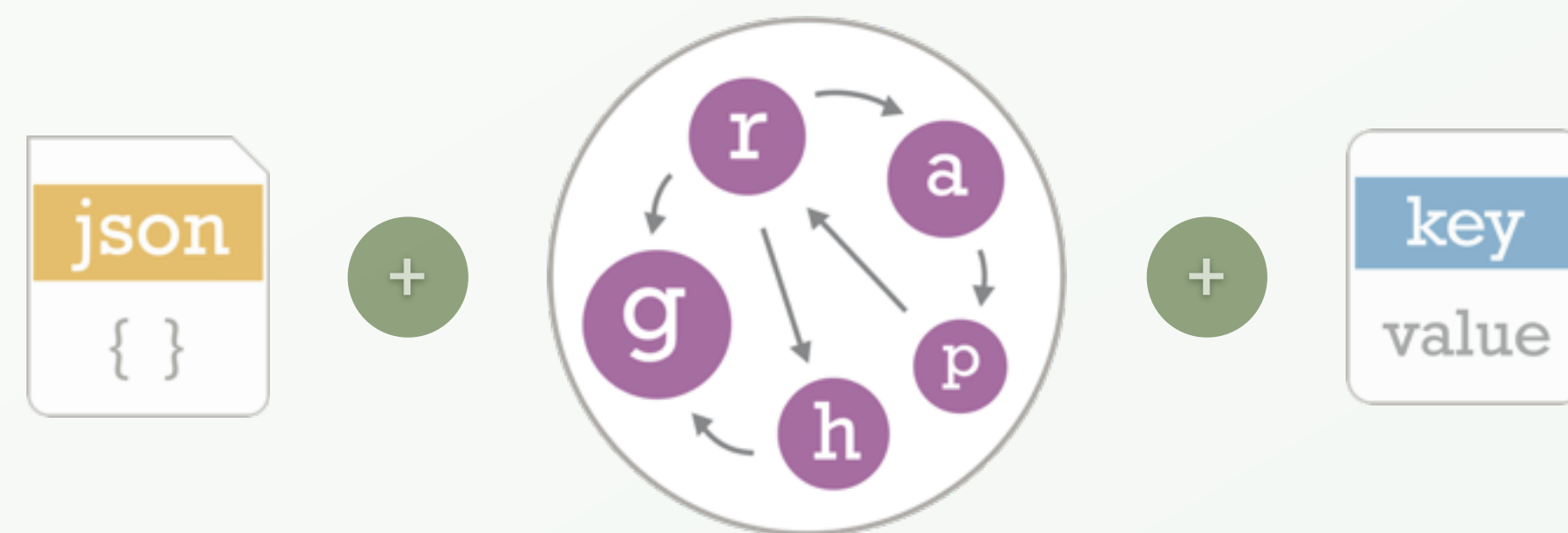


ArangoDB

*Graph Course for Freshers:
The **Shortest_Path** to first graph skills*



Welcome on board

In this course you will learn how to get started with ArangoDB's graph related features. We will use real world data of domestic flights and airports in the US. The structure of the data should be easy to understand and allows us to write many interesting queries to answer a variety of questions.

Hope you will enjoy the course!

Special thanks to @darkfrog *for his great and detailed feedback to the beta version of this course!*

What you will learn

- ▶ Basic Concepts Of Multi-Model in ArangoDB
- ▶ Basics About Graphs
- ▶ How To Import Graph Data
- ▶ First AQL Skills
- ▶ How To Use Graph Queries
 - ▶ Simple Graph Queries
 - ▶ Traverse Through a Graph
- ▶ Advanced Techniques
 - ▶ Shortest_Path
 - ▶ Pattern Matching

Table of Content

- ▶ Preparations for this Course (p.5)
 - ▶ Requirements (p.6)
 - ▶ Download & Install ArangoDB (p.7)
- ▶ Concepts in ArangoDB (p.8)
 - ▶ Multi-Model?! (p.9)
 - ▶ ArangoDB Structure (p.12)
- ▶ First Steps with ArangoDB (p.14)
 - ▶ Import Data with arangoimp (p.15)
 - ▶ First AQL Queries (p.20)
- ▶ Getting Closer to Graph Queries (p.25)
 - ▶ Importing Edges (p.25)
 - ▶ Underlying Concepts in ArangoDB (p.32)
- ▶ Traversal Query Syntax - Basics (p.36)
 - ▶ First Graph Queries Part I (p.39)
- ▶ Advanced Traversals (p.40)
 - ▶ Depth vs. Breadth First Search (p.41)
 - ▶ Breadth First Search (p.43)
 - ▶ Uniqueness Options (p.44)
 - ▶ First Graph Queries Part II (p.50)
 - ▶ Excursion: The LET Keyword in AQL (p.51)
- ▶ Advanced Graph Queries (p.55)
 - ▶ Shortest_Path (p.56)
 - ▶ Pattern Matching (p.60)
- ▶ Final Tasks (p.66)

Preparations for this Course

Requirements

&

Download and Install ArangoDB

Requirements

To follow this course and perform the described exercises, you will need:

- ▶ 4GB of RAM
- ▶ some free disk space
- ▶ an internet connection
- ▶ a web browser which supports Canvas or WebGL
- ▶ max. two hours of time

Note:



Click these symbols to get more details on a topic

Let's get started...

Download and install ArangoDB

- ▶ Go to arangodb.com/download/ to find the latest Community or Enterprise Edition for your operating system. Follow the instructions on how to download and install it for your OS. Further details can be found here:
docs.arangodb.com/latest/Manual/GettingStarted/Installing/
- ▶ Once the server booted up, open **http://localhost:8529** in your browser to access Aardvark, the ArangoDB WebUI
- ▶ Login as "root", **leave the password field empty and Login.**
After that select "_system" as your database.

Concepts of ArangoDB

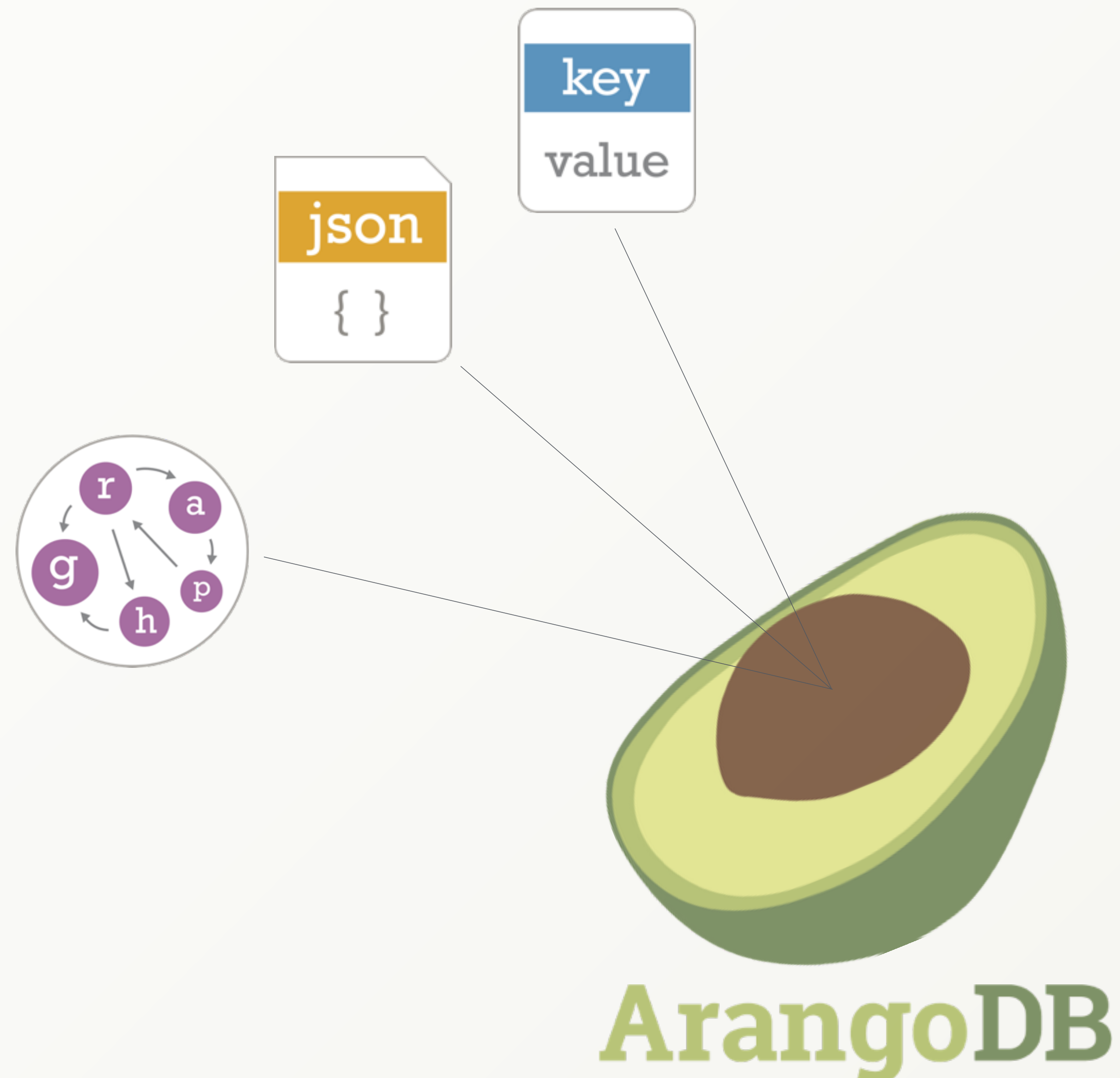
Multi-Model Approach

&

ArangoDB Structure

Multi-Model?! Part I

- ▶ ArangoDB is a **native** multi-model database
 - ▶ Multi-Model: ArangoDB supports 3 main NoSQL data models
 - ▶ Native: Supports all data models with one core and one query language (AQL)
- ▶ Unique feature of AQL:
 - ▶ Possibility to combine all three data models in a single query
 - ▶ combine joins, traversals, filters, geo-spatial operations and aggregations in your queries



Multi-Model?! Part II

How is multi-model possible at all?

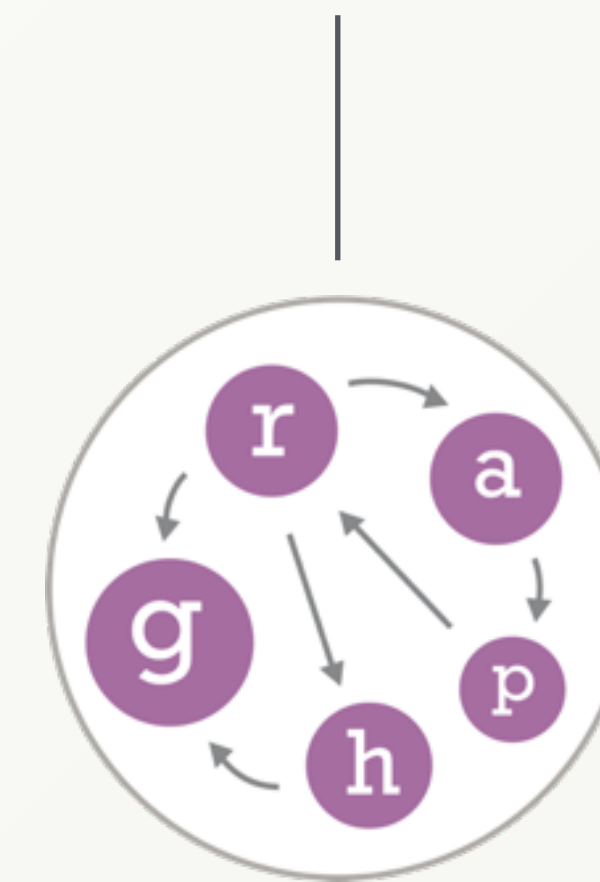
When storing just the primary key and a value you have a key/value store.



ArangoDB is a document-oriented data store using primary keys



Special *_from* and *_to* attributes pointing to other documents make up your graph in ArangoDB

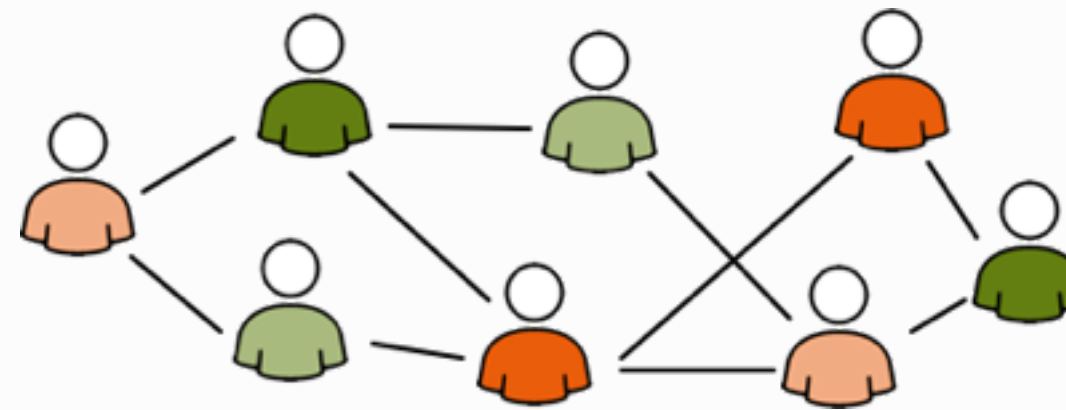


Benefits of ArangoDB's native **MULTI-MODEL** approach

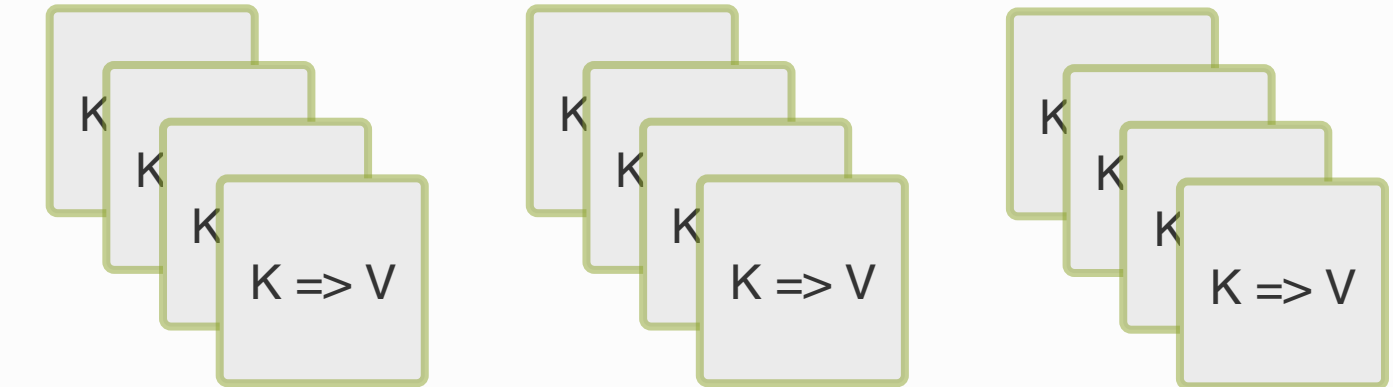
Documents - JSON

```
{
  "type": "pants",
  "waist": "32",
  "length": "34",
  "color": "blue",
  "material": "cotton"
}
{
  "type": "television",
  "diagonal size": "46",
  "hdmi inputs": "3",
  "wall mountable": "true",
  "built-in tuner": "true",
  "dynamic contrast": "50,000:1",
  "Resolution": "1920x1080"
}
```

