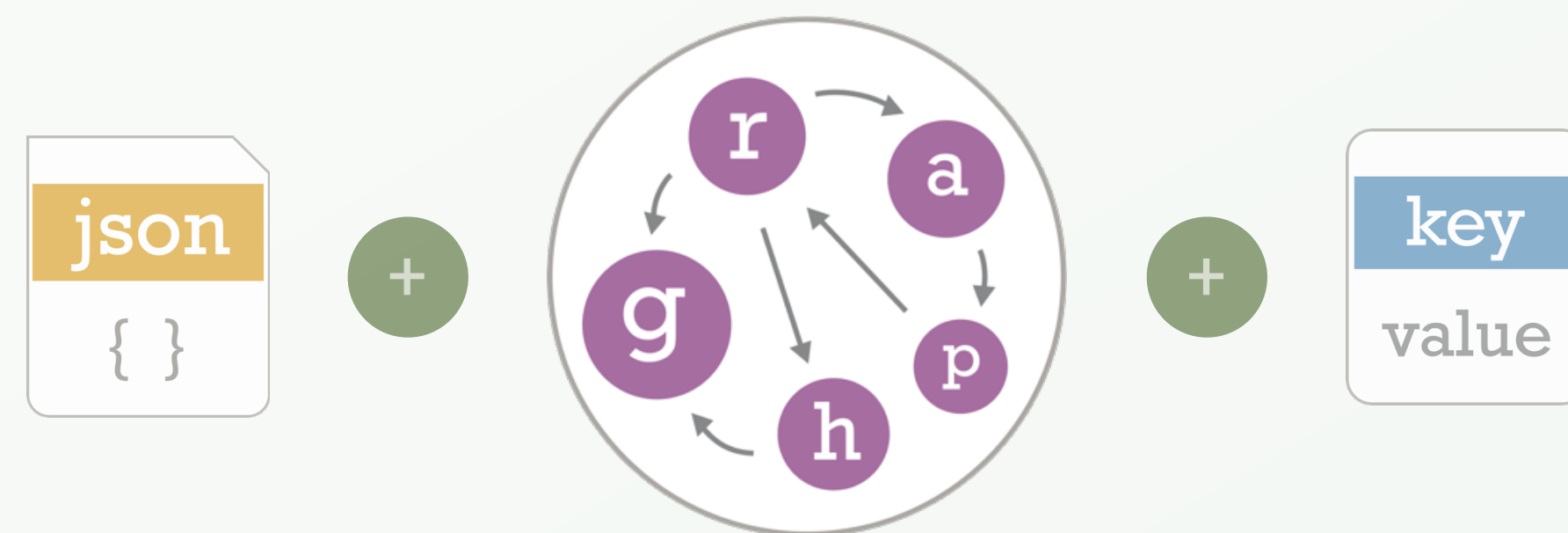


# ArangoDB

---

*Graph Course for Freshers:  
The Shortest Path to first graph skills*



**2019 Edition**

# Welcome on board

---

This is a short journey for developers, data scientists and all other interested folks. In this course you will learn how to get started with ArangoDB's graph related features and some other bits and pieces.

If you are new to ArangoDB, don't be afraid – we will start with the basic things. Also don't mind the number of pages too much, there are plenty of illustrations and exercises are optional!

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 enable you to write many interesting queries to answer a variety of questions.

Hope you will enjoy the course!

**Special thanks to @darkfrog** *for his feedback to the beta version and thousands of enthusiastic downloaders of this course!*

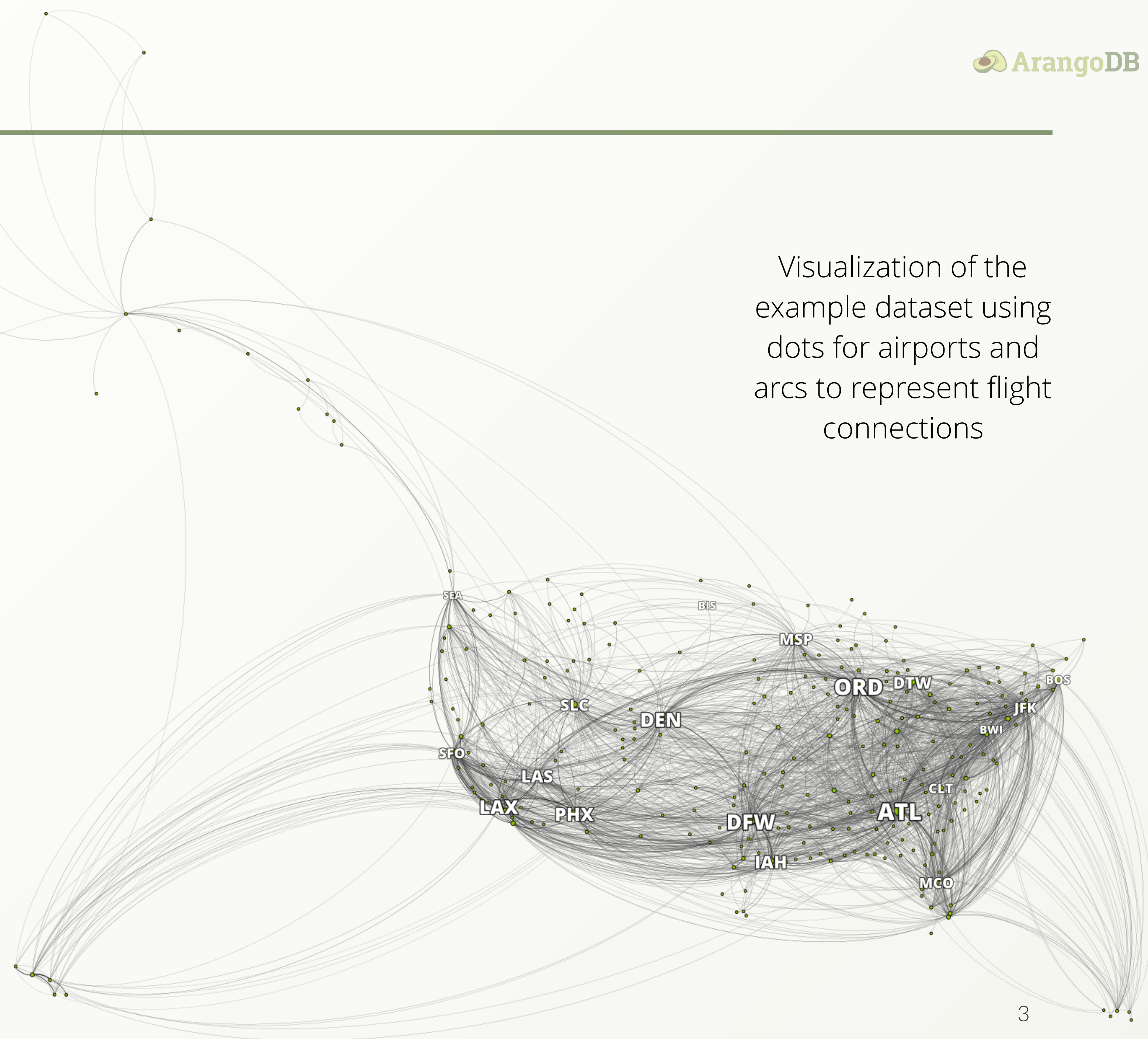




# What you will learn

- ▶ Basics about graphs, in general and in ArangoDB
- ▶ Architecture of ArangoDB and what **multi-model** is
- ▶ How to import (graph) data
- ▶ Doing queries in ArangoDB's query language **AQL**
  - ▶ Data retrieval with filtering, sorting and more
  - ▶ Simple graph queries
  - ▶ Traversing through a graph with different options
  - ▶ Shortest path queries

Visualization of the example dataset using dots for airports and arcs to represent flight connections



The symbol below indicates a link.

**If you read this course in a browser, click on links with the middle-mouse button to open a new tab!**



Graph Course page

The same goes for [underlined links](#).



