundedValue** (number): the value rounded to the closest integer

```
ROUND(2.49) // 2
ROUND(2.50) // 3
ROUND(-2.50) // -2
ROUND(-2.51) // -3
```

Rounding towards zero, also known as *trunc()* in C/C++, can be achieved with a combination of the ternary operator, CEIL() and FLOOR():

```
LET rounded = value >= 0 ? FLOOR(value) : CEIL(value)
```

SIN()

`SIN(value) → num`

Return the sine of *value*.

- **value** (number): the input value
- returns **num** (number): the sine of *value*

```
SIN(3.141592653589783 / 2) // 1
SIN(0) // 0
SIN(-3.141592653589783 / 2) // -1
SIN(RADIANS(270)) // -1
```

SQRT()

`SQRT(value) → squareRoot`

Return the square root of *value*.

- **value** (number): a number
- returns **squareRoot** (number): the square root of *value*

```
SQRT(9) // 3
SQRT(2) // 1.4142135623730951
```

Other roots can be calculated with POW() like `POW(value, 1/n)`:

```
// 4th root of 8*8*8*8 = 4096
POW(4096, 1/4) // 8

// cube root of 3*3*3 = 27
POW(27, 1/3) // 3

// square root of 3*3 = 9
POW(9, 1/2) // 3
```

STDDEV_POPULATION()

`STDDEV_POPULATION(numArray) → num`

Return the population standard deviation of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **num** (number|null): the population standard deviation of *numArray*. If the array is empty or only *null* values are contained in the array, *null* will be returned.

```
STDDEV_POPULATION( [ 1, 3, 6, 5, 2 ] ) // 1.854723699099141
```

STDDEV_SAMPLE()

`STDDEV_SAMPLE(numArray) → num`

Return the sample standard deviation of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **num** (number|null): the sample standard deviation of *numArray*. If the array is empty or only *null* values are contained in the array, *null* will be returned.

```
STDDEV_SAMPLE( [ 1, 3, 6, 5, 2 ] ) // 2.0736441353327724
```

SUM()

`SUM(numArray) → sum`

Return the sum of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **sum** (number): the total of all values in *numArray*. If the array is empty or only *null* values are contained in the array, *0* will be returned.

```
SUM( [1, 2, 3, 4] ) // 10
SUM( [null, -5, 6] ) // 1
SUM( [ ] ) // 0
```

TAN()

`TAN(value) → num`

Return the tangent of *value*.

- **value** (number): the input value
- returns **num** (number): the tangent of *value*

```
TAN(10) // 0.6483608274590866
TAN(5) // -3.380515006246586
TAN(0) // 0
```

VARIANCE_POPULATION()

`VARIANCE_POPULATION(numArray) → num`

Return the population variance of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **num** (number|null): the population variance of *numArray*. If the array is empty or only *null* values are contained in the array, *null* will be returned.

```
VARIANCE_POPULATION( [ 1, 3, 6, 5, 2 ] ) // 3.4400000000000004
```

VARIANCE_SAMPLE()

`VARIANCE_SAMPLE(array) → num`

Return the sample variance of the values in *array*.

- **numArray** (array): an array of numbers, *null* values are ignored
- returns **num** (number|null): the sample variance of *numArray*. If the array is empty or only *null* values are contained in the array, *null* will be returned.

```
VARIANCE_SAMPLE( [ 1, 3, 6, 5, 2 ] ) // 4.300000000000001
```

Date functions

AQL offers functionality to work with dates. Dates are no data types of their own in AQL (neither are they in JSON, which is usually used as format to ship data into and out of ArangoDB). Instead, dates in AQL are typically represented by either numbers (timestamps) or strings.

All functions that require dates as arguments accept the following input values:

- numeric timestamps, indicating the number of milliseconds elapsed since the UNIX epoch (i.e. January 1st 1970 00:00:00.000 UTC). An example timestamp value is *1399472349522*, which translates to *2014-05-07T14:19:09.522Z*.
- date time strings in formats *YYYY-MM-DDTHH:MM:SS.MMM*, *YYYY-MM-DD HH:MM:SS.MMM*, or *YYYY-MM-DD* Milliseconds are always optional. A timezone difference may optionally be added at the end of the string, with the hours and minutes that need to be added or subtracted to the date time value. For example, *2014-05-07T14:19:09+01:00* can be used to specify a one hour offset, and *2014-05-07T14:19:09+07:30* can be specified for seven and half hours offset. Negative offsets are also possible. Alternatively to an offset, a *Z* can be used to indicate UTC / Zulu time.

  An example value is *2014-05-07T14:19:09.522Z* meaning May 7th 2014, 14:19:09 and 522 milliseconds, UTC / Zulu time. Another example value without time component is *2014-05-07Z*.

  Please note that if no timezone offset is specified in a date string, ArangoDB will assume UTC time automatically. This is done to ensure portability of queries across servers with different timezone settings, and because timestamps will always be UTC-based.

```
DATE_HOUR( 2 * 60 * 60 * 1000 ) // 2
DATE_HOUR("1970-01-01T02:00:00") // 2
```

You are free to store age determinations of specimens, incomplete or fuzzy dates and the like in different, more appropriate ways of course. AQL's date functions will most certainly not be of any help for such dates, but you can still use language constructs like SORT (which also supports sorting of arrays) and indexes like skiplists.

Current date and time

DATE_NOW()

```
DATE_NOW() → timestamp
```

Get the current date time as numeric timestamp.

- returns **timestamp** (number): the current time as a timestamp. The return value has millisecond precision. To convert the return value to seconds, divide it by 1000.

Note that this function is evaluated on every invocation and may return different values when invoked multiple times in the same query. Assign it to a variable to use the exact same timestamp multiple times.

Conversion

DATE_TIMESTAMP() and *DATE_ISO8601()* can be used to convert ISO 8601 date time strings to numeric timestamps and numeric timestamps to ISO 8601 date time strings.

Both also support individual date components as separate function arguments, in the following order:

- year
- month
- day
- hour
- minute
- second
- millisecond

All components following *day* are optional and can be omitted. Note that no timezone offsets can be specified when using separate date components, and UTC / Zulu time will be used.

The following calls to *DATE_TIMESTAMP()* are equivalent and will all return *1399472349522*:

```
DATE_TIMESTAMP("2014-05-07T14:19:09.522")
DATE_TIMESTAMP("2014-05-07T14:19:09.522Z")
DATE_TIMESTAMP("2014-05-07 14:19:09.522")
DATE_TIMESTAMP("2014-05-07 14:19:09.522Z")
DATE_TIMESTAMP(2014, 5, 7, 14, 19, 9, 522)
DATE_TIMESTAMP(1399472349522)
```

The same is true for calls to *DATE_ISO8601()* that also accepts variable input formats:

```
DATE_ISO8601("2014-05-07T14:19:09.522Z")
DATE_ISO8601("2014-05-07 14:19:09.522Z")
DATE_ISO8601(2014, 5, 7, 14, 19, 9, 522)
DATE_ISO8601(1399472349522)
```

The above functions are all equivalent and will return *"2014-05-07T14:19:09.522Z"*.

DATE_ISO8601()

`DATE_ISO8601(date) → dateString`

Return an ISO 8601 date time string from *date*. The date time string will always use UTC / Zulu time, indicated by the *Z* at its end.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **dateString**: date and time expressed according to ISO 8601, in Zulu time

`DATE_ISO8601(year, month, day, hour, minute, second, millisecond) → dateString`

Return a ISO 8601 date time string from *date*, but allows to specify the individual date components separately. All parameters after *day* are optional.

- **year** (number): typically in the range 0..9999, e.g. *2017*
- **month** (number): 1..12 for January through December (unlike JavaScript, which uses the slightly confusing range 0..11)
- **day** (number): 1..31 (upper bound depends on number of days in month)
- **hour** (number, *optional*): 0..23
- **minute** (number, *optional*): 0..59
- **second** (number, *optional*): 0..59
- **milliseconds** (number, *optional*): 0..999
- returns **dateString**: date and time expressed according to ISO 8601, in Zulu time

DATE_TIMESTAMP()

`DATE_TIMESTAMP(date) → timestamp`

Create a UTC timestamp value from *date*. The return value has millisecond precision. To convert the return value to seconds, divide it by 1000.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **timestamp** (number): numeric timestamp

`DATE_TIMESTAMP(year, month, day, hour, minute, second, millisecond) → timestamp`

Create a UTC timestamp value, but allows to specify the individual date components separately. All parameters after *day* are optional.

- **year** (number): typically in the range 0..9999, e.g. *2017*
- **month** (number): 1..12 for January through December (unlike JavaScript, which uses the slightly confusing range 0..11)
- **day** (number): 1..31 (upper bound depends on number of days in month)
- **hour** (number, *optional*): 0..23
- **minute** (number, *optional*): 0..59
- **second** (number, *optional*): 0..59
- **milliseconds** (number, *optional*): 0..999

- returns **timestamp** (number): numeric timestamp

Negative values are not allowed, result in *null* and cause a warning. Values greater than the upper range bound overflow to the larger components (e.g. an hour of 26 is automatically turned into an additional day and two hours):

```
DATE_TIMESTAMP(2016, 12, -1) // returns null and issues a warning
DATE_TIMESTAMP(2016, 2, 32) // returns 1456963200000, which is March 3rd, 2016
DATE_TIMESTAMP(1970, 1, 1, 26) // returns 93600000, which is January 2nd, 1970, at 2 a.m.
```

IS_DATESTRING()

`IS_DATESTRING(value) → bool`

Check if an arbitrary string is suitable for interpretation as date time string.

- **value** (string): an arbitrary string
- returns **bool** (bool): *true* if *value* is a string that can be used in a date function. This includes partial dates such as *2015* or *2015-10* and strings containing invalid dates such as *2015-02-31*. The function will return *false* for all non-string values, even if some of them may be usable in date functions.

Processing

DATE_DAYOFWEEK()

`DATE_DAYOFWEEK(date) → weekdayNumber`

Return the weekday number of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **weekdayNumber** (number): 0..6 as follows:
  - 0 – Sunday
  - 1 – Monday
  - 2 – Tuesday
  - 3 – Wednesday
  - 4 – Thursday
  - 5 – Friday
  - 6 – Saturday

DATE_YEAR()

`DATE_YEAR(date) → year`

Return the year of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **year** (number): the year part of *date* as a number

DATE_MONTH()

`DATE_MONTH(date) → month`

Return the month of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **month** (number): the month part of *date* as a number

DATE_DAY()

`DATE_DAY(date) → day`

Return the day of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string

- returns **day** (number): the day part of *date* as a number

DATE_HOUR()

Return the hour of *date*.

```
DATE_HOUR(date) → hour
```

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **hour** (number): the hour part of *date* as a number

DATE_MINUTE()

```
DATE_MINUTE(date) → minute
```

Return the minute of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **minute** (number): the minute part of *date* as a number

DATE_SECOND()

```
DATE_SECOND(date) → second
```

Return the second of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **second** (number): the seconds part of *date* as a number

DATE_MILLISECOND()

```
DATE_MILLISECOND(date) → millisecond
```

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **millisecond** (number): the milliseconds part of *date* as a number

DATE_DAYOFYEAR()

```
DATE_DAYOFYEAR(date) → dayOfYear
```

Return the day of year of *date*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **dayOfYear** (number): the day of year number of *date*. The return values range from 1 to 365, or 366 in a leap year respectively.

DATE_ISOWEEK()

```
DATE_ISOWEEK(date) → weekDate
```

Return the week date of *date* according to ISO 8601.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **weekDate** (number): the ISO week date of *date*. The return values range from 1 to 53. Monday is considered the first day of the week. There are no fractional weeks, thus the last days in December may belong to the first week of the next year, and the first days in January may be part of the previous year's last week.

DATE_LEAPYEAR()

```
DATE_LEAPYEAR(date) → leapYear
```

Return whether *date* is in a leap year.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **leapYear** (bool): *true* if *date* is in a leap year, *false* otherwise

DATE_QUARTER()

```
DATE_QUARTER(date) → quarter
```

Return which quarter *date* belongs to.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **quarter** (number): the quarter of the given date (1-based):
  - 1 – January, February, March
  - 2 – April, May, June
  - 3 – July, August, September
  - 4 – October, November, December

DATE_DAYS_IN_MONTH()

Return the number of days in the month of *date*.

```
DATE_DAYS_IN_MONTH(date) → daysInMonth
```

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- returns **daysInMonth** (number): the number of days in *date*'s month (28..31)

DATE_FORMAT()

```
DATE_FORMAT(date, format) → str
```

Format a date according to the given format string.

- **date** (string|number): a date string or timestamp
- **format** (string): a format string, see below
- returns **str** (string): a formatted date string

format supports the following placeholders (case-insensitive):

- %t – timestamp, in milliseconds since midnight 1970-01-01
- %z – ISO date (0000-00-00T00:00:00.000Z)
- %w – day of week (0..6)
- %y – year (0..9999)
- %yy – year (00..99), abbreviated (last two digits)
- %yyyy – year (0000..9999), padded to length of 4
- %yyyyyy – year (-009999 .. +009999), with sign prefix and padded to length of 6
- %m – month (1..12)
- %mm – month (01..12), padded to length of 2
- %d – day (1..31)
- %dd – day (01..31), padded to length of 2
- %h – hour (0..23)
- %hh – hour (00..23), padded to length of 2
- %i – minute (0..59)
- %ii – minute (00..59), padded to length of 2
- %s – second (0..59)
- %ss – second (00..59), padded to length of 2
- %f – millisecond (0..999)
- %fff – millisecond (000..999), padded to length of 3
- %x – day of year (1..366)
- %xxx – day of year (001..366), padded to length of 3
- %k – ISO week date (1..53)
- %kk – ISO week date (01..53), padded to length of 2
- %l – leap year (0 or 1)
- %q – quarter (1..4)
- %a – days in month (28..31)
- %mmm – abbreviated English name of month (Jan..Dec)

- %mmmm – English name of month (January..December)
- %www – abbreviated English name of weekday (Sun..Sat)
- %wwww – English name of weekday (Sunday..Saturday)
- %& – special escape sequence for rare occasions
- %% – literal %
- % – ignored

`%yyyy` does not enforce a length of 4 for years before 0 and past 9999. The same format as for `%yyyyyy` will be used instead. `%yy` preserves the sign for negative years and may thus return 3 characters in total.

Single `%` characters will be ignored. Use `%%` for a literal `%`. To resolve ambiguities like in `%mmonth` (unpadded month number + the string "month") between `%mm` + "onth" and `%m` + "month", use the escape sequence `%&` : `%m%&month`.

Note that *DATE_FORMAT()* is a rather costly operation and may not be suitable for large datasets (like over 1 million dates). If possible, avoid formatting dates on server-side and leave it up to the client to do so. This function should only be used for special date comparisons or to store the formatted dates in the database. For better performance, use the primitive `DATE_*()` functions together with `CONCAT()` if possible.

Examples:

```
DATE_FORMAT(DATE_NOW(), "%q/%yyyy") // quarter and year (e.g. "3/2015")
DATE_FORMAT(DATE_NOW(), "%dd.%mm.%yyyy %hh:%ii:%ss,%fff") // e.g. "18.09.2015 15:30:49,374"
DATE_FORMAT("1969", "Summer of '%yy") // "Summer of '69"
DATE_FORMAT("2016", "%%l = %l") // "%l = 1" (2016 is a leap year)
DATE_FORMAT("2016-03-01", "%xxx%") // "063", trailing % ignored
```

Comparison and calculation

DATE_ADD()

`DATE_ADD(date, amount, unit) → isoDate`

Add *amount* given in *unit* to *date* and return the calculated date.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- **amount** (number|string): number of *unit*s to add (positive value) or subtract (negative value). It is recommended to use positive values only, and use DATE_SUBTRACT() for subtractions instead.
- **unit** (string): either of the following to specify the time unit to add or subtract (case-insensitive):
  - y, year, years
  - m, month, months
  - w, week, weeks
  - d, day, days
  - h, hour, hours
  - i, minute, minutes
  - s, second, seconds
  - f, millisecond, milliseconds
- returns **isoDate** (string): the calculated ISO 8601 date time string

```
DATE_ADD(DATE_NOW(), -1, "day") // yesterday; also see DATE_SUBTRACT()
DATE_ADD(DATE_NOW(), 3, "months") // in three months
DATE_ADD(DATE_ADD("2015-04-01", 5, "years"), 1, "month") // May 1st 2020
DATE_ADD("2015-04-01", 12*5 + 1, "months") // also May 1st 2020
DATE_ADD(DATE_TIMESTAMP(DATE_YEAR(DATE_NOW()), 12, 24), -4, "years") // Christmas four years ago
DATE_ADD(DATE_ADD("2016-02", "month", 1), -1, "day") // last day of February (29th, because 2016 is a leap year!)
```

`DATE_ADD(date, isoDuration) → isoDate`

You may also pass an ISO duration string as *amount* and leave out *unit*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- **isoDuration** (string): an ISO 8601 duration string to add to *date*, see below
- returns **isoDate** (string): the calculated ISO 8601 date time string

The format is `P_Y_M_W_DT_H_M_._S`, where underscores stand for digits and letters for time intervals - except for the separators `P` (period) and `T` (time). The meaning of the other letters are:

- Y – years
- M – months (if before T)
- W – weeks
- D – days
- H – hours
- M – minutes (if after T)
- S – seconds (optionally with 3 decimal places for milliseconds)

The string must be prefixed by a `P`. A separating `T` is only required if `H`, `M` and/or `S` are specified. You only need to specify the needed pairs of letters and numbers.

```
DATE_ADD(DATE_NOW(), "P1Y") // add 1 year
DATE_ADD(DATE_NOW(), "P3M2W") // add 3 months and 2 weeks
DATE_ADD(DATE_NOW(), "P5DT26H") // add 5 days and 26 hours (=6 days and 2 hours)
DATE_ADD("2000-01-01", "PT4H") // add 4 hours
DATE_ADD("2000-01-01", "PT30M44.4S" // add 30 minutes, 44 seconds and 400 ms
DATE_ADD("2000-01-01", "P1Y2M3W4DT5H6M7.89S" // add a bit of everything
```

DATE_SUBTRACT()

`DATE_SUBTRACT(date, amount, unit) → isoDate`

Subtract *amount* given in *unit* from *date* and return the calculated date.

It works the same as DATE_ADD(), except that it subtracts. It is equivalent to calling *DATE_ADD()* with a negative amount, except that *DATE_SUBTRACT()* can also subtract ISO durations. Note that negative ISO durations are not supported (i.e. starting with `-P`, like `-P1Y`).

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- **amount** (number|string): number of *unit*s to subtract (positive value) or add (negative value). It is recommended to use positive values only, and use DATE_ADD() for additions instead.
- **unit** (string): either of the following to specify the time unit to add or subtract (case-insensitive):
  - y, year, years
  - m, month, months
  - w, week, weeks
  - d, day, days
  - h, hour, hours
  - i, minute, minutes
  - s, second, seconds
  - f, millisecond, milliseconds
- returns **isoDate** (string): the calculated ISO 8601 date time string

`DATE_SUBTRACT(date, isoDuration) → isoDate`

You may also pass an ISO duration string as *amount* and leave out *unit*.

- **date** (number|string): numeric timestamp or ISO 8601 date time string
- **isoDuration** (string): an ISO 8601 duration string to subtract from *date*, see below
- returns **isoDate** (string): the calculated ISO 8601 date time string

The format is `P_Y_M_W_DT_H_M_._S`, where underscores stand for digits and letters for time intervals - except for the separators `P` (period) and `T` (time). The meaning of the other letters are:

- Y – years
- M – months (if before T)
- W – weeks
- D – days
- H – hours
- M – minutes (if after T)
- S – seconds (optionally with 3 decimal places for milliseconds)

The string must be prefixed by a `P`. A separating `T` is only required if `H`, `M` and/or `S` are specified. You only need to specify the needed pairs of letters and numbers.

```
DATE_SUBTRACT(DATE_NOW(), 1, "day") // yesterday
DATE_SUBTRACT(DATE_TIMESTAMP(DATE_YEAR(DATE_NOW()), 12, 24), 4, "years") // Christmas four years ago
DATE_SUBTRACT(DATE_ADD("2016-02", "month", 1), 1, "day") // last day of February (29th, because 2016 is a leap year!)
DATE_SUBTRACT(DATE_NOW(), "P4D") // four days ago
DATE_SUBTRACT(DATE_NOW(), "PT1H3M") // 1 hour and 30 minutes ago
```

DATE_DIFF()

`DATE_DIFF(date1, date2, unit, asFloat) → diff`

Calculate the difference between two dates in given time *unit*, optionally with decimal places.

- **date1** (number|string): numeric timestamp or ISO 8601 date time string
- **date2** (number|string): numeric timestamp or ISO 8601 date time string
- **unit** (string): either of the following to specify the time unit to return the difference in (case-insensitive):
  - y, year, years
  - m, month, months
  - w, week, weeks
  - d, day, days
  - h, hour, hours
  - i, minute, minutes
  - s, second, seconds
  - f, millisecond, milliseconds
- **asFloat** (boolean, *optional*): if set to *true*, decimal places will be preserved in the result. The default is *false* and an integer is returned.
- returns **diff** (number): the calculated difference as number in *unit*. The value will be negative if *date2* is before *date1*.

DATE_COMPARE()

`DATE_COMPARE(date1, date2, unitRangeStart, unitRangeEnd) → bool`

Check if two partial dates match.

- **date1** (number|string): numeric timestamp or ISO 8601 date time string
- **date2** (number|string): numeric timestamp or ISO 8601 date time string
- **unitRangeStart** (string): unit to start from, see below
- **unitRangeEnd** (string, *optional*): unit to end with, leave out to only compare the component as specified by *unitRangeStart*. An error is raised if *unitRangeEnd* is a unit before *unitRangeStart*.
- returns **bool** (bool): *true* if the dates match, *false* otherwise

The parts to compare are defined by a range of time units. The full range is: years, months, days, hours, minutes, seconds, milliseconds (in this order).

All components of *date1* and *date2* as specified by the range will be compared. You can refer to the units as:

- y, year, years
- m, month, months
- d, day, days
- h, hour, hours
- i, minute, minutes
- s, second, seconds
- f, millisecond, milliseconds

```
// Compare months and days, true on birthdays if you're born on 4th of April
DATE_COMPARE("1985-04-04", DATE_NOW(), "months", "days")

// Will only match on one day if the current year is a leap year!
// You may want to add or subtract one day from date1 to match every year.
DATE_COMPARE("1984-02-29", DATE_NOW(), "months", days")
```

```
// compare years, months and days (true, because it's the same day)
DATE_COMPARE("2001-01-01T15:30:45.678Z", "2001-01-01T08:08:08.008Z", "years", "days")
```

You can directly compare ISO date **strings** if you want to find dates before or after a certain date, or in between two dates (`>=` , `>` , `<` , `<=`). No special date function is required. Equality tests (`==` and `!=`) will only match the exact same date and time however. You may use `SUBSTRING()` to compare partial date strings, `DATE_COMPARE()` is basically a convenience function for that. However, neither is really required to limit a search to a certain day as demonstrated here:

```
FOR doc IN coll
    FILTER doc.date >= "2015-05-15" AND doc.date < "2015-05-16"
    RETURN doc
```

Every ISO date on that day is greater than or equal to `2015-05-15` in a string comparison (e.g. `2015-05-15T11:30:00.000Z`). Dates before `2015-05-15` are smaller and therefore filtered out by the first condition. Every date past `2015-05-15` is greater than this date in a string comparison, and therefore filtered out by the second condition. The result is that the time components in the dates you compare with are "ignored". The query will return every document with *date* ranging from `2015-05-15T00:00:00.000Z` to `2015-05-15T23:99:99.999Z` . It would also include `2015-05-15T24:00:00.000Z` , but that date is actually `2015-05-16T00:00:00.000Z` and can only occur if inserted manually (you may want to pass dates through DATE_ISO8601() to ensure a correct date representation).

Leap days in leap years (29th of February) must be always handled manually, if you require so (e.g. birthday checks):

```
LET today = DATE_NOW()
LET noLeapYear = NOT DATE_LEAPYEAR(today)

FOR user IN users
    LET birthday = noLeapYear AND
                   DATE_MONTH(user.birthday) == 2 AND
                   DATE_DAY(user.birthday) == 29
                   ? DATE_SUBTRACT(user.birthday, 1, "day") /* treat like 28th in non-leap years */
                   : user.birthday
    FILTER DATE_COMPARE(today, birthday, "month", "day")
    /* includes leaplings on the 28th of February in non-leap years,
     * but excludes them in leap years which do have a 29th February.
     * Replace DATE_SUBTRACT() by DATE_ADD() to include them on the 1st of March
     * in non-leap years instead (depends on local jurisdiction).
     */
    RETURN user
```

Working with dates and indices

There are two recommended ways to store timestamps in ArangoDB:

- as string with ISO 8601 UTC timestamp
- as Epoch number

The sort order of both is identical due to the sort properties of ISO date strings. You can't mix both types, numbers and strings, in a single attribute however.

You can use skiplist indices with both date types. When chosing string representations, you can work with string comparisons (less than, greater than etc.) to express time ranges in your queries while still utilizing skiplist indices:

```
arangosh> db._create("exampleTime");
arangosh> var timestamps = ["2014-05-07T14:19:09.522","2014-05-07T21:19:09.522","2014-05-
08T04:19:09.522","2014-05-08T11:19:09.522","2014-05-08T18:19:09.522"];
arangosh> for (i = 0; i < 5; i++) db.exampleTime.save({value:i, ts: timestamps[i]})
arangosh> db._query("FOR d IN exampleTime FILTER d.ts > '2014-05-07T14:19:09.522' and d.ts
< '2014-05-08T18:19:09.522' RETURN d").toArray()
```

show execution results

The first and the last timestamp in the array are excluded from the result by the `FILTER` .

Limitations

Note that dates before the year 1583 aren't allowed by the ISO 8601 standard by default, because they lie before the official introduction of the Gregorian calendar and may thus be incorrect or invalid. All AQL date functions apply the same rules to every date according to the Gregorian calendar system, even if inappropriate. That does not constitute a problem, unless you deal with dates prior to 1583 and especially years before Christ. The standard allows negative years, but requires special treatment of positive years too, if negative years are used (e.g. `+002015-05-15` and `-000753-01-01`). This is rarely used however, and AQL does not use the 7-character version for years between 0 and 9999 in ISO strings. Keep in mind that they can't be properly compared to dates outside that range. Sorting of negative dates does not result in a meaningful order, with years longer ago last, but months, days and the time components in otherwise correct order.