Graphs



Key Values



no data-model lock-in

simpler development

larger solution-space
than relational

Concepts of ArangoDB

Multi-Model Approach
&
ArangoDB Structure

The ArangoDB Structure



- ▶ ArangoDB has a storage hierarchy as other databases have as well
- ▶ You can create different **Databases** which can hold an arbitrary number of Collections
- ▶ **Collections** can hold arbitrary amounts of Documents. ArangoDB has a special type of collections for edges (later more)
- ▶ **Documents** are stored in **JSON** format and have special attributes when stored in an edge collection (later more)

Databases

_system

_myDB

Collections

Airports

mydata

Documents
(Vertices)

{JSON} {JSON} {JSON}
{JSON} {JSON} {JSON}

{JSON} {JSON} {JSON}
{JSON} {JSON} {JSON}

Documents
(Edges)

Flights

mydata

{JSON} {JSON} {JSON}
{JSON} {JSON} {JSON}

{JSON} {JSON} {JSON}
{JSON} {JSON} {JSON}

Hands On

Import Data with Arangoimp

Download and Import the Dataset -PART I

We are going to work with real world data. Here with data of domestic flights in the USA from 2008.

What we provide is a reduced version of the original dataset. Please download the training dataset here arangodb.com/arangodb_graphcourse_demodata/ and unpack it to a folder of your choice. After unpacking you should see two .csv files named "airports" and "flights".

Let's import "airports" with ArangoDB's import tool arangoimp.

- ▶ Just run the following in your console (single line):

```
arangoimp --file path to airports.csv on your machine --collection  
airports --create-collection true --type csv
```

Remember that we didn't define a password for the user "root" so please just hit return when asked for a password during import!

Download and Import the Dataset

You should see something like this in your console after putting in the import command

```
Please specify a password:
Connected to ArangoDB 'http+tcp://127.0.0.1:8529', version 3.1.16, database: '_system', username: 'root'
-----
database:          _system
collection:        airports
create:            yes
source filename:    /Users/ /GraphCourse/airports.csv
file type:         csv
quote:             "
separator:
connect timeout:    5
request timeout:    1200
-----
Starting CSV import...
2017-04-11T12:11:31Z [4125] INFO processed 32768 bytes (3%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 65536 bytes (14%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 98304 bytes (26%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 131072 bytes (38%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 163840 bytes (50%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 196608 bytes (62%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 229376 bytes (74%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 262144 bytes (86%) of input file
2017-04-11T12:11:31Z [4125] INFO processed 274781 bytes (98%) of input file

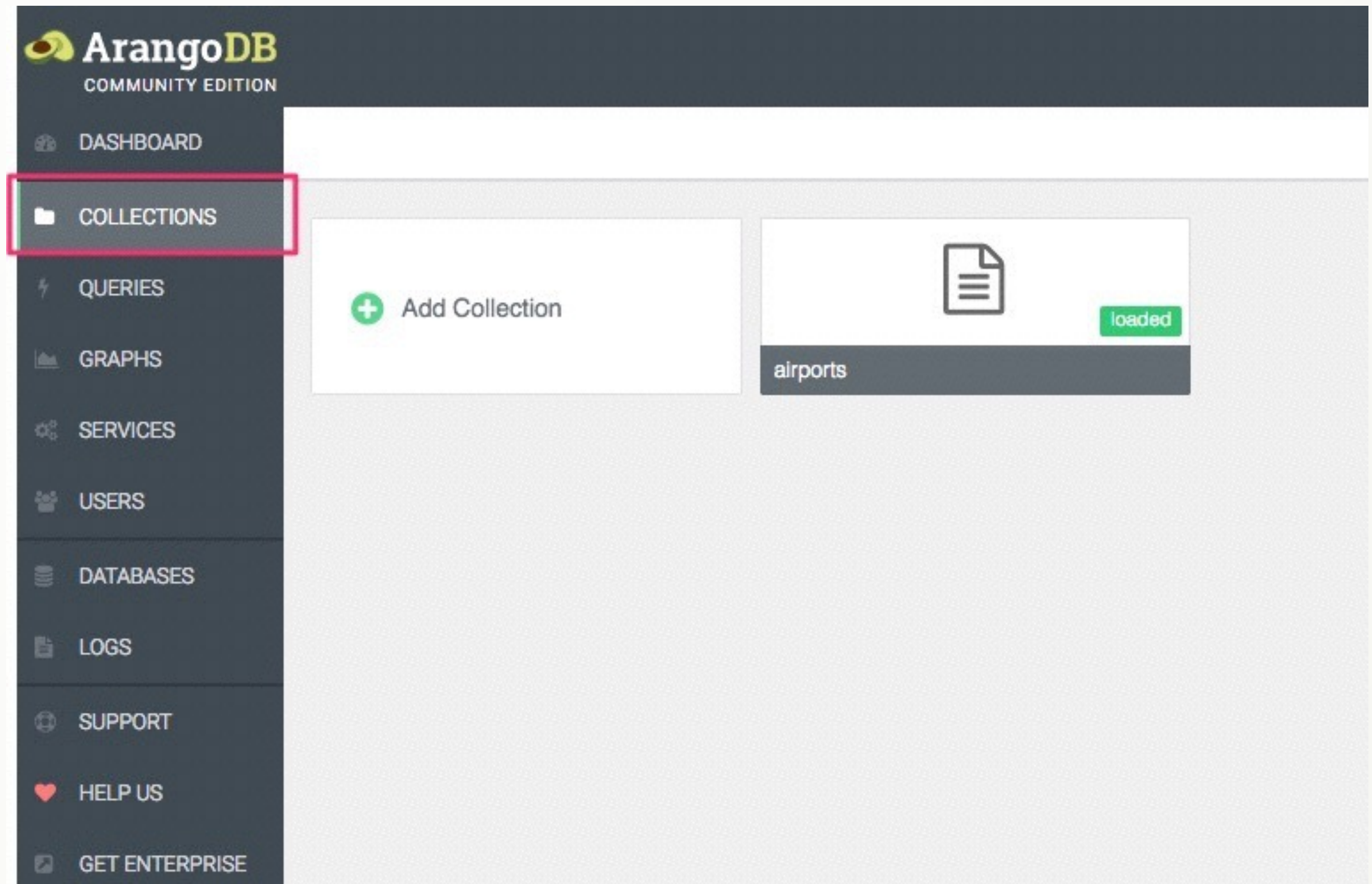
created:           3376
warnings/errors:   0
updated/replaced:  0
ignored:           0
lines read:        3377
```


What did arangoimp do?

- ▶ Created a new Document Collection (**airports**)
- ▶ Created one document for each line
- ▶ The header line is taken as attribute names
- ▶ Please note
 - ▶ We have given _key in the header line
 - ▶ The unique identifier will take this value

In ArangoDB WebUI

- ▶ Go to ArangoDB WebUI (<http://localhost:8529> in your browser) and choose "COLLECTIONS" in the menu
- ▶ Collection "airports" should be there now
- ▶ The icon indicates that it is a **document** collection, or vertex collection in the context of graphs
- ▶ Click on the collection to browse its documents



Data structure of airport documents

Attribute	Description
_key	international airport abbreviation code
_id	collection name + "/" + _key (computed property)
airport	full name of the airport
city	name of the city it belongs to
country	name of the country it is in
lat	latitude portion of the geographic location
long	longitude portion of the geographic location
state	name of the US state it is in

Example airport as shown in the document editor of the web interface:

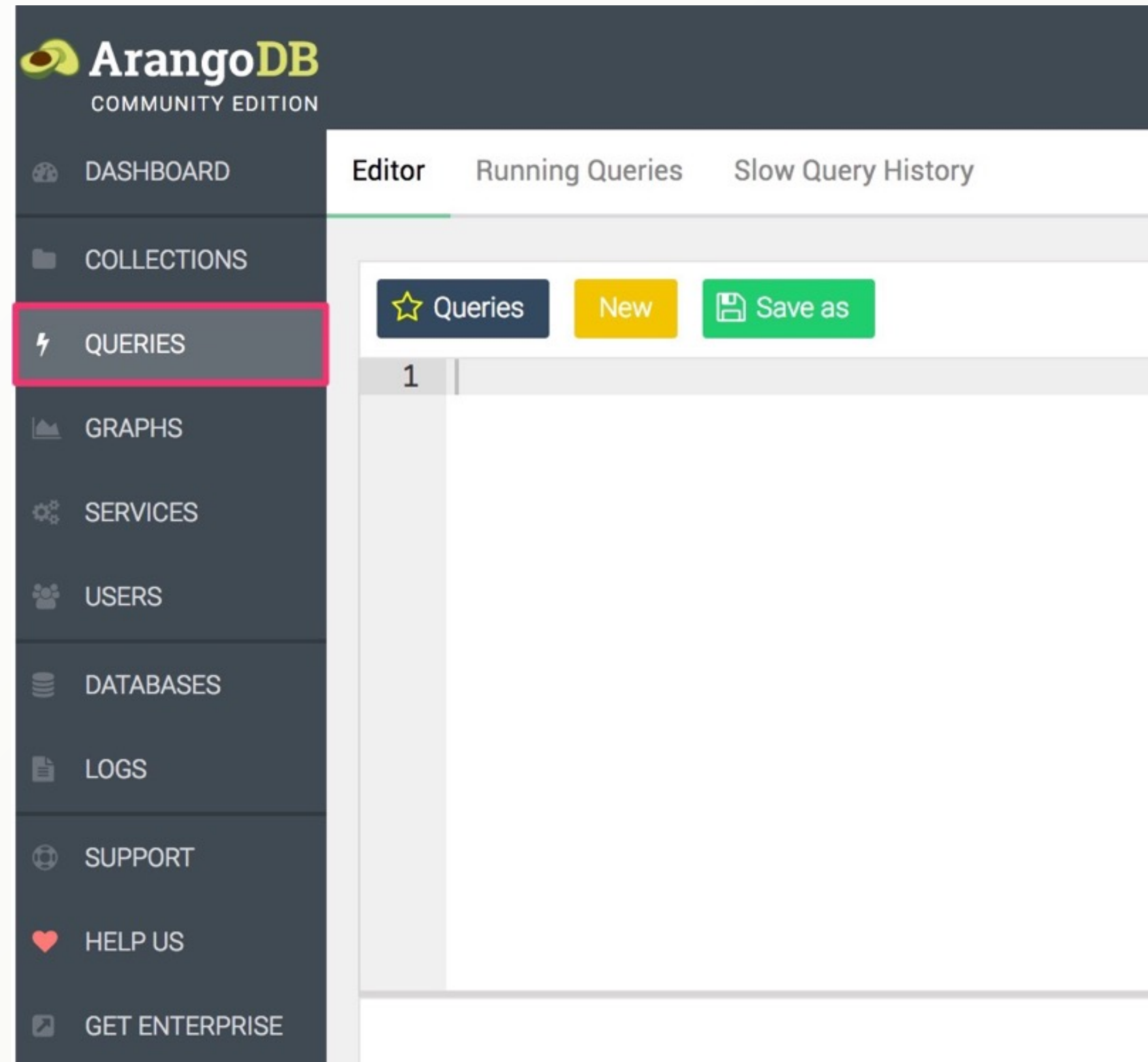
The screenshot shows the ArangoDB web interface document editor. At the top, the document's metadata is displayed: `_id: airports/BIS`, `_rev: _UdHg9Ia-_E`, and `_key: BIS`. Below this is a toolbar with icons for zooming, undo, redo, and a dropdown menu labeled 'Tree'. The main area shows a tree view of the document's fields, each with a checkbox and a label: `airport : Bismarck Municipal`, `city : Bismarck`, `country : USA`, `lat : 46.77411111`, `long : -100.7467222`, and `state : ND`.