# Table of Content

---

- ▶ Introduction (p.6)
  - ▶ Graph Basics (p.7)
  - ▶ The Example Dataset (p.12)
- ▶ Concepts of ArangoDB (p.15)
  - ▶ What is Multi-Model? (p.16)
  - ▶ ArangoDB Architecture (p.19)
- ▶ Preparations for this Course (p.22)
  - ▶ Download and Install ArangoDB (p.23)
  - ▶ Import the Dataset (p.24)
- ▶ Starting with the dataset (p.32)
  - ▶ ArangoDB Query Editor (p.33)
  - ▶ First AQL Queries – Hands on (p.35)
- ▶ Graph Traversals (p.39)
  - ▶ Traversals explained (p.40)
  - ▶ Graph Traversal Syntax (p.42)
- ▶ Traversal Options (p.44)
  - ▶ Depth vs. Breadth First Search (p.45)
  - ▶ Uniqueness Options (p.48)
  - ▶ Traversal Options – Hands on (p.52)
- ▶ Advanced Graph Queries (p.54)
  - ▶ Shortest Path (p.55)
  - ▶ Pattern Matching (p.59)
- ▶ Landing (p.60)
  - ▶ Survey and Support (p.61)
  - ▶ Exercise Solutions (p.62)

# Introduction

---

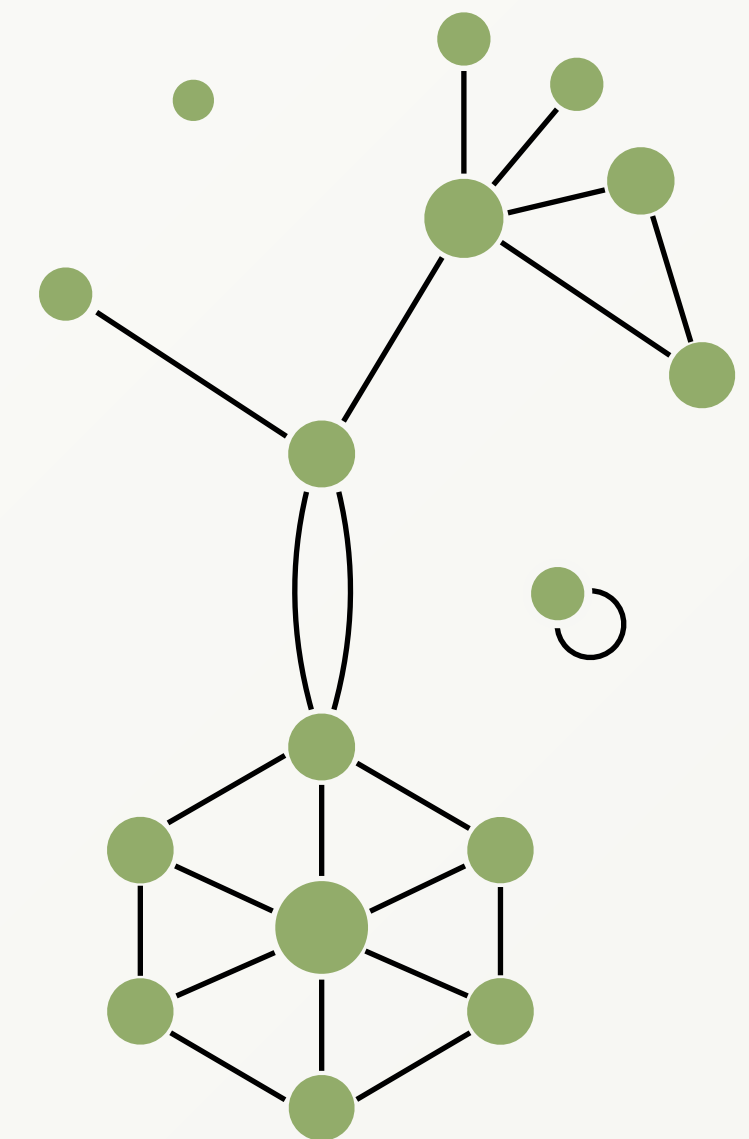
## **Graph Basics & The Example Dataset**

# Graph Basics

What is a graph? There are multiple definitions and types. A brief overview:

In discrete mathematics, a graph is defined as **set of vertices and edges**. In computing it is considered an **abstract data type** which is really good to represent connections or relations – unlike the tabular data structures of relational database systems, which are ironically very limited in expressing relations.

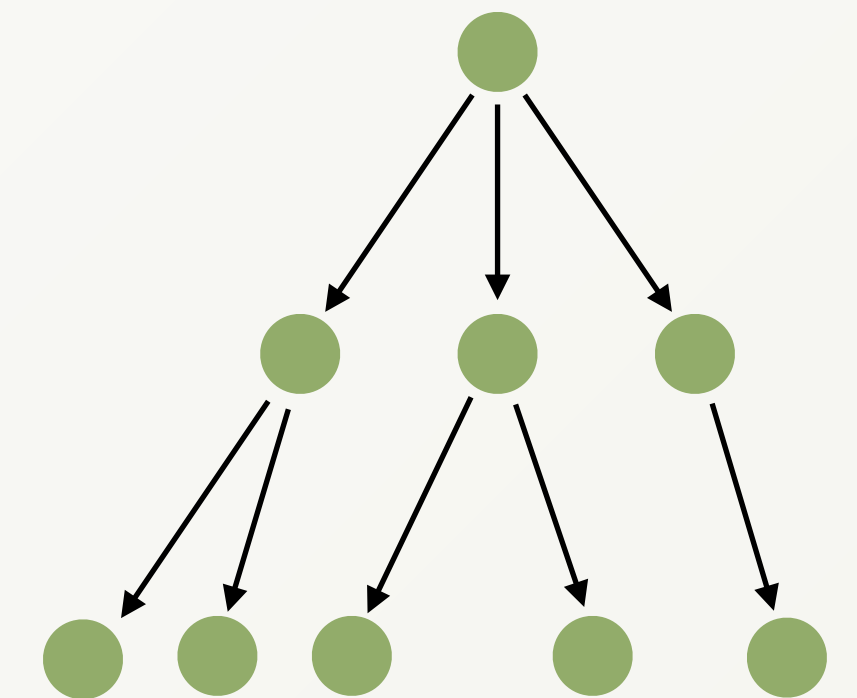
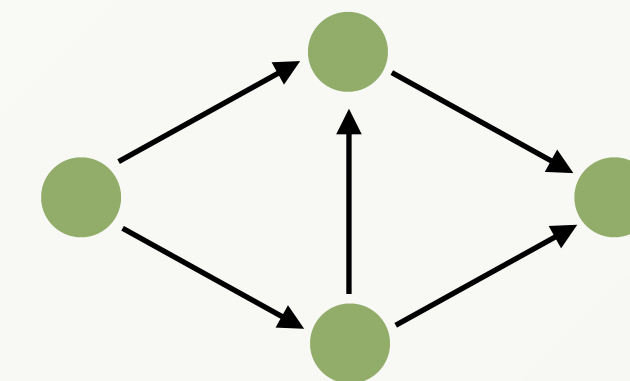
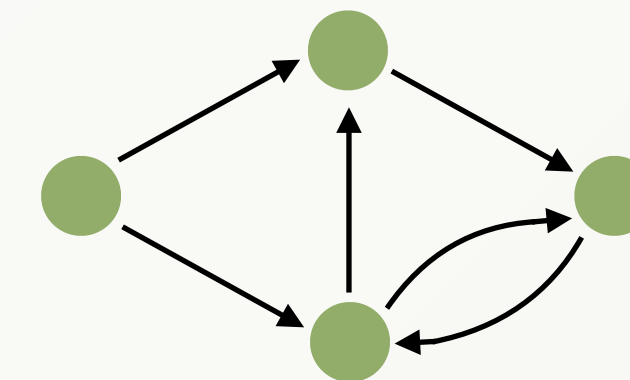
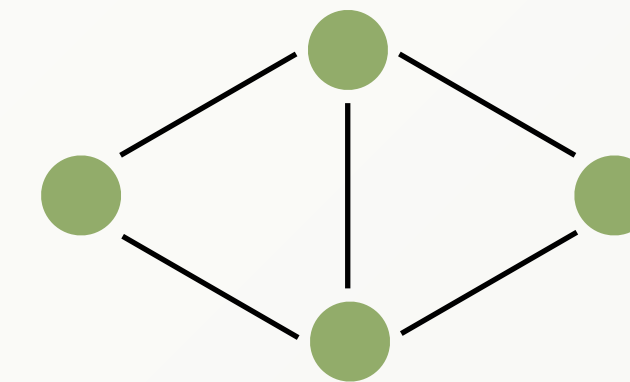
A good metaphor for graphs is to think of nodes as **circles** and edges as **lines** or **arcs**. The terms *node* and *vertex* are used interchangeably here. Usually vertices are connected by edges, making up a graph. Vertices don't have to be connected, but they may also be connected with more than one other vertex via multiple edges. You may also find vertices connected to themselves.