Leap seconds are ignored, just as they are in JavaScript as per ECMAScript Language Specifications.

Array functions

AQL provides functions for higher-level array manipulation. Also see the numeric functions for functions that work on number arrays. If you want to concatenate the elements of an array equivalent to `join()` in JavaScript, see CONCAT() and CONCAT_SEPARATOR() in the string functions chapter.

Apart from that, AQL also offers several language constructs:

- simple array access of individual elements,
- array operators for array expansion and contraction, optionally with inline filter, limit and projection,
- array comparison operators to compare each element in an array to a value or the elements of another array,
- loop-based operations using FOR, SORT, LIMIT, as well as COLLECT for grouping, which also offers efficient aggregation.

APPEND()

`APPEND(anyArray, values, unique) → newArray`

Add all elements of an array to another array. All values are added at the end of the array (right side).

- **anyArray** (array): array with elements of arbitrary type
- **values** (array): array, whose elements shall be added to *anyArray*
- **unique** (bool, *optional*): if set to *true*, only those *values* will be added that are not already contained in *anyArray*. The default is *false*.
- returns **newArray** (array): the modified array

```
APPEND([ 1, 2, 3 ], [ 5, 6, 9 ])
// [ 1, 2, 3, 5, 6, 9 ]

APPEND([ 1, 2, 3 ], [ 3, 4, 5, 2, 9 ], true)
// [ 1, 2, 3, 4, 5, 9 ]
```

COUNT()

This is an alias for LENGTH().

FIRST()

`FIRST(anyArray) → firstElement`

Get the first element of an array. It is the same as `anyArray[0]` .

- **anyArray** (array): array with elements of arbitrary type
- returns **firstElement** (any|null): the first element of *anyArray*, or *null* if the array is empty.

FLATTEN()

`FLATTEN(anyArray, depth) → flatArray`

Turn an array of arrays into a flat array. All array elements in *array* will be expanded in the result array. Non-array elements are added as they are. The function will recurse into sub-arrays up to the specified depth. Duplicates will not be removed.

- **array** (array): array with elements of arbitrary type, including nested arrays
- **depth** (number, *optional*): flatten up to this many levels, the default is 1
- returns **flatArray** (array): a flattened array

```
FLATTEN( [ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ] ] )
// [ 1, 2, 3, 4, 5, 6, 7, 8, [ 9, 10 ] ]
```

To fully flatten the example array, use a *depth* of 2:

```
FLATTEN( [ 1, 2, [ 3, 4 ], 5, [ 6, 7 ], [ 8, [ 9, 10 ] ] ], 2 )
```

```
// [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

INTERSECTION()

`INTERSECTION(array1, array2, ... arrayN) → newArray`

Return the intersection of all arrays specified. The result is an array of values that occur in all arguments.

- **arrays** (array, *repeatable*): an arbitrary number of arrays as multiple arguments (at least 2)
- returns **newArray** (array): a single array with only the elements, which exist in all provided arrays. The element order is random. Duplicates are removed.

LAST()

`LAST(anyArray) → lastElement`

Get the last element of an array. It is the same as `anyArray[-1]`.

- **anyArray** (array): array with elements of arbitrary type
- returns **lastElement** (any|null): the last element of *anyArray* or *null* if the array is empty.

LENGTH()

`LENGTH(anyArray) → length`

Determine the number of elements in an array.

- **anyArray** (array): array with elements of arbitrary type
- returns **length** (number): the number of array elements in *anyArray*.

LENGTH() can also determine the number of attribute keys of an object / document, the amount of documents in a collection and the character length of a string.

input	length
String	number of unicode characters
Number	number of unicode characters that represent the number
Array	number of elements
Object	number of first level elements
true	1
false	0
null	0

MINUS()

`MINUS(array1, array2, ... arrayN) → newArray`

Return the difference of all arrays specified.

- **arrays** (array, *repeatable*): an arbitrary number of arrays as multiple arguments (at least 2)
- returns **newArray** (array): an array of values that occur in the first array, but not in any of the subsequent arrays. The order of the result array is undefined and should not be relied on. Duplicates will be removed.

NTH()

`NTH(anyArray, position) → nthElement`

Get the element of an array at a given position. It is the same as `anyArray[position]` for positive positions, but does not support negative positions.

- **anyArray** (array): array with elements of arbitrary type

- **position** (number): position of desired element in array, positions start at 0
- returns **nthElement** (any|null): the array element at the given *position*. If *position* is negative or beyond the upper bound of the array, then *null* will be returned.

```
NTH( [ "foo", "bar", "baz" ], 2 )  // "baz"
NTH( [ "foo", "bar", "baz" ], 3 )  // null
NTH( [ "foo", "bar", "baz" ], -1 ) // null
```

OUTERSECTION()

`OUTERSECTION(array1, array2, ... arrayN) → newArray`

Return the values that occur only once across all arrays specified.

- **arrays** (array, *repeatable*): an arbitrary number of arrays as multiple arguments (at least 2)
- returns **newArray** (array): a single array with only the elements that exist only once across all provided arrays. The element order is random.

```
OUTERSECTION( [ 1, 2, 3 ], [ 2, 3, 4 ], [ 3, 4, 5 ] )
// [ 1, 5 ]
```

POP()

`POP(anyArray) → newArray`

Remove the element at the end (right side) of *array*.

- **anyArray** (array): an array with elements of arbitrary type
- returns **newArray** (array): *anyArray* without the last element. If it's already empty or has only a single element left, an empty array is returned.

```
POP( [ 1, 2, 3, 4 ] ) // [ 1, 2, 3 ]
POP( [ 1 ] ) // []
```

POSITION()

`POSITION(anyArray, search, returnIndex) → position`

Return whether *search* is contained in *array*. Optionally return the position.

- **anyArray** (array): the haystack, an array with elements of arbitrary type
- **search** (any): the needle, an element of arbitrary type
- **returnIndex** (bool, *optional*): if set to *true*, the position of the match is returned instead of a boolean. The default is *false*.
- returns **position** (bool|number): *true* if *search* is contained in *anyArray*, *false* otherwise. If *returnIndex* is enabled, the position of the match is returned (positions start at 0), or *-1* if it's not found.

PUSH()

`PUSH(anyArray, value, unique) → newArray`

Append *value* to the array specified by *anyArray*.

- **anyArray** (array): array with elements of arbitrary type
- **value** (any): an element of arbitrary type
- **unique** (bool): if set to *true*, then *value* is not added if already present in the array. The default is *false*.
- returns **newArray** (array): *anyArray* with *value* added at the end (right side)

Note: The *unique* flag only controls if *value* is added if it's already present in *anyArray*. Duplicate elements that already exist in *anyArray* will not be removed. To make an array unique, use the UNIQUE() function.

```
PUSH([ 1, 2, 3 ], 4)
// [ 1, 2, 3, 4 ]
```

```
PUSH([ 1, 2, 2, 3 ], 2, true)
// [ 1, 2, 2, 3 ]
```

REMOVE_NTH()

`REMOVE_NTH(anyArray, position) → newArray`

Remove the element at *position* from the *anyArray*.

- **anyArray** (array): array with elements of arbitrary type
- **position** (number): the position of the element to remove. Positions start at 0. Negative positions are supported, with -1 being the last array element. If *position* is out of bounds, the array is returned unmodified.
- returns **newArray** (array): *anyArray* without the element at *position*

```
REMOVE_NTH( [ "a", "b", "c", "d", "e" ], 1 )
// [ "a", "c", "d", "e" ]

REMOVE_NTH( [ "a", "b", "c", "d", "e" ], -2 )
// [ "a", "b", "c", "e" ]
```

REMOVE_VALUE()

`REMOVE_VALUE(anyArray, value, limit) → newArray`

Remove all occurrences of *value* in *anyArray*. Optionally with a *limit* to the number of removals.

- **anyArray** (array): array with elements of arbitrary type
- **value** (any): an element of arbitrary type
- **limit** (number, *optional*): cap the number of removals to this value
- returns **newArray** (array): *anyArray* with *value* removed

```
REMOVE_VALUE( [ "a", "b", "b", "a", "c" ], "a" )
// [ "b", "b", "c" ]

REMOVE_VALUE( [ "a", "b", "b", "a", "c" ], "a", 1 )
// [ "b", "b", "a", "c" ]
```

REMOVE_VALUES()

`REMOVE_VALUES(anyArray, values) → newArray`

Remove all occurrences of any of the *values* from *anyArray*.

- **anyArray** (array): array with elements of arbitrary type
- **values** (array): an array with elements of arbitrary type, that shall be removed from *anyArray*
- returns **newArray** (array): *anyArray* with all individual *values* removed

```
REMOVE_VALUES( [ "a", "a", "b", "c", "d", "e", "f" ], [ "a", "f", "d" ] )
// [ "b", "c", "e" ]
```

REVERSE()

`REVERSE(anyArray) → reversedArray`

Return an array with its elements reversed.

- **anyArray** (array): array with elements of arbitrary type
- returns **reversedArray** (array): a new array with all elements of *anyArray* in reversed order

SHIFT()

`SHIFT(anyArray) → newArray`

Remove the element at the start (left side) of *anyArray*.

- **anyArray** (array): array with elements with arbitrary type
- returns **newArray** (array): *anyArray* without the left-most element. If *anyArray* is already empty or has only one element left, an empty array is returned.

```
SHIFT( [ 1, 2, 3, 4 ] ) // [ 2, 3, 4 ]
SHIFT( [ 1 ] ) // []
```

SLICE()

`SLICE(anyArray, start, length) → newArray`

Extract a slice of *anyArray*.

- **anyArray** (array): array with elements of arbitrary type
- **start** (number): start extraction at this element. Positions start at 0. Negative values indicate positions from the end of the array.
- **length** (number, *optional*): extract up to *length* elements, or all elements from *start* up to *length* if negative (exclusive)
- returns **newArray** (array): the specified slice of *anyArray*. If *length* is not specified, all array elements starting at *start* will be returned.

```
SLICE( [ 1, 2, 3, 4, 5 ], 0, 1 ) // [ 1 ]
SLICE( [ 1, 2, 3, 4, 5 ], 1, 2 ) // [ 2, 3 ]
SLICE( [ 1, 2, 3, 4, 5 ], 3 ) // [ 4, 5 ]
SLICE( [ 1, 2, 3, 4, 5 ], 1, -1 ) // [ 2, 3, 4 ]
SLICE( [ 1, 2, 3, 4, 5 ], 0, -2 ) // [ 1, 2, 3 ]
SLICE( [ 1, 2, 3, 4, 5 ], -3, 2 ) // [ 3, 4 ]
```

UNION()

`UNION(array1, array2, ... arrayN) → newArray`

Return the union of all arrays specified.

- **arrays** (array, *repeatable*): an arbitrary number of arrays as multiple arguments (at least 2)
- returns **newArray** (array): all array elements combined in a single array, in any order

```
UNION(
    [ 1, 2, 3 ],
    [ 1, 2 ]
)
// [ 1, 2, 3, 1, 2 ]
```

Note: No duplicates will be removed. In order to remove duplicates, please use either UNION_DISTINCT() or apply UNIQUE() on the result of *UNION()*:

```
UNIQUE(
    UNION(
        [ 1, 2, 3 ],
        [ 1, 2 ]
    )
)
// [ 1, 2, 3 ]
```

UNION_DISTINCT()

`UNION_DISTINCT(array1, array2, ... arrayN) → newArray`

Return the union of distinct values of all arrays specified.

- **arrays** (array, *repeatable*): an arbitrary number of arrays as multiple arguments (at least 2)
- returns **newArray** (array): the elements of all given arrays in a single array, without duplicates, in any order

```
UNION_DISTINCT(
    [ 1, 2, 3 ],
    [ 1, 2 ]
```

```
)
// [ 1, 2, 3 ]
```

UNIQUE()

`UNIQUE(anyArray) → newArray`

Return all unique elements in *anyArray*. To determine uniqueness, the function will use the comparison order.

- **anyArray** (array): array with elements of arbitrary type
- returns **newArray** (array): *anyArray* without duplicates, in any order

UNSHIFT()

`UNSHIFT(anyArray, value, unique) → newArray`

Prepend *value* to *anyArray*.

- **anyArray** (array): array with elements of arbitrary type
- **value** (any): an element of arbitrary type
- **unique** (bool): if set to *true*, then *value* is not added if already present in the array. The default is *false*.
- returns **newArray** (array): *anyArray* with *value* added at the start (left side)

Note: The *unique* flag only controls if *value* is added if it's already present in *anyArray*. Duplicate elements that already exist in *anyArray* will not be removed. To make an array unique, use the UNIQUE() function.

```
UNSHIFT( [ 1, 2, 3 ], 4 ) // [ 4, 1, 2, 3 ]
UNSHIFT( [ 1, 2, 3 ], 2, true ) // [ 1, 2, 3 ]
```

Document functions

AQL provides below listed functions to operate on objects / document values. Also see object access for additional language constructs.

ATTRIBUTES()

`ATTRIBUTES(document, removeInternal, sort) → strArray`

Return the attribute keys of the *document* as an array. Optionally omit system attributes.

- **document** (object): an arbitrary document / object
- **removeInternal** (bool, *optional*): whether all system attributes (*_key*, *_id* etc., every attribute key that starts with an underscore) shall be omitted in the result. The default is *false*.
- **sort** (bool, *optional*): optionally sort the resulting array alphabetically. The default is *false* and will return the attribute names in any order.
- returns **strArray** (array): the attribute keys of the input *document* as an array of strings

```
ATTRIBUTES( { "foo": "bar", "_key": "123", "_custom": "yes" } )
// [ "foo", "_key", "_custom" ]

ATTRIBUTES( { "foo": "bar", "_key": "123", "_custom": "yes" }, true )
// [ "foo" ]

ATTRIBUTES( { "foo": "bar", "_key": "123", "_custom": "yes" }, false, true )
// [ "_custom", "_key", "foo" ]
```

Complex example to count how often every attribute key occurs in the documents of *collection* (expensive on large collections):

```
LET attributesPerDocument = (
    FOR doc IN collection RETURN ATTRIBUTES(doc, true)
)
FOR attributeArray IN attributesPerDocument
    FOR attribute IN attributeArray
        COLLECT attr = attribute WITH COUNT INTO count
        SORT count DESC
        RETURN {attr, count}
```

COUNT()

This is an alias for LENGTH().

HAS()

`HAS(document, attributeName) → isPresent`

Test whether an attribute is present in the provided document.

- **document** (object): an arbitrary document / object
- **attributeName** (string): the attribute key to test for
- returns **isPresent** (bool): *true* if *document* has an attribute named *attributeName*, and *false* otherwise. An attribute with a falsy value (*0*, *false*, empty string *""*) or *null* is also considered as present and returns *true*.

```
HAS( { name: "Jane" }, "name" ) // true
HAS( { name: "Jane" }, "age" ) // false
HAS( { name: null }, "name" ) // true
```

Note that the function checks if the specified attribute exists. This is different from similar ways to test for the existance of an attribute, in case the attribute has a falsy value or is not present (implicitly *null* on object access):

```
!!{ name: "" }.name        // false
HAS( { name: "" }, "name") // true
```

```
{ name: null }.name == null   // true
{ }.name == null              // true
HAS( { name: null }, "name" ) // true
HAS( { }, "name" )            // false
```

Note that `HAS()` can not utilize indexes. If it's not necessary to distinguish between explicit and implicit *null* values in your query, you may use an equality comparison to test for *null* and create a non-sparse index on the attribute you want to test against:

```
FILTER !HAS(doc, "name")    // can not use indexes
FILTER IS_NULL(doc, "name") // can not use indexes
FILTER doc.name == null     // can utilize non-sparse indexes
```

IS_SAME_COLLECTION()

`IS_SAME_COLLECTION(collectionName, documentHandle) → bool`

collection id as the collection specified in *collection*. *document* can either be a document handle string, or a document with an *_id* attribute. The function does not validate whether the collection actually contains the specified document, but only compares the name of the specified collection with the collection name part of the specified document. If *document* is neither an object with an *id* attribute nor a *string* value, the function will return *null* and raise a warning.

- **collectionName** (string): the name of a collection as string
- **documentHandle** (string|object): a document identifier string (e.g. *_users/1234*) or a regular document from a collection. Passing either a non-string or a non-document or a document without an *_id* attribute will result in an error.
- returns **bool** (bool): return *true* if the collection of *documentHandle* is the same as *collectionName*, otherwise *false*

```
// true
IS_SAME_COLLECTION( "_users", "_users/my-user" )
IS_SAME_COLLECTION( "_users", { _id: "_users/my-user" } )

// false
IS_SAME_COLLECTION( "_users", "foobar/baz")
IS_SAME_COLLECTION( "_users", { _id: "something/else" } )
```

KEEP()

`KEEP(document, attributeName1, attributeName2, ... attributeNameN) → doc`

Keep only the attributes *attributeName* to *attributeNameN* of *document*. All other attributes will be removed from the result.

- **document** (object): a document / object
- **attributeNames** (string, *repeatable*): an arbitrary number of attribute names as multiple arguments
- returns **doc** (object): a document with only the specified attributes on the top-level

```
KEEP(doc, "firstname", "name", "likes")
```

`KEEP(document, attributeNameArray) → doc`

- **document** (object): a document / object
- **attributeNameArray** (array): an array of attribute names as strings
- returns **doc** (object): a document with only the specified attributes on the top-level

```
KEEP(doc, [ "firstname", "name", "likes" ])
```

LENGTH()

`LENGTH(doc) → attrCount`

Determine the number of attribute keys of an object / document.

- **doc** (object): a document / object
- returns **attrCount** (number): the number of attribute keys in *doc*, regardless of their values

LENGTH() can also determine the number of elements in an array, the amount of documents in a collection and the character length of a string.

MATCHES()

`MATCHES(document, examples, returnIndex) → match`

Compare the given *document* against each example document provided. The comparisons will be started with the first example. All attributes of the example will be compared against the attributes of *document*. If all attributes match, the comparison stops and the result is returned. If there is a mismatch, the function will continue the comparison with the next example until there are no more examples left.

The *examples* can be an array of 1..n example documents or a single document, with any number of attributes each.

Note that *MATCHES()* can not utilize indexes.

- **document** (object): document to determine whether it matches any example
- **examples** (object|array): a single document, or an array of documents to compare against. Specifying an empty array is not allowed.
- **returnIndex** (bool): by setting this flag to *true*, the index of the example that matched will be returned (starting at offset 0), or *-1* if there was no match. The default is *false* and makes the function return a boolean.
- returns **match** (bool|number): if *document* matches one of the examples, *true* is returned, otherwise *false*. A number is returned instead if *returnIndex* is used.

```
LET doc = {
    name: "jane",
    age: 27,
    active: true
}
RETURN MATCHES(doc, { age: 27, active: true } )
```

This will return *true*, because all attributes of the example are present in the document.

```
RETURN MATCHES(
    { "test": 1 },
    [
        { "test": 1, "foo": "bar" },
        { "foo": 1 },
        { "test": 1 }
    ], true)
```

This will return *2*, because the third example matches, and because the *returnIndex* flag is set to *true*.

MERGE()

`MERGE(document1, document2, ... documentN) → mergedDocument`

Merge the documents *document1* to *documentN* into a single document. If document attribute keys are ambiguous, the merged result will contain the values of the documents contained later in the argument list.

- **documents** (object, *repeatable*): an arbitrary number of documents as multiple arguments (at least 2)
- returns **mergedDocument** (object): a combined document

Note that merging will only be done for top-level attributes. If you wish to merge sub-attributes, use MERGE_RECURSIVE() instead.

Two documents with distinct attribute names can easily be merged into one:

```
MERGE(
    { "user1": { "name": "Jane" } },
    { "user2": { "name": "Tom" } }
)
// { "user1": { "name": "Jane" }, "user2": { "name": "Tom" } }
```

When merging documents with identical attribute names, the attribute values of the latter documents will be used in the end result:

```
MERGE(
    { "users": { "name": "Jane" } },
```

```
    { "users": { "name": "Tom" } }
)
// { "users": { "name": "Tom" } }
```

`MERGE(docArray) → mergedDocument`

MERGE works with a single array parameter, too. This variant allows combining the attributes of multiple objects in an array into a single object.

- **docArray** (array): an array of documents, as sole argument
- returns **mergedDocument** (object): a combined document

```
MERGE(
    [
        { foo: "bar" },
        { quux: "quetzalcoatl", ruled: true },
        { bar: "baz", foo: "done" }
    ]
)
```

This will now return:

```
{
    "foo": "done",
    "quux": "quetzalcoatl",
    "ruled": true,
    "bar": "baz"
}
```

MERGE_RECURSIVE()

`MERGE_RECURSIVE(document1, document2, ... documentN) → mergedDocument`

Recursively merge the documents *document1* to *documentN* into a single document. If document attribute keys are ambiguous, the merged result will contain the values of the documents contained later in the argument list.

- **documents** (object, *repeatable*): an arbitrary number of documents as multiple arguments (at least 2)
- returns **mergedDocument** (object): a combined document

For example, two documents with distinct attribute names can easily be merged into one:

```
MERGE_RECURSIVE(
    { "user-1": { "name": "Jane", "livesIn": { "city": "LA" } } },
    { "user-1": { "age": 42, "livesIn": { "state": "CA" } } }
)
// { "user-1": { "name": "Jane", "livesIn": { "city": "LA", "state": "CA" }, "age": 42 } }
```

MERGE_RECURSIVE() does not support the single array parameter variant that *MERGE* offers.

PARSE_IDENTIFIER()

`PARSE_IDENTIFIER(documentHandle) → parts`

Parse a document handle and return its individual parts a separate attributes.

This function can be used to easily determine the collection name and key of a given document.

- **documentHandle** (string|object): a document identifier string (e.g. *_users/1234*) or a regular document from a collection. Passing either a non-string or a non-document or a document without an *_id* attribute will result in an error.
- returns **parts** (object): an object with the attributes *collection* and *key*

```
PARSE_IDENTIFIER("_users/my-user")
// { "collection": "_users", "key": "my-user" }

PARSE_IDENTIFIER( { "_id": "mycollection/mykey", "value": "some value" } )
// { "collection": "mycollection", "key": "mykey" }
```

TRANSLATE()

`TRANSLATE(value, lookupDocument, defaultValue) → mappedValue`

Look up the specified *value* in the *lookupDocument*. If *value* is a key in *lookupDocument*, then *value* will be replaced with the lookup value found. If *value* is not present in *lookupDocument*, then *defaultValue* will be returned if specified. If no *defaultValue* is specified, *value* will be returned unchanged.

- **value** (string): the value to encode according to the mapping
- **lookupDocument** (object): a key/value mapping as document
- **defaultValue** (any, *optional*): a fallback value in case *value* is not found
- returns **mappedValue** (any): the encoded value, or the unaltered *value* or *defaultValue* (if supplied) in case it couldn't be mapped

```
TRANSLATE("FR", { US: "United States", UK: "United Kingdom", FR: "France" } )
// "France"

TRANSLATE(42, { foo: "bar", bar: "baz" } )
// 42

TRANSLATE(42, { foo: "bar", bar: "baz" }, "not found!")
// "not found!"
```

UNSET()

`UNSET(document, attributeName1, attributeName2, ... attributeNameN) → doc`

Remove the attributes *attributeName1* to *attributeNameN* from *document*. All other attributes will be preserved.

- **document** (object): a document / object
- **attributeNames** (string, *repeatable*): an arbitrary number of attribute names as multiple arguments (at least 1)
- returns **doc** (object): *document* without the specified attributes on the top-level

```
UNSET( doc, "_id", "_key", "foo", "bar" )
```

`UNSET(document, attributeNameArray) → doc`

- **document** (object): a document / object
- **attributeNameArray** (array): an array of attribute names as strings
- returns **doc** (object): *document* without the specified attributes on the top-level

```
UNSET( doc, [ "_id", "_key", "foo", "bar" ] )
```

UNSET_RECURSIVE()

`UNSET_RECURSIVE(document, attributeName1, attributeName2, ... attributeNameN) → doc`

Recursively remove the attributes *attributeName1* to *attributeNameN* from *document* and its sub-documents. All other attributes will be preserved.

- **document** (object): a document / object
- **attributeNames** (string, *repeatable*): an arbitrary number of attribute names as multiple arguments (at least 1)
- returns **doc** (object): *document* without the specified attributes on all levels (top-level as well as nested objects)

```
UNSET_RECURSIVE( doc, "_id", "_key", "foo", "bar" )
```

`UNSET_RECURSIVE(document, attributeNameArray) → doc`

- **document** (object): a document / object
- **attributeNameArray** (array): an array of attribute names as strings
- returns **doc** (object): *document* without the specified attributes on all levels (top-level as well as nested objects)

```
UNSET_RECURSIVE( doc, [ "_id", "_key", "foo", "bar" ] )
```

VALUES()

`VALUES(document, removeInternal) → anyArray`

Return the attribute values of the *document* as an array. Optionally omit system attributes.

- **document** (object): a document / object
- **removeInternal** (bool, *optional*): if set to *true*, then all internal attributes (such as *_id*, *_key* etc.) are removed from the result
- returns **anyArray** (array): the values of *document* returned in any order

```
VALUES( { "_key": "users/jane", "name": "Jane", "age": 35 } )
// [ "Jane", 35, "users/jane" ]

VALUES( { "_key": "users/jane", "name": "Jane", "age": 35 }, true )
// [ "Jane", 35 ]
```

ZIP()

`ZIP(keys, values) → doc`

Return a document object assembled from the separate parameters *keys* and *values*.

keys and *values* must be arrays and have the same length.

- **keys** (array): an array of strings, to be used as attribute names in the result
- **values** (array): an array with elements of arbitrary types, to be used as attribute values
- returns **doc** (object): a document with the keys and values assembled

```
ZIP( [ "name", "active", "hobbies" ], [ "some user", true, [ "swimming", "riding" ] ] )
// { "name": "some user", "active": true, "hobbies": [ "swimming", "riding" ] }
```

Geo functions

Geo index functions

AQL offers the following functions to filter data based on geo indexes. These functions require the collection to have at least one geo index. If no geo index can be found, calling this function will fail with an error at runtime. There is no error when explaining the query however.

NEAR()