Hands On

First AQL Queries

ArangoDB Query Editor

- ▶ Choose "QUERIES" in the ArangoDB WebUI
- ▶ It brings up the AQL query editor to write query code supported by syntax highlighting, submit and execute it



ArangoDB Query Editor

Opens saved queries

Clears current query

Saves current query

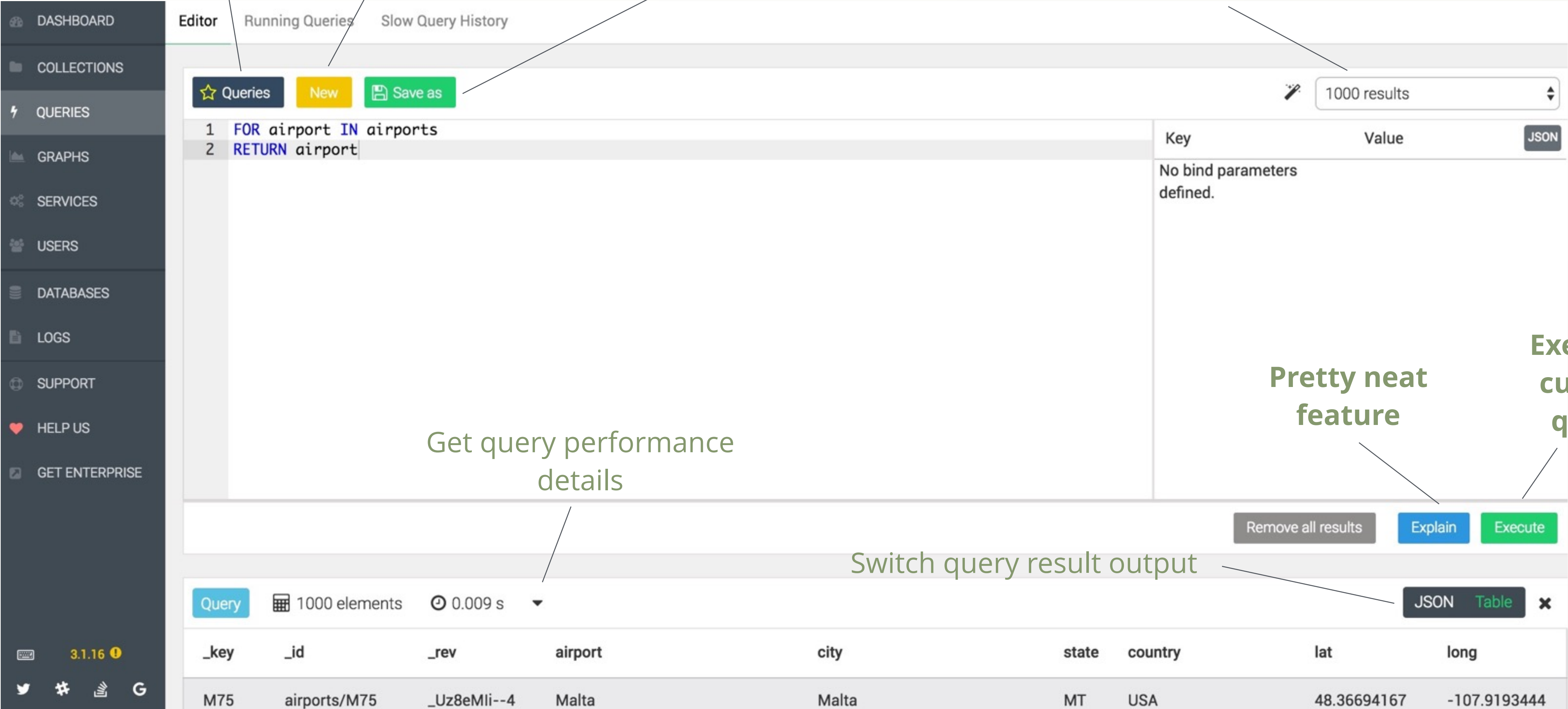
Set limit for results shown

Get query performance details

Pretty neat feature

Executes current query

Switch query result output



The screenshot shows the ArangoDB Query Editor interface. On the left is a dark sidebar with navigation links: DASHBOARD, COLLECTIONS, QUERIES (highlighted), GRAPHS, SERVICES, USERS, DATABASES, LOGS, SUPPORT, HELP US, and GET ENTERPRISE. The main area has tabs for Editor, Running Queries, and Slow Query History. The Editor tab is active, showing a query editor with two lines of code: `1 FOR airport IN airports` and `2 RETURN airport`. Above the editor are buttons for 'Queries' (with a star icon), 'New' (yellow), and 'Save as' (green). To the right of the editor is a dropdown menu set to '1000 results'. Below the editor is a large area for query results, currently displaying 'No bind parameters defined.' At the bottom right of the editor area are buttons for 'Remove all results', 'Explain', and 'Execute'. Below the editor area is a status bar showing 'Query', '1000 elements', and '0.009 s'. To the right of the status bar are buttons for 'JSON' and 'Table' (highlighted in green), with a close button 'x'. At the very bottom is a table with columns: _key, _id, _rev, airport, city, state, country, lat, and long. The first row of data shows: M75, airports/M75, _Uz8eMli--4, Malta, Malta, MT, USA, 48.36694167, -107.9193444.

_key	_id	_rev	airport	city	state	country	lat	long
M75	airports/M75	_Uz8eMli--4	Malta	Malta	MT	USA	48.36694167	-107.9193444

Hands on: First AQL Queries

Now that we have created our first collection and imported "airports" into ArangoDB, let us do some queries. Copy below queries into the editor and

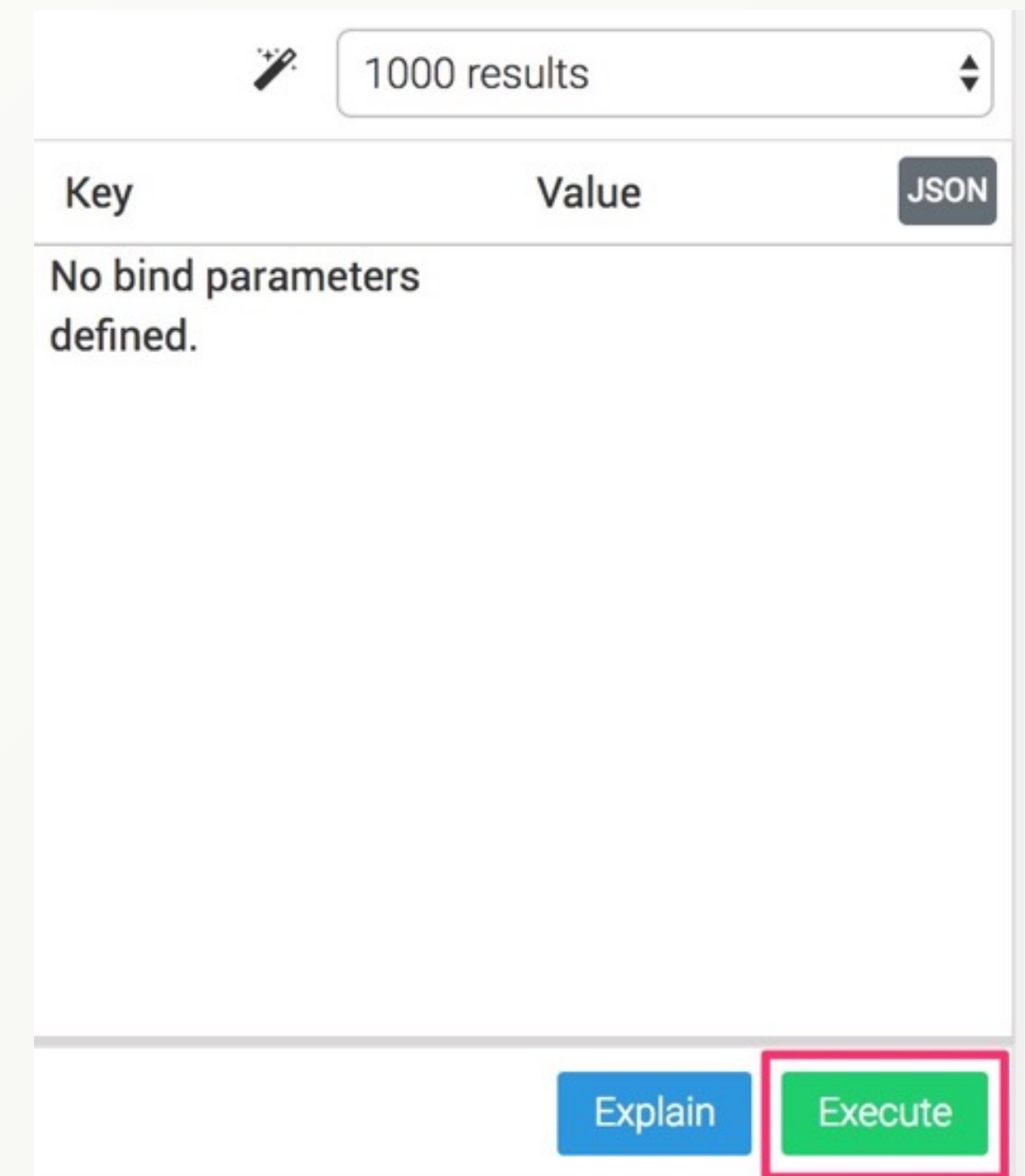
- click EXECUTE or hit
- Ctrl + Return, respectively
- Cmd + Return (Mac keyboard)

- ▶ Return all airports in "airports"

```
FOR airport IN airports  
  RETURN airport
```

- ▶ Return only the airports located in California

```
FOR airport IN airports  
  FILTER airport.state == "CA"  
  RETURN airport
```



Hands on: First AQL Queries

You can do a lot more with AQL, but that is beyond the scope of this course. Here is just one more example that demonstrates aggregation:

- ▶ Return the number of airports for each state

```
FOR airport IN airports
COLLECT state = airport.state
WITH COUNT INTO counter
RETURN {state, counter}
```

Great... You just wrote your first AQL queries.

COLLECT in AQL documentation



Getting Closer to Graphs Queries

Importing Edges

&

Underlying Concepts in ArangoDB

Download and Import the Dataset -PART II

A few steps back we imported "airports" which are the vertices of our graph. To complete our graph dataset, we also need edges to connect the vertices. In our case the edges are the flights.

Let's import "flights" as an EDGE COLLECTION with ArangoDB's import tool arangoimp.

- ▶ Just run the following in your console (single line):

```
arangoimp --file path to flights.csv on your machine --collection  
flights --create-collection true --type csv --create-collection-type  
edge
```

Importing flights.csv might take a few moments to complete.

What did arangoimp do?

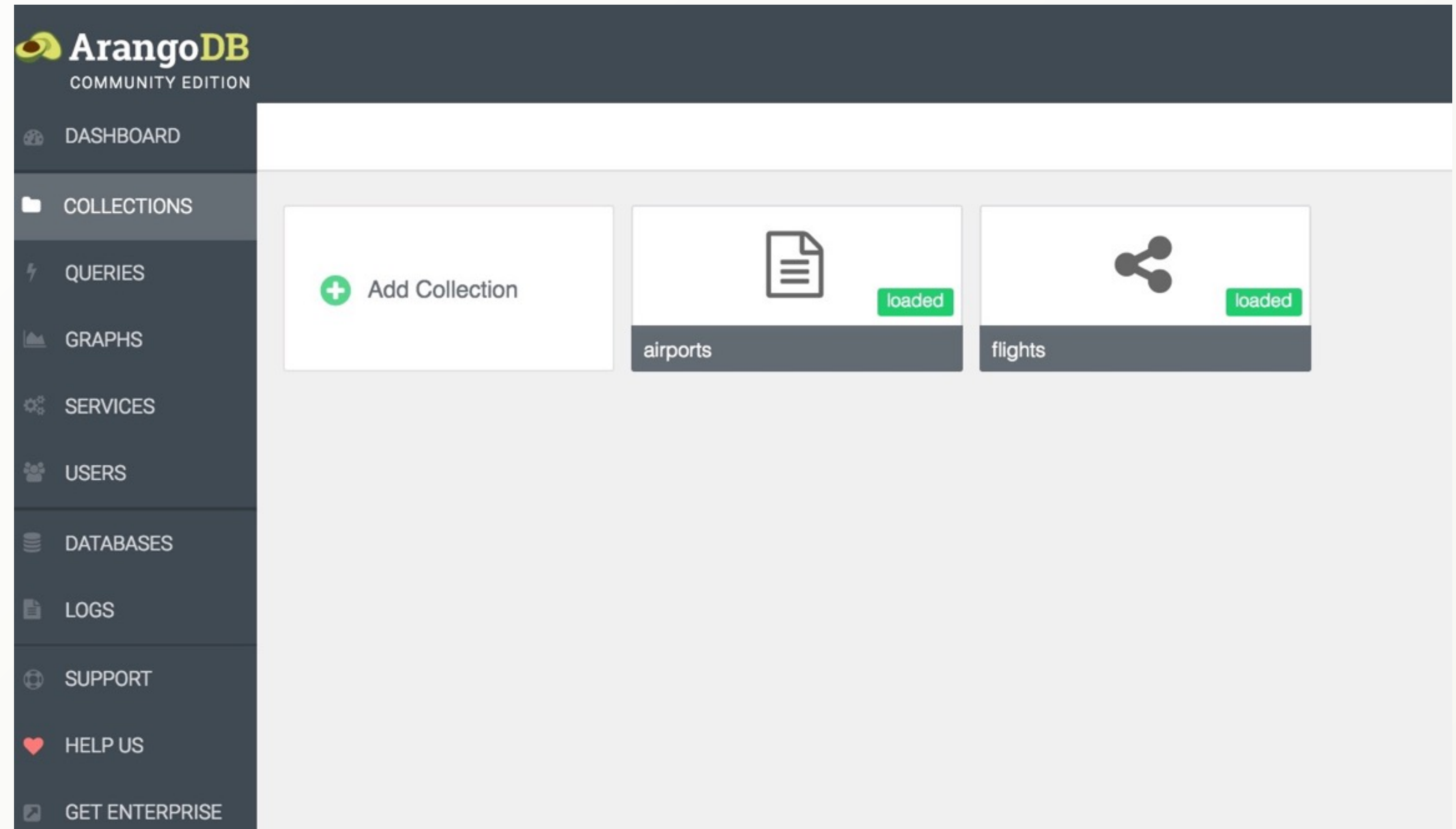
- ▶ Created a new Edge Collection (*flights*)
- ▶ Created one edge for each line
- ▶ Created an edge index for fast lookups

Note

- ▶ The *_from* and *_to* attributes forming the graph are the IDs of departure and arrival airports

In ArangoDB WebUI

- ▶ Go to ArangoDB WebUI (<http://localhost:8529> in your browser) and choose "COLLECTIONS" in the menu
- ▶ Edge Collection "**flights**" should be there now
- ▶ The type of the collection is indicated by a different icon for **edge** collections
- ▶ Click the flights collection to browse its edge documents