● Vertex  
— Edge

Important types of graphs:

- ▶ **Undirected** – edges connect pairs of nodes without having a notion of direction
- ▶ **Directed** – edges have a direction associated with them (the lines/arcs have arrow heads in depictions)
- ▶ **DAG** – *Directed Acyclic Graph*: edges have a direction and there are no loops. In the most simple case, this means that if you have vertices A and B and an edge from A to B, then there must not be another edge from B to A. One example for a DAG is a tree topology.



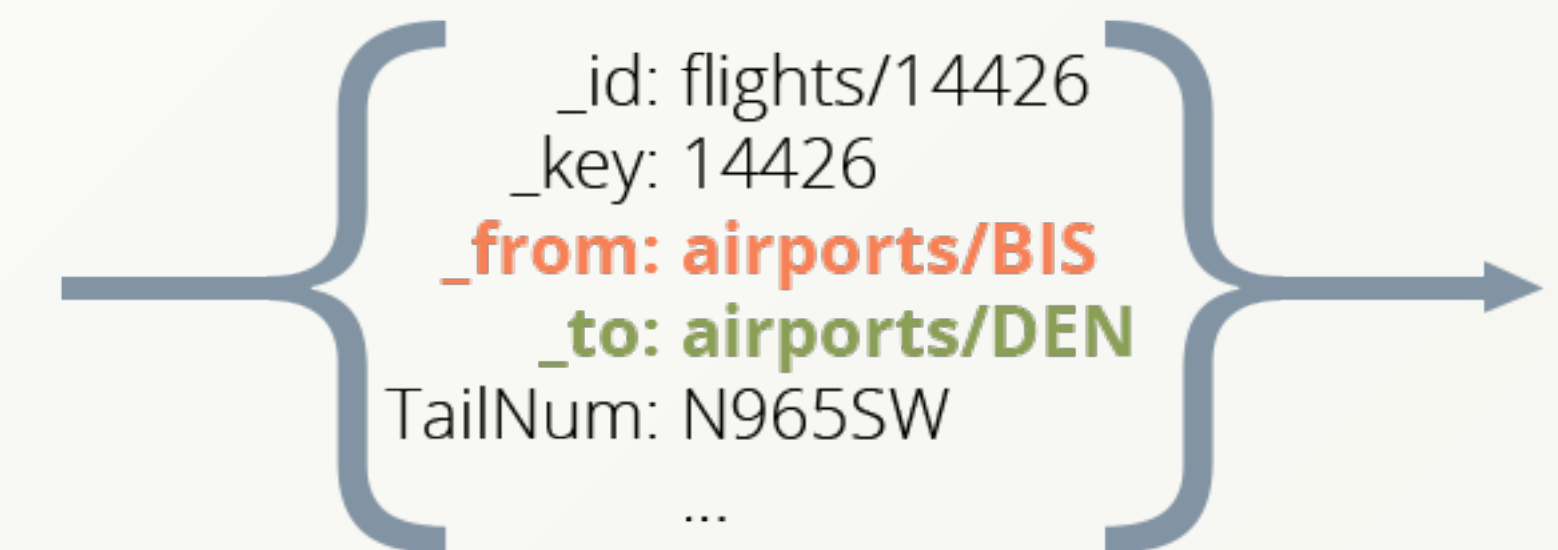
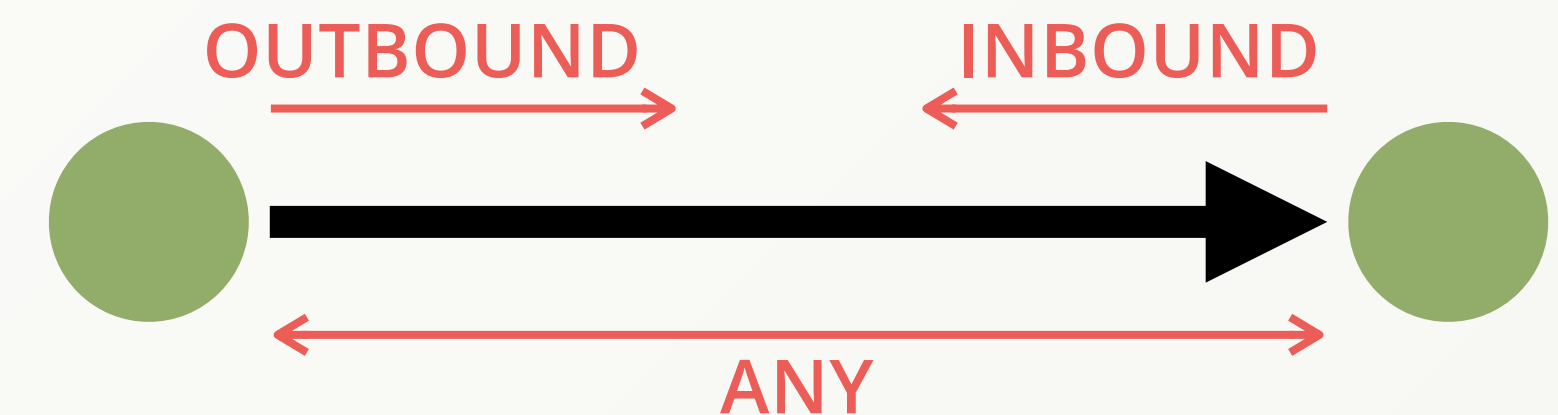
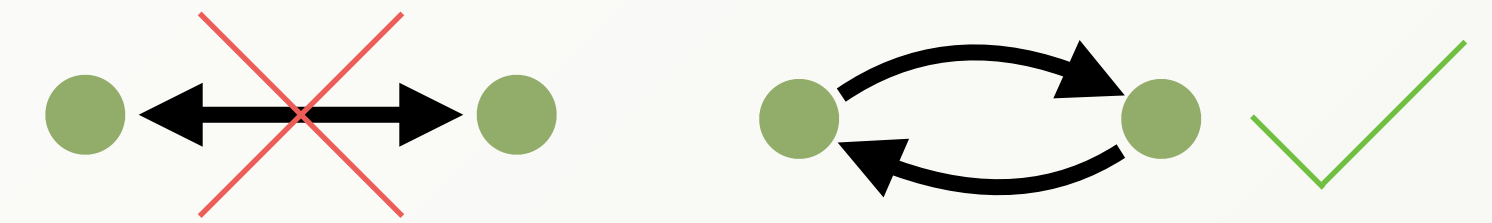


# Graph Basics

In ArangoDB, each edge has a single direction, it can't point both ways at once. This model is also known as *oriented* graph.

Moreover, **edges are always directed**, but you can ignore the direction (follow in **ANY** direction) when you walk through the graph, or follow edges in reverse direction (**INBOUND**) instead of going in the direction they actually point to (**OUTBOUND**). Walking through a graph is called *traversal*.