```
NEAR(coll, latitude, longitude, limit, distanceName) → docArray
```

Return at most *limit* documents from collection *coll* that are near *latitude* and *longitude*. The result contains at most *limit* documents, returned sorted by distance, with closest distances being returned first. If more than *limit* documents qualify, with the distance being exactly the same among multiple documents around the limit, it is undefined which of the qualifying documents are returned. Optionally, the distances in meters between the specified coordinate (*latitude* and *longitude*) and the document coordinates can be returned as well. To make use of that, the desired attribute name for the distance result has to be specified in the *distanceName* argument. The result documents will contain the distance value in an attribute of that name.

- **coll** (collection): a collection
- **latitude** (number): the latitude portion of the search coordinate
- **longitude** (number): the longitude portion of the search coordinate
- **limit** (number, *optional*): cap the result to at most this number of documents. The default is 100. If more documents than *limit* are found, it is undefined which ones will be returned.
- **distanceName** (string, *optional*): include the distance to the search coordinate in each document in the result (in meters), using the attribute name *distanceName*
- returns **docArray** (array): an array of documents, sorted by distance (shortest distance first)

WITHIN()

```
WITHIN(coll, latitude, longitude, radius, distanceName) → docArray
```

Return all documents from collection *coll* that are within a radius of *radius* around the specified coordinate (*latitude* and *longitude*). The documents returned are sorted by distance to the search coordinate, with the closest distances being returned first. Optionally, the distance in meters between the search coordinate and the document coordinates can be returned as well. To make use of that, an attribute name for the distance result has to be specified in the *distanceName* argument. The result documents will contain the distance value in an attribute of that name.

- **coll** (collection): a collection
- **latitude** (number): the latitude portion of the search coordinate
- **longitude** (number): the longitude portion of the search coordinate
- **radius** (number): radius in meters
- **distanceName** (string, *optional*): include the distance to the search coordinate in each document in the result (in meters), using the attribute name *distanceName*
- returns **docArray** (array): an array of documents, sorted by distance (shortest distance first)

WITHIN_RECTANGLE()

```
WITHIN_RECTANGLE(coll, latitude1, longitude1, latitude2, longitude2) → docArray
```

Return all documents from collection *coll* that are positioned inside the bounding rectangle with the points (*latitude1*, *longitude1*) and (*latitude2*, *longitude2*). There is no guaranteed order in which the documents are returned.

- **coll** (collection): a collection
- **latitude1** (number): the bottom-left latitude portion of the search coordinate
- **longitude1** (number): the bottom-left longitude portion of the search coordinate
- **latitude2** (number): the top-right latitude portion of the search coordinate
- **longitude2** (number): the top-right longitude portion of the search coordinate

- returns **docArray** (array): an array of documents, in random order

Geo utility functions

The following helper functions do **not use any geo index**. On large datasets, it is advisable to use them in combination with index-accelerated geo functions to limit the number of calls to these non-accelerated functions because of their computational costs.

DISTANCE()

`DISTANCE(latitude1, longitude1, latitude2, longitude2) → distance`

Calculate the distance between two arbitrary coordinates in meters (as birds would fly). The value is computed using the haversine formula, which is based on a spherical Earth model. It's fast to compute and is accurate to around 0.3%, which is sufficient for most use cases such as location-aware services.

- **latitude1** (number): the latitude portion of the first coordinate
- **longitude1** (number): the longitude portion of the first coordinate
- **latitude2** (number): the latitude portion of the second coordinate
- **longitude2** (number): the longitude portion of the second coordinate
- returns **distance** (number): the distance between both coordinates in meters

```
// Distance between Brandenburg Gate (Berlin) and ArangoDB headquarters (Cologne)
DISTANCE(52.5163, 13.3777, 50.9322, 6.94) // 476918.89688380965 (~477km)

// Sort a small number of documents based on distance to Central Park (New York)
FOR doc IN documentSubset // e.g. documents returned by a traversal
  SORT DISTANCE(doc.latitude, doc.longitude, 40.78, -73.97)
  RETURN doc
```

IS_IN_POLYGON()

Determine whether a coordinate is inside a polygon.

`IS_IN_POLYGON(polygon, latitude, longitude) → bool`

- **polygon** (array): an array of arrays with 2 elements each, representing the points of the polygon in the format *[lat, lon]*
- **latitude** (number): the latitude portion of the search coordinate
- **longitude** (number): the longitude portion of the search coordinate
- returns **bool** (bool): *true* if the point (*latitude*, *longitude*) is inside the *polygon* or *false* if it's not. The result is undefined (can be *true* or *false*) if the specified point is exactly on a boundary of the polygon.

```
// will check if the point (lat 4, lon 7) is contained inside the polygon
IS_IN_POLYGON( [ [ 0, 0 ], [ 0, 10 ], [ 10, 10 ], [ 10, 0 ] ], 4, 7 )
```

`IS_IN_POLYGON(polygon, coord, useLonLat) → bool`

The 2nd parameter can alternatively be specified as an array with two values.

By default, each array element in *polygon* is expected to be in the format *[lat, lon]*. This can be changed by setting the 3rd parameter to *true* to interpret the points as *[lon, lat]*. *coord* will then also be interpreted in the same way.

- **polygon** (array): an array of arrays with 2 elements each, representing the points of the polygon
- **coord** (array): the search coordinate as a number array with two elements
- **useLonLat** (bool, *optional*): if set to *true*, the coordinates in *polygon* and the search coordinate *coord* will be interpreted as *[lon, lat]*. The default is *false* and the format *[lat, lon]* is expected.
- returns **bool** (bool): *true* if the point *coord* is inside the *polygon* or *false* if it's not. The result is undefined (can be *true* or *false*) if the specified point is exactly on a boundary of the polygon.

```
// will check if the point (lat 4, lon 7) is contained inside the polygon
IS_IN_POLYGON( [ [ 0, 0 ], [ 0, 10 ], [ 10, 10 ], [ 10, 0 ] ], [ 4, 7 ] )

// will check if the point (lat 4, lon 7) is contained inside the polygon
IS_IN_POLYGON( [ [ 0, 0 ], [ 10, 0 ], [ 10, 10 ], [ 0, 10 ] ], [ 7, 4 ], true )
```

Fulltext functions

AQL offers the following functions to filter data based on fulltext indexes.

Currently, fulltext indexes are not yet supported with the RocksDB storage engine. Thus the function `FULLTEXT()` will be unavailable when using this storage engine. To use fulltext indexes, please use the MMFiles storage engine for the time being.

FULLTEXT()

```
FULLTEXT(coll, attribute, query, limit) → docArray
```

Return all documents from collection *coll*, for which the attribute *attribute* matches the fulltext search phrase *query*, optionally capped to *limit* results.

Note: the *FULLTEXT()* function requires the collection *coll* to have a fulltext index on *attribute*. If no fulltext index is available, this function will fail with an error at runtime. It doesn't fail when explaining the query however.

- **coll** (collection): a collection
- **attribute** (string): the attribute name of the attribute to search in
- **query** (string): a fulltext search expression as described below
- **limit** (number, *optional*): if set to a non-zero value, it will cap the result to at most this number of documents
- returns **docArray** (array): an array of documents

FULLTEXT() is not meant to be used as an argument to *FILTER*, but rather to be used as the expression of a *FOR* statement:

```
FOR oneMail IN FULLTEXT(emails, "body", "banana,-apple")
    RETURN oneMail._id
```

query is a comma-separated list of sought words (or prefixes of sought words). To distinguish between prefix searches and complete-match searches, each word can optionally be prefixed with either the *prefix:* or *complete:* qualifier. Different qualifiers can be mixed in the same query. Not specifying a qualifier for a search word will implicitly execute a complete-match search for the given word:

- *FULLTEXT(emails, "body", "banana")* Will look for the word *banana* in the attribute *body* of the collection *collection*.
- *FULLTEXT(emails, "body", "banana,orange")* Will look for both words *banana* and *orange* in the mentioned attribute. Only those documents will be returned that contain both words.
- *FULLTEXT(emails, "body", "prefix:head")* Will look for documents that contain any words starting with the prefix *head*.
- *FULLTEXT(emails, "body", "prefix:head,complete:aspirin")* Will look for all documents that contain a word starting with the prefix *head* and that also contain the (complete) word *aspirin*. Note: specifying *complete* is optional here.
- *FULLTEXT(emails, "body", "prefix:cent,prefix:subst")* Will look for all documents that contain a word starting with the prefix *cent* and that also contain a word starting with the prefix *subst*.

If multiple search words (or prefixes) are given, then by default the results will be AND-combined, meaning only the logical intersection of all searches will be returned. It is also possible to combine partial results with a logical OR, and with a logical NOT:

- *FULLTEXT(emails, "body", "+this,+text,+document")* Will return all documents that contain all the mentioned words. Note: specifying the + symbols is optional here.
- *FULLTEXT(emails, "body", "banana,|apple")* Will return all documents that contain either (or both) words *banana* or *apple*.
- *FULLTEXT(emails, "body", "banana,-apple")* Will return all documents that contain the word *banana*, but do not contain the word *apple*.
- *FULLTEXT(emails, "body", "banana,pear,-cranberry")* Will return all documents that contain both the words *banana* and *pear*, but do not contain the word *cranberry*.

No precedence of logical operators will be honored in a fulltext query. The query will simply be evaluated from left to right.

Miscellaneous functions

Control flow functions

NOT_NULL()

`NOT_NULL(alternative, ...) → value`

Return the first element that is not *null*, and *null* if all alternatives are *null* themselves. It is also known as `COALESCE()` in SQL.

- **alternative** (any, *repeatable*): input of arbitrary type
- returns **value** (any): first non-null parameter, or *null* if all arguments are *null*

FIRST_LIST()

Return the first alternative that is an array, and *null* if none of the alternatives is an array.

- **alternative** (any, *repeatable*): input of arbitrary type
- returns **list** (list|null): array / list or null

FIRST_DOCUMENT()

`FIRST_DOCUMENT(value) → doc`

Return the first alternative that is a document, and *null* if none of the alternatives is a document.

- **alternative** (any, *repeatable*): input of arbitrary type
- returns **doc** (object|null): document / object or null

Ternary operator

For conditional evaluation, check out the ternary operator.

Database functions

COLLECTION_COUNT()

`COLLECTION_COUNT(coll) → count`

Determine the amount of documents in a collection. LENGTH() is preferred.

COLLECTIONS()

`COLLECTIONS() → docArray`

Return an array of collections.

- returns **docArray** (array): each collection as a document with attributes *name* and *_id* in an array

COUNT()

This is an alias for LENGTH().

CURRENT_USER()

`CURRENT_USER() → userName`

Return the name of the current user.

The current user is the user account name that was specified in the *Authorization* HTTP header of the request. It will only be populated if authentication on the server is turned on, and if the query was executed inside a request context. Otherwise, the return value of this function will be *null*.

- returns **userName** (string|null): the current user name, or *null* if authentication is disabled

DOCUMENT()

`DOCUMENT(collection, id) → doc`

Return the document which is uniquely identified by its *id*. ArangoDB will try to find the document using the *_id* value of the document in the specified collection.

If there is a mismatch between the *collection* passed and the collection specified in *id*, then *null* will be returned. Additionally, if the *collection* matches the collection value specified in *id* but the document cannot be found, *null* will be returned.

This function also allows *id* to be an array of ids. In this case, the function will return an array of all documents that could be found.

It is also possible to specify a document key instead of an id, or an array of keys to return all documents that can be found.

- **collection** (string): name of a collection
- **id** (string|array): a document handle string (consisting of collection name and document key), a document key, or an array of both document handle strings and document keys
- returns **doc** (document|array|null): the content of the found document, an array of all found documents or *null* if nothing was found

```
DOCUMENT( users, "users/john" )
DOCUMENT( users, "john" )

DOCUMENT( users, [ "users/john", "users/amy" ] )
DOCUMENT( users, [ "john", "amy" ] )
```

`DOCUMENT(id) → doc`

The function can also be used with a single parameter *id* as follows:

- **id** (string|array): either a document handle string (consisting of collection name and document key) or an array of document handle strings
- returns **doc** (document|null): the content of the found document or *null* if nothing was found

```
DOCUMENT("users/john")
DOCUMENT( [ "users/john", "users/amy" ] )
```

Please also consider to use `DOCUMENT` in conjunction with `WITH`

LENGTH()

`LENGTH(coll) → documentCount`

Determine the amount of documents in a collection.

It calls COLLECTION_COUNT() internally.

- **coll** (collection): a collection (not string)
- returns **documentCount** (number): the total amount of documents in *coll*

LENGTH() can also determine the number of elements in an array, the number of attribute keys of an object / document and the character length of a string.

Hash functions

`HASH(value) → hashNumber`

Calculate a hash value for *value*.

- **value** (any): an element of arbitrary type

- returns **hashNumber** (number): a hash value of *value*

value is not required to be a string, but can have any data type. The calculated hash value will take the data type of *value* into account, so for example the number *1* and the string *"1"* will have different hash values. For arrays the hash values will be creared if the arrays contain exactly the same values (including value types) in the same order. For objects the same hash values will be created if the objects have exactly the same attribute names and values (including value types). The order in which attributes appear inside objects is not important for hashing.

The hash value returned by this function is a number. The hash algorithm is not guaranteed to remain the same in future versions of ArangoDB. The hash values should therefore be used only for temporary calculations, e.g. to compare if two documents are the same, or for grouping values in queries.

Function calling

APPLY()

`APPLY(functionName, arguments) → retVal`

Dynamically call the function *funcName* with the arguments specified. Arguments are given as array and are passed as separate parameters to the called function.

Both built-in and user-defined functions can be called.

- **funcName** (string): a function name
- **arguments** (array, *optional*): an array with elements of arbitrary type
- returns **retVal** (any): the return value of the called function

```
APPLY( "SUBSTRING", [ "this is a test", 0, 7 ] )
// "this is"
```

CALL()

`CALL(funcName, arg1, arg2, ... argN) → retVal`

Dynamically call the function *funcName* with the arguments specified. Arguments are given as multiple parameters and passed as separate parameters to the called function.

Both built-in and user-defined functions can be called.

- **funcName** (string): a function name
- **args** (any, *repeatable*): an arbitrary number of elements as multiple arguments, can be omitted
- returns **retVal** (any): the return value of the called function

```
CALL( "SUBSTRING", "this is a test", 0, 4 )
// "this"
```

Internal functions

The following functions are used during development of ArangoDB as a database system, primarily for unit testing. They are not intended to be used by end users, especially not in production environments.

FAIL()

`FAIL(reason)`

Let a query fail on purpose. Can be used in a conditional branch, or to verify if lazy evaluation / short circuiting is used for instance.

- **reason** (string): an error message
- returns nothing, because the query is aborted

```
RETURN 1 == 1 ? "okay" : FAIL("error") // "okay"
```

```
RETURN 1 == 1 || FAIL("error") ? true : false // true
RETURN 1 == 2 && FAIL("error") ? true : false // false
RETURN 1 == 1 && FAIL("error") ? true : false // aborted with error
```

NOOPT()

`NOOPT(expression) → retVal`

No-operation that prevents query compile-time optimizations. Constant expressions can be forced to be evaluated at runtime with this.

If there is a C++ implementation as well as a JavaScript implementation of an AQL function, then it will enforce the use of the C++ version.

- **expression** (any): arbitray expression
- returns **retVal** (any): the return value of the *expression*

```
// differences in execution plan (explain)
FOR i IN 1..3 RETURN (1 + 1)      // const assignment
FOR i IN 1..3 RETURN NOOPT(1 + 1) // simple expression

NOOPT( RAND() ) // C++ implementation
V8( RAND() )    // JavaScript implementation
```

PASSTHRU()

`PASSTHRU(value) → retVal`

This function is marked as non-deterministic so its argument withstands query optimization.

- **value** (any): a value of arbitrary type
- returns **retVal** (any): *value*, without optimizations

SLEEP()

`SLEEP(seconds) → null`

Wait for a certain amount of time before continuing the query.

- **seconds** (number): amount of time to wait
- returns a *null* value

```
SLEEP(1)    // wait 1 second
SLEEP(0.02) // wait 20 milliseconds
```

V8()

`V8(expression) → retVal`

No-operation that enforces the usage of the V8 JavaScript engine. If there is a JavaScript implementation of an AQL function, for which there is also a C++ implementation, the JavaScript version will be used.

- **expression** (any): arbitray expression
- returns **retVal** (any): the return value of the *expression*