Data structure of flight documents

Attribute	Description
_from	Origin (airport _id)
_to	Destination (airport _id)
Year	Year of flight (here: 2008)
Month	Month of flight (1..12)
DayOfMonth	Day of flight (1..31)
DayOfWeek	Weekday (1 = Monday .. 7 = Sunday)
DepTime	Actual departure time (local, hhmm)
ArrTime	Actual arrival time (local, hhmm)
FlightNum	Flight number
TailNum	Plane tail number
UniqueCarrier	Unique carrier code
Distance	Travel distance in miles

Example flight as shown in the document editor of the web interface:

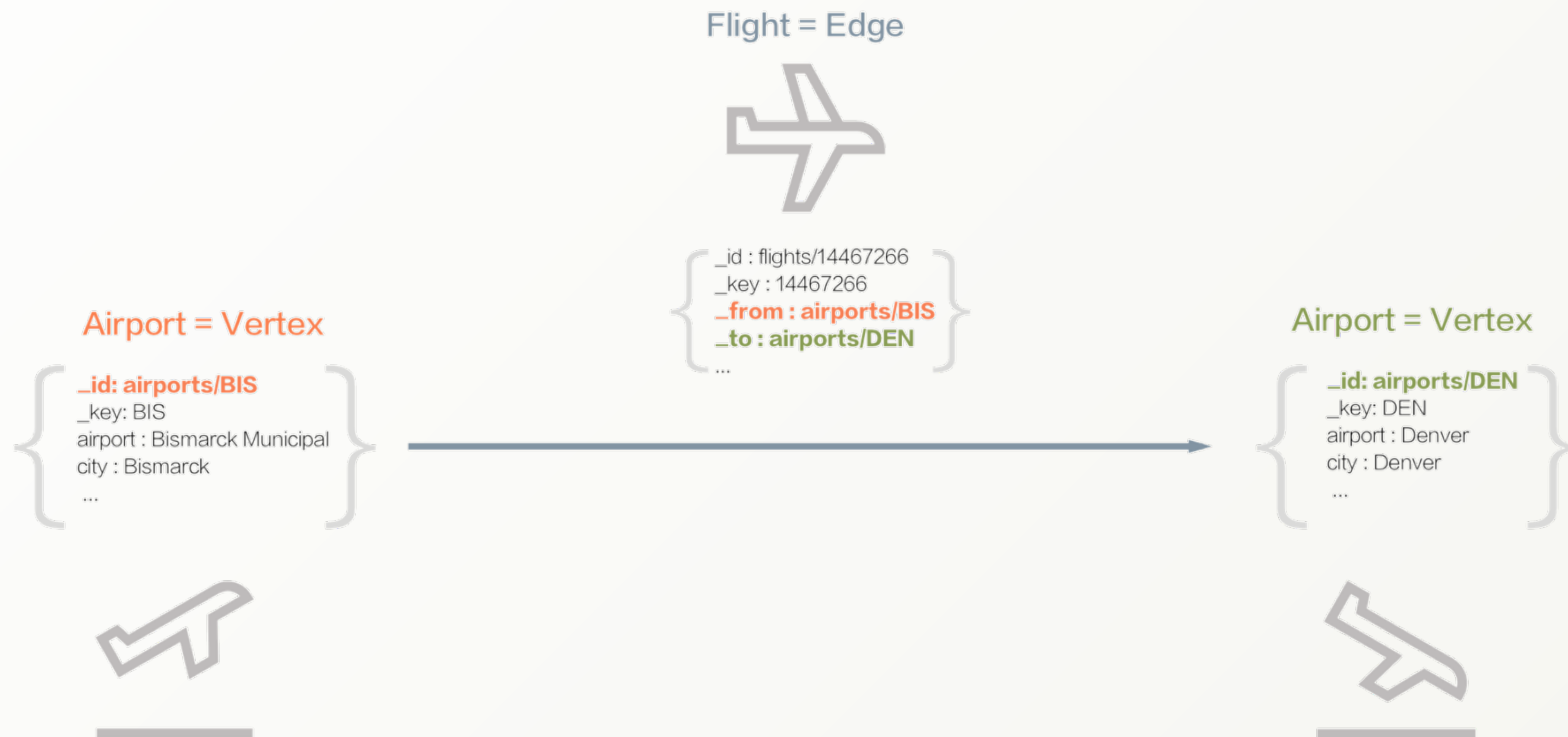
The screenshot shows the ArangoDB web interface document editor. At the top, the document's metadata is displayed: `_id: flights/94578`, `_rev: _U6n6ZP2--M`, `_key: 94578`, `_from: airports/MSP`, and `_to: airports/JFK`. Below this is a toolbar with icons for expand/collapse, refresh, and a dropdown menu labeled 'Tree'. The main area shows a tree view of the document's fields, each with a collapse icon and a label: `Year : 2008`, `Month : 1`, `DayOfMonth : 1`, `DayOfWeek : 2`, `DepTime : 712`, `ArrTime : 1059`, `UniqueCarrier : NW`, `FlightNum : 736`, `TailNum : N319NB`, and `Distance : 1028`.

Edge Collections in ArangoDB

- ▶ Place to hold relations
 - ▶ Comparable with SQL-many-to-many-Relation-Tables
- ▶ System attributes in documents:
 - ▶ `_from`: `_id` value of the source vertex
 - ▶ `_to`: `_id` value of the target vertex
- ▶ Always an Edge-Index for each edge collection
- ▶ Building block of graphs

How do airports & flights form a graph?

Airports are the vertices, flights are the edges. The `_id` attribute of airport documents is used for the `_from` and `_to` attributes in the edge documents to link airports together by flights.



Getting Closer to Graphs Queries

Importing Edges

&

Underlying Concepts in ArangoDB

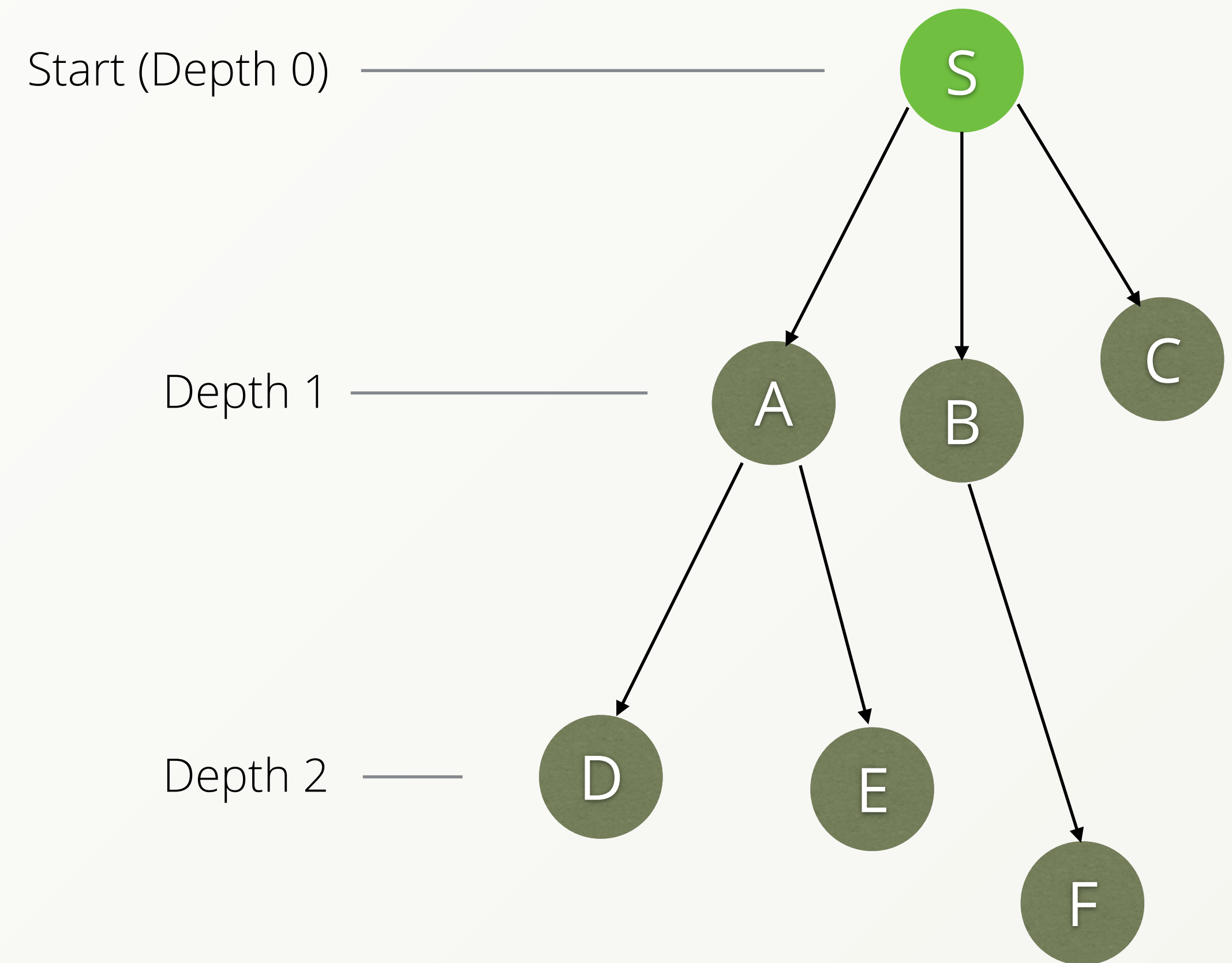
Typical Graph Queries

- ▶ Give me all friends of Alice
- ▶ Give me all friends-of-friends of Alice
- ▶ What is the linking path between Alice and Bob?
- ▶ Which train stations can I reach if I am allowed to drive a distance of 6 stations on my ticket?
- ▶ Pattern Matching:
 - ▶ Give me all users that share two hobbies with Alice
 - ▶ Give me all products that at least one of my friends has bought together with the products I already own, ordered by how many friends have bought it and the products rating, but only 20 of them.

Traversal Part I

Using a graph database allows you to traverse through your data efficiently. Traversal means to walk along edges of a graph in certain ways, optionally with some filters.

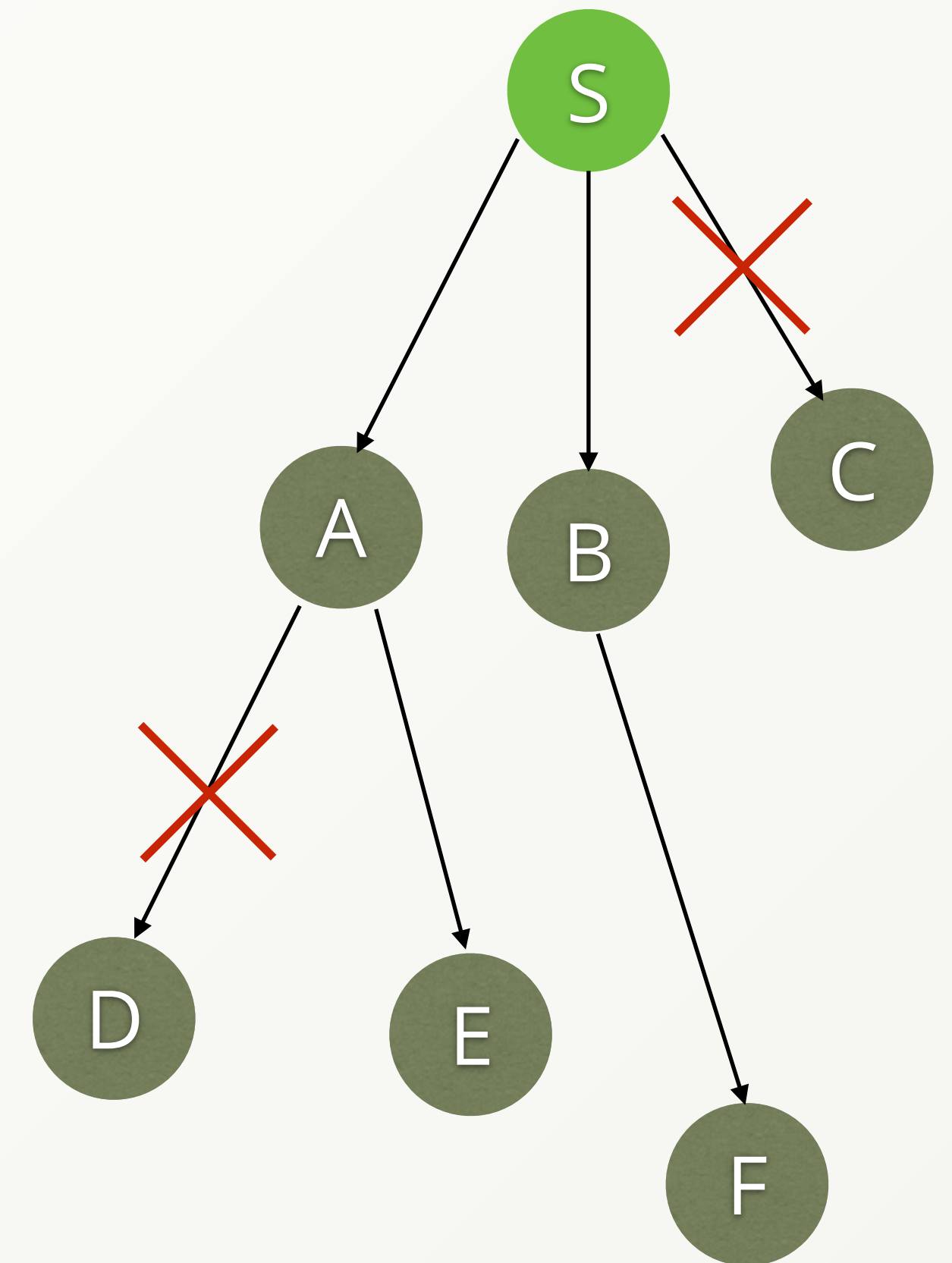
- ▶ The starting vertex in a traversal (S) has a traversal depth of zero.
- ▶ At depth = 1 are the direct neighbors of S (A, B and C).
- ▶ Their neighbor vertices in turn are at depth = 2 (D, E and F).