ArangoDB allows you to store all kinds of graphs in different shapes and sizes, with and without cycles. You can save one or more edges between two vertices or even the same vertex. Also note that edges are **full-fledged JSON documents**, which means you can store as much information on the edges as you want!



# Graph Basics

---

A few examples what can be answered by graph queries with the example dataset in mind:

- ▶ Give me all flights departing from **JFK** (airport in New York)
- ▶ Give me all flights landing in **LAX** (airport in Los Angeles) on January 5th
- ▶ Which **airports** can I reach with up to one stopover?  
(From one or multiple starting airports)
- ▶ Shortest Path:
  - ▶ What is the minimum amount of stopovers to fly from **BIS**  
(Bismarck Municipal Airport in North Dakota) to **LAX** and where is the stopover?
- ▶ Pattern Matching:
  - ▶ Departing from **BIS**, which flight to **JFK** with one stopover  
(at least 20 minutes time for the transit) is the quickest and via which airport?

# Graph Basics

---

Typical use cases for graph databases and "graphy" queries are:

- ▶ 360° View (Market Data, Customer, User, ...)
- ▶ Artificial Intelligence
- ▶ Dependency Management
- ▶ Fraud Detection
- ▶ Identity & Access Management
- ▶ Knowledge Graph
- ▶ Master Data Management
- ▶ Network Infrastructure
- ▶ Recommendation Engine
- ▶ Risk Management
- ▶ Social Media Management

Whenever the depth of your search is unknown (how many edges to follow), then graph queries are easier to write and more efficient to compute compared to other query patterns.



# The Example Dataset

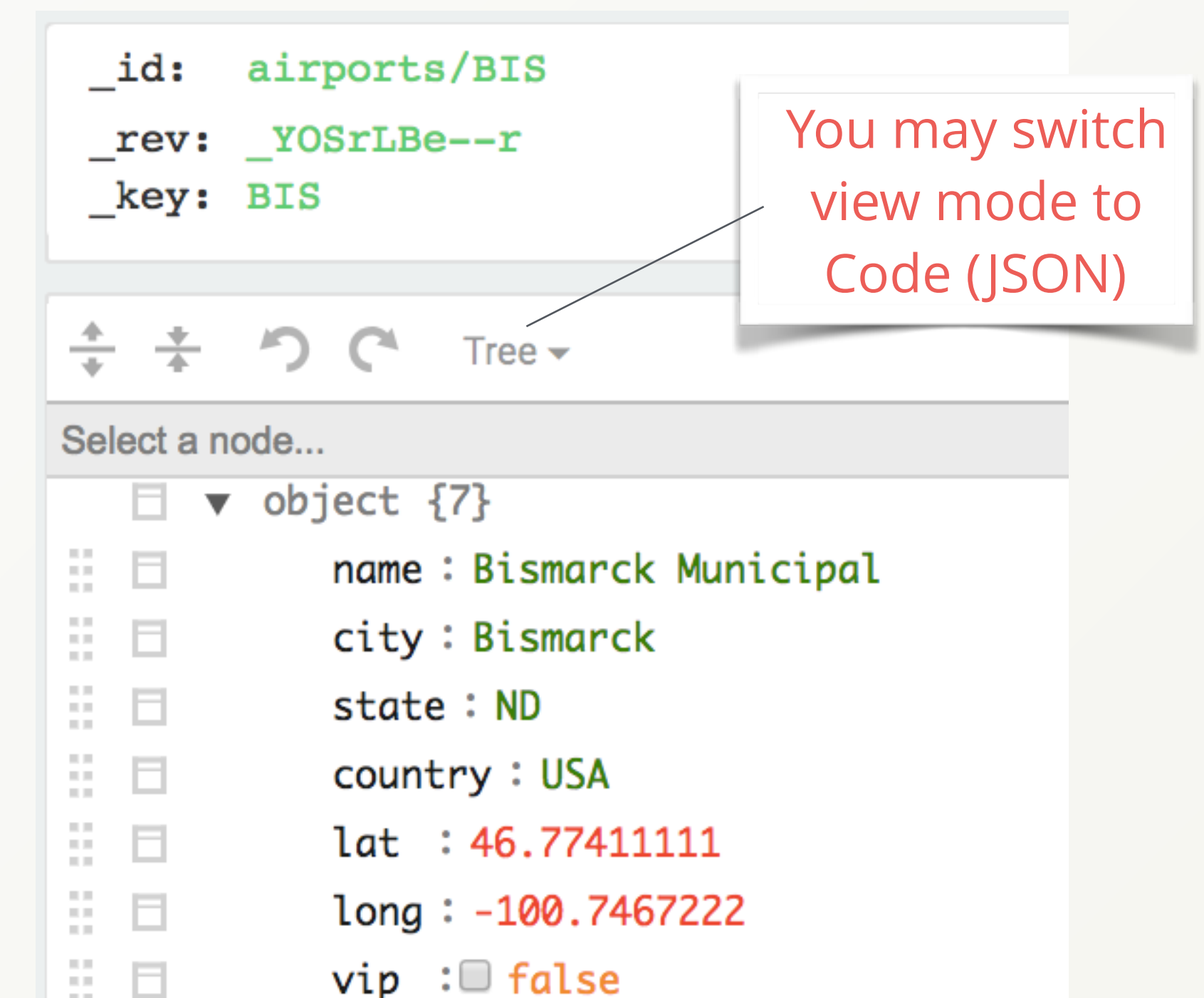
We took a dataset of US airports and flights, augmented and simplified it. Included are more than 3,000 airports and roughly 300,000 flights from January 1st to 15th, 2008.

Data structure of **airport** documents:

Attribute	Description
<b>_key</b>	international airport abbreviation code
<b>_id</b>	collection name + "/" + _key (computed property)
<b>name</b>	full name of the airport
<b>city</b>	name of the associated city
<b>country</b>	name of the country it is in (USA)
<b>lat</b>	latitude portion of the geographic location
<b>long</b>	longitude portion of the geographic location
<b>state</b>	name of the US state it is in
<b>vip</b>	airport with premium lounge? (true or false) *

\* We marked a few airports randomly for example queries shown later

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: _YOSrLBe--r`, and `_key: BIS`. Below this is a toolbar with icons for expand/collapse, refresh, and a dropdown menu currently set to 'Tree'. A callout box with a red border and text points to the 'Tree' dropdown, stating: "You may switch view mode to Code (JSON)". The main area shows a tree view of the document structure. Under 'Select a node...', there is a collapsed 'object {7}' node. Expanding it reveals the following fields: `name: Bismarck Municipal`, `city: Bismarck`, `state: ND`, `country: USA`, `lat: 46.77411111`, `long: -100.7467222`, and `vip: false`. The values for latitude and longitude are highlighted in red in the original image.

# The Example Dataset

Data structure of **flights** documents:

Attribute	Description
_from	Origin (airport _id)
_to	Destination (airport _id)
Year	Year of flight (here: 2008)
Month	Month of flight (1..12)
Day	Day of flight (1..31)
DayOfWeek	Weekday (1 = Monday .. 7 = Sunday)
DepTime	Actual departure time (local, <i>hhmm</i> as number)
ArrTime	Actual arrival time (local, <i>hhmm</i> as number)
DepTimeUTC	Departure time (coord. universal time, ISO string)
ArrTimeUTC	Arrival time (coordinated universal time, ISO string)
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:

```
_id: flights/1986
_rev: _Y008IKK--H   _from: airports/MSP
_key: 1986           _to: airports/JFK
```

Tree ▾

Select a node...