```
// differences in execution plan (explain)
FOR i IN 1..3 RETURN (1 + 1)          // const assignment
FOR i IN 1..3 RETURN V8(1 + 1)        // const assignment
FOR i IN 1..3 RETURN NOOPT(V8(1 + 1)) // v8 expression
```

Graphs in AQL

There are multiple ways to work with graphs in ArangoDB, as well as different ways to query your graphs using AQL.

The two options in managing graphs are to either use

- named graphs where ArangoDB manages the collections involved in one graph, or
- graph functions on a combination of document and edge collections.

Named graphs can be defined through the graph-module or via the web interface. The definition contains the name of the graph, and the vertex and edge collections involved. Since the management functions are layered on top of simple sets of document and edge collections, you can also use regular AQL functions to work with them.

Both variants (named graphs and loosely coupled collection sets a.k.a. anonymous graphs) are supported by the AQL language constructs for graph querying. These constructs make full use of optimizations and therefore best performance is to be expected:

- AQL Traversals to follow edges connected to a start vertex, up to a variable depth. It can be combined with AQL filter conditions.
- AQL Shortest Path to find the vertices and edges between two given vertices, with as few hops as possible.

These types of queries are only useful if you use edge collections and/or graphs in your data model.

Graph traversals in AQL

General query idea

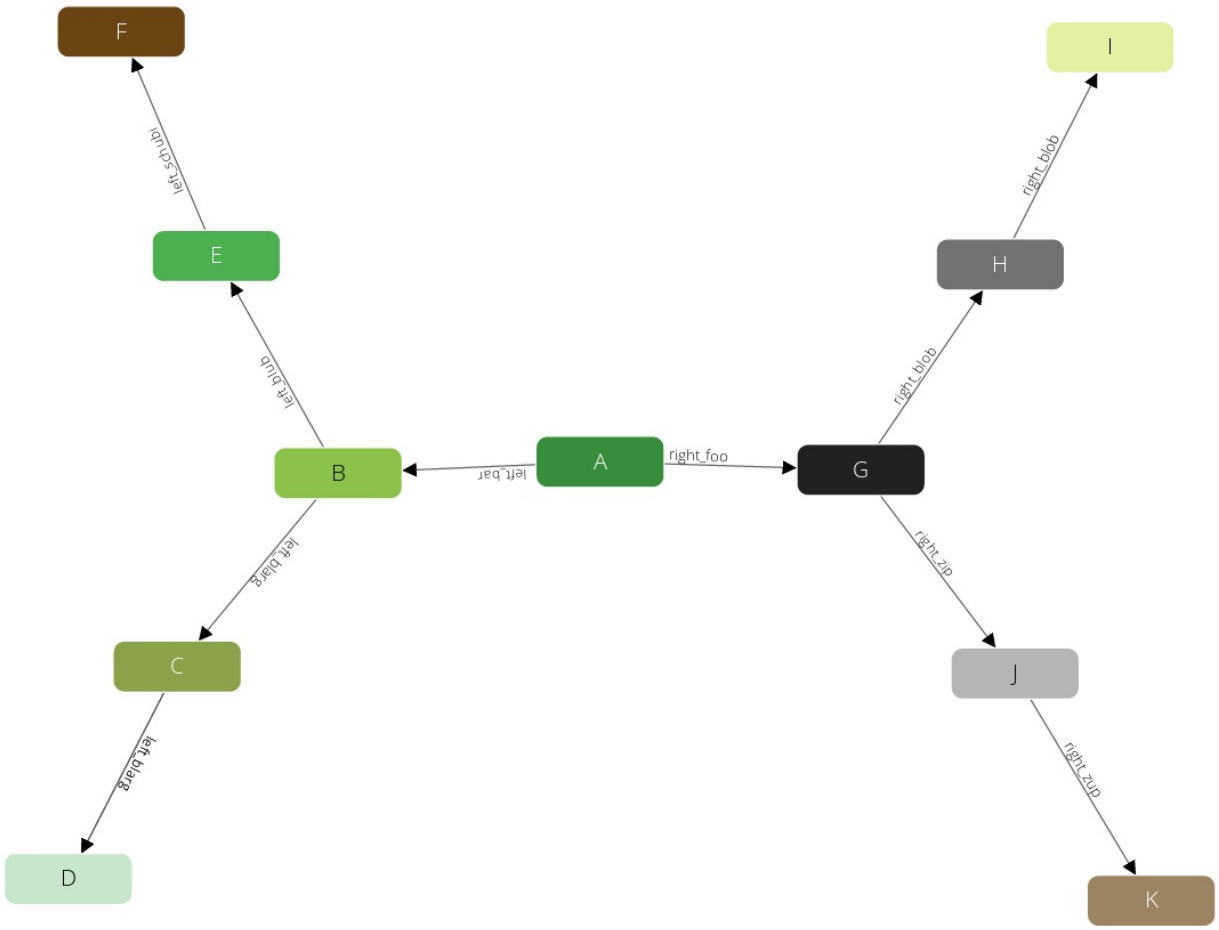
A traversal starts at one specific document (*startVertex*) and follows all edges connected to this document. For all documents (*vertices*) that are targeted by these edges it will again follow all edges connected to them and so on. It is possible to define how many of these follow iterations should be executed at least (*min* depth) and at most (*max* depth).

For all vertices that were visited during this process in the range between *min* depth and *max* depth iterations you will get a result in form of a set with three items:

1. The visited vertex.
2. The edge pointing to it.
3. The complete path from startVertex to the visited vertex as object with an attribute *edges* and an attribute *vertices*, each a list of the coresponding elements. These lists are sorted, which means the first element in *vertices* is the *startVertex* and the last is the visited vertex, and the n-th element in *edges* connects the n-th element with the (n+1)-th element in *vertices*.

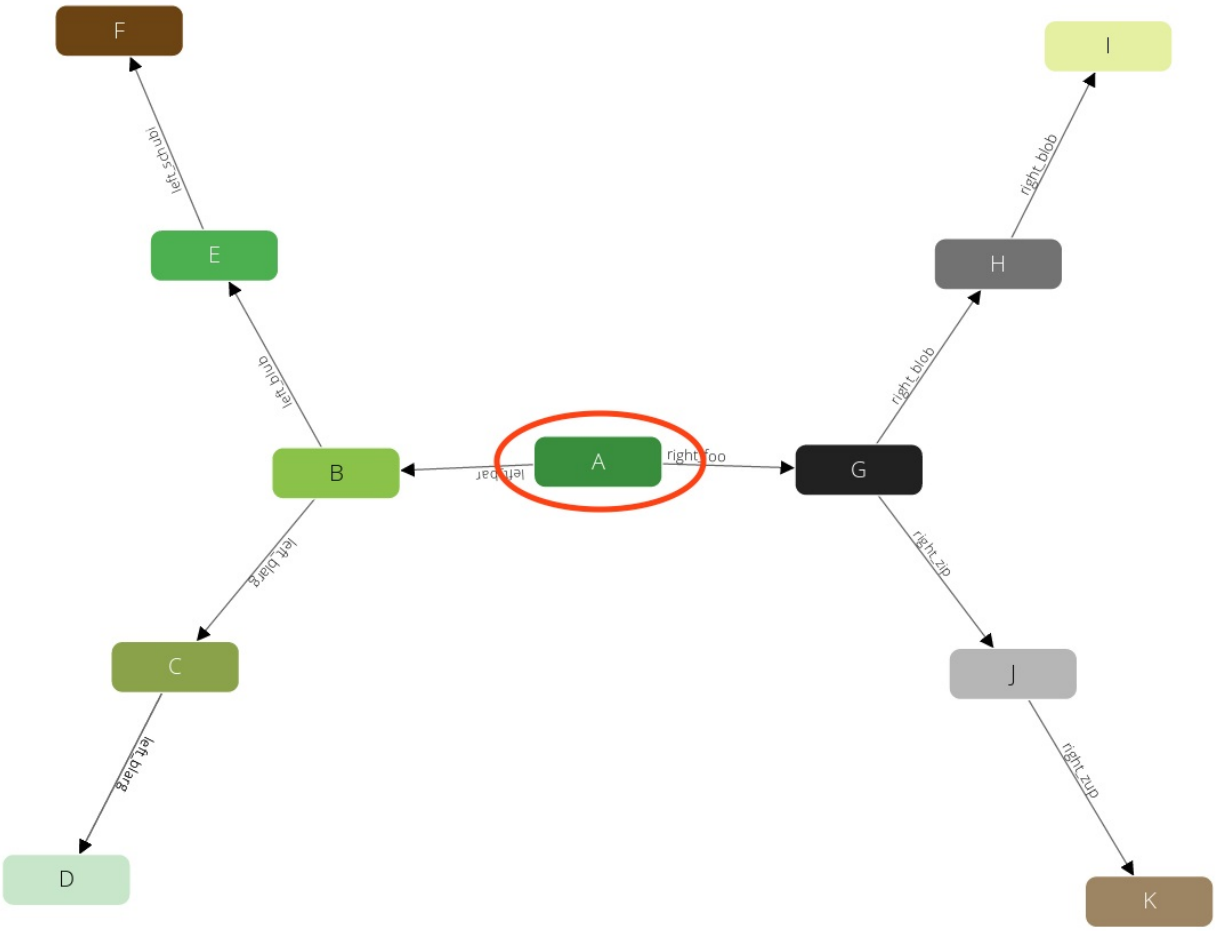
Example execution

Let's take a look at a simple example to explain how it works. This is the graph that we are going to traverse:

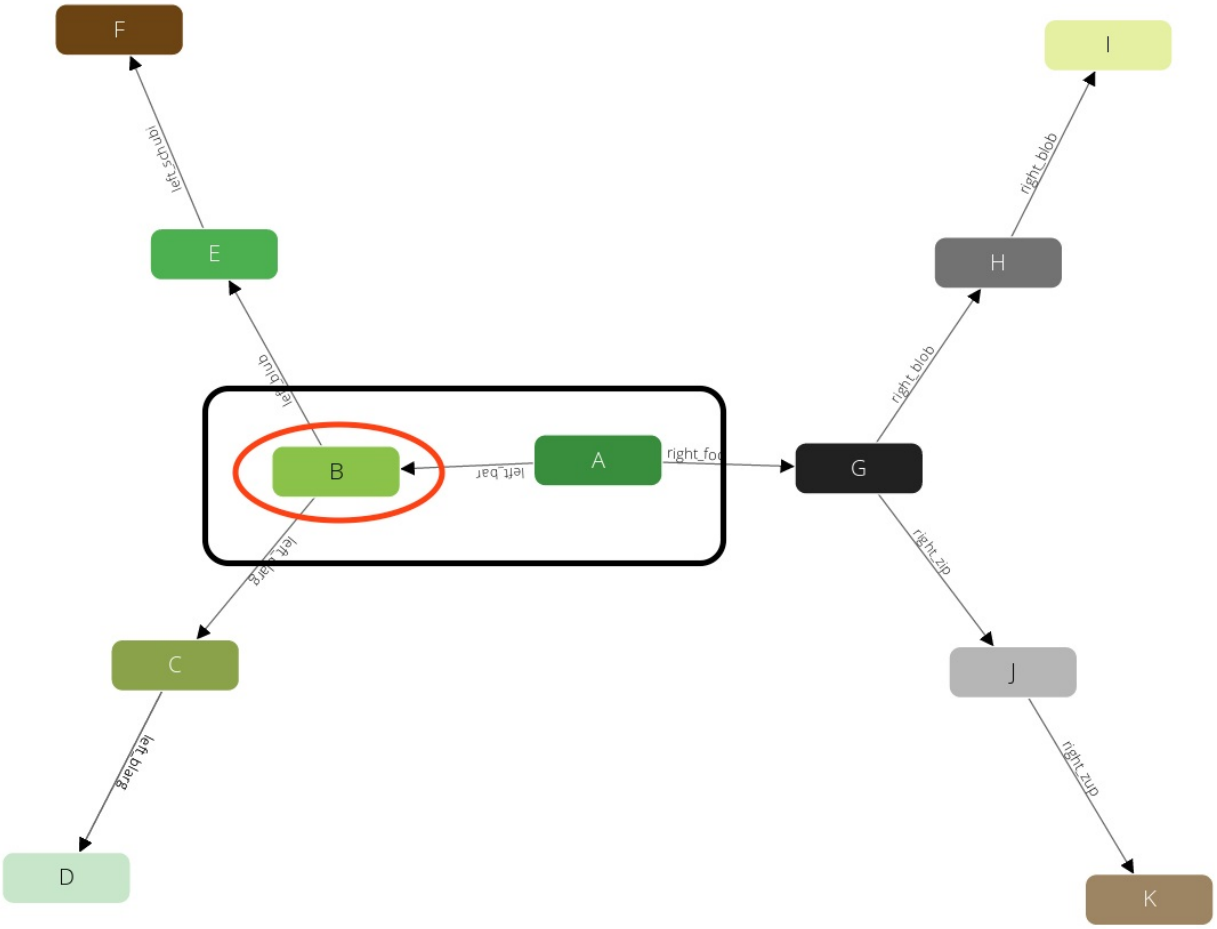


We use the following parameters for our query:

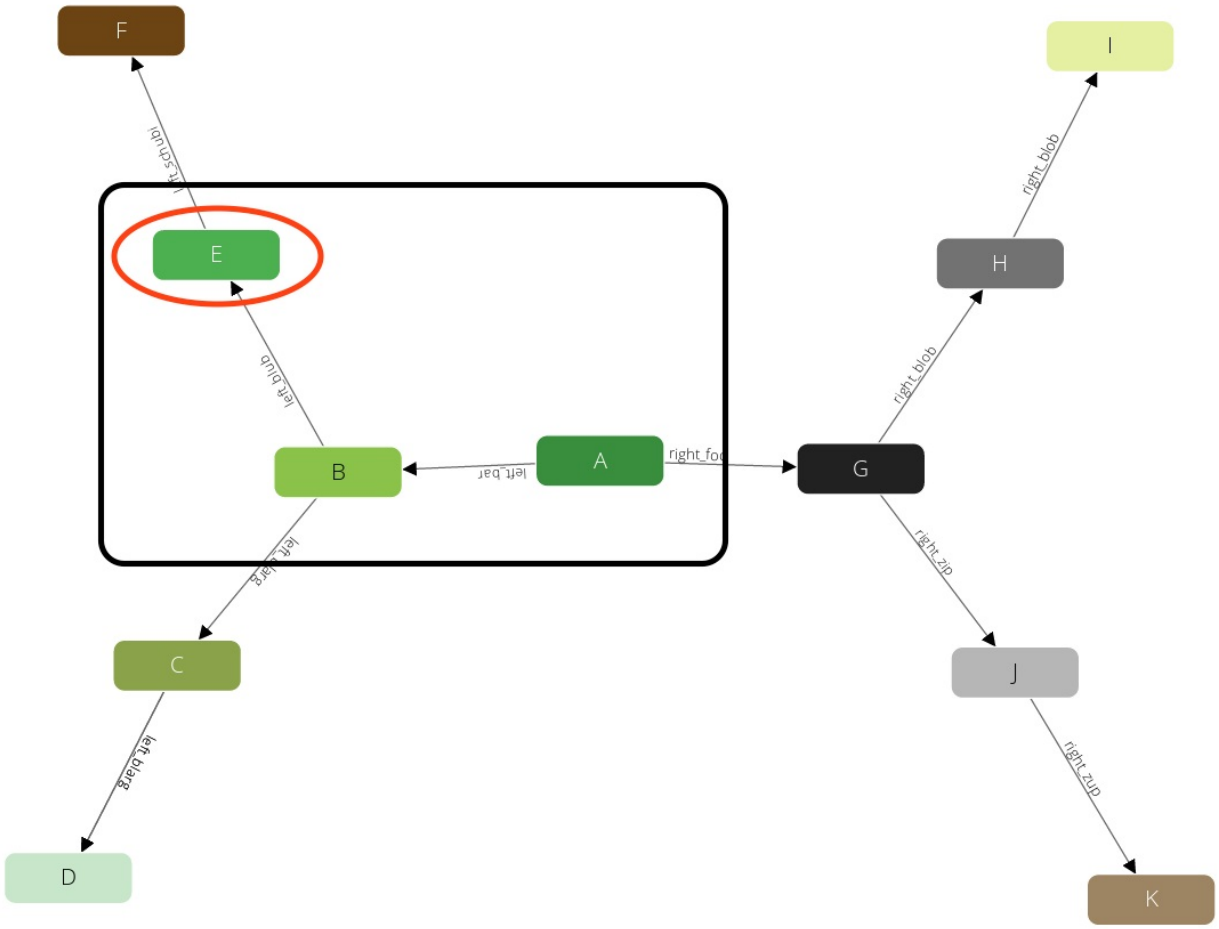
1. We start at the vertex **A**.
2. We use a *min* depth of 1.
3. We use a *max* depth of 2.
4. We follow only in *OUTBOUND* direction of edges



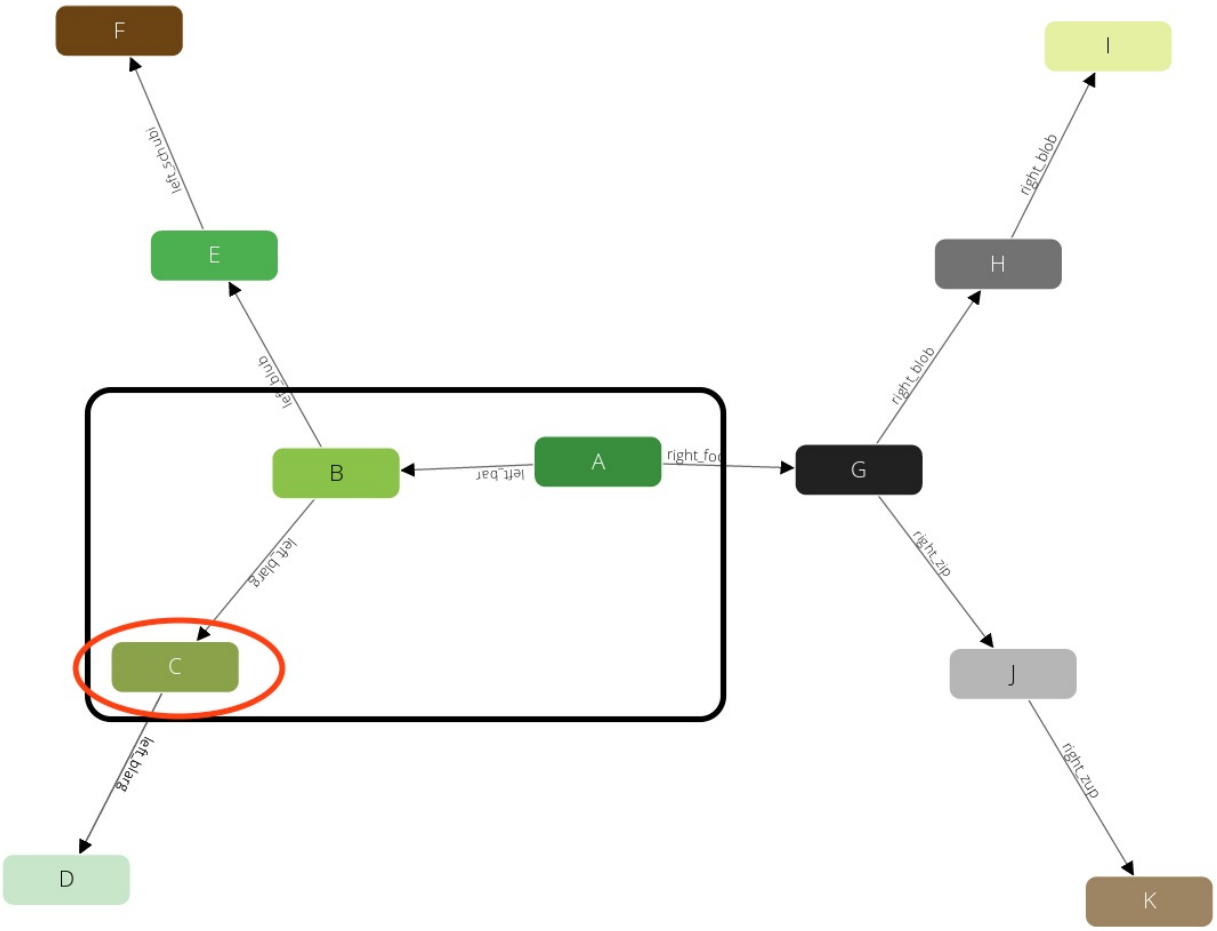
Now it walks to one of the direct neighbors of **A**, say **B** (note: ordering is not guaranteed!):



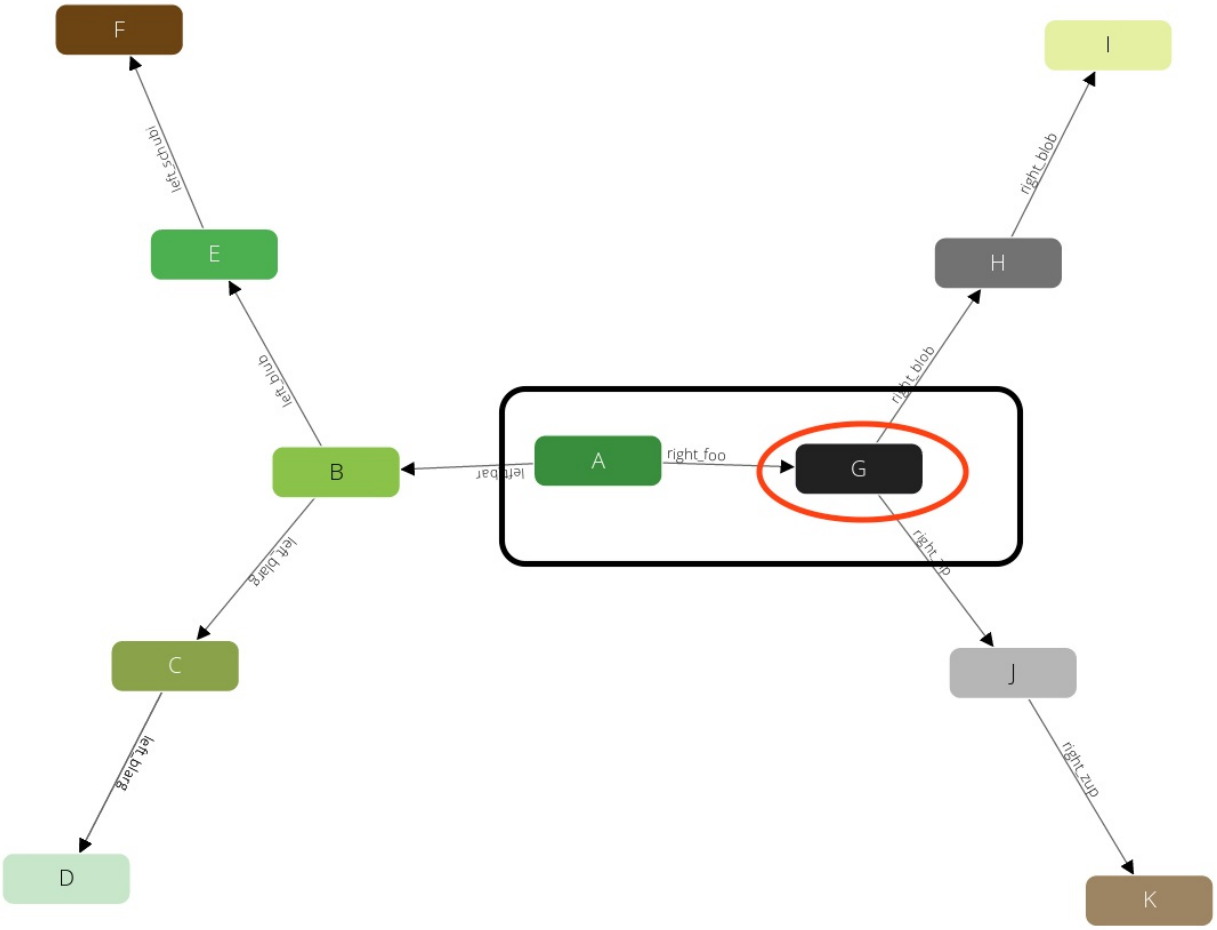
The query will remember the state (red circle) and will emit the first result **A** → **B** (black box). This will also prevent the traverser to be trapped in cycles. Now again it will visit one of the direct neighbors of **B**, say **E**:



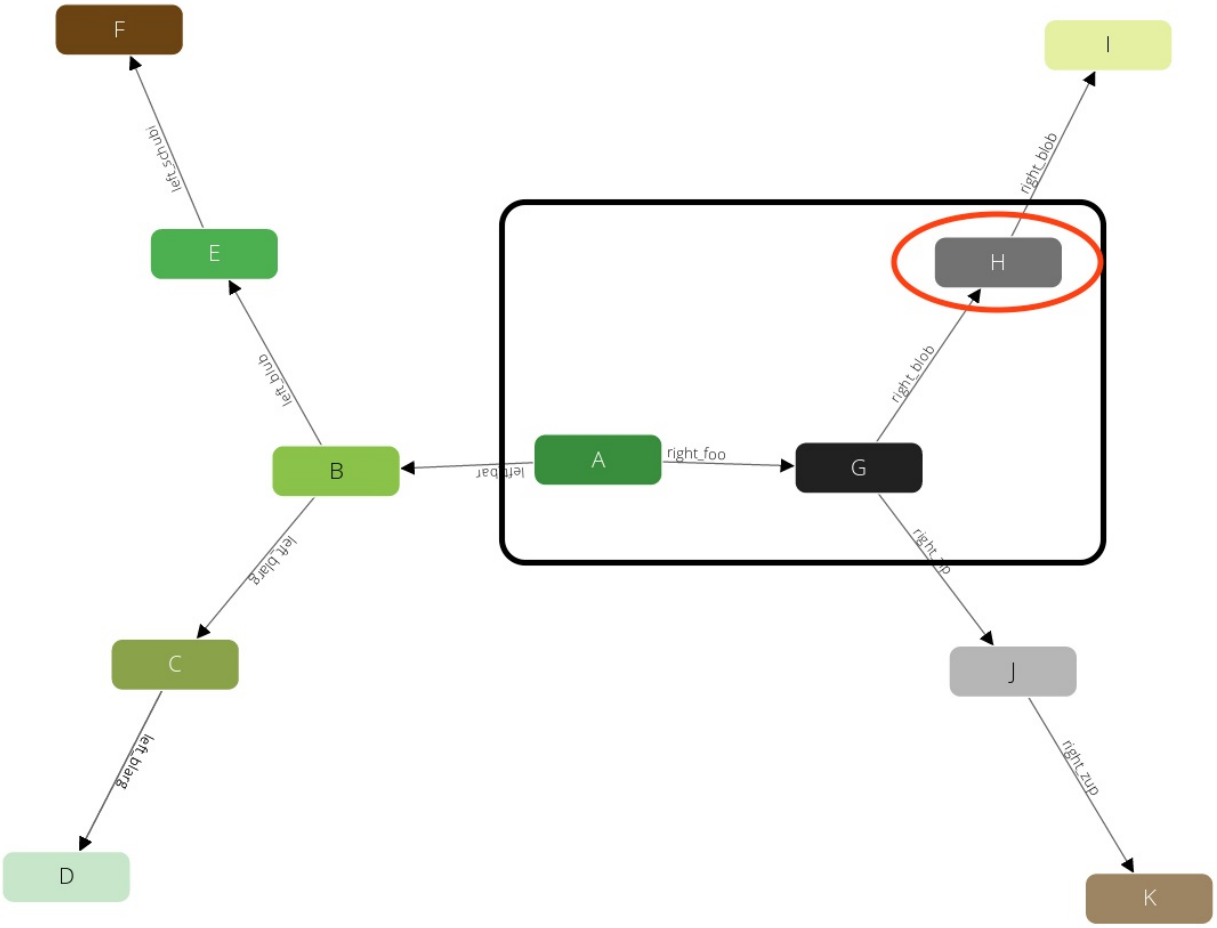
We have limited the query with a *max* depth of *2*, so it will not pick any neighbor of **E**, as the path from **A** to **E** already requires *2* steps. Instead, we will go back one level to **B** and continue with any other direct neighbor there:



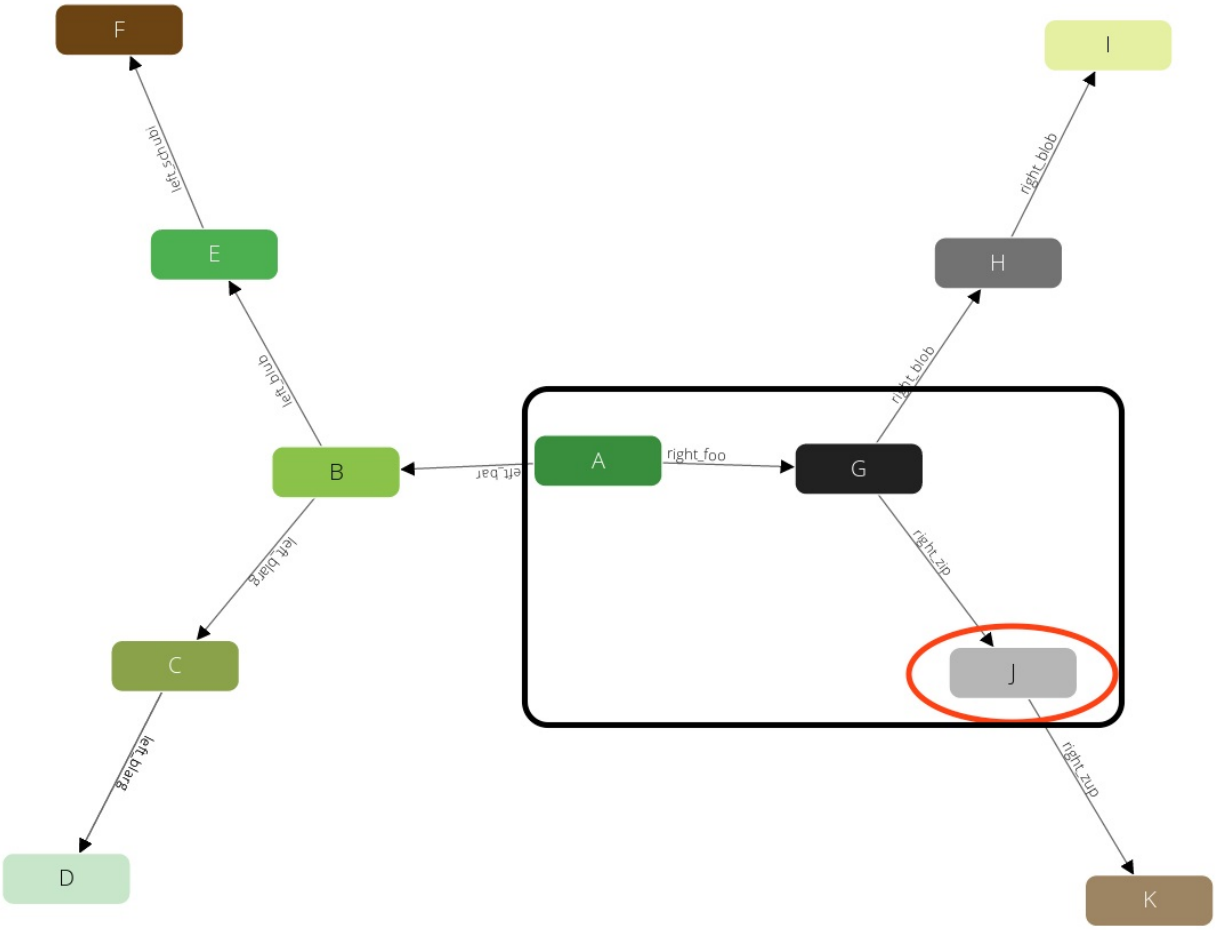
Again after we produced this result we will step back to **B**. But there is no neighbor of **B** left that we have not yet visited. Hence we go another step back to **A** and continue with any other neighbor there.



And identical to the iterations before we will visit **H**:



And **J**:



After these steps there is no further result left. So all together this query has returned the following paths:

1. **A → B**
2. **A → B → E**
3. **A → B → C**
4. **A → G**
5. **A → G → H**
6. **A → G → J**

Syntax

Now let's see how we can write a query that follows this schema. You have two options here, you can either use a named graph or a set of edge collections (anonymous graph).

Working with named graphs

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  GRAPH graphName
  [OPTIONS options]
```

- `FOR` : emits up to three variables:
  - **vertex** (object): the current vertex in a traversal
  - **edge** (object, *optional*): the current edge in a traversal
  - **path** (object, *optional*): representation of the current path with two members:
    - `vertices` : an array of all vertices on this path
    - `edges` : an array of all edges on this path
- `IN` `min..max` : the minimal and maximal depth for the traversal:
  - **min** (number, *optional*): edges and vertices returned by this query will start at the traversal depth of *min* (thus edges and vertices below will not be returned). If not specified, it defaults to 1. The minimal possible value is 0.
  - **max** (number, *optional*): up to *max* length paths are traversed. If omitted, *max* defaults to *min*. Thus only the vertices and edges in the range of *min* are returned. *max* can not be specified without *min*.
- `OUTBOUND|INBOUND|ANY` : follow outgoing, incoming, or edges pointing in either direction in the traversal; Please note that this can't be replaced by a bind parameter.
- **startVertex** (string|object): a vertex where the traversal will originate from. This can be specified in the form of an ID string or in the form of a document with the attribute `_id` . All other values will lead to a warning and an empty result. If the specified document does not exist, the result is empty as well and there is no warning.
- `GRAPH` **graphName** (string): the name identifying the named graph. Its vertex and edge collections will be looked up.
- `OPTIONS` **options** (object, *optional*): used to modify the execution of the traversal. Only the following attributes have an effect, all others are ignored:
  - **uniqueVertices** (string): optionally ensure vertex uniqueness
    - "path" – it is guaranteed that there is no path returned with a duplicate vertex
    - "global" – it is guaranteed that each vertex is visited at most once during the traversal, no matter how many paths lead from the start vertex to this one. If you start with a `min depth > 1` a vertex that was found before *min* depth might not be returned at all (it still might be part of a path). **Note:** Using this configuration the result is not deterministic any more. If there are multiple paths from *startVertex* to *vertex*, one of those is picked.
    - "none" (default) – no uniqueness check is applied on vertices
  - **uniqueEdges** (string): optionally ensure edge uniqueness
    - "path" (default) – it is guaranteed that there is no path returned with a duplicate edge
    - "global" – it is guaranteed that each edge is visited at most once during the traversal, no matter how many paths lead from the start vertex to this edge. If you start with a `min depth > 1` , an edge that was found before *min* depth might not be returned at all (it still might be part of a path). **Note:** Using this configuration the result is not deterministic any more. If there are multiple paths from *startVertex* over *edge* one of those is picked.
    - "none" – no uniqueness check is applied on edges. **Note:** Using this configuration the traversal will follow cycles in edges.
  - **bfs** (bool): optionally use the alternative breadth-first traversal algorithm

- true – the traversal will be executed breadth-first. The results will first contain all vertices at depth 1. Than all vertices at depth 2 and so on.
- false (default) – the traversal will be executed depth-first. It will first return all paths from *min* depth to *max* depth for one vertex at depth 1. Than for the next vertex at depth 1 and so on.