Traversal Part II

A traversal with depth 2 might look like the following:

- ▶ We first pick a start vertex (S)
- ▶ We follow all edges of S
- ▶ We apply filters on edges (C doesn't meet filter condition)
- ▶ We iterate down one of the new vertices (A)
- ▶ We apply filters on edges (D doesn't meet filter condition)
- ▶ The next vertex (E) is in desired depth. Return the path $S \rightarrow A \rightarrow E$
- ▶ Go back to the next unfinished vertex (B)
- ▶ We iterate down on (B)
- ▶ We apply filters on edges (F meets condition)
- ▶ The next vertex (F) is in desired depth. Return the path $S \rightarrow B \rightarrow F$



Basic Graph Query Syntax Part I

Before we do our first graph queries we should spend some time on the underlying concepts of the query options. We will go through the keywords and basic options step-by-step:

Basic Query Structure

```
FOR vertex[, edge[, path]]
IN [min[..max]]
OUTBOUND|INBOUND|ANY startVertex
edgeCollection[, more...]
```

Explanation

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: defines the minimal and maximal depth for the traversal. If not specified it defaults to 1!



Traversal in AQL documentation



Basic Graph Query Syntax Part II

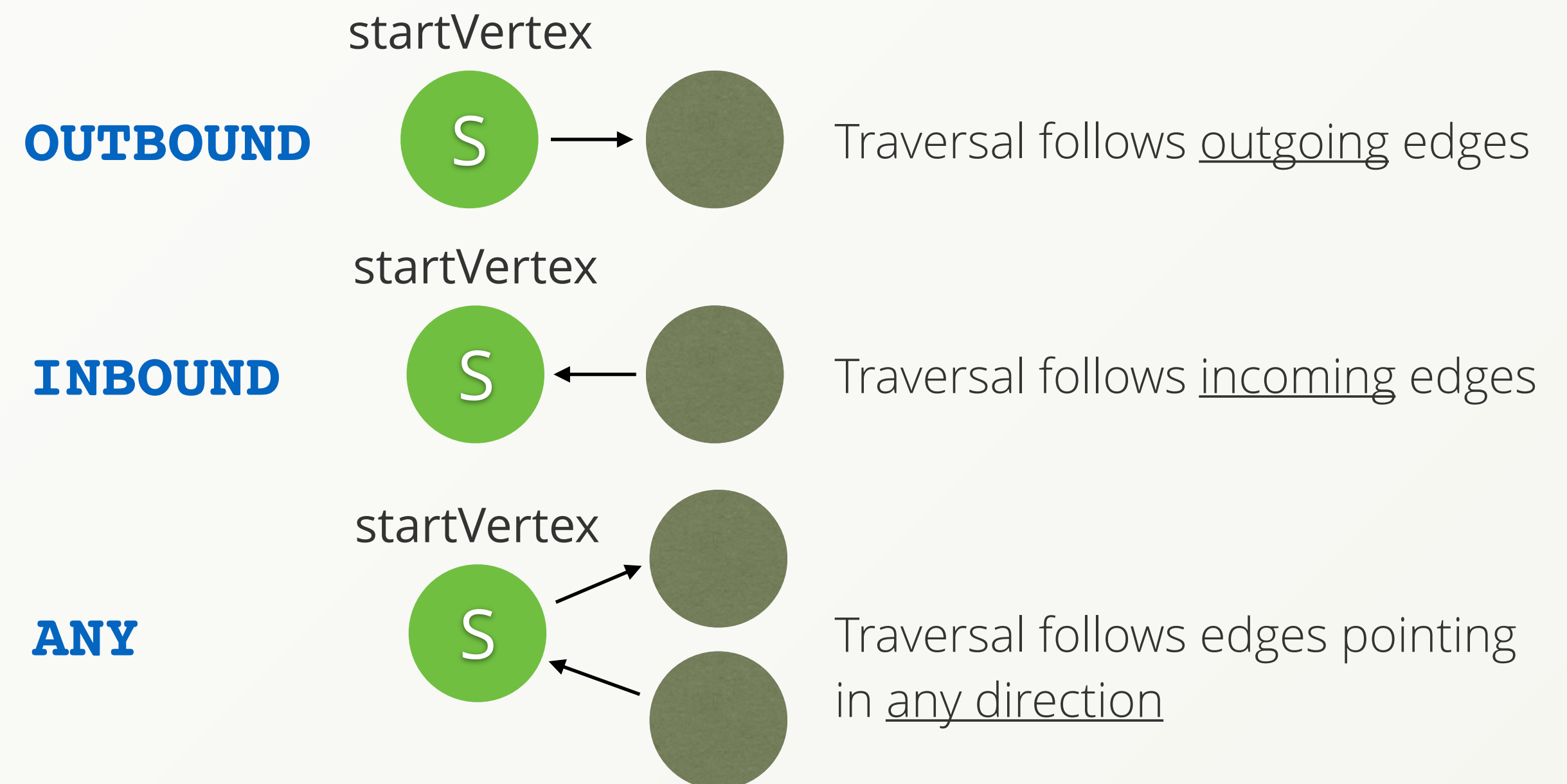
Before we do our first graph queries we should spend some time on the underlying concepts of the query options. We will go through the keywords and basic options step-by-step:

Basic Query Structure

```
FOR vertex[, edge[, path]]
  IN [min[..max]]
  OUTBOUND|INBOUND|ANY startVertex
  edgeCollection[, more...]
```

Explanation

OUTBOUND/INBOUND/ANY defines the direction of your search



Traversal in AQL documentation

edgeCollection: one or more names of collections holding the edges that we want to consider in the traversal (anonymous graph)

Hands on: First Graph Queries

Now that we have created our first edge collection, imported "flights" into ArangoDB and are familiar with the basic concepts. Let us do some graph queries.

- ▶ Return all airports I can reach from Los Angeles International (LAX)

```
FOR airport IN OUTBOUND 'airports/LAX' flights  
RETURN DISTINCT airport
```

- ▶ Return 10 flights from LAX and their destination

```
FOR airport, flight IN OUTBOUND 'airports/LAX' flights  
LIMIT 10  
RETURN {airport, flight}
```


Advanced Traversals

Depth vs. Breadth First Search & Uniqueness Options

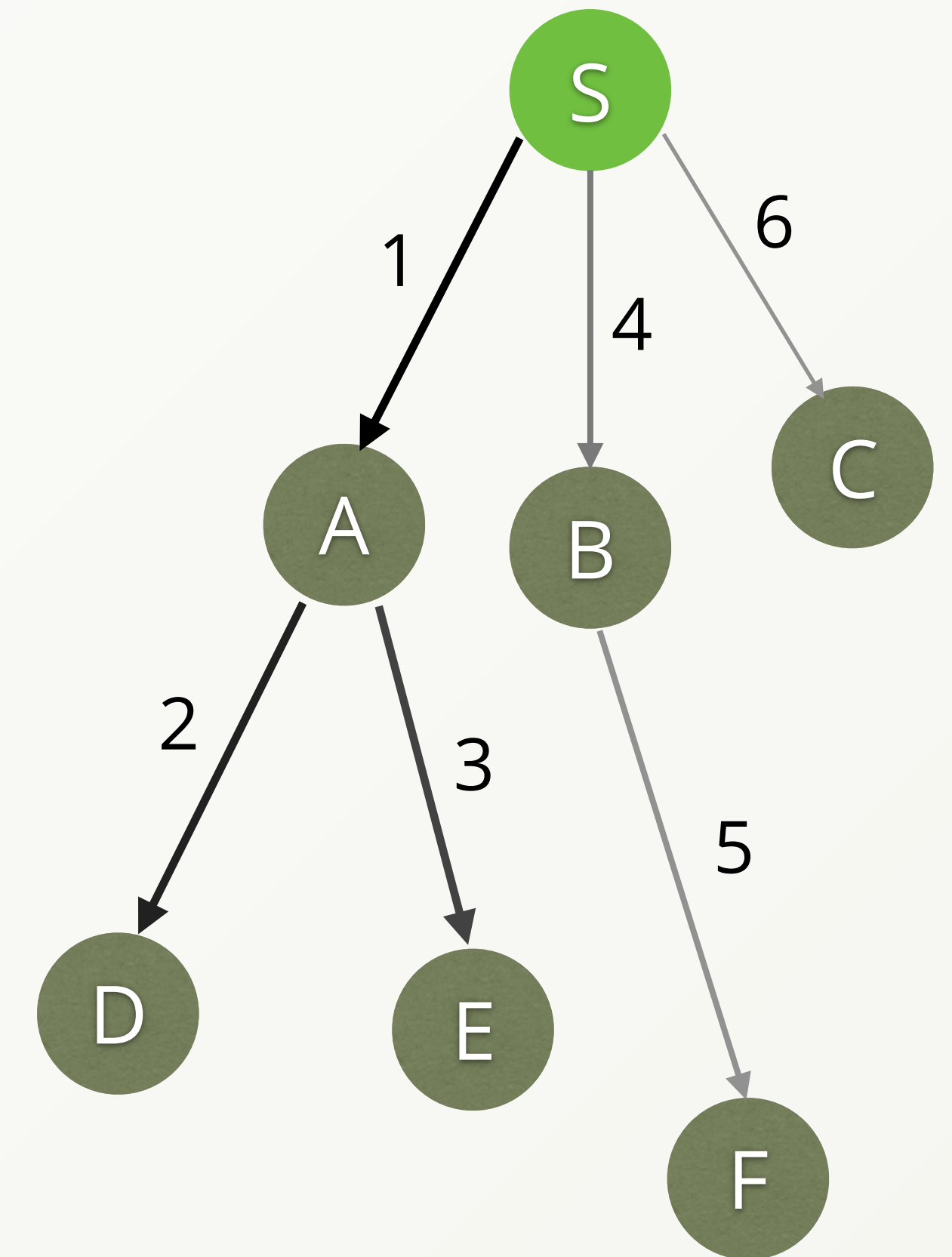
Depth or Breadth First?

For all who already took a look into the documentation for traversals recognized that there are also **OPTIONS** for the behavior of the traversal.

For traversals with a minimum depth greater than 2, you have two options how to traverse the graph:

- ▶ **Depth-first** (default): continue down the edges from the start vertex to the last vertex on that path or until the maximum traversal depth is reached, then walk down the other paths.
- ▶ **Breadth-first** (optional): follow all edges from the start vertex to the next level, then follow all edges of their neighbors by another level and continue this pattern until there are no more edges to follow or the maximum traversal depth is reached.

Depth-first search



Depth or Breadth First?

Both algorithms return the same amount of paths if all other traversal options are the same, but the order in which edges are followed and vertices are visited is different.

With a variable traversal depth of 1..2, the following paths would be found:

Depth-first

$S \rightarrow A$
 $S \rightarrow A \rightarrow D$
 $S \rightarrow A \rightarrow E$

 $S \rightarrow B$
 $S \rightarrow B \rightarrow F$

 $S \rightarrow C$

Breadth-first

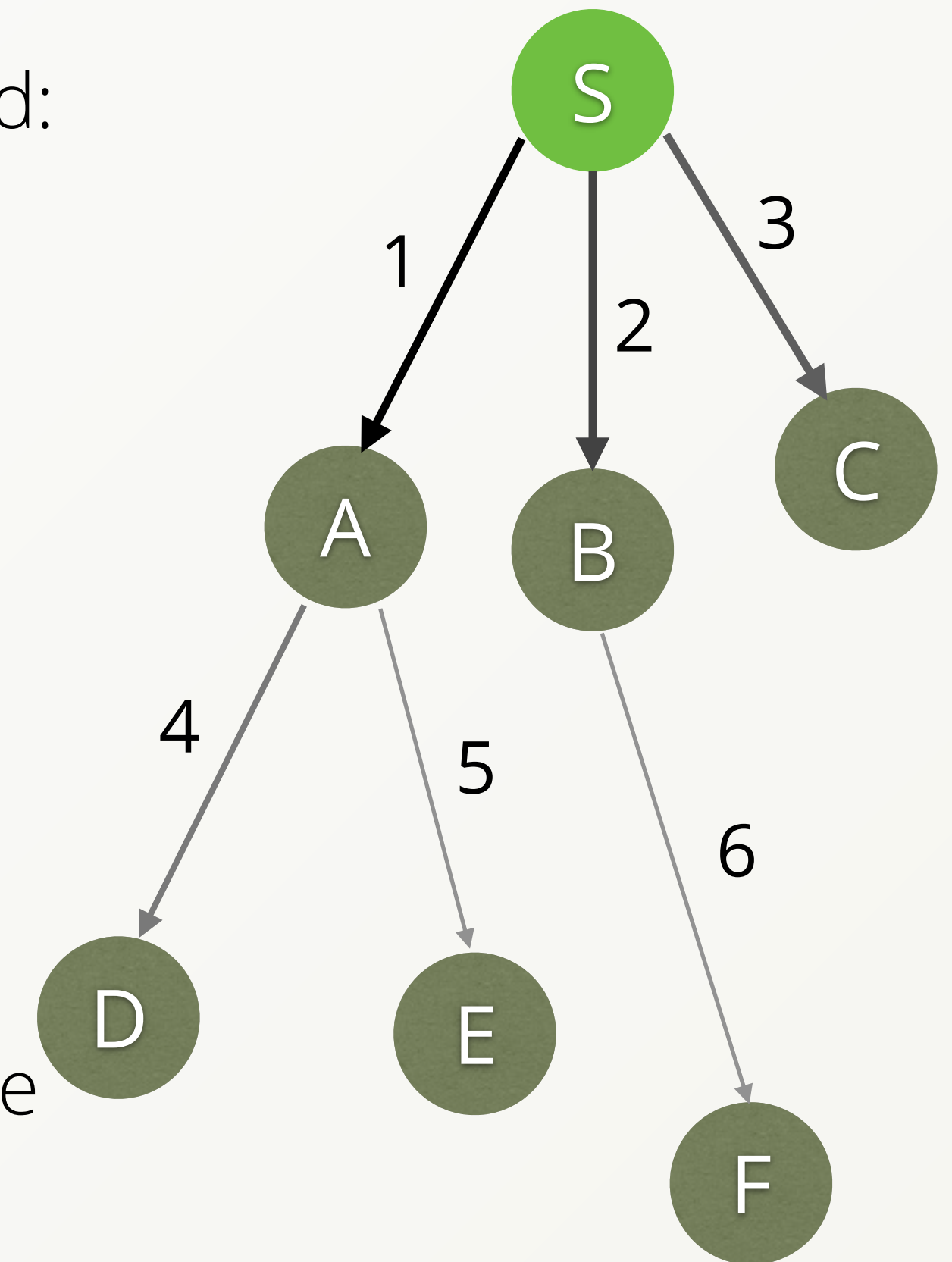
$S \rightarrow A$
 $S \rightarrow B$
 $S \rightarrow C$

 $S \rightarrow A \rightarrow D$
 $S \rightarrow A \rightarrow E$

 $S \rightarrow B \rightarrow F$

Note that there is no particular order in which edges of a single vertex are followed. Hence, $S \rightarrow C$ may be returned before $S \rightarrow A$ and $S \rightarrow B$. Shorter paths are returned before longer paths using breadth-first search still.