- object {12}
  - Year : 2008
  - Month : 1
  - Day : 1
  - DayOfWeek : 2
  - DepTime : 712
  - ArrTime : 1059
  - DepTimeUTC : 2008-01-01T13:12:00.000Z
  - ArrTimeUTC : 2008-01-01T15:59:00.000Z
  - UniqueCarrier : NW
  - FlightNum : 736
  - TailNum : N319NB
  - Distance : 1028



# The Example Dataset

Here are some example documents from both collections (JSON view mode):

## airports

```
{
  "_key": "JFK",
  "_id": "airports/JFK",
  "_rev": "_Y0008KG--T",
  "name": "John F Kennedy Intl",
  "city": "New York",
  "state": "NY",
  "country": "USA",
  "lat": 40.63975111,
  "long": -73.77892556,
  "vip": true
}
```

```
{
  "_key": "BIS",
  "_id": "airports/BIS",
  "_rev": "_Y0SrLBe--r",
  "name": "Bismarck Municipal",
  "city": "Bismarck",
  "state": "ND",
  "country": "USA",
  "lat": 46.77411111,
  "long": -100.7467222,
  "vip": false
}
```

## flights

```
{
  "_key": "25471",
  "_id": "flights/25471",
  "_from": "airports/BIS",
  "_to": "airports/MSP",
  "_rev": "_Y008JXG--f",
  "Year": 2008,
  "Month": 1,
  "Day": 2,
  "DayOfWeek": 3,
  "DepTime": 1055,
  "ArrTime": 1224,
  "DepTimeUTC": "2008-01-02T16:55:00.000Z",
  "ArrTimeUTC": "2008-01-02T18:24:00.000Z",
  "UniqueCarrier": "9E",
  "FlightNum": 5660,
  "TailNum": "85069E",
  "Distance": 386
}
```

```
{
  "_key": "71374",
  "_id": "flights/71374",
  "_from": "airports/JFK",
  "_to": "airports/DCA",
  "_rev": "_Y008LYG--N",
  "Year": 2008,
  "Month": 1,
  "Day": 4,
  "DayOfWeek": 5,
  "DepTime": 1604,
  "ArrTime": 1724,
  "DepTimeUTC": "2008-01-04T21:04:00.000Z",
  "ArrTimeUTC": "2008-01-04T22:24:00.000Z",
  "UniqueCarrier": "MQ",
  "FlightNum": 4755,
  "TailNum": "N854AE",
  "Distance": 213
}
```



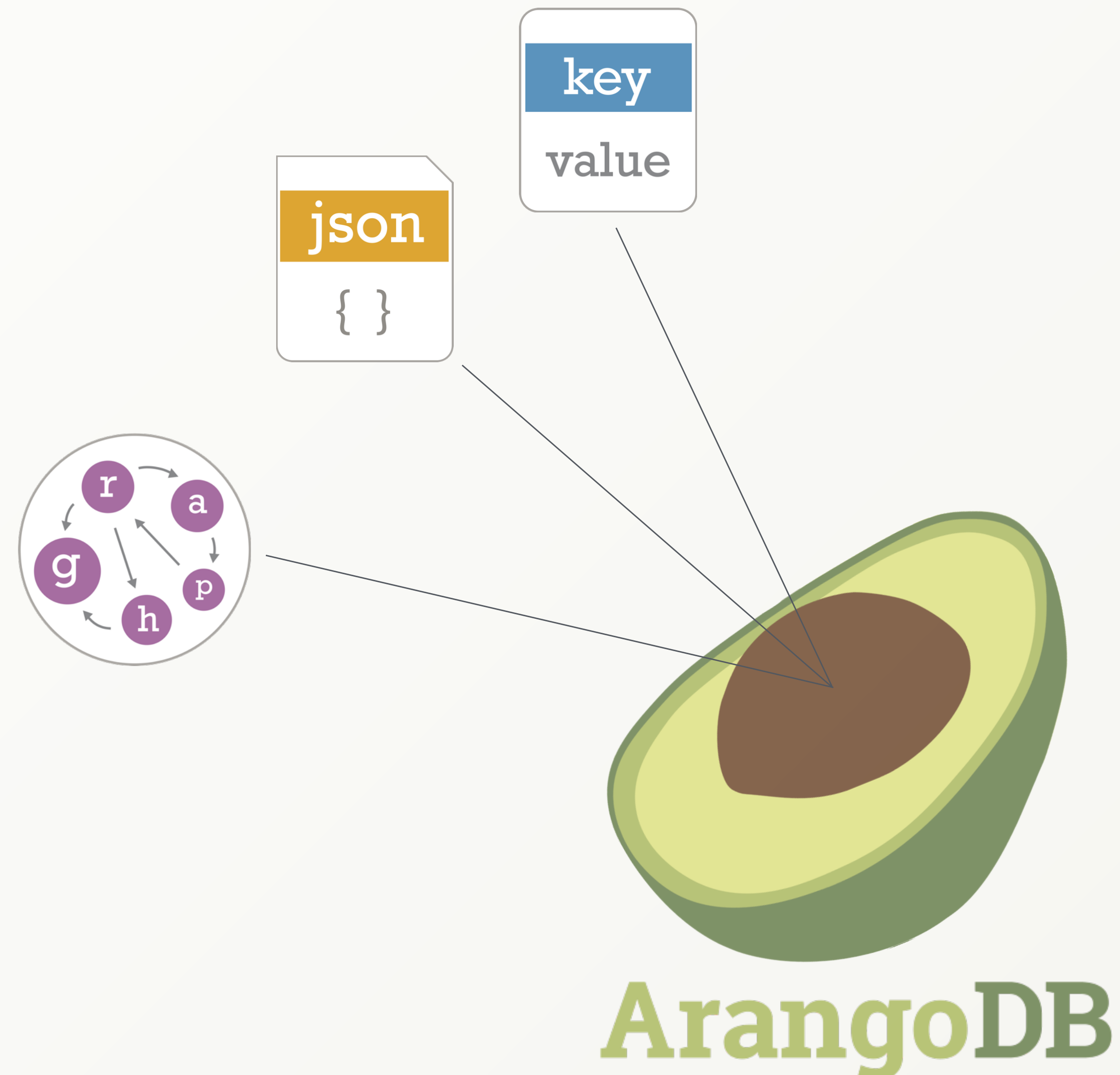
# Concepts of ArangoDB

---

## **What is Multi-Model? & ArangoDB Architecture**

# What is Multi-Model?

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



# What is Multi-Model?

How is multi-model possible at all?

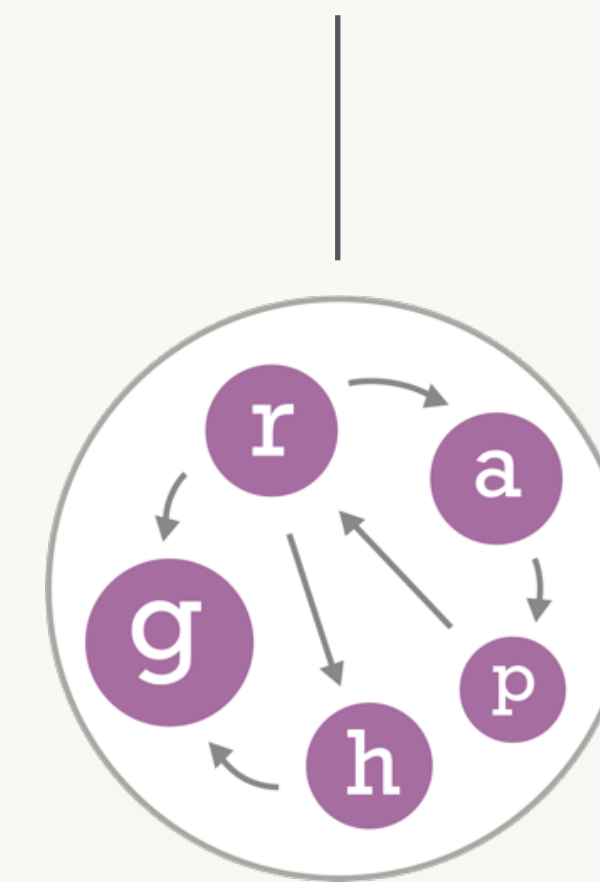
If you store a JSON document and treat it as opaque value under a primary key then you have a key/value store.