Working with collection sets

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  edgeCollection1, ..., edgeCollectionN
  [OPTIONS options]
```

Instead of `GRAPH graphName` you may specify a list of edge collections. Vertex collections are determined by the edges in the edge collections. The rest of the behavior is similar to the named version. If the same edge collection is specified multiple times, it will behave as if it were specified only once. Specifying the same edge collection is only allowed when the collections do not have conflicting traversal directions.

Traversing in mixed directions

For traversals with a list of edge collections you can optionally specify the direction for some of the edge collections. Say for example you have three edge collections *edges1*, *edges2* and *edges3*, where in *edges2* the direction has no relevance but in *edges1* and *edges3* the direction should be taken into account. In this case you can use *OUTBOUND* as general traversal direction and *ANY* specifically for *edges2* as follows:

```
FOR vertex IN OUTBOUND
  startVertex
  edges1, ANY edges2, edges3
```

All collections in the list that do not specify their own direction will use the direction defined after `IN`. This allows to use a different direction for each collection in your traversal.

Using filters and the explainer to extrapolate the costs

All three variables emitted by the traversals might as well be used in filter statements. For some of these filter statements the optimizer can detect that it is possible to prune paths of traversals earlier, hence filtered results will not be emitted to the variables in the first place. This may significantly improve the performance of your query. Whenever a filter is not fulfilled, the complete set of *vertex*, *edge* and *path* will be skipped. All paths with a length greater than *max* will never be computed.

In the current state, `AND` combined filters can be optimized, but `OR` combined filters cannot.

Filtering on paths

Filtering on paths allows for the most powerful filtering and may have the highest impact on performance. Using the path variable you can filter on specific iteration depths. You can filter for absolute positions in the path by specifying a positive number (which then qualifies for the optimizations), or relative positions to the end of the path by specifying a negative number.

Filtering edges on the path

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.edges[0].theTruth == true
  RETURN p
```

will filter all paths where the start edge (index 0) has the attribute *theTruth* equal to *true*. The resulting paths will be up to 5 items long.

Filtering vertices on the path

Similar to filtering the edges on the path you can also filter the vertices:

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.vertices[1]._key == "G"
  RETURN p
```

Combining several filters

And of course you can combine these filters in any way you like:

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.edges[0].theTruth == true
     AND p.edges[1].theFalse == false
  FILTER p.vertices[1]._key == "G"
  RETURN p
```

The query will filter all paths where the first edge has the attribute *theTruth* equal to *true*, the first vertex is "G" and the second edge has the attribute *theFalse* equal to *false*. The resulting paths will be up to 5 items long.

Note: Although we have defined a *min* of 1, we will only get results of depth 2. This is because for all results in depth 1 the second edge does not exist and hence cannot fulfill the condition here.

Filter on the entire path

With the help of array comparison operators filters can also be defined on the entire path, like ALL edges should have theTruth == true:

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.edges[*].theTruth ALL == true
  RETURN p
```

Or NONE of the edges should have theTruth == true:

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.edges[*].theTruth NONE == true
  RETURN p
```

Both examples above are recognized by the optimizer and can potentially use other indexes than the edge index.

It is also possible to define that at least one edge on the path has to fulfill the condition:

```
FOR v, e, p IN 1..5 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
  FILTER p.edges[*].theTruth ANY == true
  RETURN p
```

It is guaranteed that at least one, but potentially more edges fulfill the condition. All of the above filters can be defined on vertices in the exact same way.

Filtering on the path vs. filtering on vertices or edges

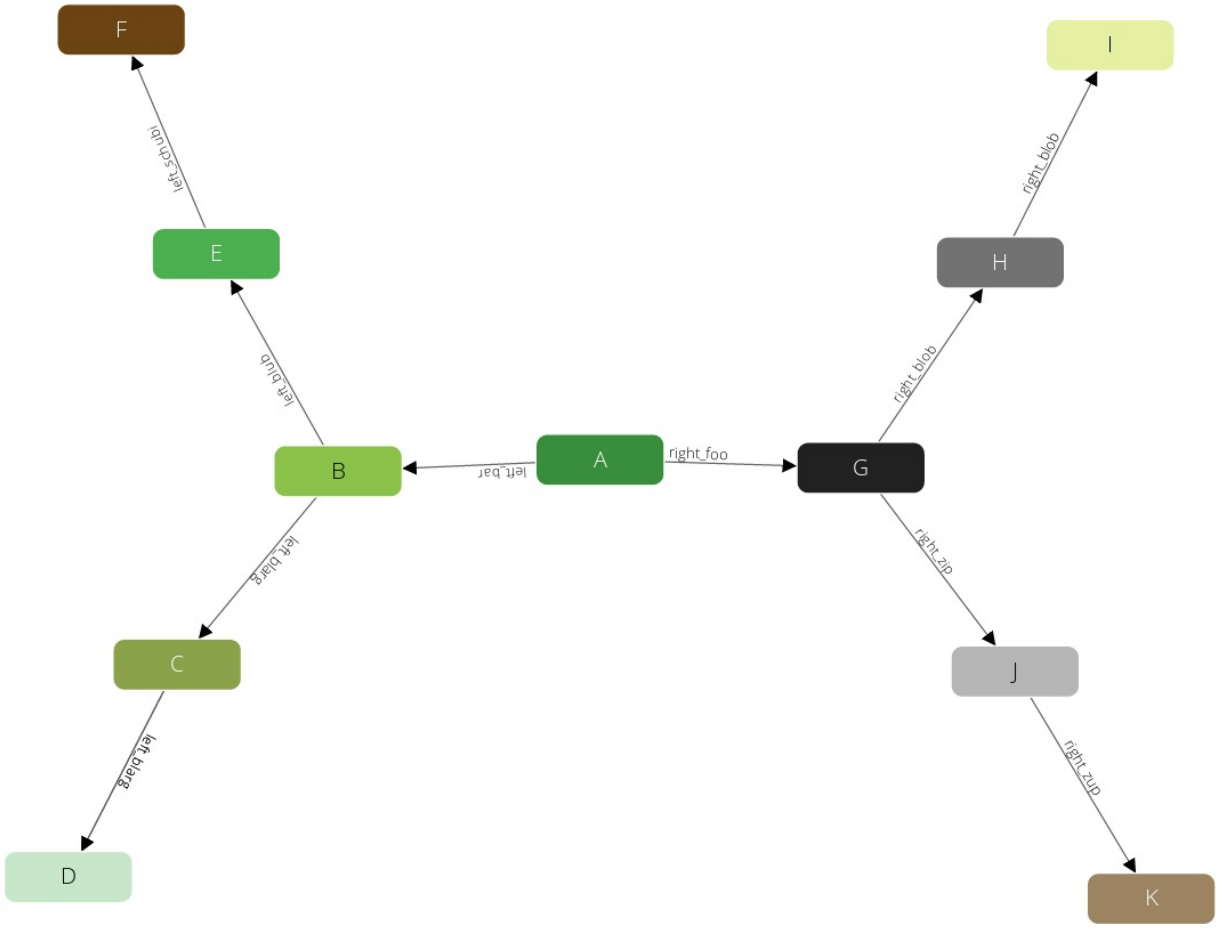
Filtering on the path influences the Iteration on your graph. If certain conditions aren't met, the traversal may stop continuing along this path.

In contrast filters on vertex or edge only express whether you're interestet in the actual value of these documents. Thus, it influences the list of returned documents (if you return v or e) similar as specifying a non-null `min` value. If you specify a min value of 2, the traversal over the first two nodes of these paths has to be executed - you just won't see them in your result array.

Similar are filters on vertices or edges - the traverser has to walk along these nodes, since you may be interested in documents further down the path.

Examples

We will create a simple symmetric traversal demonstration graph:




```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("traversalGraph");
arangosh> db.circles.toArray();
arangosh> db.edges.toArray();
```

show execution results

To get started we select the full graph. For better overview we only return the vertex IDs:

```
arangosh> db._query("FOR v IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph' RETURN
v._key");
arangosh> db._query("FOR v IN 1..3 OUTBOUND 'circles/A' edges RETURN v._key");
```

show execution results

We can nicely see that it is heading for the first outer vertex, then goes back to the branch to descend into the next tree. After that it returns to our start node, to descend again. As we can see both queries return the same result, the first one uses the named graph, the second uses the edge collections directly.

Now we only want the elements of a specific depth (min = max = 2), the ones that are right behind the fork:

```
arangosh> db._query("FOR v IN 2..2 OUTBOUND 'circles/A' GRAPH 'traversalGraph' return
v._key");
arangosh> db._query("FOR v IN 2 OUTBOUND 'circles/A' GRAPH 'traversalGraph' return
v._key");
```

show execution results

As you can see, we can express this in two ways: with or without *max* parameter in the expression.

Filter examples

Now let's start to add some filters. We want to cut of the branch on the right side of the graph, we may filter in two ways:

- we know the vertex at depth 1 has `_key == G`
- we know the `label` attribute of the edge connecting **A** to **G** is `right_foo`

```
arangosh> db._query("FOR v, e, p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
FILTER p.vertices[1]._key != 'G' RETURN v._key");
arangosh> db._query("FOR v, e, p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
FILTER p.edges[0].label != 'right_foo' RETURN v._key");
```

show execution results

As we can see all vertices behind **G** are skipped in both queries. The first filters on the vertex `_key`, the second on an edge label. Note again, as soon as a filter is not fulfilled for any of the three elements `v`, `e` or `p`, the complete set of these will be excluded from the result.

We also may combine several filters, for instance to filter out the right branch (**G**), and the **E** branch:

```
arangosh> db._query("FOR v,e,p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph' FILTER
p.vertices[1]._key != 'G' FILTER p.edges[1].label != 'left_blub' return v._key");
arangosh> db._query("FOR v,e,p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph' FILTER
p.vertices[1]._key != 'G' AND    p.edges[1].label != 'left_blub' return v._key");
```

show execution results

As you can see, combining two `FILTER` statements with an `AND` has the same result.

Comparing OUTBOUND / INBOUND / ANY

All our previous examples traversed the graph in *OUTBOUND* edge direction. You may however want to also traverse in reverse direction (*INBOUND*) or both (*ANY*). Since `circles/A` only has outbound edges, we start our queries from `circles/E`:

```
arangosh> db._query("FOR v IN 1..3 OUTBOUND 'circles/E' GRAPH 'traversalGraph' return
v._key");
arangosh> db._query("FOR v IN 1..3 INBOUND 'circles/E' GRAPH 'traversalGraph' return
v._key");
arangosh> db._query("FOR v IN 1..3 ANY 'circles/E' GRAPH 'traversalGraph' return v._key");
```

show execution results

The first traversal will only walk in the forward (*OUTBOUND*) direction. Therefore from **E** we only can see **F**. Walking in reverse direction (*INBOUND*), we see the path to **A**: **B** → **A**.

Walking in forward and reverse direction (*ANY*) we can see a more diverse result. First of all, we see the simple paths to **F** and **A**. However, these vertices have edges in other directions and they will be traversed.

Note: The traverser may use identical edges multiple times. For instance, if it walks from **E** to **F**, it will continue to walk from **F** to **E** using the same edge once again. Due to this we will see duplicate nodes in the result.

Please also consider to use `WITH` to specify the collections you expect to be involved.

Please note that the direction can't be passed in by a bind parameter.

Use the AQL explainer for optimizations

Now let's have a look what the optimizer does behind the curtain and inspect traversal queries using the explainer:

```
arangosh> db._explain("FOR v,e,p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph' LET
```

```
localScopeVar = RAND() > 0.5 FILTER p.edges[0].theTruth != localScopeVar RETURN v._key",
{}, {colors: false});
arangosh> db._explain("FOR v,e,p IN 1..3 OUTBOUND 'circles/A' GRAPH 'traversalGraph'
FILTER p.edges[0].label == 'right_foo' RETURN v._key", {}, {colors: false});
```

show execution results

We now see two queries: In one we add a variable *localScopeVar*, which is outside the scope of the traversal itself - it is not known inside of the traverser. Therefore, this filter can only be executed after the traversal, which may be undesired in large graphs. The second query on the other hand only operates on the path, and therefore this condition can be used during the execution of the traversal. Paths that are filtered out by this condition won't be processed at all.

And finally clean it up again:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> examples.dropGraph("traversalGraph");
true
```

If this traversal is not powerful enough for your needs, like you cannot describe your conditions as AQL filter statements, then you might want to have a look at manually crafted traversers.

Also see how to combine graph traversals.

Shortest Path in AQL

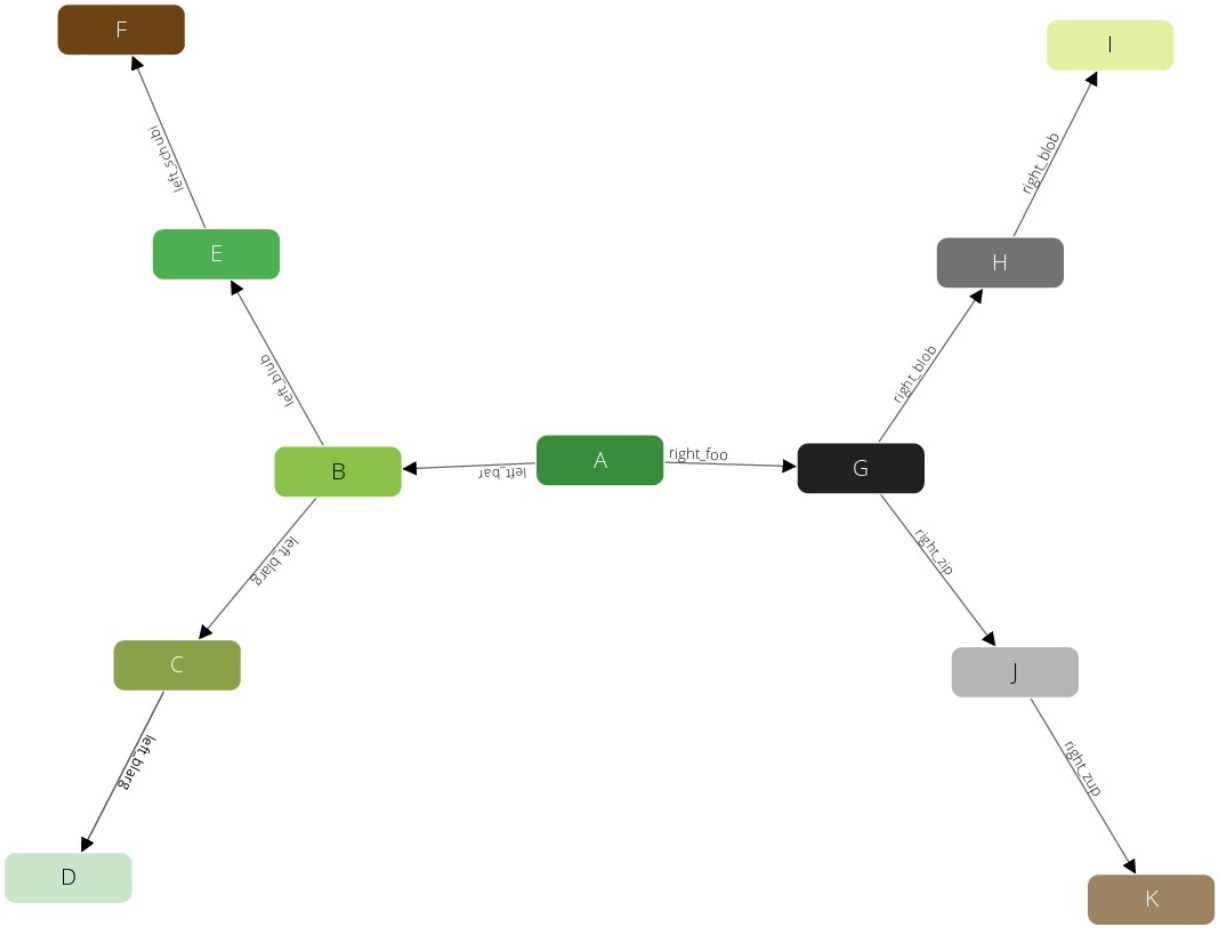
General query idea

This type of query is supposed to find the shortest path between two given documents (*startVertex* and *targetVertex*) in your graph. For all vertices on this shortest path you will get a result in form of a set with two items:

1. The vertex on this path.
2. The edge pointing to it.

Example execution

Let's take a look at a simple example to explain how it works. This is the graph that we are going to find a shortest path on:



Now we use the following parameters for our query:

1. We start at the vertex **A**.
2. We finish with the vertex **D**.

So obviously we will have the vertices **A**, **B**, **C** and **D** on the shortest path in exactly this order. Than the shortest path statement will return the following pairs:

Vertex	Edge
A	null
B	A → B
C	B → C
D	C → D

Note: The first edge will always be `null` because there is no edge pointing to the *startVertex*.

Syntax

Now let's see how we can write a shortest path query. You have two options here, you can either use a named graph or a set of edge collections (anonymous graph).

Working with named graphs

```
FOR vertex[, edge]
  IN OUTBOUND|INBOUND|ANY SHORTEST_PATH
  startVertex TO targetVertex
  GRAPH graphName
  [OPTIONS options]
```

- `FOR` : emits up to two variables:
  - **vertex** (object): the current vertex on the shortest path
  - **edge** (object, *optional*): the edge pointing to the vertex
- `IN` `OUTBOUND|INBOUND|ANY` : defines in which direction edges are followed (outgoing, incoming, or both)
- **startVertex** `TO` **targetVertex** (both string|object): the two vertices between which the shortest path will be computed. This can be specified in the form of an ID string or in the form of a document with the attribute `_id`. All other values will lead to a warning and an empty result. If one of the specified documents does not exist, the result is empty as well and there is no warning.
- `GRAPH` **graphName** (string): the name identifying the named graph. Its vertex and edge collections will be looked up.
- `OPTIONS` **options** (object, *optional*): used to modify the execution of the traversal. Only the following attributes have an effect, all others are ignored:
  - **weightAttribute** (string): a top-level edge attribute that should be used to read the edge weight. If the attribute is not existent or not numeric, the *defaultWeight* will be used instead.
  - **defaultWeight** (number): this value will be used as fallback if there is no *weightAttribute* in the edge document, or if it's not a number. The default is 1.

Working with collection sets

```
FOR vertex[, edge]
  IN OUTBOUND|INBOUND|ANY SHORTEST_PATH
  startVertex TO targetVertex
  edgeCollection1, ..., edgeCollectionN
  [OPTIONS options]
```

Instead of `GRAPH graphName` you may specify a list of edge collections (anonymous graph). The involved vertex collections are determined by the edges of the given edge collections. The rest of the behavior is similar to the named version.

Traversing in mixed directions

For shortest path with a list of edge collections you can optionally specify the direction for some of the edge collections. Say for example you have three edge collections *edges1*, *edges2* and *edges3*, where in *edges2* the direction has no relevance, but in *edges1* and *edges3* the direction should be taken into account. In this case you can use *OUTBOUND* as general search direction and *ANY* specifically for *edges2* as follows:

```
FOR vertex IN OUTBOUND SHORTEST_PATH
  startVertex TO targetVertex
  edges1, ANY edges2, edges3
```

All collections in the list that do not specify their own direction will use the direction defined after *IN* (here: *OUTBOUND*). This allows to use a different direction for each collection in your path search.

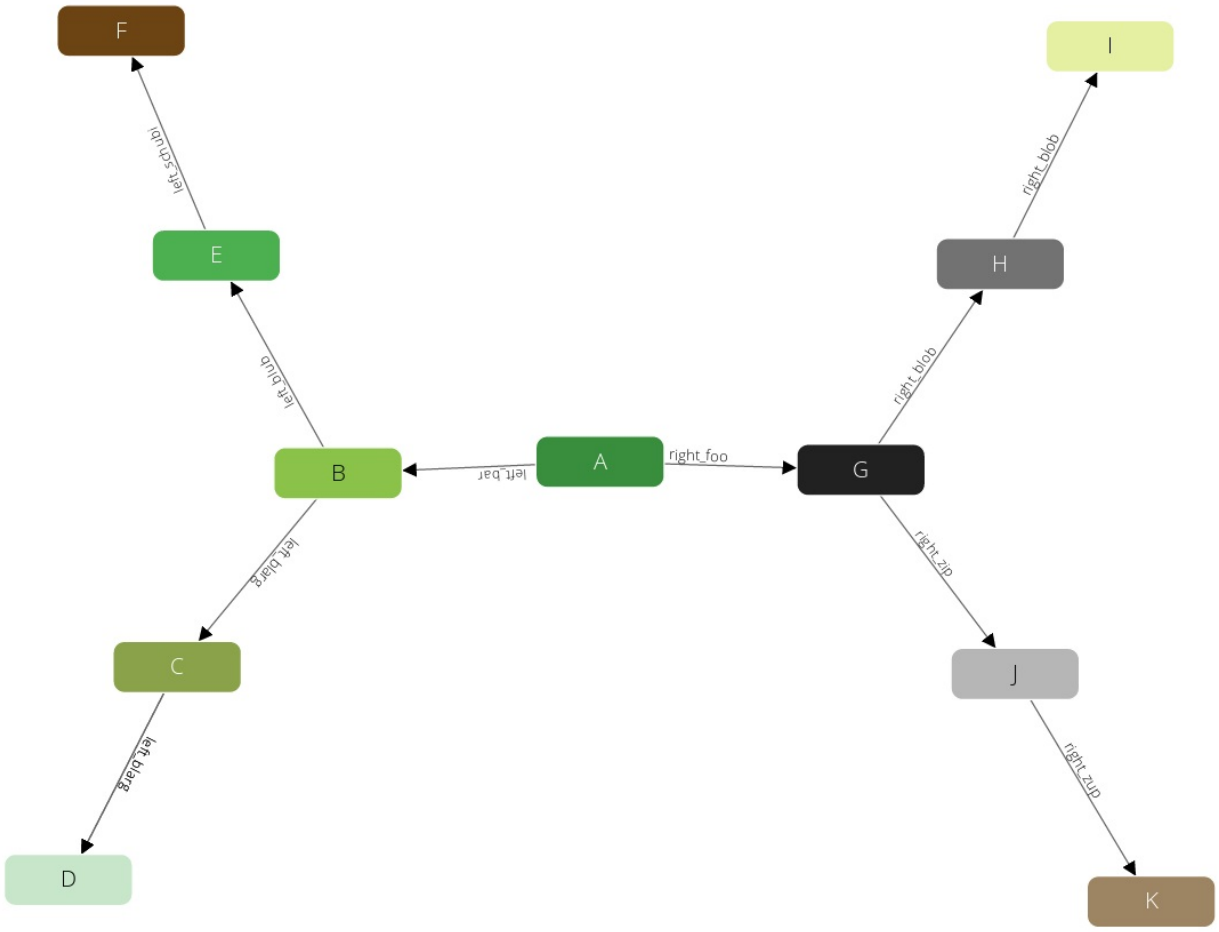
Conditional shortest path

The SHORTEST_PATH computation will only find an unconditioned shortest path. With this construct it is not possible to define a condition like: "Find the shortest path where all edges are of type *X*". If you want to do this, use a normal Traversal instead with the option `{bfs: true}` in combination with `LIMIT 1`.

Please also consider to use `WITH` to specify the collections you expect to be involved.

Examples

We will create a simple symmetric traversal demonstration graph:




```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var graph = examples.loadGraph("traversalGraph");
arangosh> db.circles.toArray();
arangosh> db.edges.toArray();
```

show execution results

We start with the shortest path from **A** to **D** as above:

```
arangosh> db._query("FOR v, e IN OUTBOUND SHORTEST_PATH 'circles/A' TO 'circles/D' GRAPH
'traversalGraph' RETURN [v._key, e._key]");
arangosh> db._query("FOR v, e IN OUTBOUND SHORTEST_PATH 'circles/A' TO 'circles/D' edges
RETURN [v._key, e._key]");
```

show execution results

We can see our expectations are fulfilled. We find the vertices in the correct ordering and the first edge is *null*, because no edge is pointing to the start vertex on t his path.

We can also compute shortest paths based on documents found in collections:

```
arangosh> db._query("FOR a IN circles FILTER a._key == 'A' FOR d IN circles FILTER d._key
== 'D' FOR v, e IN OUTBOUND SHORTEST_PATH a TO d GRAPH 'traversalGraph' RETURN [v._key,
e._key]");
arangosh> db._query("FOR a IN circles FILTER a._key == 'A' FOR d IN circles FILTER d._key
== 'D' FOR v, e IN OUTBOUND SHORTEST_PATH a TO d edges RETURN [v._key, e._key]");
```

show execution results
And finally clean it up again:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> examples.dropGraph("traversalGraph");
true
```

Advanced features

This section covers additional, powerful AQL features, which you may wanna look into once you made yourself familiar with the basics of the query language.

- Array operators: Shorthands for array manipulation

Array Operators

Array expansion

In order to access a named attribute from all elements in an array easily, AQL offers the shortcut operator *[*]* for array variable expansion.

Using the *[*]* operator with an array variable will iterate over all elements in the array, thus allowing to access a particular attribute of each element. It is required that the expanded variable is an array. The result of the *[*]* operator is again an array.

To demonstrate the array expansion operator, let's go on with the following three example *users* documents:

```
[
  {
    name: "john",
    age: 35,
    friends: [
      { name: "tina", age: 43 },
      { name: "helga", age: 52 },
      { name: "alfred", age: 34 }
    ]
  },
  {
    name: "yves",
    age: 24,
    friends: [
      { name: "sergei", age: 27 },
      { name: "tiffany", age: 25 }
    ]
  },
  {
    name: "sandra",
    age: 40,
    friends: [
      { name: "bob", age: 32 },
      { name: "elena", age: 48 }
    ]
  }
]
```

With the *[*]* operator it becomes easy to query just the names of the friends for each user:

```
FOR u IN users
  RETURN { name: u.name, friends: u.friends[*].name }
```

This will produce:

```
[
  { "name" : "john", "friends" : [ "tina", "helga", "alfred" ] },
  { "name" : "yves", "friends" : [ "sergei", "tiffany" ] },
  { "name" : "sandra", "friends" : [ "bob", "elena" ] }
]
```

This is a shortcut for the longer, semantically equivalent query:

```
FOR u IN users
  RETURN { name: u.name, friends: (FOR f IN u.friends RETURN f.name) }
```

Array contraction

In order to collapse (or flatten) results in nested arrays, AQL provides the *[**]* operator. It works similar to the *[*]* operator, but additionally collapses nested arrays.

How many levels are collapsed is determined by the amount of asterisk characters used. *[**]* collapses one level of nesting - just like `FLATTEN(array)` or `FLATTEN(array, 1)` would do -, *[***]* collapses two levels - the equivalent to `FLATTEN(array, 2)` - and so on.

Let's compare the array expansion operator with an array contraction operator. For example, the following query produces an array of friend names per user:

```
FOR u IN users
  RETURN u.friends[*].name
```

As we have multiple users, the overall result is a nested array:

```
[
  [
    "tina",
    "helga",
    "alfred"
  ],
  [
    "sergei",
    "tiffany"
  ],
  [
    "bob",
    "elena"
  ]
]
```

If the goal is to get rid of the nested array, we can apply the *[**]* operator on the result. But simply appending *[**]* to the query won't help, because *u.friends* is not a nested (multi-dimensional) array, but a simple (one-dimensional) array. Still, the *[**]* can be used if it has access to a multi-dimensional nested result.