Breadth-first search



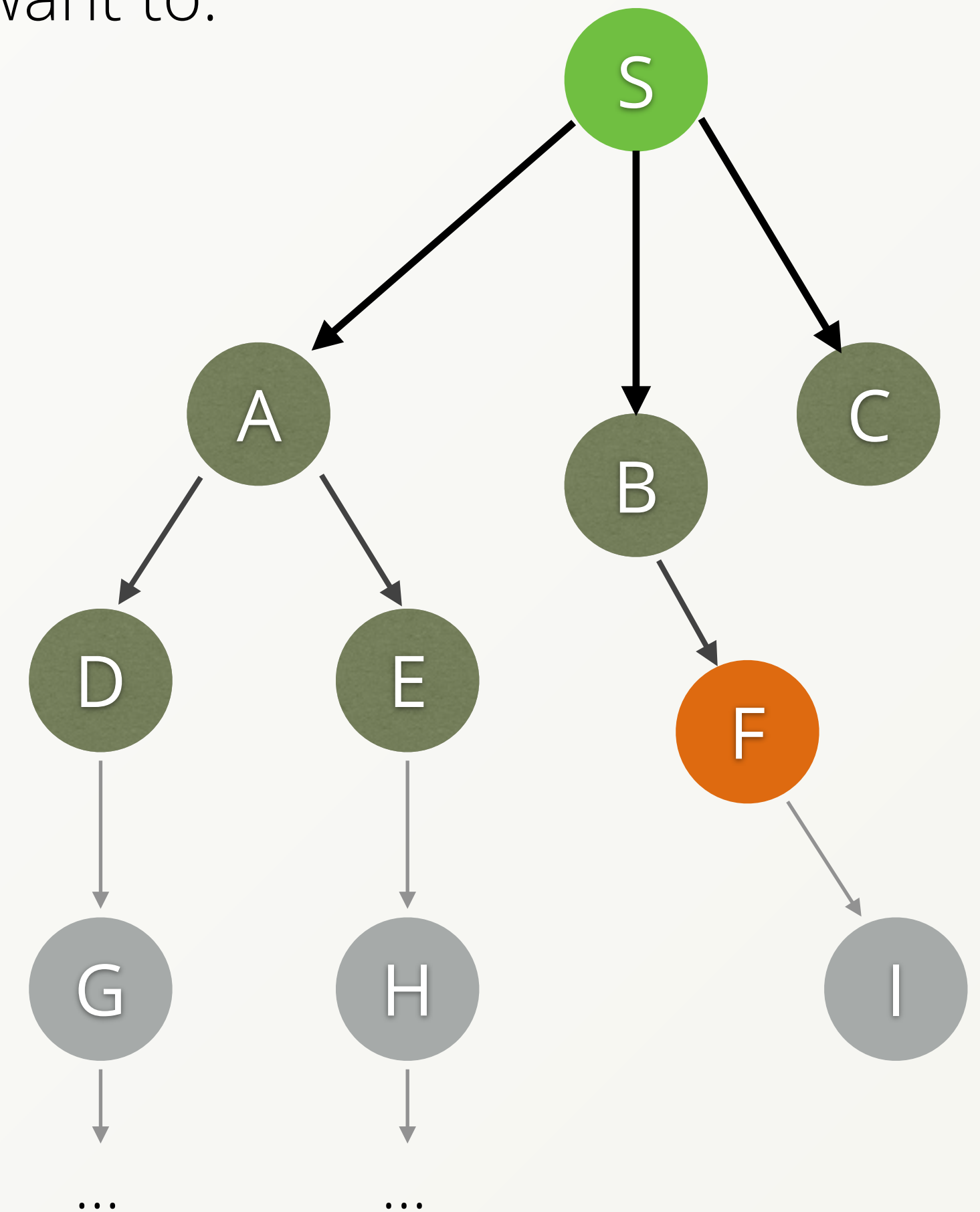
Breadth First

Breadth-first search can significantly improve performance if used together with filters and limits that prune long paths early on.

Whether it is applicable depends on the use case. For example, you want to:

- ▶ Traverse a graph from vertex S with depth 1..10
- ▶ Find 1 vertex that fulfills your criteria, lets assume vertex F meets your conditions
- ▶ Depth-first might follow the edge to A first, then all the way down up to 10 hops to D, G, E, H and more
- ▶ Breadth-first however finds F at depth 2 and never visits vertices past that level if you limit the query to a single match:

```
FOR v IN 1..10 OUTBOUND 'verts/S' edges
  OPTIONS {bfs: true}
  FILTER v._key == 'F'
  LIMIT 1
  RETURN v
```



Advanced Traversals

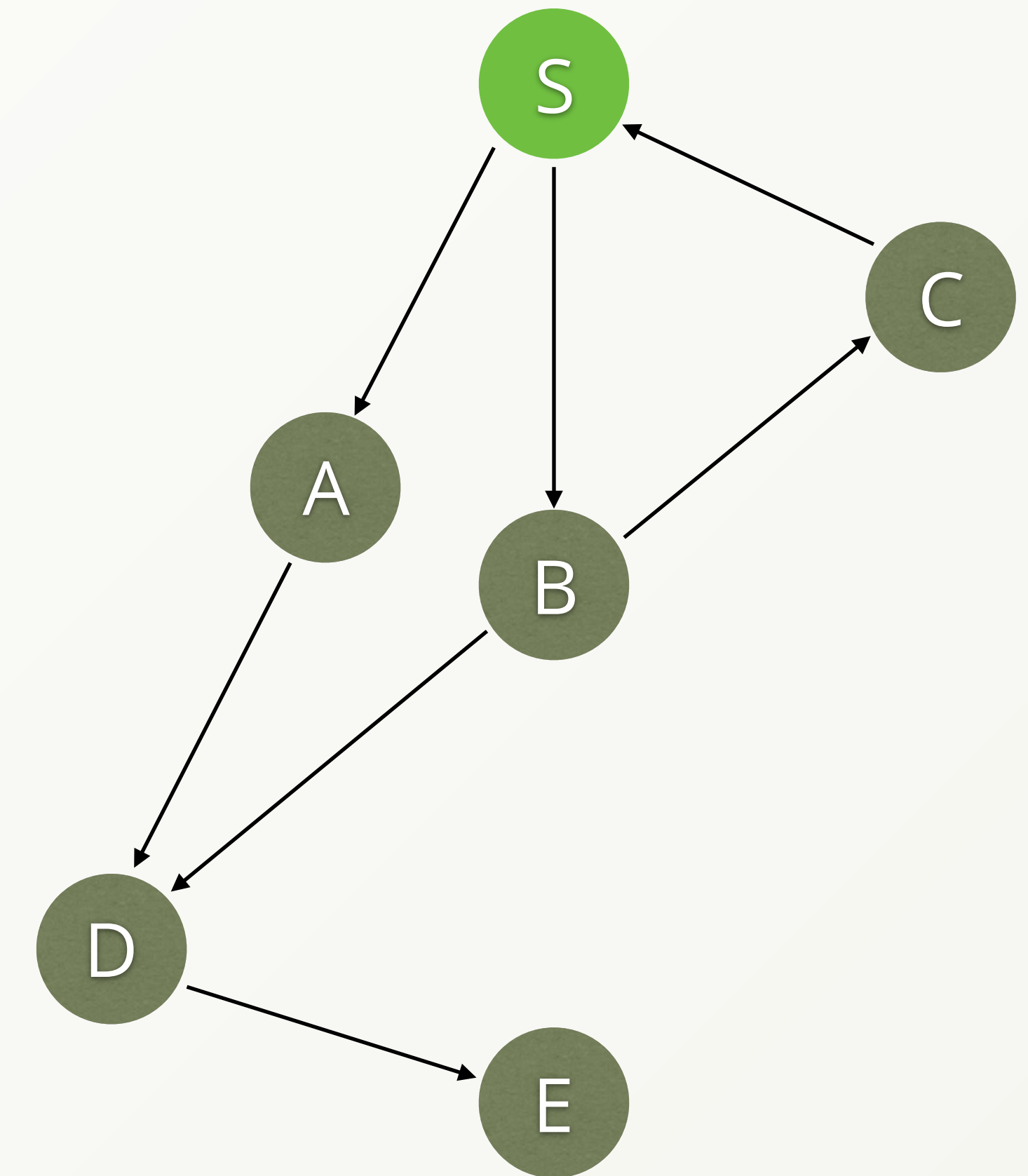
Depth vs. Breadth First Search
&
Uniqueness Options

Uniqueness Options Part I

Not every graph has just a single path from a chosen start vertex to its connected vertices. There may even be cycles in a graph.

- ▶ By default, the traversal along any of the paths is stopped if an edge is encountered again, that has already been visited. It keeps your traversals from running around in circles until the maximum traversal depth is reached. It is a safe guard to not produce a plethora of unwanted paths.
- ▶ Duplicate vertices on a path are allowed unless the traversal is configured otherwise.

Graph with cycle $S \rightarrow B \rightarrow C \rightarrow S$ and multiple paths from S to E



Uniqueness Options Part II

The following query specifies the uniqueness options explicitly, although the ones shown are used by default anyway:

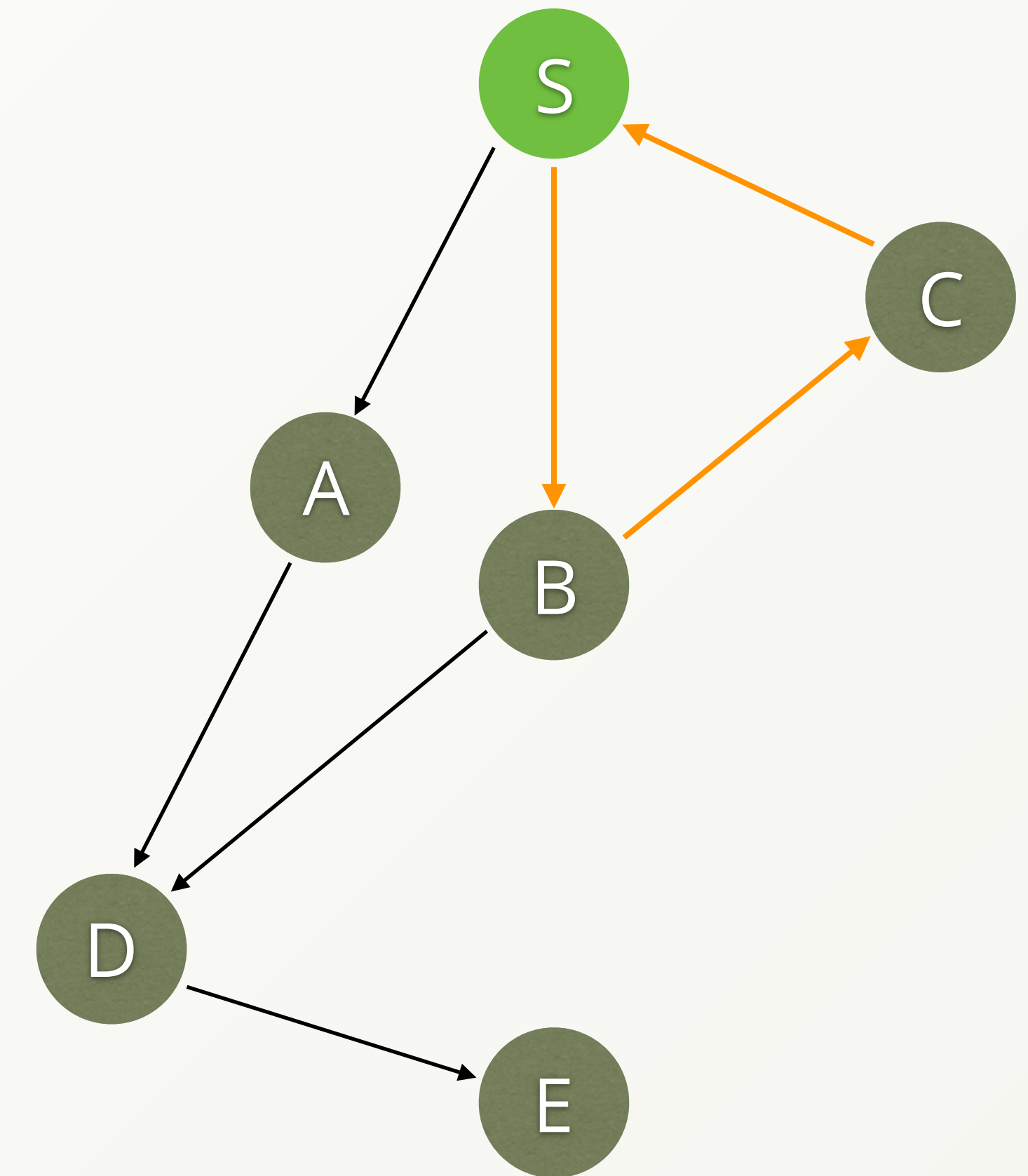
```
FOR v, e, p IN 1..5 OUTBOUND 'verts/S' edges
  OPTIONS {
    uniqueVertices: 'none',
    uniqueEdges: 'path'
  }
RETURN CONCAT_SEPARATOR('-', p.vertices[*]._key)
```

We use the path variable p , which is emitted by the traversal, and concatenate all vertex keys of the paths neatly as single string per path, like "S->A->D->E". The array expansion operator $[*]$ is used for convenience.