ArangoDB is a document-oriented data store using primary keys



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





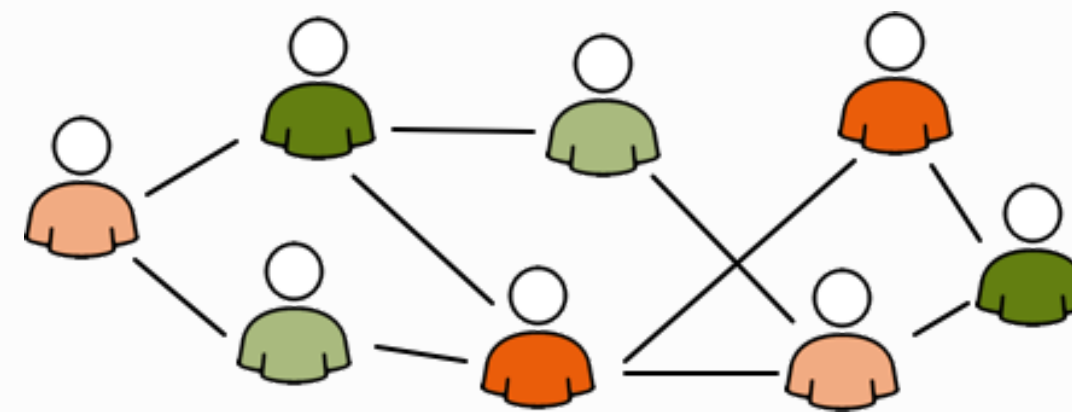
# What is Multi-Model?

## Benefits of ArangoDB's **NATIVE MULTI-MODEL** approach

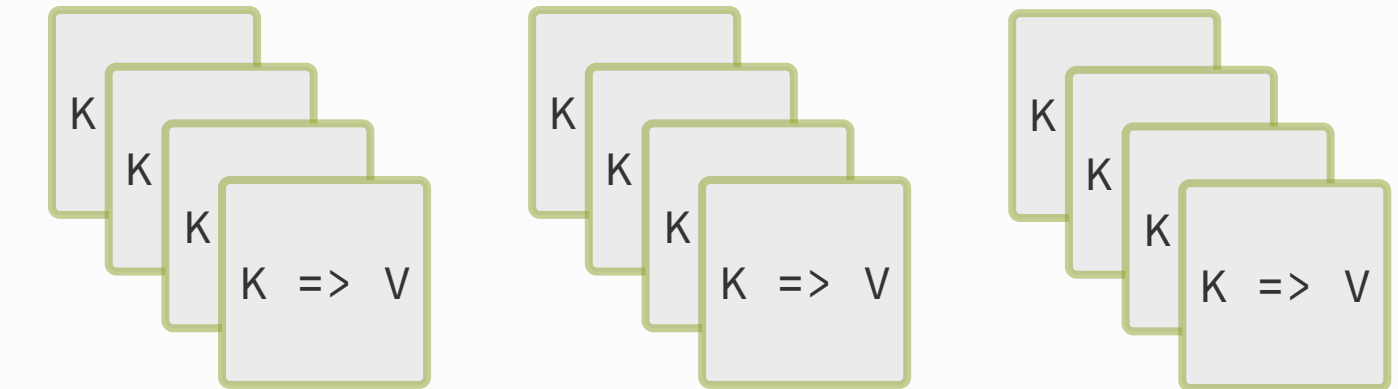
Documents - JSON

```
{
  "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 model

# ArangoDB Architecture

ArangoDB has a storage hierarchy like other databases have too:

- ▶ You can create different **Databases** which can hold an arbitrary number of collections. There is a default database called *\_system*
- ▶ **Collections** can hold arbitrary amounts of documents. There are two collection types: document and edge collections
- ▶ **Documents** are stored in **JSON** format. A document is a JSON object at the top-level, whose attribute names are strings and the values can be null, true, false, numbers, strings, arrays and nested objects. There are also system attributes (*\_key*, *\_id*, *\_rev*, for edges also *\_from*, *\_to*)



Databases

*\_system*

myDB1

Collections

Airports

myColl1

Documents  
(Vertices)

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

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

Collections

Flights

myEdges1

Documents  
(Edges)

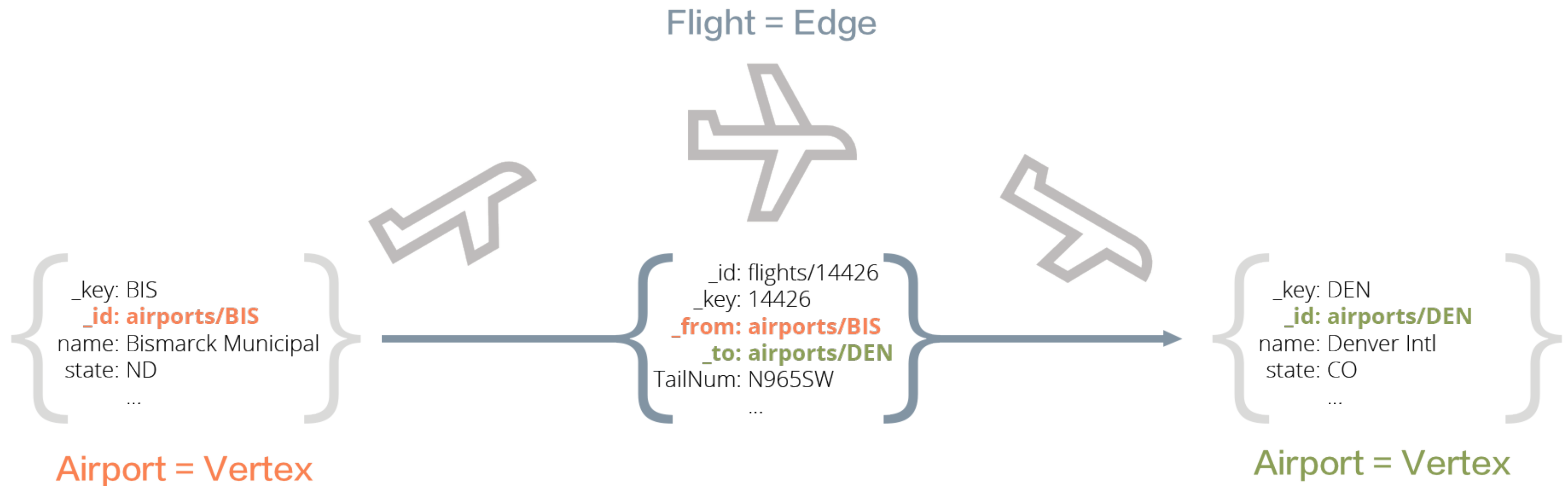
{JSON} {JSON} {JSON}  
{JSON} {JSON} {JSON}

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

# ArangoDB Architecture

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.





# ArangoDB Architecture

---

Edge collections in summary:

- ▶ **Place to hold relations**
  - ▶ Comparable with many-to-many relations in SQL systems (cross tables)
- ▶ **Contain documents, but with special attributes**
  - ▶ *\_from: \_id* value of the source vertex
  - ▶ *\_to: \_id* value of the target vertex
- ▶ **Built-in edge index for every edge collection**
- ▶ **Building block of graphs**

# Preparations for this Course

---

**Download and Install ArangoDB  
&  
Import the Dataset**

# Download and Install ArangoDB

- ▶ Go to [arangodb.com/download/](https://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. We recommend to set a password for the default user *root*. Further details can be found here: [docs.arangodb.com/latest/Manual/Installation/](https://docs.arangodb.com/latest/Manual/Installation/)
- ▶ Once the server is booted up, open **http://localhost:8529** in your browser to access *Aardvark*, the ArangoDB WebUI
- ▶ Login with your credentials, e.g. as *root*. If you did not set a password, then leave the password field empty.
- ▶ Next, select a database, e.g. the default *\_system* database.



# Import the Dataset – Airports

---

- ▶ Download the example dataset here:  
[arangodb.com/arangodb\\_graphcourse\\_demodata/](https://arangodb.com/arangodb_graphcourse_demodata/)
- ▶ Unpack it to a folder of your choice.  
After unpacking you should see two .csv files named **airports.csv** and **flights.csv**
- ▶ Import the airports with ArangoDB's import tool *arangoimport*.  
Run the following on your command line (single line):

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

You can specify **--server.username name** to use another user than *root*.  
If you did not set a password or if the server has authentication disabled then just hit return when asked for a password.

If ArangoDB is in your *PATH* environment variable, then you can run the binaries by their name from any working directory. Otherwise specify the full path.



# Import the Dataset – Airports

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