We can extend above query as follows and still create the same nested result:

```
RETURN (
  FOR u IN users RETURN u.friends[*].name
)
```

By now appending the *[**]* operator at the end of the query...

```
RETURN (
  FOR u IN users RETURN u.friends[*].name
)[**]
```

... the query result becomes:

```
[
  [
    "tina",
    "helga",
    "alfred",
    "sergei",
    "tiffany",
    "bob",
    "elena"
  ]
]
```

Note that the elements are not de-duplicated. For a flat array with only unique elements, a combination of UNIQUE() and FLATTEN() is advisable.

Inline expressions

It is possible to filter elements while iterating over an array, to limit the amount of returned elements and to create a projection using the current array element. Sorting is not supported by this shorthand form.

These inline expressions can follow array expansion and contraction operators *[* ...]*, *[** ...]* etc. The keywords *FILTER*, *LIMIT* and *RETURN* must occur in this order if they are used in combination, and can only occur once:

```
anyArray[* FILTER conditions LIMIT skip,limit RETURN projection]
```

Example with nested numbers and array contraction:

```
LET arr = [ [ 1, 2 ], 3, [ 4, 5 ], 6 ]
RETURN arr[** FILTER CURRENT % 2 == 0]
```

All even numbers are returned in a flat array:

```
[
  [ 2, 4, 6 ]
]
```

Complex example with multiple conditions, limit and projection:

```
FOR u IN users
    RETURN {
        name: u.name,
        friends: u.friends[* FILTER CONTAINS(CURRENT.name, "a") AND CURRENT.age > 40
            LIMIT 2
            RETURN CONCAT(CURRENT.name, " is ", CURRENT.age)
        ]
    }
```

No more than two computed strings based on *friends* with an `a` in their name and older than 40 years are returned per user:

```
[
  {
    "name": "john",
    "friends": [
      "tina is 43",
      "helga is 52"
    ]
  },
  {
    "name": "sandra",
    "friends": [
      "elena is 48"
    ]
  },
  {
    "name": "yves",
    "friends": []
  }
]
```

Inline filter

To return only the names of friends that have an *age* value higher than the user herself, an inline *FILTER* can be used:

```
FOR u IN users
  RETURN { name: u.name, friends: u.friends[* FILTER CURRENT.age > u.age].name }
```

The pseudo-variable *CURRENT* can be used to access the current array element. The *FILTER* condition can refer to *CURRENT* or any variables valid in the outer scope.

Inline limit

The number of elements returned can be restricted with *LIMIT*. It works the same as the limit operation. *LIMIT* must come after *FILTER* and before *RETURN*, if they are present.

```
FOR u IN users
  RETURN { name: u.name, friends: u.friends[* LIMIT 1].name }
```

Above example returns one friend each:

```
[
  { "name": "john", "friends": [ "tina" ] },
  { "name": "sandra", "friends": [ "bob" ] },
  { "name": "yves", "friends": [ "sergei" ] }
]
```

A number of elements can also be skipped and up to *n* returned:

```
FOR u IN users
  RETURN { name: u.name, friends: u.friends[* LIMIT 1,2].name }
```

The example query skips the first friend and returns two friends at most per user:

```
[
  { "name": "john", "friends": [ "helga", "alfred" ] },
  { "name": "sandra", "friends": [ "elena" ] },
  { "name": "yves", "friends": [ "tiffany" ] }
]
```

Inline projection

To return a projection of the current element, use *RETURN*. If a *FILTER* is also present, *RETURN* must come later.

```
FOR u IN users
  RETURN u.friends[* RETURN CONCAT(CURRENT.name, " is a friend of ", u.name)]
```

The above will return:

```
[
  [
    "tina is a friend of john",
    "helga is a friend of john",
    "alfred is a friend of john"
  ],
  [
    "sergei is a friend of yves",
    "tiffany is a friend of yves"
  ],
  [
    "bob is a friend of sandra",
    "elena is a friend of sandra"
  ]
]
```

Usual Query Patterns Examples

Those pages contain some common query patterns with examples. For better understandability the query results are also included directly below each query.

Normally you would want to run queries on data stored in collections. This section will provide several examples for that.

Some of the following example queries are executed on a collection 'users' with the data provided here below.

Things to consider when running queries on collections

Note that all documents created in any collections will automatically get the following server-generated attributes:

- *_id*: A unique id, consisting of collection name and a server-side sequence value
- *_key*: The server sequence value
- *_rev*: The document's revision id

Whenever you run queries on the documents in collections, don't be surprised if these additional attributes are returned as well.

Please also note that with real-world data, you might want to create additional indexes on the data (left out here for brevity). Adding indexes on attributes that are used in *FILTER* statements may considerably speed up queries. Furthermore, instead of using attributes such as *id*, *from* and *to*, you might want to use the built-in *_id*, *_from* and *_to* attributes. Finally, edge collections provide a nice way of establishing references / links between documents. These features have been left out here for brevity as well.

Example data

Some of the following example queries are executed on a collection *users* with the following initial data:

```
[
  { "id": 100, "name": "John", "age": 37, "active": true, "gender": "m" },
  { "id": 101, "name": "Fred", "age": 36, "active": true, "gender": "m" },
  { "id": 102, "name": "Jacob", "age": 35, "active": false, "gender": "m" },
  { "id": 103, "name": "Ethan", "age": 34, "active": false, "gender": "m" },
  { "id": 104, "name": "Michael", "age": 33, "active": true, "gender": "m" },
  { "id": 105, "name": "Alexander", "age": 32, "active": true, "gender": "m" },
  { "id": 106, "name": "Daniel", "age": 31, "active": true, "gender": "m" },
  { "id": 107, "name": "Anthony", "age": 30, "active": true, "gender": "m" },
  { "id": 108, "name": "Jim", "age": 29, "active": true, "gender": "m" },
  { "id": 109, "name": "Diego", "age": 28, "active": true, "gender": "m" },
  { "id": 200, "name": "Sophia", "age": 37, "active": true, "gender": "f" },
  { "id": 201, "name": "Emma", "age": 36,  "active": true, "gender": "f" },
  { "id": 202, "name": "Olivia", "age": 35, "active": false, "gender": "f" },
  { "id": 203, "name": "Madison", "age": 34, "active": true, "gender": "f" },
  { "id": 204, "name": "Chloe", "age": 33, "active": true, "gender": "f" },
  { "id": 205, "name": "Eva", "age": 32, "active": false, "gender": "f" },
  { "id": 206, "name": "Abigail", "age": 31, "active": true, "gender": "f" },
  { "id": 207, "name": "Isabella", "age": 30, "active": true, "gender": "f" },
  { "id": 208, "name": "Mary", "age": 29, "active": true, "gender": "f" },
  { "id": 209, "name": "Mariah", "age": 28, "active": true, "gender": "f" }
]
```

For some of the examples, we'll also use a collection *relations* to store relationships between users. The example data for *relations* are as follows:

```
[
  { "from": 209, "to": 205, "type": "friend" },
  { "from": 206, "to": 108, "type": "friend" },
  { "from": 202, "to": 204, "type": "friend" },
  { "from": 200, "to": 100, "type": "friend" },
  { "from": 205, "to": 101, "type": "friend" },
  { "from": 209, "to": 203, "type": "friend" },
  { "from": 200, "to": 203, "type": "friend" },
  { "from": 100, "to": 208, "type": "friend" },
  { "from": 101, "to": 209, "type": "friend" },
  { "from": 206, "to": 102, "type": "friend" },
```

```
  { "from": 104, "to": 100, "type": "friend" },
  { "from": 104, "to": 108, "type": "friend" },
  { "from": 108, "to": 209, "type": "friend" },
  { "from": 206, "to": 106, "type": "friend" },
  { "from": 204, "to": 105, "type": "friend" },
  { "from": 208, "to": 207, "type": "friend" },
  { "from": 102, "to": 108, "type": "friend" },
  { "from": 207, "to": 203, "type": "friend" },
  { "from": 203, "to": 106, "type": "friend" },
  { "from": 202, "to": 108, "type": "friend" },
  { "from": 201, "to": 203, "type": "friend" },
  { "from": 105, "to": 100, "type": "friend" },
  { "from": 100, "to": 109, "type": "friend" },
  { "from": 207, "to": 109, "type": "friend" },
  { "from": 103, "to": 203, "type": "friend" },
  { "from": 208, "to": 104, "type": "friend" },
  { "from": 105, "to": 104, "type": "friend" },
  { "from": 103, "to": 208, "type": "friend" },
  { "from": 203, "to": 107, "type": "boyfriend" },
  { "from": 107, "to": 203, "type": "girlfriend" },
  { "from": 208, "to": 109, "type": "boyfriend" },
  { "from": 109, "to": 208, "type": "girlfriend" },
  { "from": 106, "to": 205, "type": "girlfriend" },
  { "from": 205, "to": 106, "type": "boyfriend" },
  { "from": 103, "to": 209, "type": "girlfriend" },
  { "from": 209, "to": 103, "type": "boyfriend" },
  { "from": 201, "to": 102, "type": "boyfriend" },
  { "from": 102, "to": 201, "type": "girlfriend" },
  { "from": 206, "to": 100, "type": "boyfriend" },
  { "from": 100, "to": 206, "type": "girlfriend" }
]
```

Counting

Amount of documents in a collection

To return the count of documents that currently exist in a collection, you can call the LENGTH() function:

```
RETURN LENGTH(collection)
```

This type of call is optimized since 2.8 (no unnecessary intermediate result is built up in memory) and it is therefore the prefered way to determine the count. Internally, COLLECTION_COUNT() is called.

In earlier versions with `COLLECT ... WITH COUNT INTO` available (since 2.4), you may use the following code instead of *LENGTH()* for better performance:

```
FOR doc IN collection
    COLLECT WITH COUNT INTO length
    RETURN length
```

Data-modification queries

The following operations can be used to modify data of multiple documents with one query. This is superior to fetching and updating the documents individually with multiple queries. However, if only a single document needs to be modified, ArangoDB's specialized data-modification operations for single documents might execute faster.

Updating documents

To update existing documents, we can either use the *UPDATE* or the *REPLACE* operation. *UPDATE* updates only the specified attributes in the found documents, and *REPLACE* completely replaces the found documents with the specified values.

We'll start with an *UPDATE* query that rewrites the gender attribute in all documents:

```
FOR u IN users
  UPDATE u WITH { gender: TRANSLATE(u.gender, { m: 'male', f: 'female' }) } IN users
```

To add new attributes to existing documents, we can also use an *UPDATE* query. The following query adds an attribute *numberOfLogins* for all users with status active:

```
FOR u IN users
  FILTER u.active == true
  UPDATE u WITH { numberOfLogins: 0 } IN users
```

Existing attributes can also be updated based on their previous value:

```
FOR u IN users
  FILTER u.active == true
  UPDATE u WITH { numberOfLogins: u.numberOfLogins + 1 } IN users
```

The above query will only work if there was already a *numberOfLogins* attribute present in the document. If it is unsure whether there is a *numberOfLogins* attribute in the document, the increase must be made conditional:

```
FOR u IN users
  FILTER u.active == true
  UPDATE u WITH {
    numberOfLogins: HAS(u, 'numberOfLogins') ? u.numberOfLogins + 1 : 1
  } IN users
```

Updates of multiple attributes can be combined in a single query:

```
FOR u IN users
  FILTER u.active == true
  UPDATE u WITH {
    lastLogin: DATE_NOW(),
    numberOfLogins: HAS(u, 'numberOfLogins') ? u.numberOfLogins + 1 : 1
  } IN users
```

Note than an update query might fail during execution, for example because a document to be updated does not exist. In this case, the query will abort at the first error. In single-server mode, all modifications done by the query will be rolled back as if they never happened.

Replacing documents

To not just partially update, but completely replace existing documents, use the *REPLACE* operation. The following query replaces all documents in the collection backup with the documents found in collection users. Documents common to both collections will be replaced. All other documents will remain unchanged. Documents are compared using their *_key* attributes:

```
FOR u IN users
```

```
  REPLACE u IN backup
```

The above query will fail if there are documents in collection users that are not in collection backup yet. In this case, the query would attempt to replace documents that do not exist. If such case is detected while executing the query, the query will abort. In single-server mode, all changes made by the query will also be rolled back.

To make the query succeed for such case, use the *ignoreErrors* query option:

```
FOR u IN users
  REPLACE u IN backup OPTIONS { ignoreErrors: true }
```

Removing documents

Deleting documents can be achieved with the *REMOVE* operation. To remove all users within a certain age range, we can use the following query:

```
FOR u IN users
  FILTER u.active == true && u.age >= 35 && u.age <= 37
  REMOVE u IN users
```

Creating documents

To create new documents, there is the *INSERT* operation. It can also be used to generate copies of existing documents from other collections, or to create synthetic documents (e.g. for testing purposes). The following query creates 1000 test users in collection users with some attributes set:

```
FOR i IN 1..1000
  INSERT {
    id: 100000 + i,
    age: 18 + FLOOR(RAND() * 25),
    name: CONCAT('test', TO_STRING(i)),
    active: false,
    gender: i % 2 == 0 ? 'male' : 'female'
  } IN users
```

Copying data from one collection into another

To copy data from one collection into another, an *INSERT* operation can be used:

```
FOR u IN users
  INSERT u IN backup
```

This will copy over all documents from collection users into collection backup. Note that both collections must already exist when the query is executed. The query might fail if backup already contains documents, as executing the insert might attempt to insert the same document (identified by *_key* attribute) again. This will trigger a unique key constraint violation and abort the query. In single-server mode, all changes made by the query will also be rolled back. To make such copy operation work in all cases, the target collection can be emptied before, using a *REMOVE* query.

Handling errors

In some cases it might be desirable to continue execution of a query even in the face of errors (e.g. "document not found"). To continue execution of a query in case of errors, there is the *ignoreErrors* option.

To use it, place an *OPTIONS* keyword directly after the data modification part of the query, e.g.

```
FOR u IN users
  REPLACE u IN backup OPTIONS { ignoreErrors: true }
```

This will continue execution of the query even if errors occur during the *REPLACE* operation. It works similar for *UPDATE*, *INSERT*, and *REMOVE*.

Altering substructures

To modify lists in documents we have to work with temporary variables. We will collect the sublist in there and alter it. We choose a simple boolean filter condition to make the query better comprehensible.

First lets create a collection with a sample:

```
database = db._create('complexCollection')
database.save({
  "topLevelAttribute" : "a",
  "subList" : [
    {
      "attributeToAlter" : "oldValue",
      "filterByMe" : true
    },
    {
      "attributeToAlter" : "moreOldValues",
      "filterByMe" : true
    },
    {
      "attributeToAlter" : "unchangedValue",
      "filterByMe" : false
    }
  ]
})
```

Heres the Query which keeps the *subList* on *alteredList* to update it later:

```
FOR document in complexCollection
  LET alteredList = (
    FOR element IN document.subList
       LET newItem = (! element.filterByMe ?
                      element :
                      MERGE(element, { attributeToAlter: "shiny New Value" }))
       RETURN newItem)
  UPDATE document WITH { subList:  alteredList } IN complexCollection
```

While the query as it is is now functional:

```
db.complexCollection.toArray()
[
  {
    "_id" : "complexCollection/392671569467",
    "_key" : "392671569467",
    "_rev" : "392799430203",
    "topLevelAttribute" : "a",
    "subList" : [
      {
        "filterByMe" : true,
        "attributeToAlter" : "shiny New Value"
      },
      {
        "filterByMe" : true,
        "attributeToAlter" : "shiny New Value"
      },
      {
        "filterByMe" : false,
        "attributeToAlter" : "unchangedValue"
      }
    ]
  }
]
```

It will probably be soonish a performance bottleneck, since it **modifies** all documents in the collection **regardless whether the values change or not**. Therefore we want to only *UPDATE* the documents if we really change their value. Hence we employ a second *FOR* to test whether *subList* will be altered or not:

```
FOR document in complexCollection
  LET willUpdateDocument = (
    FOR element IN docToAlter.subList
      FILTER element.filterByMe LIMIT 1 RETURN 1)

  FILTER LENGTH(willUpdateDocument) > 0

  LET alteredList = (
    FOR element IN document.subList
       LET newItem = (! element.filterByMe ?
                      element :
                      MERGE(element, { attributeToAlter: "shiny New Value" }))
       RETURN newItem)

  UPDATE document WITH { subList:  alteredList } IN complexCollection
```

Combining queries

Subqueries

Wherever an expression is allowed in AQL, a subquery can be placed. A subquery is a query part that can introduce its own local variables without affecting variables and values in its outer scope(s).

It is required that subqueries be put inside parentheses *(* and *)* to explicitly mark their start and end points:

```
FOR p IN persons
  LET recommendations = (
    FOR r IN recommendations
      FILTER p.id == r.personId
      SORT p.rank DESC
      LIMIT 10
      RETURN r
  )
  RETURN { person : p, recommendations : recommendations }
```

```
FOR p IN persons
  COLLECT city = p.city INTO g
  RETURN {
    city : city,
    numPersons : LENGTH(g),
    maxRating: MAX(
      FOR r IN g
      RETURN r.p.rating
    )}
```

Subqueries may also include other subqueries.

Note that subqueries always return a result **array**, even if there is only a single return value:

```
RETURN ( RETURN 1 )
```

```
[ [ 1 ] ]
```

To avoid such a nested data structure, FIRST() can be used for example:

```
RETURN FIRST( RETURN 1 )
```

```
[ 1 ]
```

Projections and Filters

Returning unaltered documents

To return three complete documents from collection *users*, the following query can be used:

```
FOR u IN users
  LIMIT 0, 3
  RETURN u
```

```
[
  {
    "_id" : "users/229886047207520",
    "_rev" : "229886047207520",
    "_key" : "229886047207520",
    "active" : true,
    "id" : 206,
    "age" : 31,
    "gender" : "f",
    "name" : "Abigail"
  },
  {
    "_id" : "users/229886045175904",
    "_rev" : "229886045175904",
    "_key" : "229886045175904",
    "active" : true,
    "id" : 101,
    "age" : 36,
    "name" : "Fred",
    "gender" : "m"
  },
  {
    "_id" : "users/229886047469664",
    "_rev" : "229886047469664",
    "_key" : "229886047469664",
    "active" : true,
    "id" : 208,
    "age" : 29,
    "name" : "Mary",
    "gender" : "f"
  }
]
```

Note that there is a *LIMIT* clause but no *SORT* clause. In this case it is not guaranteed which of the user documents are returned. Effectively the document return order is unspecified if no *SORT* clause is used, and you should not rely on the order in such queries.

Projections

To return a projection from the collection *users* use a modified *RETURN* instruction:

```
FOR u IN users
  LIMIT 0, 3
  RETURN {
    "user" : {
      "isActive" : u.active ? "yes" : "no",
      "name" : u.name
    }
  }
```

```
[
  {
    "user" : {
      "isActive" : "yes",
      "name" : "John"
    }
```

```
  },
  {
    "user" : {
      "isActive" : "yes",
      "name" : "Anthony"
    }
  },
  {
    "user" : {
      "isActive" : "yes",
      "name" : "Fred"
    }
  }
]
```

Filters

To return a filtered projection from collection *users*, you can use the *FILTER* keyword. Additionally, a *SORT* clause is used to have the result returned in a specific order:

```
FOR u IN users
  FILTER u.active == true && u.age >= 30
  SORT u.age DESC
  LIMIT 0, 5
  RETURN {
    "age" : u.age,
    "name" : u.name
  }
```

```
[
  {
    "age" : 37,
      "name" : "Sophia"
  },
  {
    "age" : 37,
    "name" : "John"
  },
  {
    "age" : 36,
    "name" : "Emma"
  },
  {
    "age" : 36,
    "name" : "Fred"
  },
  {
    "age" : 34,
    "name" : "Madison"
  }
]
```

Joins

So far we have only dealt with one collection (*users*) at a time. We also have a collection *relations* that stores relationships between users. We will now use this extra collection to create a result from two collections.

First of all, we'll query a few users together with their friends' ids. For that, we'll use all *relations* that have a value of *friend* in their *type* attribute. Relationships are established by using the *friendOf* and *thisUser* attributes in the *relations* collection, which point to the *userId* values in the *users* collection.

Join tuples

We'll start with a SQL-ish result set and return each tuple (user name, friends userId) separately. The AQL query to generate such result is:

```
FOR u IN users
  FILTER u.active == true
  LIMIT 0, 4
  FOR f IN relations
    FILTER f.type == "friend" && f.friendOf == u.userId
    RETURN {
      "user" : u.name,
      "friendId" : f.thisUser
    }
```

```
[
  {
    "user" : "Abigail",
    "friendId" : 108
  },
  {
    "user" : "Abigail",
    "friendId" : 102
  },
  {
    "user" : "Abigail",
    "friendId" : 106
  },
  {
    "user" : "Fred",
    "friendId" : 209
  },
  {
    "user" : "Mary",
    "friendId" : 207
  },
  {
    "user" : "Mary",
    "friendId" : 104
  },
  {
    "user" : "Mariah",
    "friendId" : 203
  },
  {
    "user" : "Mariah",
    "friendId" : 205
  }
]
```

We iterate over the collection users. Only the 'active' users will be examined. For each of these users we will search for up to 4 friends. We locate friends by comparing the *userId* of our current user with the *friendOf* attribute of the *relations* document. For each of those relations found we return the users name and the userId of the friend.

Horizontal lists

Note that in the above result, a user can be returned multiple times. This is the SQL way of returning data. If this is not desired, the friends' ids of each user can be returned in a horizontal list. This will return each user at most once.

The AQL query for doing so is:

```
FOR u IN users
  FILTER u.active == true LIMIT 0, 4
  RETURN {
    "user" : u.name,
    "friendIds" : (
      FOR f IN relations
        FILTER f.friendOf == u.userId && f.type == "friend"
        RETURN f.thisUser
    )
  }
```

```
[
  {
    "user" : "Abigail",
    "friendIds" : [
      108,
      102,
      106
    ]
  },
  {
    "user" : "Fred",
    "friendIds" : [
      209
    ]
  },
  {
    "user" : "Mary",
    "friendIds" : [
      207,
      104
    ]
  },
  {
    "user" : "Mariah",
    "friendIds" : [
      203,
      205
    ]
  }
]
```

In this query we are still iterating over the users in the *users* collection and for each matching user we are executing a subquery to create the matching list of related users.

Self joins

To not only return friend ids but also the names of friends, we could "join" the *users* collection once more (something like a "self join"):

```
FOR u IN users
  FILTER u.active == true
  LIMIT 0, 4
  RETURN {
    "user" : u.name,
    "friendIds" : (
      FOR f IN relations
        FILTER f.friendOf == u.userId && f.type == "friend"
        FOR u2 IN users
          FILTER f.thisUser == u2.useId
          RETURN u2.name
    )
  }
```

```
[
```

```
  {
    "user" : "Abigail",
    "friendIds" : [
      "Jim",
      "Jacob",
      "Daniel"
    ]
  },
  {
    "user" : "Fred",
    "friendIds" : [
      "Mariah"
    ]
  },
  {
    "user" : "Mary",
    "friendIds" : [
      "Isabella",
      "Michael"
    ]
  },
  {
    "user" : "Mariah",
    "friendIds" : [
      "Madison",
      "Eva"
    ]
  }
]
```

This query will then again in term fetch the clear text name of the friend from the users collection. So here we iterate the users collection, and for each hit the relations collection, and for each hit once more the users collection.

Outer joins

Lets find the lonely people in our database - those without friends.

```
FOR user IN users
  LET friendList = (
    FOR f IN relations
      FILTER f.friendOf == u.userId
      RETURN 1
  )
  FILTER LENGTH(friendList) == 0
  RETURN { "user" : user.name }
```

```
[
  {
    "user" : "Abigail"
  },
  {
    "user" : "Fred"
  }
]
```

So, for each user we pick the list of their friends and count them. The ones where count equals zero are the lonely people. Using *RETURN 1* in the subquery saves even more precious CPU cycles and gives the optimizer more alternatives.

Index usage

Especially on joins you should make sure indices can be used to speed up your query. Please note that sparse indices don't qualify for joins:

In joins you typically would also want to join documents not containing the property you join with. However sparse indices don't contain references to documents that don't contain the indexed attributes - thus they would be missing from the join operation. For that reason you should provide non-sparse indices.

Pitfalls

Since we're free of schemata, there is by default no way to tell the format of the documents. So, if your documents don't contain an attribute, it defaults to null. We can however check our data for accuracy like this:

```
RETURN LENGTH(FOR u IN users FILTER u.userId == null RETURN 1)
```

```
[
  10000
]
```

```
RETURN LENGTH(FOR f IN relations FILTER f.friendOf == null RETURN 1)
```

```
[
  10000
]
```

So if the above queries return 10k matches each, the result of the Join tuples query will become 100,000,000 items larger and use much memory plus computation time. So it is generally a good idea to revalidate that the criteria for your join conditions exist.

Using indices on the properties can speed up the operation significantly. You can use the explain helper to revalidate your query actually uses them.

If you work with joins on edge collections you would typically aggregate over the internal fields *_id*, *_from* and *_to* (where *_id* equals *userId*, *_from friendOf* and *_to* would be *thisUser* in our examples). ArangoDB implicitly creates indices on them.

Grouping

To group results by arbitrary criteria, AQL provides the *COLLECT* keyword. *COLLECT* will perform a grouping, but no aggregation. Aggregation can still be added in the query if required.

Ensuring uniqueness

COLLECT can be used to make a result set unique. The following query will return each distinct `age` attribute value only once:

```
FOR u IN users
    COLLECT age = u.age
    RETURN age
```

This is grouping without tracking the group values, but just the group criterion (*age*) value.

Grouping can also be done on multiple levels using *COLLECT*:

```
FOR u IN users
    COLLECT status = u.status, age = u.age
    RETURN { status, age }
```

Alternatively *RETURN DISTINCT* can be used to make a result set unique. *RETURN DISTINCT* supports a single criterion only:

```
FOR u IN users
    RETURN DISTINCT u.age
```

Note: the order of results is undefined for *RETURN DISTINCT*.