Array expansion in AQL documentation

Graph with cycle $S \rightarrow B \rightarrow C \rightarrow S$ and multiple paths from S to E



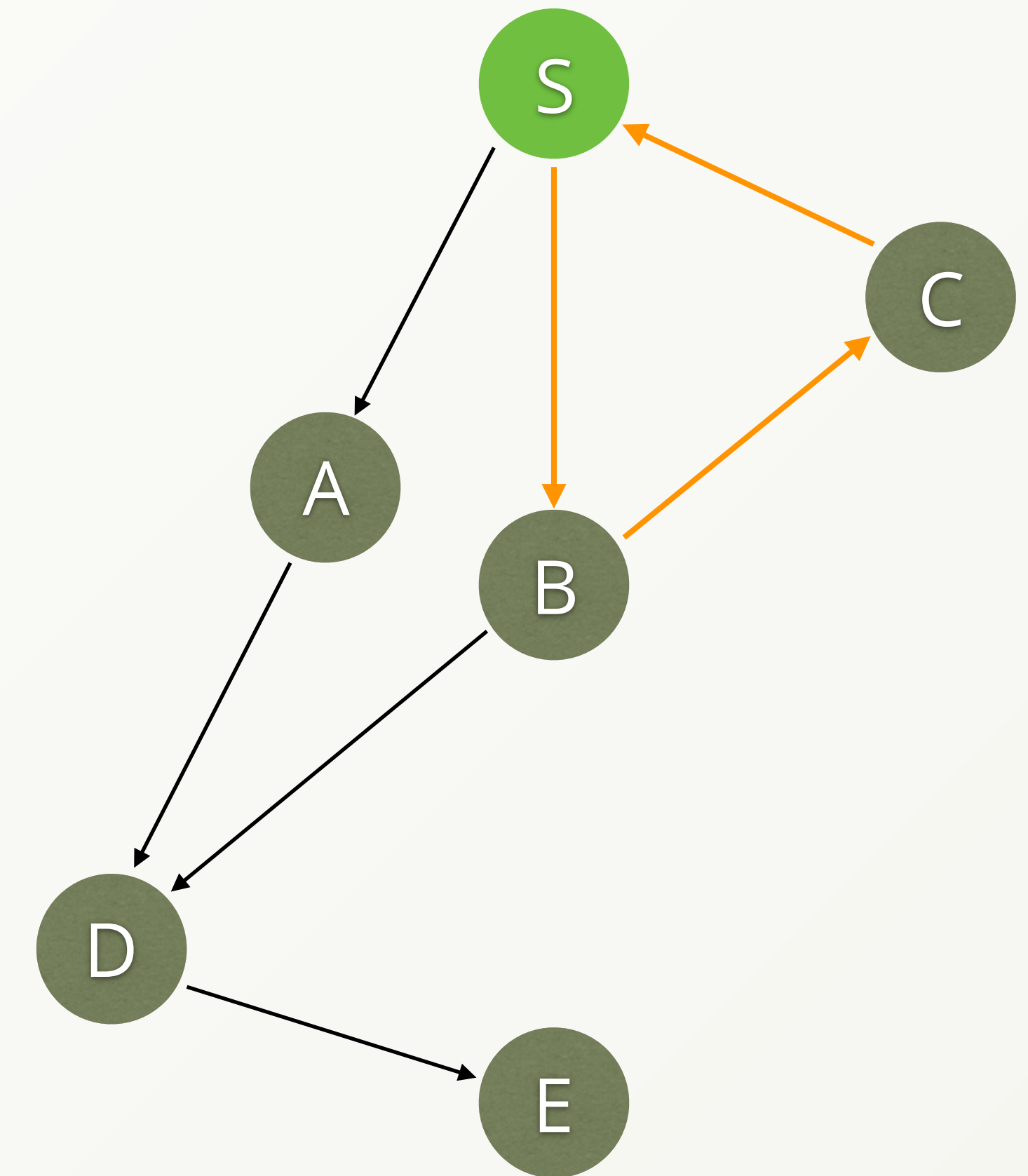
Uniqueness Options Part III

The query finds a total of 10 paths. One of them is $S \rightarrow B \rightarrow C \rightarrow S$. The start vertex is also the last vertex on that path, which is possible because uniqueness of vertices is not ensured.

A path such as $S \rightarrow B \rightarrow C \rightarrow S \rightarrow B \rightarrow C$ is not present in the result, because uniqueness of edges for paths avoids following the same edge twice.

- ▶ **uniqueEdges:** `'none'` would make the traverser follow the edge from S to B and from B to C again. It would only stop there, because the maximum depth of 5 is reached.

Graph with cycle $S \rightarrow B \rightarrow C \rightarrow S$ and multiple paths from S to E



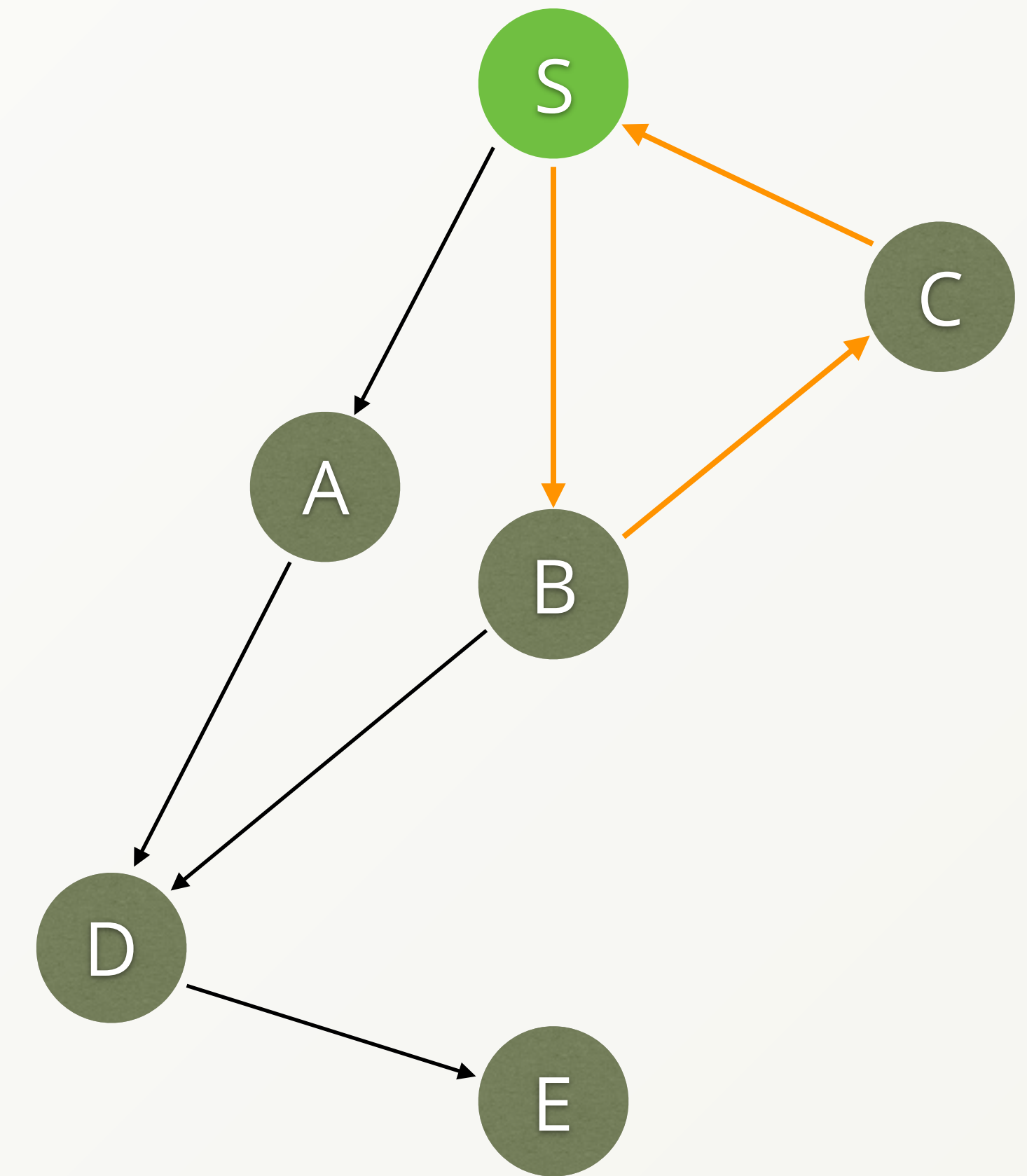
Uniqueness Options Part IV

To stop the start vertex (or other vertices) from being visited more than once, we can enable uniqueness for vertices in two ways:

- ▶ **uniqueVertices:** `'path'` ensures no duplicate vertices on each individual path.
- ▶ **uniqueVertices:** `'global'` ensures every reachable vertex to be visited once for the entire traversal.

It requires **bfs:** `true` (breadth-first search). It is not supported for depth-first search, because the results would be completely non-deterministic (varying between query runs), as there is no rule in which order the traverser follows the edges of a vertex. The uniqueness rule would lead to randomly excluded paths whenever there are multiple paths to choose from, of which it would take one.

Graph with cycle $S \rightarrow B \rightarrow C \rightarrow S$ and multiple paths from S to E



Uniqueness Options Part V

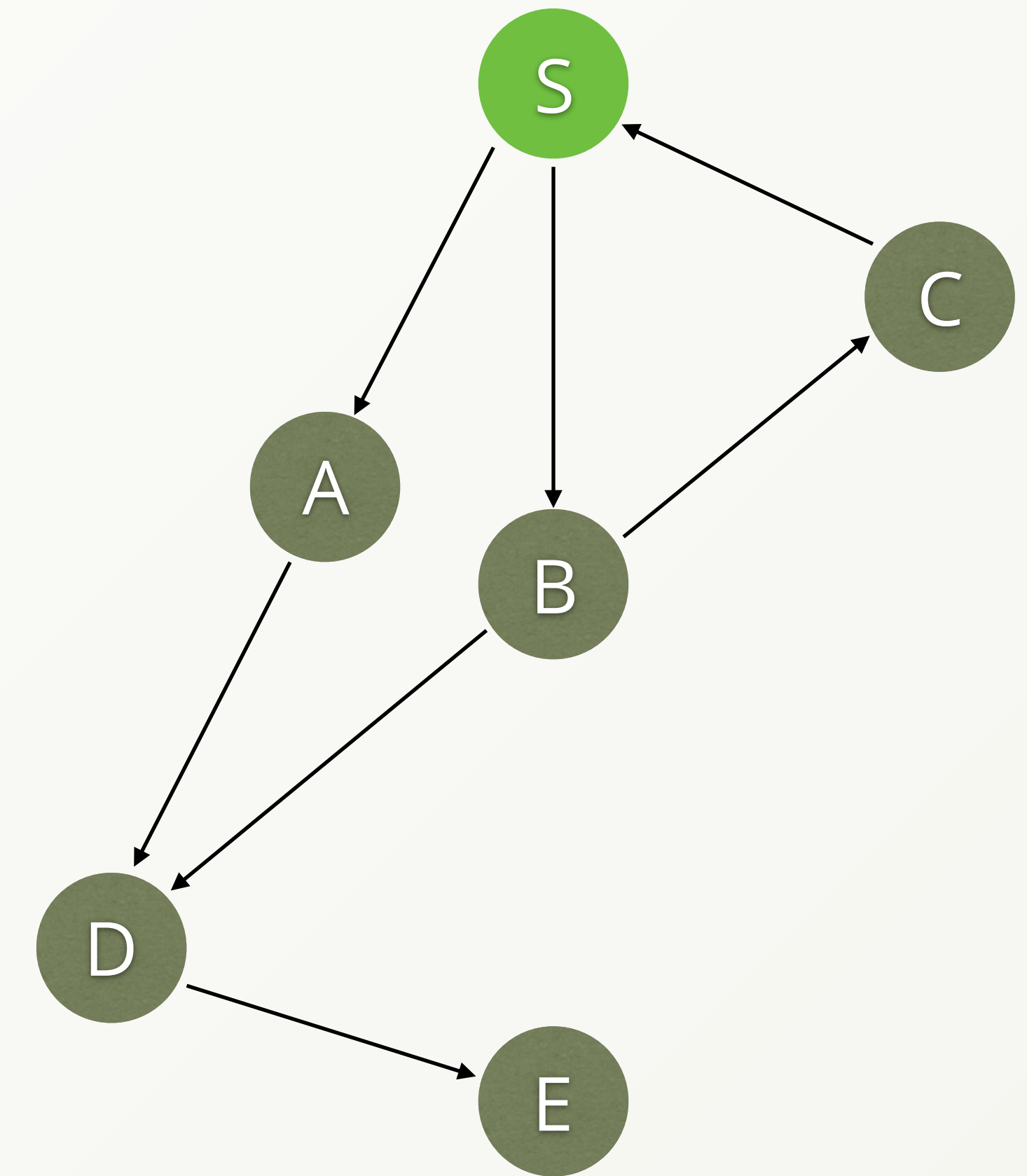
```
FOR v IN 0..5 OUTBOUND 'verts/S' edges
  OPTIONS {
    bfs: true,
    uniqueVertices: 'global'
  }
RETURN v._key
```

The query gives us all vertex keys of this example graph exactly once. Path or uniqueness of vertices would give us a lot of duplicates instead, 14 in total.

Which edges are actually followed in this traversal is not deterministic, but since it is breadth-first search, every reachable vertex is guaranteed to be visited one way or another.

Note: A depth of zero makes the traversal include the start vertex, which would otherwise only be accessible via the emitted path variable like `p.vertices[0]`.

Graph with cycle $S \rightarrow B \rightarrow C \rightarrow S$ and multiple paths from S to E



Hands on: First Graph Queries Part II

For our domestic flights example we might want to have all airports directly reachable from a given airport. Let's see which airports we can reach from Los Angeles

- ▶ Return all airports directly reachable from LAX

```
FOR airport IN OUTBOUND 'airports/LAX' flights
  OPTIONS {bfs: true, uniqueVertices: 'global'}
  RETURN airport
```

- ▶ Compare the execution times to this earlier shown query, which returns the same airports:

```
FOR airport IN OUTBOUND 'airports/LAX' flights
  RETURN DISTINCT airport
```

You will see a significant performance improvement.