```

Simrans-Air:~ Simran$ arangoimport --file /Users/Simran/Downloads/GraphCourse_DemoData_ArangoDB-2/airports.csv
[ --type csv --collection airports --create-collection
Please specify a password:
Connected to ArangoDB 'http+tcp://127.0.0.1:8529', version 3.4.0, database: '_system', username: 'root'
-----
database:          _system
collection:        airports
create:            yes
create database:    no
source filename:    /Users/Simran/Downloads/GraphCourse_DemoData_ArangoDB-2/airports.csv
file type:          csv
quote:             "
separator:
threads:           2
connect timeout:    5
request timeout:    1200
-----
Starting CSV import...
2019-02-19T05:04:55Z [833] INFO processed 32768 bytes (3%) of input file
2019-02-19T05:04:55Z [833] INFO processed 65536 bytes (16%) of input file
2019-02-19T05:04:55Z [833] INFO processed 98304 bytes (29%) of input file
2019-02-19T05:04:55Z [833] INFO processed 131072 bytes (42%) of input file
2019-02-19T05:04:55Z [833] INFO processed 163840 bytes (55%) of input file
2019-02-19T05:04:55Z [833] INFO processed 196608 bytes (69%) of input file
2019-02-19T05:04:55Z [833] INFO processed 229376 bytes (82%) of input file
2019-02-19T05:04:55Z [833] INFO processed 247769 bytes (95%) of input file

created:           3375
warnings/errors:   0
updated/replaced:  0
ignored:           0
lines read:        3377
Simrans-Air:~ Simran$

```



# Import the Dataset – Airports

What did *arangoimport* do?

- ▶ Created a new *document* collection (**airports**) with a primary index on *\_key*
- ▶ Created one document for each line of the CSV file (except the first line and last, empty line)
- ▶ The first line is the header defining the attribute names

Note:

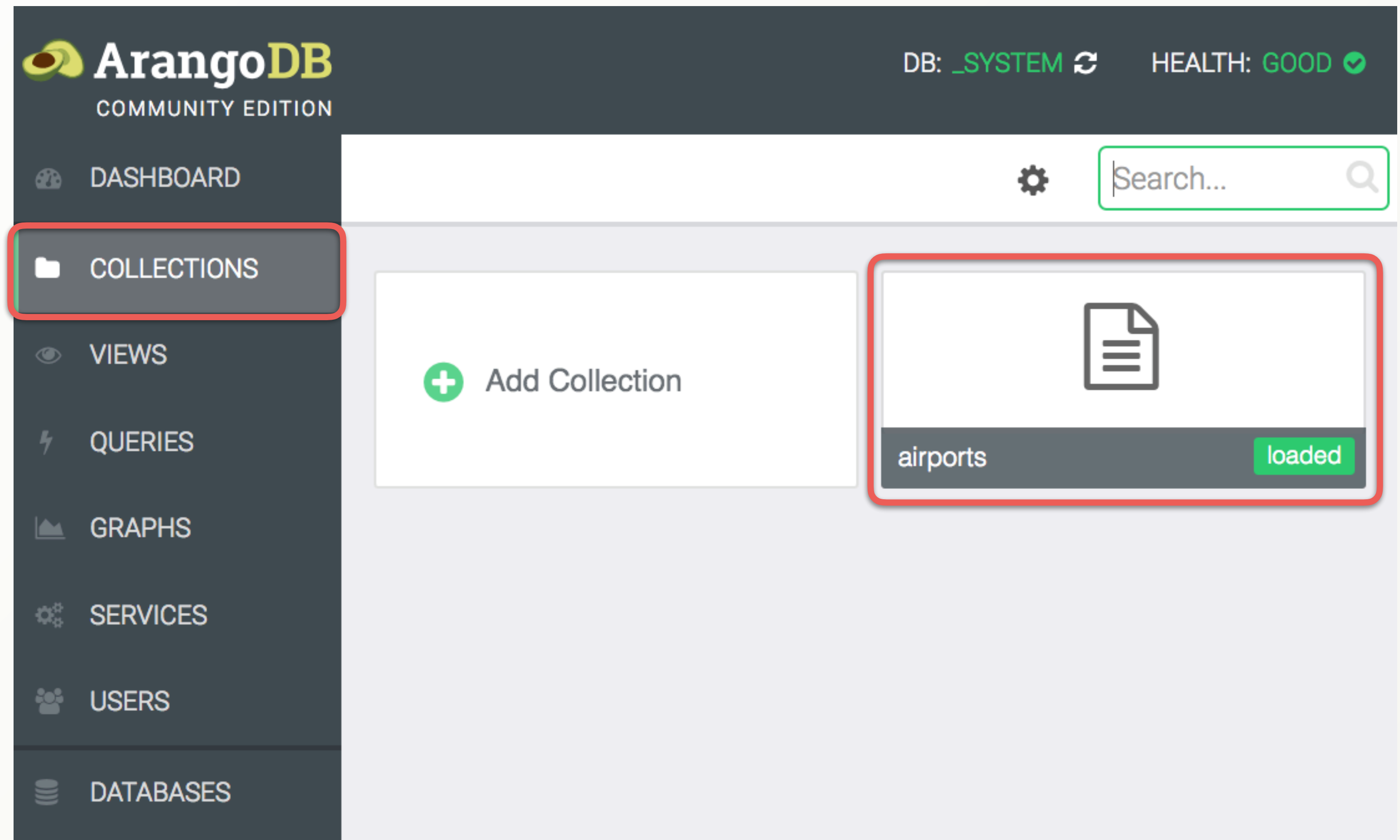
- ▶ Airport codes are provided as *\_key* attribute in the CSV file
- ▶ The *\_key* attribute is the primary key which uniquely identifies documents within a collection. Therefore, we will be able to retrieve airports via their airport code utilizing the primary index