Fetching group values

To group users by age, and return the names of the users with the highest ages, we'll issue a query like this:

```
FOR u IN users
    FILTER u.active == true
    COLLECT age = u.age INTO usersByAge
    SORT age DESC LIMIT 0, 5
    RETURN {
        age,
        users: usersByAge[*].u.name
    }
```

```
[
  { "age": 37, "users": [ "John", "Sophia" ] },
  { "age": 36, "users": [ "Fred", "Emma" ] },
  { "age": 34, "users": [ "Madison" ] },
  { "age": 33, "users": [ "Chloe", "Michael" ] },
  { "age": 32, "users": [ "Alexander" ] }
]
```

The query will put all users together by their *age* attribute. There will be one result document per distinct *age* value (let aside the *LIMIT*). For each group, we have access to the matching document via the *usersByAge* variable introduced in the *COLLECT* statement.

Variable Expansion

The *usersByAge* variable contains the full documents found, and as we're only interested in user names, we'll use the expansion operator *[*]* to extract just the *name* attribute of all user documents in each group:

```
usersByAge[*].u.name
```

The *[*]* expansion operator is just a handy short-cut. We could also write a subquery:

```
( FOR temp IN usersByAge RETURN temp.u.name )
```

Grouping by multiple criteria

To group by multiple criteria, we'll use multiple arguments in the *COLLECT* clause. For example, to group users by *ageGroup* (a derived value we need to calculate first) and then by *gender*, we'll do:

```
FOR u IN users
    FILTER u.active == true
    COLLECT ageGroup = FLOOR(u.age / 5) * 5,
            gender = u.gender INTO group
    SORT ageGroup DESC
    RETURN {
        ageGroup,
        gender
    }
```

```
[
  { "ageGroup": 35, "gender": "f" },
  { "ageGroup": 35, "gender": "m" },
  { "ageGroup": 30, "gender": "f" },
  { "ageGroup": 30, "gender": "m" },
  { "ageGroup": 25, "gender": "f" },
  { "ageGroup": 25, "gender": "m" }
]
```

Counting group values

If the goal is to count the number of values in each group, AQL provides the special *COLLECT WITH COUNT INTO* syntax. This is a simple variant for grouping with an additional group length calculation:

```
FOR u IN users
    FILTER u.active == true
    COLLECT ageGroup = FLOOR(u.age / 5) * 5,
            gender = u.gender WITH COUNT INTO numUsers
    SORT ageGroup DESC
    RETURN {
        ageGroup,
        gender,
        numUsers
    }
```

```
[
  { "ageGroup": 35, "gender": "f", "numUsers": 2 },
  { "ageGroup": 35, "gender": "m", "numUsers": 2 },
  { "ageGroup": 30, "gender": "f", "numUsers": 4 },
  { "ageGroup": 30, "gender": "m", "numUsers": 4 },
  { "ageGroup": 25, "gender": "f", "numUsers": 2 },
  { "ageGroup": 25, "gender": "m", "numUsers": 2 }
]
```

Aggregation

Adding further aggregation is also simple in AQL by using an *AGGREGATE* clause in the *COLLECT*:

```
FOR u IN users
    FILTER u.active == true
    COLLECT ageGroup = FLOOR(u.age / 5) * 5,
            gender = u.gender
    AGGREGATE numUsers = LENGTH(1),
              minAge = MIN(u.age),
              maxAge = MAX(u.age)
    SORT ageGroup DESC
```

```
    RETURN {
        ageGroup,
        gender,
        numUsers,
        minAge,
        maxAge
    }
```

```
[
  {
    "ageGroup": 35,
    "gender": "f",
    "numUsers": 2,
    "minAge": 36,
    "maxAge": 39,
  },
  {
    "ageGroup": 35,
    "gender": "m",
    "numUsers": 2,
    "minAge": 35,
    "maxAge": 39,
  },
  ...
]
```

We have used the aggregate functions *LENGTH* here (it returns the length of a array). This is the equivalent to SQL's `SELECT g, COUNT(*) FROM ... GROUP BY g` . In addition to *LENGTH* AQL also provides *MAX*, *MIN*, *SUM* and *AVERAGE*, *VARIANCE_POPULATION*, *VARIANCE_SAMPLE*, *STDDEV_POPULATION* and *STDDEV_SAMPLE* as basic aggregation functions.

In AQL all aggregation functions can be run on arrays only. If an aggregation function is run on anything that is not an array, a warning will be produced and the result will be *null*.

Using an *AGGREGATE* clause will ensure the aggregation is run while the groups are built in the collect operation. This is normally more efficient than collecting all group values for all groups and then doing a post-aggregation.

Post-aggregation

Aggregation can also be performed after a *COLLECT* operation using other AQL constructs, though performance-wise this is often inferior to using *COLLECT* with *AGGREGATE*.

The same query as before can be turned into a post-aggregation query as shown below. Note that this query will build and pass on all group values for all groups inside the variable *g*, and perform the aggregation at the latest possible stage:

```
FOR u IN users
    FILTER u.active == true
    COLLECT ageGroup = FLOOR(u.age / 5) * 5,
            gender = u.gender INTO g
    SORT ageGroup DESC
    RETURN {
        ageGroup,
        gender,
        numUsers: LENGTH(g[*]),
        minAge: MIN(g[*].u.age),
        maxAge: MAX(g[*].u.age)
    }
```

```
[
  {
    "ageGroup": 35,
    "gender": "f",
    "numUsers": 2,
    "minAge": 36,
    "maxAge": 39,
  },
  {
    "ageGroup": 35,
    "gender": "m",
```

```
    "numUsers": 2,
    "minAge": 35,
    "maxAge": 39,
  },
  ...
]
```

This is in constrast to the previous query that used an *AGGREGATE* clause to perform the aggregation during the collect operation, at the earliest possible stage.

Post-filtering aggregated data

To filter the results of a grouping or aggregation operation (i.e. something similar to *HAVING* in SQL), simply add another *FILTER* clause after the *COLLECT* statement.

For example, to get the 3 *ageGroup*s with the most users in them:

```
FOR u IN users
    FILTER u.active == true
    COLLECT ageGroup = FLOOR(u.age / 5) * 5 INTO group
    LET numUsers = LENGTH(group)
    FILTER numUsers > 2 /* group must contain at least 3 users in order to qualify */
    SORT numUsers DESC
    LIMIT 0, 3
    RETURN {
        "ageGroup": ageGroup,
        "numUsers": numUsers,
        "users": group[*].u.name
    }
```

```
[
  {
    "ageGroup": 30,
    "numUsers": 8,
    "users": [
      "Abigail",
      "Madison",
      "Anthony",
      "Alexander",
      "Isabella",
      "Chloe",
      "Daniel",
      "Michael"
    ]
  },
  {
    "ageGroup": 25,
    "numUsers": 4,
    "users": [
      "Mary",
      "Mariah",
      "Jim",
      "Diego"
    ]
  },
  {
    "ageGroup": 35,
    "numUsers": 4,
    "users": [
      "Fred",
      "John",
      "Emma",
      "Sophia"
    ]
  }
]
```

To increase readability, the repeated expression *LENGTH(group)* was put into a variable *numUsers*. The *FILTER* on *numUsers* is the equivalent an SQL *HAVING* clause.

Combining Graph Traversals

Finding the start vertex via a geo query

Our first example will locate the start vertex for a graph traversal via a geo index. We use the city graph and its geo indices:

```
arangosh> var examples = require("@arangodb/graph-examples/example-graph.js");
arangosh> var g = examples.loadGraph("routeplanner");
arangosh> var bonn=[50.7340, 7.0998];
arangosh> db._query(`FOR startCity IN
........>             WITHIN(germanCity, @lat, @long, @radius)
........>               RETURN startCity`,
........>   {lat: bonn[0], long: bonn[1], radius: 400000}
........> ).toArray()
```

show execution results

We search all german cities in a range of 400 km around the ex-capital **Bonn**: **Hamburg** and **Cologne**. We won't find **Paris** since its in the `frenchCity` collection.

```
arangosh> db._query(`FOR startCity IN
........>             WITHIN(germanCity, @lat, @long, @radius)
........>               FOR v, e, p IN 1..1 OUTBOUND startCity
........>                 GRAPH 'routeplanner'
........>     RETURN {startcity: startCity._key, traversedCity: v}`,
........> {
........>  lat: bonn[0],
........>  long: bonn[1],
........>  radius: 400000
........> } ).toArray()
```

show execution results

The geo index query returns us `startCity` (**Cologne** and **Hamburg**) which we then use as starting point for our graph traversal. For simplicity we only return their direct neighbours. We format the return result so we can see from which `startCity` the traversal came.

Alternatively we could use a `LET` statement with a subquery to group the traversals by their `startCity` efficiently:

```
arangosh> db._query(`FOR startCity IN
........>            WITHIN(germanCity, @lat, @long, @radius)
........>              LET oneCity = (FOR v, e, p IN 1..1 OUTBOUND startCity
........>                GRAPH 'routeplanner' RETURN v)
........>              return {startCity: startCity._key, connectedCities: oneCity}`,
........> {
........>  lat: bonn[0],
........>  long: bonn[1],
........>  radius: 400000
........> } ).toArray();
```

show execution results

Finally, we clean up again:

```
arangosh> examples.dropGraph("routeplanner");
true
```

Queries without collections

Following is a query that returns a string value. The result string is contained in an array because the result of every valid query is an array:

```
RETURN "this will be returned"
[
  "this will be returned"
]
```

Here is a query that creates the cross products of two arrays and runs a projection on it, using a few of AQL's built-in functions:

```
FOR year in [ 2011, 2012, 2013 ]
  FOR quarter IN [ 1, 2, 3, 4 ]
    RETURN {
      "y" : "year",
      "q" : quarter,
      "nice" : CONCAT(quarter, "/", year)
    }
[
  { "y" : "year", "q" : 1, "nice" : "1/2011" },
  { "y" : "year", "q" : 2, "nice" : "2/2011" },
  { "y" : "year", "q" : 3, "nice" : "3/2011" },
  { "y" : "year", "q" : 4, "nice" : "4/2011" },
  { "y" : "year", "q" : 1, "nice" : "1/2012" },
  { "y" : "year", "q" : 2, "nice" : "2/2012" },
  { "y" : "year", "q" : 3, "nice" : "3/2012" },
  { "y" : "year", "q" : 4, "nice" : "4/2012" },
  { "y" : "year", "q" : 1, "nice" : "1/2013" },
  { "y" : "year", "q" : 2, "nice" : "2/2013" },
  { "y" : "year", "q" : 3, "nice" : "3/2013" },
  { "y" : "year", "q" : 4, "nice" : "4/2013" }
]
```

Extending AQL with User Functions

AQL comes with a built-in set of functions, but it is not a fully-featured programming language.

To add missing functionality or to simplify queries, users may add their own functions to AQL in the selected database. These functions are written in JavaScript, and are deployed via an API; see Registering Functions.

In order to avoid conflicts with existing or future built-in function names, all user defined functions (**UDF**) have to be put into separate namespaces. Invoking a UDF is then possible by referring to the fully-qualified function name, which includes the namespace, too; see Conventions.

Technical Details

Known Limitations

UDFs have some implications you should be aware of. Otherwise they can introduce serious effects on the performance of your queries and the resource usage in ArangoDB.

Since the optimizer doesn't know anything about the nature of your function, **the optimizer can't use indices for UDFs**. So you should never lean on a UDF as the primary criterion for a `FILTER` statement to reduce your query result set. Instead, put a another `FILTER` statement in front of it. You should make sure that this `FILTER` **statement** is effective to reduce the query result before passing it to your UDF.

Rule of thumb is, the closer the UDF is to your final `RETURN` statement (or maybe even inside it), the better.

When used in clusters, UDFs are always executed on the coordinator.

Using UDFs in clusters may result in a higher resource allocation in terms of used V8 contexts and server threads. If you run out of these resources, your query may abort with a **cluster backend unavailable** error.

To overcome these mentioned limitations, you may want to increase the number of available V8 contexts (at the expense of increased memory usage), and the number of available server threads.

Deployment Details

Internally, UDFs are stored in a system collection named `_aqlfunctions` of the selected database. When an AQL statement refers to such a UDF, it is loaded from that collection. The UDFs will be exclusively available for queries in that particular database.

Since the coordinator doesn't have own local collections, the `_aqlfunctions` collection is sharded across the cluster. Therefore (as usual), it has to be accessed through a coordinator - you mustn't talk to the shards directly. Once it is in the `_aqlfunctions` collection, it is available on all coordinators without additional effort.

Keep in mind that system collections are excluded from dumps created with arangodump by default. To include AQL UDF in a dump, the dump needs to be started with the option *--include-system-collections true*.

Conventions

Naming

Built-in AQL functions that are shipped with ArangoDB reside in the namespace *_aql*, which is also the default namespace to look in if an unqualified function name is found.

To refer to a user-defined AQL function, the function name must be fully qualified to also include the user-defined namespace. The *::* symbol is used as the namespace separator. Users can create a multi-level hierarchy of function groups if required:

```
MYGROUP::MYFUNC()
MYFUNCTIONS::MATH::RANDOM()
```

Note: Adding user functions to the *_aql* namespace is disallowed and will fail.

User function names are case-insensitive like all function names in AQL.

Variables and side effects

User functions can take any number of input arguments and should provide one result via a `return` statement. User functions should be kept purely functional and thus free of side effects and state, and state modification.

Modification of global variables is unsupported, as is changing the data of any collection from inside an AQL user function.

User function code is late-bound, and may thus not rely on any variables that existed at the time of declaration. If user function code requires access to any external data, it must take care to set up the data by itself.

All AQL user function-specific variables should be introduced with the `var` keyword in order to not accidentally access already defined variables from outer scopes. Not using the `var` keyword for own variables may cause side effects when executing the function.

Here is an example that may modify outer scope variables `i` and `name`, making the function **not** side-effect free:

```
function (values) {
  for (i = 0; i < values.length; ++i) {
    name = values[i];
    if (name === "foo") {
      return i;
    }
  }
  return null;
}
```

The above function can be made free of side effects by using the `var` or `let` keywords, so the variables become function-local variables:

```
function (values) {
  for (var i = 0; i < values.length; ++i) {
    var name = values[i];
    if (name === "foo") {
      return i;
    }
  }
  return null;
}
```

Input parameters

In order to return a result, a user function should use a `return` instruction rather than modifying its input parameters.

AQL user functions are allowed to modify their input parameters for input parameters that are null, boolean, numeric or string values. Modifying these input parameter types inside a user function should be free of side effects. However, user functions should not modify input parameters if the parameters are arrays or objects and as such passed by reference, as that may modify variables and state outside of the user function itself.

Return values

User functions must only return primitive types (i.e. *null*, boolean values, numeric values, string values) or aggregate types (arrays or objects) composed of these types. Returning any other JavaScript object type (Function, Date, RegExp etc.) from a user function may lead to undefined behavior and should be avoided.

Enforcing strict mode

By default, any user function code will be executed in *sloppy mode*, not *strict* or *strong mode*. In order to make a user function run in strict mode, use `"use strict"` explicitly inside the user function, e.g.:

```
function (values) {
  "use strict"

  for (var i = 0; i < values.length; ++i) {
    var name = values[i];
    if (name === "foo") {
      return i;
    }
  }
  return null;
}
```

Any violation of the strict mode will trigger a runtime error.

Registering and Unregistering User Functions

AQL user functions can be registered in the selected database using the *aqlfunctions* object as follows:

```
var aqlfunctions = require("@arangodb/aql/functions");
```

To register a function, the fully qualified function name plus the function code must be specified. This can easily be done in arangosh. The HTTP Interface also offers User Functions management.

Documents in the *_aqlfunctions* collection (or any other system collection) should not be accessed directly, but only via the dedicated interfaces. Otherwise you might see caching issues or accidentally break something. The interfaces will ensure the correct format of the documents and invalidate the UDF cache.

Registering an AQL user function

For testing, it may be sufficient to directly type the function code in the shell. To manage more complex code, you may write it in the code editor of your choice and save it as file. For example:

```
/* path/to/file.js */
'use strict';

function greeting(name) {
    if (name === undefined) {
        name = "World";
    }
    return `Hello ${name}!`;
}

module.exports = greeting;
```

Then require it in the shell in order to register a user-defined function:

```
arangosh> var func = require("path/to/file.js");
arangosh> aqlfunctions.register("HUMAN::GREETING", func, true);
```

Note that a return value of *false* means that the function `HUMAN::GREETING` was newly created, and not that it failed to register. *true* is returned if a function of that name existed before and was just updated.

`aqlfunctions.register(name, code, isDeterministic)`

Registers an AQL user function, identified by a fully qualified function name. The function code in *code* must be specified as a JavaScript function or a string representation of a JavaScript function. If the function code in *code* is passed as a string, it is required that the string evaluates to a JavaScript function definition.

If a function identified by *name* already exists, the previous function definition will be updated. Please also make sure that the function code does not violate the Conventions for AQL functions.

The *isDeterministic* attribute can be used to specify whether the function results are fully deterministic (i.e. depend solely on the input and are the same for repeated calls with the same input values). It is not used at the moment but may be used for optimizations later.

The registered function is stored in the selected database's system collection *_aqlfunctions*.

The function returns *true* when it updates/replaces an existing AQL function of the same name, and *false* otherwise. It will throw an exception when it detects syntactially invalid function code.

Examples

```
require("@arangodb/aql/functions").register("MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT",
function (celsius) {
  return celsius * 1.8 + 32;
});
```

The function code will not be executed in *strict mode* or *strong mode* by default. In order to make a user function being run in strict mode, use `use strict` explicitly, e.g.:

```
require("@arangodb/aql/functions").register("MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT",
function (celsius) {
  "use strict";
  return celsius * 1.8 + 32;
});
```

You can access the name under which the AQL function is registered by accessing the `name` property of `this` inside the JavaScript code:

```
require("@arangodb/aql/functions").register("MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT",
function (celsius) {
  "use strict";
  if (typeof celsius === "undefined") {
    const error = require("@arangodb").errors.ERROR_QUERY_FUNCTION_ARGUMENT_NUMBER_MISMATCH;
    AQL_WARNING(error.code, require("util").format(error.message, this.name, 1, 1));
  }
  return celsius * 1.8 + 32;
});
```

`AQL_WARNING()` is automatically available to the code of user-defined functions. The error code and message is retrieved via `@arangodb` module. The *argument number mismatch* message has placeholders, which we can substitute using format():

```
invalid number of arguments for function '%s()', expected number of arguments: minimum: %d, maximum: %d
```

In the example above, `%s` is replaced by `this.name` (the AQL function name), and both `%d` placeholders by `1` (number of expected arguments). If you call the function without an argument, you will see this:

```
arangosh> db._query("RETURN MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT()")
[object ArangoQueryCursor, count: 1, hasMore: false, warning: 1541 - invalid
number of arguments for function 'MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT()',
expected number of arguments: minimum: 1, maximum: 1]

[
  null
]
```

Deleting an existing AQL user function

`aqlfunctions.unregister(name)`

Unregisters an existing AQL user function, identified by the fully qualified function name.

Trying to unregister a function that does not exist will result in an exception.

Examples

```
require("@arangodb/aql/functions").unregister("MYFUNCTIONS::TEMPERATURE::CELSIUSTOFAHRENHEIT");
```

Unregister Group

delete a group of AQL user functions `aqlfunctions.unregisterGroup(prefix)`

Unregisters a group of AQL user function, identified by a common function group prefix.

This will return the number of functions unregistered.

Examples

```
require("@arangodb/aql/functions").unregisterGroup("MYFUNCTIONS::TEMPERATURE");

require("@arangodb/aql/functions").unregisterGroup("MYFUNCTIONS");
```

Listing all AQL user functions

`aqlfunctions.toArray()`

Returns all previously registered AQL user functions, with their fully qualified names and function code.

The result may optionally be restricted to a specified group of functions by specifying a group prefix:

`aqlfunctions.toArray(prefix)`

Examples

To list all available user functions:

```
require("@arangodb/aql/functions").toArray();
```

To list all available user functions in the *MYFUNCTIONS* namespace:

```
require("@arangodb/aql/functions").toArray("MYFUNCTIONS");
```

To list all available user functions in the *MYFUNCTIONS::TEMPERATURE* namespace:

```
require("@arangodb/aql/functions").toArray("MYFUNCTIONS::TEMPERATURE");
```

AQL Execution and Performance

This chapter describes AQL features related to query executions and query performance.

- Execution statistics: A query that has been executed also returns statistics about its execution.
- Query parsing: Clients can use ArangoDB to check if a given AQL query is syntactically valid.
- Query execution plan: If it is unclear how a given query will perform, clients can retrieve a query's execution plan from the AQL query optimizer without actually executing the query; this is called explaining.
- The AQL query optimizer: AQL queries are sent through an optimizer before execution. The task of the optimizer is to create an initial execution plan for the query, look for optimization opportunities and apply them.
- The AQL query result cache: an optional query result cache is used to avoid repeated calculation of the same query results.

Query statistics

A query that has been executed will always return execution statistics. Execution statistics can be retrieved by calling `getExtra()` on the cursor. The statistics are returned in the return value's `stats` attribute:

```
arangosh> db._query(`
........>   FOR i IN 1..@count INSERT
........>     { _key: CONCAT('anothertest', TO_STRING(i)) }
........>     INTO mycollection`,
........>  {count: 100},
........>  {},
........>  {fullCount: true}
........> ).getExtra();
arangosh> db._query({
........>  "query": `FOR i IN 200..@count INSERT
........>              { _key: CONCAT('anothertest', TO_STRING(i)) }
........>              INTO mycollection`,
........>  "bindVars": {count: 300},
........>  "options": { fullCount: true}
........> }).getExtra();
```

show execution results

The meaning of the statistics attributes is as follows:

- *writesExecuted*: the total number of data-modification operations successfully executed. This is equivalent to the number of documents created, updated or removed by `INSERT`, `UPDATE`, `REPLACE` or `REMOVE` operations.
- *writesIgnored*: the total number of data-modification operations that were unsuccessful, but have been ignored because of query option `ignoreErrors`.
- *scannedFull*: the total number of documents iterated over when scanning a collection without an index. Documents scanned by subqueries will be included in the result, but not no operations triggered by built-in or user-defined AQL functions.
- *scannedIndex*: the total number of documents iterated over when scanning a collection using an index. Documents scanned by subqueries will be included in the result, but not no operations triggered by built-in or user-defined AQL functions.
- *filtered*: the total number of documents that were removed after executing a filter condition in a `FilterNode`. Note that `IndexRangeNode`s can also filter documents by selecting only the required index range from a collection, and the `filtered` value only indicates how much filtering was done by `FilterNode`s.
- *fullCount*: the total number of documents that matched the search condition if the query's final `LIMIT` statement were not present. This attribute will only be returned if the `fullCount` option was set when starting the query and will only contain a sensible value if the query contained a `LIMIT` operation on the top level.

Parsing queries

Clients can use ArangoDB to check if a given AQL query is syntactically valid. ArangoDB provides an HTTP REST API for this.

A query can also be parsed from the ArangoShell using `ArangoStatement`'s `parse` method. The `parse` method will throw an exception if the query is syntactically invalid. Otherwise, it will return the some information about the query.

The return value is an object with the collection names used in the query listed in the `collections` attribute, and all bind parameters listed in the `bindVars` attribute. Additionally, the internal representation of the query, the query's abstract syntax tree, will be returned in the `AST` attribute of the result. Please note that the abstract syntax tree will be returned without any optimizations applied to it.

```
arangosh> var stmt = db._createStatement(
........> "FOR doc IN @@collection FILTER doc.foo == @bar RETURN doc");
arangosh> stmt.parse();
```

show execution results

Explaining queries

If it is unclear how a given query will perform, clients can retrieve a query's execution plan from the AQL query optimizer without actually executing the query. Getting the query execution plan from the optimizer is called *explaining*.

An explain will throw an error if the given query is syntactically invalid. Otherwise, it will return the execution plan and some information about what optimizations could be applied to the query. The query will not be executed.

Explaining a query can be achieved by calling the HTTP REST API. A query can also be explained from the ArangoShell using `ArangoStatement`'s `explain` method.

By default, the query optimizer will return what it considers to be the *optimal plan*. The optimal plan will be returned in the `plan` attribute of the result. If `explain` is called with option `allPlans` set to `true`, all plans will be returned in the `plans` attribute instead. The result object will also contain an attribute *warnings*, which is an array of warnings that occurred during optimization or execution plan creation.

Each plan in the result is an object with the following attributes:

- *nodes*: the array of execution nodes of the plan. The list of available node types can be found here
- *estimatedCost*: the total estimated cost for the plan. If there are multiple plans, the optimizer will choose the plan with the lowest total cost.
- *collections*: an array of collections used in the query
- *rules*: an array of rules the optimizer applied. The list of rules can be found here
- *variables*: array of variables used in the query (note: this may contain internal variables created by the optimizer)

Here is an example for retrieving the execution plan of a simple query:

```
arangosh> var stmt = db._createStatement(
........> "FOR user IN _users RETURN user");
arangosh> stmt.explain();
```

show execution results

As the output of `explain` is very detailed, it is recommended to use some scripting to make the output less verbose:

```
arangosh> var formatPlan = function (plan) {
........>    return { estimatedCost: plan.estimatedCost,
........>        nodes: plan.nodes.map(function(node) {
........> return node.type; }) }; };
arangosh> formatPlan(stmt.explain().plan);
```

show execution results

If a query contains bind parameters, they must be added to the statement **before** `explain` is called:

```
arangosh> var stmt = db._createStatement(
........> `FOR doc IN @@collection FILTER doc.user == @user RETURN doc`
........> );
arangosh> stmt.bind({ "@collection" : "_users", "user" : "root" });
arangosh> stmt.explain();
```

show execution results

In some cases the AQL optimizer creates multiple plans for a single query. By default only the plan with the lowest total estimated cost is kept, and the other plans are discarded. To retrieve all plans the optimizer has generated, `explain` can be called with the option `allPlans` set to `true`.

In the following example, the optimizer has created two plans:

```
arangosh> var stmt = db._createStatement(
........> "FOR user IN _users FILTER user.user == 'root' RETURN user");
arangosh> stmt.explain({ allPlans: true }).plans.length;
1
```

To see a slightly more compact version of the plan, the following transformation can be applied:

```
arangosh> stmt.explain({ allPlans: true }).plans.map(
........> function(plan) { return formatPlan(plan); });
```

show execution results

`explain` will also accept the following additional options:

- *maxPlans*: limits the maximum number of plans that are created by the AQL query optimizer
- *optimizer.rules*: an array of to-be-included or to-be-excluded optimizer rules can be put into this attribute, telling the optimizer to include or exclude specific rules. To disable a rule, prefix its name with a `-`, to enable a rule, prefix it with a `+`. There is also a pseudo-rule `all`, which will match all optimizer rules.