What happens is that **RETURN DISTINCT** de-duplicates airports only after the traversal has returned all vertices (huge intermediate result), whereas **uniqueVertices: 'global'** is a traversal option that instructs the traverser to ignore duplicates right away.

Excursion

The LET keyword in AQL

Hands on: Storing Results in Variables Part I

Results of simple expressions as well as of entire subqueries can be stored in variables. To declare a variable, use the **LET keyword** followed by the variable name, an equal sign and the expression. The code must be in parentheses if the expression is a subquery.

```
LET h = FLOOR(f.DepTime / 100)
LET m = f.DepTime % 100
LET formatted = CONCAT(h, ':', m)
RETURN { hours: h, minutes: m, formatted }
```

The time format is like 1245. Hours and minutes can be separated with some math: divide by 100 and round off the decimal places for the hours part and modulo 100 for the division remainder to get the minutes part.

The calculated values are stored in two different variables h and m. These variables are then used to create another variable formatted like „12:45“ and all three are returned as result.

RETURN { myVariable } is a short form of
RETURN { myVariable: myVariable } if you wondered.

[LET in AQL documentation](#)



Hands on: Storing Results in Variables Part I

In below example, the hours and minutes of the departure time are pre-calculated and stored in the variables `h` and `m`. They are later used to create an ISO timestamp together with the date attributes of the data:

```
FOR f IN flights
  FILTER f._from == 'airports/BIS'
  LIMIT 100
  LET h = FLOOR(f.DepTime / 100)
  LET m = f.DepTime % 100
  RETURN {
    year: f.Year,
    month: f.Month,
    day: f.DayofMonth,
    time: f.DepTime,
    iso: DATE_ISO8601(f.Year, f.Month, f.DayofMonth, h, m)
  }
```

DATE_ISO8601 in AQL documentation



Hands on: Storing Results in Variables Part II

Your results in „Table View“ should look like this

Query

100 elements

0.180 s

JSON

Table

year	month	day	time	iso
2008	1	1	632	2008-01-01T06:32:00.000Z
2008	1	15	1910	2008-01-15T19:10:00.000Z
2008	1	15	1856	2008-01-15T18:56:00.000Z
2008	1	15	1633	2008-01-15T16:33:00.000Z
2008	1	15	1431	2008-01-15T14:31:00.000Z
2008	1	15	1305	2008-01-15T13:05:00.000Z
2008	1	15	746	2008-01-15T07:46:00.000Z
2008	1	15	620	2008-01-15T06:20:00.000Z
2008	1	14	1913	2008-01-14T19:13:00.000Z
2008	1	14	1851	2008-01-14T18:51:00.000Z

Advanced Graph Queries

Shortest_Path & Pattern Matching

Shortest_Path - What is it?

A shortest path query finds a connection between two given documents (startVertex and targetVertex) with the fewest amount of edges.

With our domestic flights dataset we could search for e.g. the connection between two airports with the fewest stops.

The shortest path algorithm needs a startVertex and a targetVertex. Let's have a look at an example query.

[Shortest_Path in AQL documentation](#)



Shortest_Path - Query Structure



Source: Google Maps

- Find the shortest path between airports BIS and JFK.

```
FOR v IN OUTBOUND  
SHORTEST_PATH 'airports/BIS'  
TO 'airports/JFK' flights  
RETURN v
```

We defined "BIS" as our startVertex and JFK as our targetVertex.

Hands on: Shortest_Path - Part II



Source: Google Maps

The result of the previous Shortest_Path query will show you, that you have to change in e.g. DEN (Denver) to get to JFK

Note that the Shortest_Path can return different results. It just finds and returns one of possibly multiple shortest paths. In this case it found BIS→DEN→JFK

Hands on: Shortest_Path - Part III

Sometimes you just want the length of the shortest path. To achieve this you can use LET again.

- ▶ Return the minimum number of flights from BIS to JFK

```
LET airports = (
  FOR v IN OUTBOUND
  SHORTEST_PATH 'airports/BIS'
  TO 'airports/JFK' flights
  RETURN v
)
RETURN LENGTH(airports) - 1
```

Your result should be 2.

Notes

- ▶ we placed a "-1" at the end of the query to not count the endVertex as a step!
- ▶ **using the shortest path algorithm you can not apply filters.**
Therefore you need to use Pattern Matching

Advanced Graph Queries

Shortest_Path
&
Pattern Matching

Hands on: Pattern Matching - Part I

Let's use a real-world question to explain how Pattern Matching works.

Q: I want to find the connection between BIS and JFK with the lowest total travel time.

Answering this question is a bit more complex so let's go through it step-by-step...

STEP 1

We just want to get from BIS to JFK (We don't care about time, day or month)

Note: We already know that 2 steps is the shortest path, that's why "**IN 2 OUTBOUND**"

```
FOR v, e, p IN 2 OUTBOUND 'airports/BIS' flights
  FILTER v._id == 'airports/JFK'
  LIMIT 5
  RETURN p
```

The result shows a graph but when switching to JSON view you will see 5 different flight plans from BIS → JFK

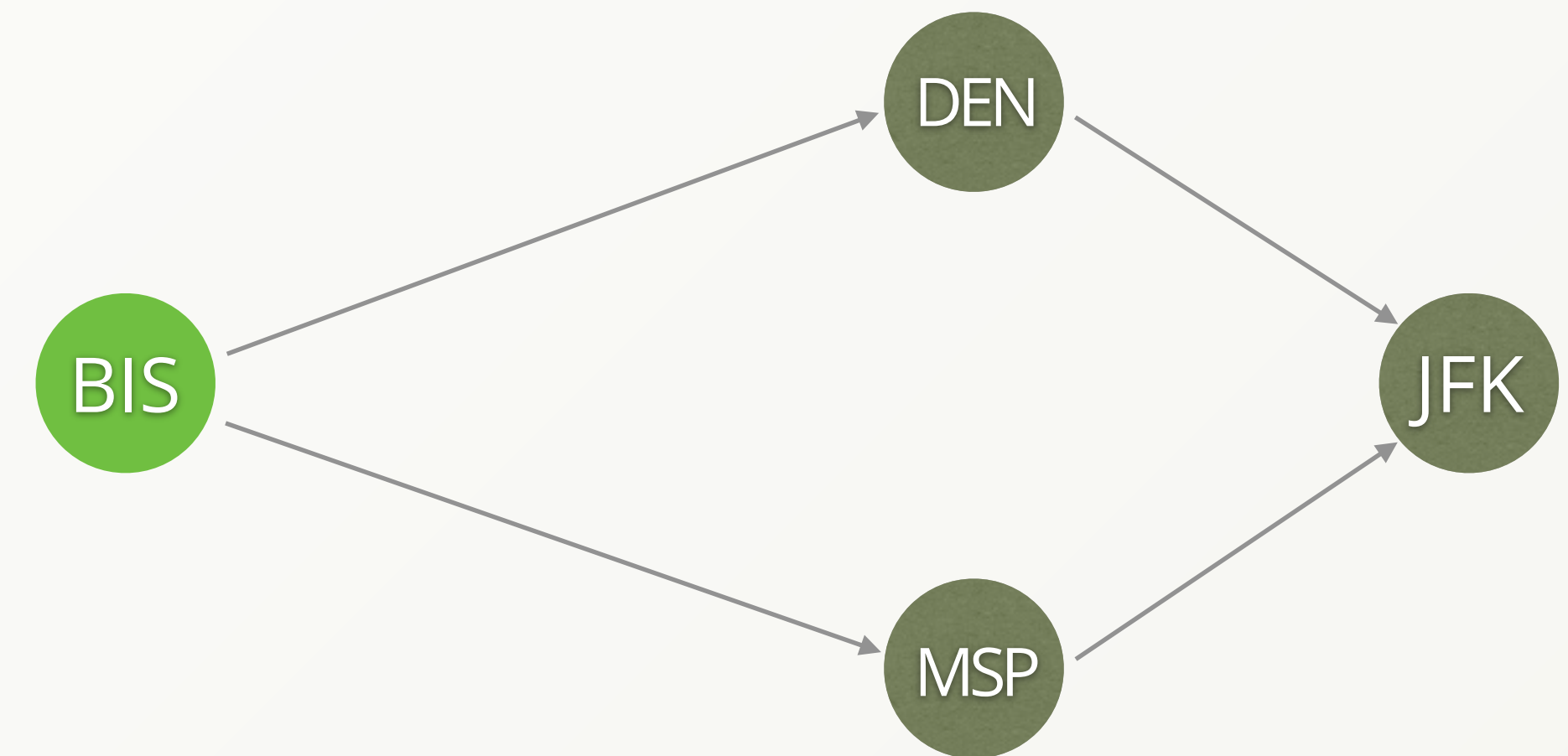
Hands on: Pattern Matching - Part II

STEP 2

We make sure that we fly on the same date on **each** segment, here on New Year (Month = 1, DayofMonth = 1).

```
FOR v, e, p IN 2 OUTBOUND 'airports/BIS' flights
  FILTER v._id == 'airports/JFK'
  FILTER p.edges[*].Month ALL == 1
  FILTER p.edges[*].DayofMonth ALL == 1
  LIMIT 5
  RETURN p
```

- ▶ Note that the result now shows two possibilities. In this case we could also fly via MSP (Minneapolis).



Array comparison operators (ALL, ANY, NONE) in AQL documentation

Hands on: Pattern Matching - Part III

STEP 3

We make sure that we pick those flights which have the lowest total flight time. For that, we calculate arrival time at target airport minus departure time at start airport and use the result to sort in ascending order. Finally, we return the top 5 flights as well as the flight time.

```
FOR v, e, p IN 2 OUTBOUND 'airports/BIS' flights
  FILTER v._id == 'airports/JFK'
  FILTER p.edges[*].Month ALL == 1
  FILTER p.edges[*].DayofMonth ALL == 1
  LET flightTime = e.ArrTime - p.edges[0].DepTime
  SORT flightTime ASC
  LIMIT 5
  RETURN { flight: p, time: flightTime }
```

Hands on: Pattern Matching - Part III

STEP 3 - Result

Have a look at the results! You will see that the first flightTime is negative.

```
71         "lat": 40.63975111,  
72         "long": -73.77892556  
73     }  
74 ]  
75 },  
76 "time": -1288  
77 },  
78 {  
79     "flight": {  
80         "edges": [  
81             {  
82                 "_key": "7523393",  
83                 "_id": "flights/7523393",  
84                 "_from": "airports/BIS",
```

So we need a 4th step to **make sure that Departure and Arrival time don't overlap.**

Hands on: Pattern Matching - Part IV

Final Step

Let's put in the final part of the query to get the best flights to JFK. In this case we assume that we need 20min to get our connecting flight to JFK. By adding another filter that ensures that the next plane does not take off before we landed with the previous one, plus another 20min for the transit, we won't see negative flight times anymore and get viable flight connections:

```
FOR v, e, p IN 2 OUTBOUND 'airports/BIS' flights
  FILTER v._id == 'airports/JFK'
  FILTER p.edges[0].ArrTime + 20 < p.edges[1].DepTime
  FILTER p.edges[*].Month ALL == 1
  FILTER p.edges[*].DayofMonth ALL == 1
  LET flightTime = e.ArrTime - p.edges[0].DepTime
  SORT flightTime ASC
  LIMIT 5
  RETURN { flight: p, time: flightTime }
```

Have a look into the results and you will see, that our best flight would not be via Denver (DEN) but via Minneapolis (MSP) — another possible shortest path.

Final tasks :)

Add-ons

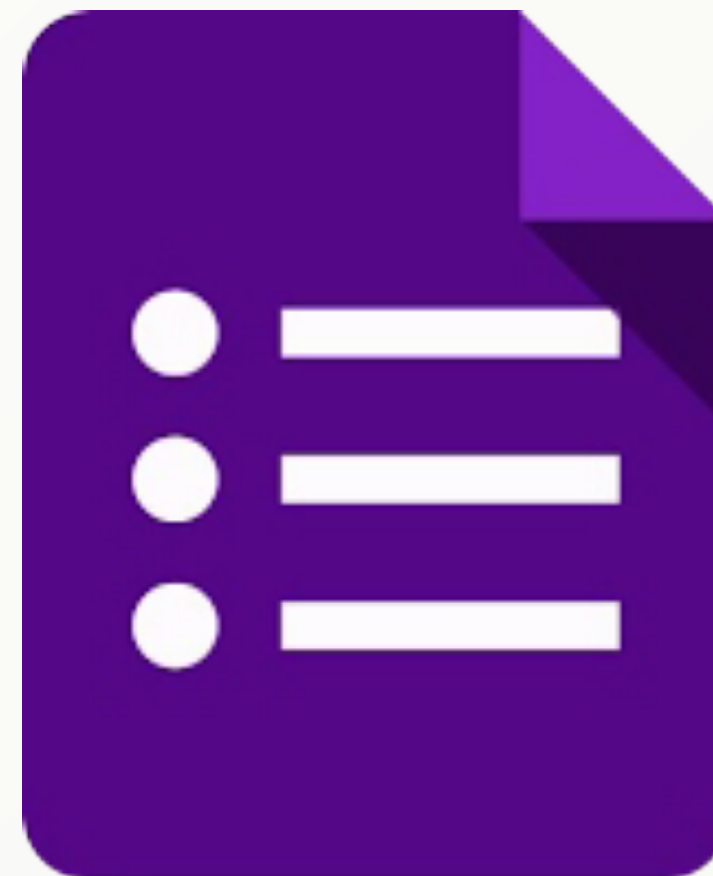
&

Support ArangoDB

We hope you enjoyed the course and it helped you to get started.

What would you like to learn next ?

Tell us with 3 clicks:



You can support ArangoDB in multiple ways...

Feedback to the course



Add "ArangoDB" to your skills



Tweet about ArangoDB



Hope you enjoyed this course!

Your ArangoDB team