The following example disables all optimizer rules but `remove-redundant-calculations`:

```
arangosh> stmt.explain({ optimizer: {
........> rules: [ "-all", "+remove-redundant-calculations" ] } });
```

show execution results

The contents of an execution plan are meant to be machine-readable. To get a human-readable version of a query's execution plan, the following commands can be used:

```
arangosh> var query = "FOR doc IN mycollection FILTER doc.value > 42 RETURN doc";
arangosh> require("@arangodb/aql/explainer").explain(query, {colors:false});
```

show execution results

The above command prints the query's execution plan in the ArangoShell directly, focusing on the most important information.

The AQL query optimizer

AQL queries are sent through an optimizer before execution. The task of the optimizer is to create an initial execution plan for the query, look for optimization opportunities and apply them. As a result, the optimizer might produce multiple execution plans for a single query. It will then calculate the costs for all plans and pick the plan with the lowest total cost. This resulting plan is considered to be the *optimal plan*, which is then executed.

The optimizer is designed to only perform optimizations if they are *safe*, in the meaning that an optimization should not modify the result of a query. A notable exception to this is that the optimizer is allowed to change the order of results for queries that do not explicitly specify how results should be sorted.

Execution plans

The `explain` command can be used to query the optimal executed plan or even all plans the optimizer has generated. Additionally, `explain` can reveal some more information about the optimizer's view of the query.

Inspecting plans using the explain helper

The `explain` method of `ArangoStatement` as shown in the next chapters creates very verbose output. You can work on the output programmatically, or use this handsome tool that we created to generate a more human readable representation.

You may use it like this: (we disable syntax highlighting here)

```
arangosh> db._create("test");
arangosh> for (i = 0; i < 100; ++i) { db.test.save({ value: i }); }
arangosh> db.test.ensureIndex({ type: "skiplist", fields: [ "value" ] });
arangosh> var explain = require("@arangodb/aql/explainer").explain;
arangosh> explain("FOR i IN test FILTER i.value > 97 SORT i.value RETURN i.value",
{colors:false});
```

show execution results

Execution plans in detail

Let's have a look at the raw json output of the same execution plan using the `explain` method of `ArangoStatement`:

```
arangosh> stmt = db._createStatement("FOR i IN test FILTER i.value > 97 SORT i.value
RETURN i.value");
arangosh> stmt.explain();
```

show execution results
As you can see, the result details are very verbose so we will not show them in full in the next sections. Instead, let's take a closer look at the results step by step.

Execution nodes

In general, an execution plan can be considered to be a pipeline of processing steps. Each processing step is carried out by a so-called *execution node*

The `nodes` attribute of the `explain` result contains these *execution nodes* in the *execution plan*. The output is still very verbose, so here's a shorted form of it:

```
arangosh> stmt.explain().plan.nodes.map(function (node) { return node.type; });
```

show execution results

Note that the list of nodes might slightly change in future versions of ArangoDB if new execution node types get added or the optimizer create somewhat more optimized plans).

When a plan is executed, the query execution engine will start with the node at the bottom of the list (i.e. the *ReturnNode*).

The *ReturnNode*'s purpose is to return data to the caller. It does not produce data itself, so it will ask the node above itself, this is the *CalculationNode* in our example. *CalculationNode*s are responsible for evaluating arbitrary expressions. In our example query, the *CalculationNode* will evaluate the value of `i.value`, which is needed by the *ReturnNode*. The calculation will be applied for all data the *CalculationNode* gets from the node above it, in our example the *IndexNode*.

Finally, all of this needs to be done for documents of collection `test`. This is where the *IndexNode* enters the game. It will use an index (thus its name) to find certain documents in the collection and ship it down the pipeline in the order required by `SORT i.value`. The *IndexNode* itself has a *SingletonNode* as its input. The sole purpose of a *SingletonNode* node is to provide a single empty document as input for other processing steps. It is always the end of the pipeline.

Here's a summary:

- SingletonNode: produces an empty document as input for other processing steps.
- IndexNode: iterates over the index on attribute `value` in collection `test` in the order required by `SORT i.value`.
- CalculationNode: evaluates the result of the calculation `i.value > 97` to `true` or `false`
- CalculationNode: calculates return value `i.value`
- ReturnNode: returns data to the caller

Optimizer rules

Note that in the example, the optimizer has optimized the `SORT` statement away. It can do it safely because there is a sorted skiplist index on `i.value`, which it has picked in the *IndexNode*. As the index values are iterated over in sorted order anyway, the extra *SortNode* would have been redundant and was removed.

Additionally, the optimizer has done more work to generate an execution plan that avoids as much expensive operations as possible. Here is the list of optimizer rules that were applied to the plan:

```
arangosh> stmt.explain().plan.rules;
```

show execution results
Here is the meaning of these rules in context of this query:

- `move-calculations-up`: moves a *CalculationNode* as far up in the processing pipeline as possible
- `move-filters-up`: moves a *FilterNode* as far up in the processing pipeline as possible
- `remove-redundant-calculations`: replaces references to variables with references to other variables that contain the exact same result. In the example query, `i.value` is calculated multiple times, but each calculation inside a loop iteration would produce the same value. Therefore, the expression result is shared by several nodes.
- `remove-unnecessary-calculations`: removes *CalculationNode*s whose result values are not used in the query. In the example this happens due to the `remove-redundant-calculations` rule having made some calculations unnecessary.
- `use-indexes`: use an index to iterate over a collection instead of performing a full collection scan. In the example case this makes sense, as the index can be used for filtering and sorting.
- `remove-filter-covered-by-index`: remove an unnessary filter whose functionality is already covered by an index. In this case the index only returns documents matching the filter.
- `use-index-for-sort`: removes a `SORT` operation if it is already satisfied by traversing over a sorted index

Note that some rules may appear multiple times in the list, with number suffixes. This is due to the same rule being applied multiple times, at different positions in the optimizer pipeline.

Collections used in a query

The list of collections used in a plan (and query) is contained in the `collections` attribute of a plan:

```
arangosh> stmt.explain().plan.collections
```

show execution results

The `name` attribute contains the name of the `collection`, and `type` is the access type, which can be either `read` or `write`.

Variables used in a query

The optimizer will also return a list of variables used in a plan (and query). This list will contain auxiliary variables created by the optimizer itself. This list can be ignored by end users in most cases.

Cost of a query

For each plan the optimizer generates, it will calculate the total cost. The plan with the lowest total cost is considered to be the optimal plan. Costs are estimates only, as the actual execution costs are unknown to the optimizer. Costs are calculated based on heuristics that are hard-coded into execution nodes. Cost values do not have any unit.

Retrieving all execution plans

To retrieve not just the optimal plan but a list of all plans the optimizer has generated, set the option `allPlans` to `true`:

This will return a list of all plans in the `plans` attribute instead of in the `plan` attribute:

```
arangosh> stmt.explain({ allPlans: true });
```

show execution results

Retrieving the plan as it was generated by the parser / lexer

To retrieve the plan which closely matches your query, you may turn off most optimization rules (i.e. cluster rules cannot be disabled if you're running the explain on a cluster coordinator) set the option `rules` to `-all`:

This will return an unoptimized plan in the `plan`:

```
arangosh> stmt.explain({ optimizer: { rules: [ "-all" ] } });
```

show execution results
Note that some optimizations are already done at parse time (i.e. evaluate simple constant calculation as `1 + 1`)

Turning specific optimizer rules off

Optimizer rules can also be turned on or off individually, using the `rules` attribute. This can be used to enable or disable one or multiple rules. Rules that shall be enabled need to be prefixed with a `+`, rules to be disabled should be prefixed with a `-`. The pseudo-rule `all` matches all rules.

Rules specified in `rules` are evaluated from left to right, so the following works to turn on just the one specific rule:

```
arangosh> stmt.explain({ optimizer: { rules: [ "-all", "+use-index-range" ] } });
```

show execution results
By default, all rules are turned on. To turn off just a few specific rules, use something like this:

```
arangosh> stmt.explain({ optimizer: { rules: [ "-use-index-range", "-use-index-for-sort" ]
} });
```

show execution results
The maximum number of plans created by the optimizer can also be limited using the `maxNumberOfPlans` attribute:

```
arangosh> stmt.explain({ maxNumberOfPlans: 1 });
```

show execution results

Optimizer statistics

The optimizer will return statistics as a part of an `explain` result.

The following attributes will be returned in the `stats` attribute of an `explain` result:

- `plansCreated` : total number of plans created by the optimizer
- `rulesExecuted` : number of rules executed (note: an executed rule does not indicate a plan was actually modified by a rule)
- `rulesSkipped` : number of rules skipped by the optimizer

Warnings

For some queries, the optimizer may produce warnings. These will be returned in the `warnings` attribute of the `explain` result:

```
arangosh> var stmt = db._createStatement("FOR i IN 1..10 RETURN 1 / 0")
arangosh> stmt.explain().warnings;
```

show execution results
There is an upper bound on the number of warning a query may produce. If that bound is reached, no further warnings will be returned.

Things to consider for optimizing queries

While the optimizer can fix some things in queries, its not allowed to take some assumptions, that you, the user, knowing what queries are intended to do can take. It may pull calculations to the front of the execution, but it may not cross certain borders.

So in certain cases you may want to move calculations in your query, so they're cheaper. Even more expensive is if you have calculacions that are executed in javascript:

```
arangosh> db._explain('FOR x IN 1..10 LET then=DATE_NOW() FOR y IN 1..10 LET
now=DATE_NOW() LET nowstr=CONCAT(now, x, y, then) RETURN nowstr', {}, {colors: false})
arangosh> db._explain('LET now=DATE_NOW() FOR x IN 1..10 FOR y IN 1..10 LET
nowstr=CONCAT(now, x, y, now) RETURN nowstr', {}, {colors: false})
```

show execution results
You can see, that the optimizer found `1..10` is specified twice, but can be done first one time.

While you may see time passing by during the execution of the query and its calls to `DATE_NOW()` this may not be the desired thing in first place. The queries V8 Expressions will however also use significant resources, since its executed 10 x 10 times => 100 times. Now if we don't care for the time ticking by during the query execution, we may fetch the time once at the startup of the query, which will then only give us one V8 expression at the very start of the query.

Next to bringing better performance, this also obeys the DRY principle.

Optimization in a cluster

When you're running AQL in the cluster, the parsing of the query is done on the coordinator. The coordinator then chops the query into snipets, which are to remain on the coordinator, and others that are to be distributed over the network to the shards. The cutting sites are interconnected via *Scatter*-, *Gather*- and *RemoteNodes*.

These nodes mark the network borders of the snippets. The optimizer strives to reduce the amount of data transfered via these network interfaces by pushing `FILTER`s out to the shards, as it is vital to the query performance to reduce that data amount to transfer over the network links.

Snippets marked with **DBS** are executed on the shards, **COOR** ones are excuted on the coordinator.

As usual, the optimizer can only take certain assumptions for granted when doing so, i.e. user-defined functions have to be executed on the coordinator. If in doubt, you should modify your query to reduce the number interconnections between your snippets.

When optimizing your query you may want to look at simpler parts of it first.

List of execution nodes

The following execution node types will appear in the output of `explain`:

- *SingletonNode*: the purpose of a *SingletonNode* is to produce an empty document that is used as input for other processing steps. Each execution plan will contain exactly one *SingletonNode* as its top node.
- *EnumerateCollectionNode*: enumeration over documents of a collection (given in its *collection* attribute) without using an index.
- *IndexNode*: enumeration over one or many indexes (given in its *indexes* attribute) of a collection. The index ranges are specified in the *condition* attribute of the node.
- *EnumerateListNode*: enumeration over a list of (non-collection) values.
- *FilterNode*: only lets values pass that satisfy a filter condition. Will appear once per *FILTER* statement.
- *LimitNode*: limits the number of results passed to other processing steps. Will appear once per *LIMIT* statement.
- *CalculationNode*: evaluates an expression. The expression result may be used by other nodes, e.g. *FilterNode*, *EnumerateListNode*, *SortNode* etc.
- *SubqueryNode*: executes a subquery.
- *SortNode*: performs a sort of its input values.
- *AggregateNode*: aggregates its input and produces new output variables. This will appear once per *COLLECT* statement.
- *ReturnNode*: returns data to the caller. Will appear in each read-only query at least once. Subqueries will also contain *ReturnNode*s.
- *InsertNode*: inserts documents into a collection (given in its *collection* attribute). Will appear exactly once in a query that contains an *INSERT* statement.
- *RemoveNode*: removes documents from a collection (given in its *collection* attribute). Will appear exactly once in a query that contains a *REMOVE* statement.
- *ReplaceNode*: replaces documents in a collection (given in its *collection* attribute). Will appear exactly once in a query that contains a *REPLACE* statement.
- *UpdateNode*: updates documents in a collection (given in its *collection* attribute). Will appear exactly once in a query that contains an *UPDATE* statement.
- *UpsertNode*: upserts documents in a collection (given in its *collection* attribute). Will appear exactly once in a query that contains an *UPSERT* statement.
- *NoResultsNode*: will be inserted if *FILTER* statements turn out to be never satisfiable. The *NoResultsNode* will pass an empty result set into the processing pipeline.

For queries in the cluster, the following nodes may appear in execution plans:

- *ScatterNode*: used on a coordinator to fan-out data to one or multiple shards.
- *GatherNode*: used on a coordinator to aggregate results from one or many shards into a combined stream of results.
- *DistributeNode*: used on a coordinator to fan-out data to one or multiple shards, taking into account a collection's shard key.
- *RemoteNode*: a *RemoteNode* will perform communication with another ArangoDB instances in the cluster. For example, the cluster coordinator will need to communicate with other servers to fetch the actual data from the shards. It will do so via *RemoteNode*s. The data servers themselves might again pull further data from the coordinator, and thus might also employ *RemoteNode*s. So, all of the above cluster relevant nodes will be accompanied by a *RemoteNode*.

List of optimizer rules

The following optimizer rules may appear in the `rules` attribute of a plan:

- `move-calculations-up`: will appear if a *CalculationNode* was moved up in a plan. The intention of this rule is to move calculations up in the processing pipeline as far as possible (ideally out of enumerations) so they are not executed in loops if not required. It is also quite common that this rule enables further optimizations to kick in.
- `move-filters-up`: will appear if a *FilterNode* was moved up in a plan. The intention of this rule is to move filters up in the processing pipeline as far as possible (ideally out of inner loops) so they filter results as early as possible.
- `sort-in-values`: will appear when the values used as right-hand side of an `IN` operator will be pre-sorted using an extra function call. Pre-sorting the comparison array allows using a binary search in-list lookup with a logarithmic complexity instead of the default linear complexity in-list lookup.
- `remove-unnecessary-filters`: will appear if a *FilterNode* was removed or replaced. *FilterNode*s whose filter condition will always evaluate to *true* will be removed from the plan, whereas *FilterNode* that will never let any results pass will be replaced with a *NoResultsNode*.
- `remove-redundant-calculations`: will appear if redundant calculations (expressions with the exact same result) were found in the query. The optimizer rule will then replace references to the redundant expressions with a single reference, allowing other optimizer

rules to remove the then-unneeded *CalculationNode*s.

- `remove-unnecessary-calculations` : will appear if *CalculationNode*s were removed from the query. The rule will removed all calculations whose result is not referenced in the query (note that this may be a consequence of applying other optimizations).
- `remove-redundant-sorts` : will appear if multiple *SORT* statements can be merged into fewer sorts.
- `interchange-adjacent-enumerations` : will appear if a query contains multiple *FOR* statements whose order were permuted. Permutation of *FOR* statements is performed because it may enable further optimizations by other rules.
- `remove-collect-variables` : will appear if an *INTO* clause was removed from a *COLLECT* statement because the result of *INTO* is not used. May also appear if a result of a *COLLECT* statement's *AGGREGATE* variables is not used.
- `propagate-constant-attributes` : will appear when a constant value was inserted into a filter condition, replacing a dynamic attribute value.
- `replace-or-with-in` : will appear if multiple *OR*-combined equality conditions on the same variable or attribute were replaced with an *IN* condition.
- `remove-redundant-or` : will appear if multiple *OR* conditions for the same variable or attribute were combined into a single condition.
- `use-indexes` : will appear when an index is used to iterate over a collection. As a consequence, an *EnumerateCollectionNode* was replaced with an *IndexNode* in the plan.
- `remove-filter-covered-by-index` : will appear if a *FilterNode* was removed or replaced because the filter condition is already covered by an *IndexNode*.
- `remove-filter-covered-by-traversal` : will appear if a *FilterNode* was removed or replaced because the filter condition is already covered by an *TraversalNode*.
- `use-index-for-sort` : will appear if an index can be used to avoid a *SORT* operation. If the rule was applied, a *SortNode* was removed from the plan.
- `move-calculations-down` : will appear if a *CalculationNode* was moved down in a plan. The intention of this rule is to move calculations down in the processing pipeline as far as possible (below *FILTER*, *LIMIT* and *SUBQUERY* nodes) so they are executed as late as possible and not before their results are required.
- `patch-update-statements` : will appear if an *UpdateNode* was patched to not buffer its input completely, but to process it in smaller batches. The rule will fire for an *UPDATE* query that is fed by a full collection scan, and that does not use any other indexes and subqueries.
- `optimize-traversals` : will appear if either the edge or path output variable in an AQL traversal was optimized away, or if a *FILTER* condition from the query was moved in the *TraversalNode* for early pruning of results.
- `inline-subqueries` : will appear when a subquery was pulled out in its surrounding scope, e.g. `FOR x IN (FOR y IN collection FILTER y.value >= 5 RETURN y.test) RETURN x.a` would become `FOR tmp IN collection FILTER tmp.value >= 5 LET x = tmp.test RETURN x.a`
- `geo-index-optimizer` : will appear when a geo index is utilized.
- `remove-sort-rand` : will appear when a *SORT RAND()* expression is removed by moving the random iteration into an *EnumerateCollectionNode*. This optimizer rule is specific for the MMFiles storage engine.
- `reduce-extraction-to-projection` : will appear when an *EnumerationCollectionNode* that would have extracted an entire document was modified to return only a projection of each document. This optimizer rule is specific for the RocksDB storage engine.

The following optimizer rules may appear in the `rules` attribute of cluster plans:

- `distribute-in-cluster` : will appear when query parts get distributed in a cluster. This is not an optimization rule, and it cannot be turned off.
- `scatter-in-cluster` : will appear when scatter, gather, and remote nodes are inserted into a distributed query. This is not an optimization rule, and it cannot be turned off.
- `distribute-filtercalc-to-cluster` : will appear when filters are moved up in a distributed execution plan. Filters are moved as far up in the plan as possible to make result sets as small as possible as early as possible.
- `distribute-sort-to-cluster` : will appear if sorts are moved up in a distributed query. Sorts are moved as far up in the plan as possible to make result sets as small as possible as early as possible.
- `remove-unnecessary-remote-scatter` : will appear if a RemoteNode is followed by a ScatterNode, and the ScatterNode is only followed by calculations or the SingletonNode. In this case, there is no need to distribute the calculation, and it will be handled centrally.
- `undistribute-remove-after-enum-coll` : will appear if a RemoveNode can be pushed into the same query part that enumerates over the documents of a collection. This saves inter-cluster roundtrips between the EnumerateCollectionNode and the RemoveNode.

Note that some rules may appear multiple times in the list, with number suffixes. This is due to the same rule being applied multiple times, at different positions in the optimizer pipeline.

The AQL query result cache

AQL provides an optional query result cache.

The purpose of the query cache is to avoid repeated calculation of the same query results. It is useful if data-reading queries repeat a lot and there are not many write queries.

The query cache is transparent so users do not need to manually invalidate results in it if underlying collection data are modified.

Modes

The cache can be operated in the following modes:

- `off` : the cache is disabled. No query results will be stored
- `on` : the cache will store the results of all AQL queries unless their `cache` attribute flag is set to `false`
- `demand` : the cache will store the results of AQL queries that have their `cache` attribute set to `true`, but will ignore all others

The mode can be set at server startup and later changed at runtime.

Query eligibility

The query cache will consider two queries identical if they have exactly the same query string. Any deviation in terms of whitespace, capitalization etc. will be considered a difference. The query string will be hashed and used as the cache lookup key. If a query uses bind parameters, these will also be hashed and used as the cache lookup key.

That means even if the query string for two queries is identical, the query cache will treat them as different queries if they have different bind parameter values. Other components that will become part of a query's cache key are the `count` and `fullCount` attributes.

If the cache is turned on, the cache will check at the very start of execution whether it has a result ready for this particular query. If that is the case, the query result will be served directly from the cache, which is normally very efficient. If the query cannot be found in the cache, it will be executed as usual.

If the query is eligible for caching and the cache is turned on, the query result will be stored in the query cache so it can be used for subsequent executions of the same query.

A query is eligible for caching only if all of the following conditions are met:

- the server the query executes on is not a coordinator
- the query string is at least 8 characters long
- the query is a read-only query and does not modify data in any collection
- no warnings were produced while executing the query
- the query is deterministic and only uses deterministic functions

The usage of non-deterministic functions leads to a query not being cachable. This is intentional to avoid caching of function results which should rather be calculated on each invocation of the query (e.g. `RAND()` or `DATE_NOW()`).

The query cache considers all user-defined AQL functions to be non-deterministic as it has no insight into these functions.

Cache invalidation

The query cache results are fully or partially invalidated automatically if queries modify the data of collections that were used during the computation of the cached query results. This is to protect users from getting stale results from the query cache.

This also means that if the cache is turned on, then there is an additional cache invalidation check for each data-modification operation (e.g. insert, update, remove, truncate operations as well as AQL data-modification queries).

Example

If the result of the following query is present in the query cache, then either modifying data in collection `users` or in collection `organizations` will remove the already computed result from the cache:

```
FOR user IN users
  FOR organization IN organizations
    FILTER user.organization == organization._key
    RETURN { user: user, organization: organization }
```

Modifying data in other collections than the named two will not lead to this query result being removed from the cache.

Performance considerations

The query cache is organized as a hash table, so looking up whether a query result is present in the cache is relatively fast. Still, the query string and the bind parameter used in the query will need to be hashed. This is a slight overhead that will not be present if the cache is turned off or a query is marked as not cacheable.

Additionally, storing query results in the cache and fetching results from the cache requires locking via an R/W lock. While many thread can read in parallel from the cache, there can only be a single modifying thread at any given time. Modifications of the query cache contents are required when a query result is stored in the cache or during cache invalidation after data-modification operations. Cache invalidation will require time proportional to the number of cached items that need to be invalidated.

There may be workloads in which enabling the query cache will lead to a performance degradation. It is not recommended to turn the query cache on in workloads that only modify data, or that modify data more often than reading it. Turning on the query cache will also provide no benefit if queries are very diverse and do not repeat often. In read-only or read-mostly workloads, the query cache will be beneficial if the same queries are repeated lots of times.

In general, the query cache will provide the biggest improvements for queries with small result sets that take long to calculate. If query results are very big and most of the query time is spent on copying the result from the cache to the client, then the cache will not provide much benefit.

Global configuration

The query cache can be configured at server start using the configuration parameter `--query.cache-mode`. This will set the cache mode according to the descriptions above.

After the server is started, the cache mode can be changed at runtime as follows:

```
require("@arangodb/aql/cache").properties({ mode: "on" });
```

The maximum number of cached results in the cache for each database can be configured at server start using the configuration parameter `--query.cache-entries`. This parameter can be used to put an upper bound on the number of query results in each database's query cache and thus restrict the cache's memory consumption.

The value can also be adjusted at runtime as follows:

```
require("@arangodb/aql/cache").properties({ maxResults: 200 });
```

Per-query configuration

When a query is sent to the server for execution and the cache is set to `on` or `demand`, the query executor will look into the query's `cache` attribute. If the query cache mode is `on`, then not setting this attribute or setting it to anything but `false` will make the query executor consult the query cache. If the query cache mode is `demand`, then setting the `cache` attribute to `true` will make the executor look for the query in the query cache. When the query cache mode is `off`, the executor will not look for the query in the cache.

The `cache` attribute can be set as follows via the `db._createStatement()` function:

```
var stmt = db._createStatement({
  query: "FOR doc IN users LIMIT 5 RETURN doc",
```

```
  cache: true  /* cache attribute set here */
});

stmt.execute();
```

When using the `db._query()` function, the `cache` attribute can be set as allows:

```
db._query({
  query: "FOR doc IN users LIMIT 5 RETURN doc",
  cache: true  /* cache attribute set here */
});
```

The `cache` attribute can be set via the HTTP REST API `POST /_api/cursor`, too.

Each query result returned will contain a `cached` attribute. This will be set to `true` if the result was retrieved from the query cache, and `false` otherwise. Clients can use this attribute to check if a specific query was served from the cache or not.

Restrictions

Query results that are returned from the query cache do not contain any execution statistics, meaning their *extra.stats* attribute will not be present. Additionally, query results returned from the cache will not contain profile information even if the *profile* option was set to true when invoking the query.

Common Errors

String concatenation

In AQL, strings must be concatenated using the CONCAT() function. Joining them together with the `+` operator is not supported. Especially as JavaScript programmer it is easy to walk into this trap:

```
RETURN "foo" + "bar" // [ 0 ]
RETURN "foo" + 123   // [ 123 ]
RETURN "123" + 200   // [ 323 ]
```

The arithmetic plus operator expects numbers as operands, and will try to implicitly cast them to numbers if they are of different type. `"foo"` and `"bar"` are casted to `0` and then added to together (still zero). If an actual number is added, that number will be returned (adding zero doesn't change the result). If the string is a valid string representation of a number, then it is casted to a number. Thus, adding `"123"` and `200` results in two numbers being added up to `323`.

To concatenate elements (with implicit casting to string for non-string values), do:

```
RETURN CONCAT("foo", "bar") // [ "foobar" ]
RETURN CONCAT("foo", 123)   // [ "foo123" ]
RETURN CONCAT("123", 200)   // [ "123200" ]
```