```
airports.csv
1  "_key","name","city","state","country","lat","long","vip"
2  "00M","Thigpen ","Bay Springs","MS","USA",31.95376472,-89.23450472,false
3  "00R","Livingston Municipal","Livingston","TX","USA",30.68586111,-95.01792778,false
4  "00V","Meadow Lake","Colorado Springs","CO","USA",38.94574889,-104.5698933,false
5  "01G","Perry-Warsaw","Perry","NY","USA",42.74134667,-78.05208056,false
6  "01J","Hilliard Airpark","Hilliard","FL","USA",30.6880125,-81.90594389,false
7  "01M","Tishomingo County","Belmont","MS","USA",34.49166667,-88.20111111,false
8  "02A","Gragg-Wade ","Clanton","AL","USA",32.85048667,-86.61145333,false
9  "02C","Capitol","Brookfield","WI","USA",43.08751,-88.17786917,false
10 "02G","Columbiana County","East Liverpool","OH","USA",40.67331278,-80.64140639,false
11 "03D","Memphis Memorial","Memphis","MO","USA",40.44725889,-92.22696056,false
12 "04M","Calhoun County","Pittsboro","MS","USA",33.93011222,-89.34285194,false
13 "04Y","Hawley Municipal","Hawley","MN","USA",46.88384889,-96.35089861,false
14 "05C","Griffith-Merrillville ","Griffith","IN","USA",41.51961917,-87.40109333,false
15 "05F","Gatesville - City/County","Gatesville","TX","USA",31.42127556,-97.79696778,false
16 "05U","Eureka","Eureka","NV","USA",39.60416667,-116.0050597,false
17 "06A","Moton Municipal","Tuskegee","AL","USA",32.46047167,-85.68003611,false
18 "06C","Schaumburg","Chicago/Schaumburg","IL","USA",41.98934083,-88.10124278,false
19 "06D","Rolla Municipal","Rolla","ND","USA",48.88434111,-99.62087694,false
20 "06M","Eupora Municipal","Eupora","MS","USA",33.53456583,-89.31256917,false
21 "06N","Randall ","Middletown","NY","USA",41.43156583,-74.39191722,false
22 "06U","Jackpot/Hayden ","Jackpot","NV","USA",41.97602222,-114.6580911,false
23 "07C","DeKalb County","Auburn","IN","USA",41.30716667,-85.06433333,false
24 "07F","Gladewater Municipal","Gladewater","TX","USA",32.52883861,-94.97174556,false
25 "07G","Fitch H Beach","Charlotte","MI","USA",42.57450861,-84.81143139,false
26 "07K","Central City Municipal","Central City","NE","USA",41.11668056,-98.05033639,false
27 "08A","Wetumpka Municipal","Wetumpka","AL","USA",32.52943944,-86.32822139,false
28 "08D","Stanley Municipal","Stanley","ND","USA",48.30079861,-102.4063514,false
29 "08K","Harvard State","Harvard","NE","USA",40.65138528,-98.07978667,false
30 "08M","Carthage-Leake County","Carthage","MS","USA",32.76124611,-89.53007139,false
31 "09A","Butler-Choctaw County","Butler","AL","USA",32.11931306,-88.1274625,false
32 "09J","Jekyll Island","Jekyll Island","GA","USA",31.07447222,-81.42777778,false
33 "09K","Sargent Municipal","Sargent","NE","USA",41.63695083,-99.34038139,false
34 "09M","Charleston Municipal","Charleston","MS","USA",33.99150222,-90.078145,false
35 "09W","South Capitol Street","Washington","DC","USA",38.86872333,-77.00747583,false
36 "0A3","Smithville Municipal","Smithville","TN","USA",35.98531194,-85.80931806,false
37 "0A8","Bibb County","Centreville","AL","USA",32.93679056,-87.08888306,false
38 "0A9","Elizabethton Municipal","Elizabethton","TN","USA",36.37094306,-82.17374111,false
39 "0AK","Pilot Station","Pilot Station","AK","USA",61.93396417,-162.8929358,false
40 "0B1","Col. Dyke ","Bethel","ME","USA",44.42506444,-70.80784778,false
41 "0B4","Hartington Municipal","Hartington","NE","USA",42.60355556,-97.25263889,false
42 "0B5","Turners Falls","Montague","MA","USA",42.59136361,-72.52275472,false
```

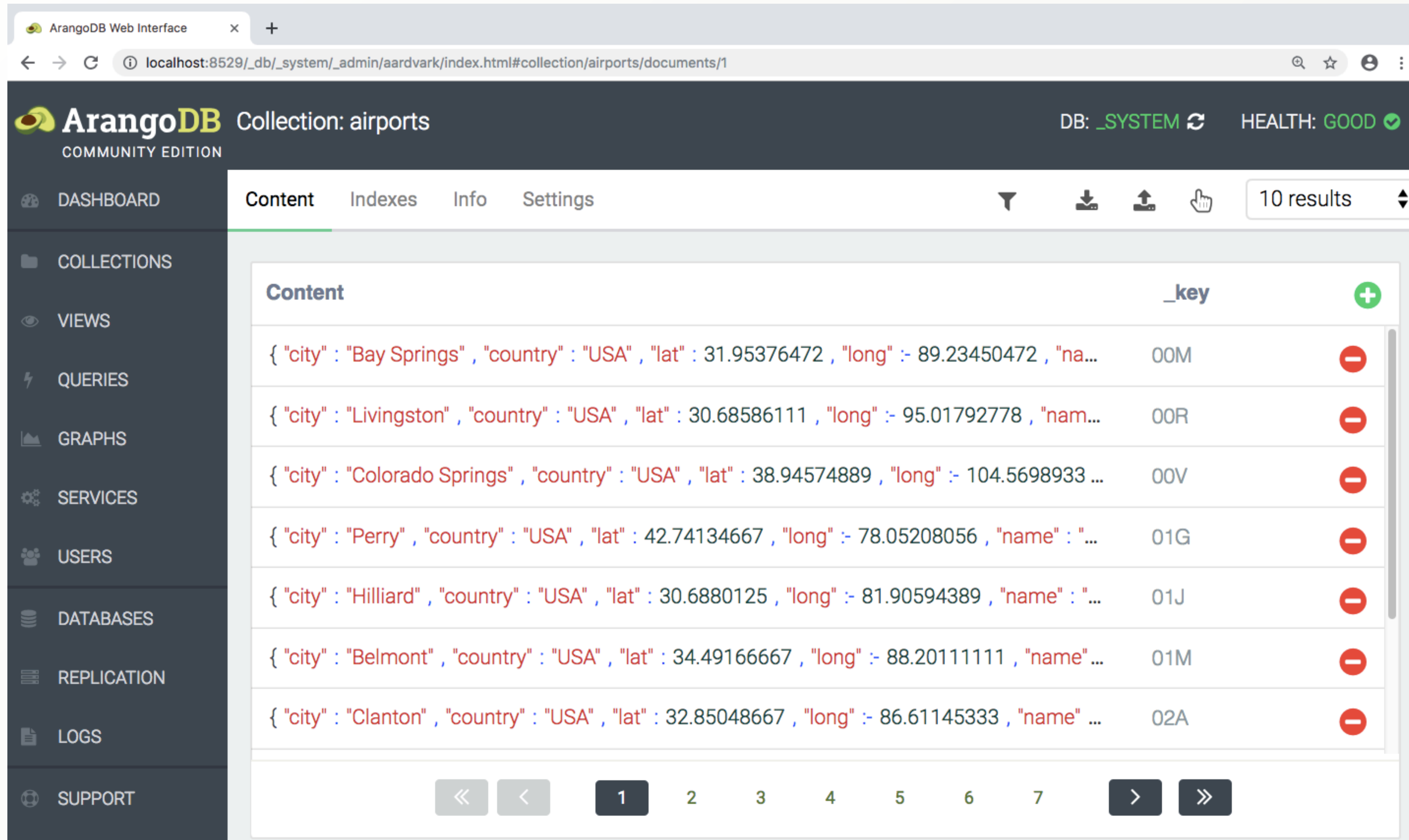


# Import the Dataset – Airports

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



# Import the Dataset – Airports



ArangoDB Web Interface

Collection: airports

DB: **\_SYSTEM** HEALTH: **GOOD**

Content Indexes Info Settings

10 results

Content	_key	
{ "city" : "Bay Springs" , "country" : "USA" , "lat" : 31.95376472 , "long" :- 89.23450472 , "na...	00M	-
{ "city" : "Livingston" , "country" : "USA" , "lat" : 30.68586111 , "long" :- 95.01792778 , "nam...	00R	-
{ "city" : "Colorado Springs" , "country" : "USA" , "lat" : 38.94574889 , "long" :- 104.5698933 ...	00V	-
{ "city" : "Perry" , "country" : "USA" , "lat" : 42.74134667 , "long" :- 78.05208056 , "name" : "...	01G	-
{ "city" : "Hilliard" , "country" : "USA" , "lat" : 30.6880125 , "long" :- 81.90594389 , "name" : "...	01J	-
{ "city" : "Belmont" , "country" : "USA" , "lat" : 34.49166667 , "long" :- 88.20111111 , "name" ...	01M	-
{ "city" : "Clanton" , "country" : "USA" , "lat" : 32.85048667 , "long" :- 86.61145333 , "name" ...	02A	-

<< < 1 2 3 4 5 6 7 > >>



# Import the Dataset – Flights

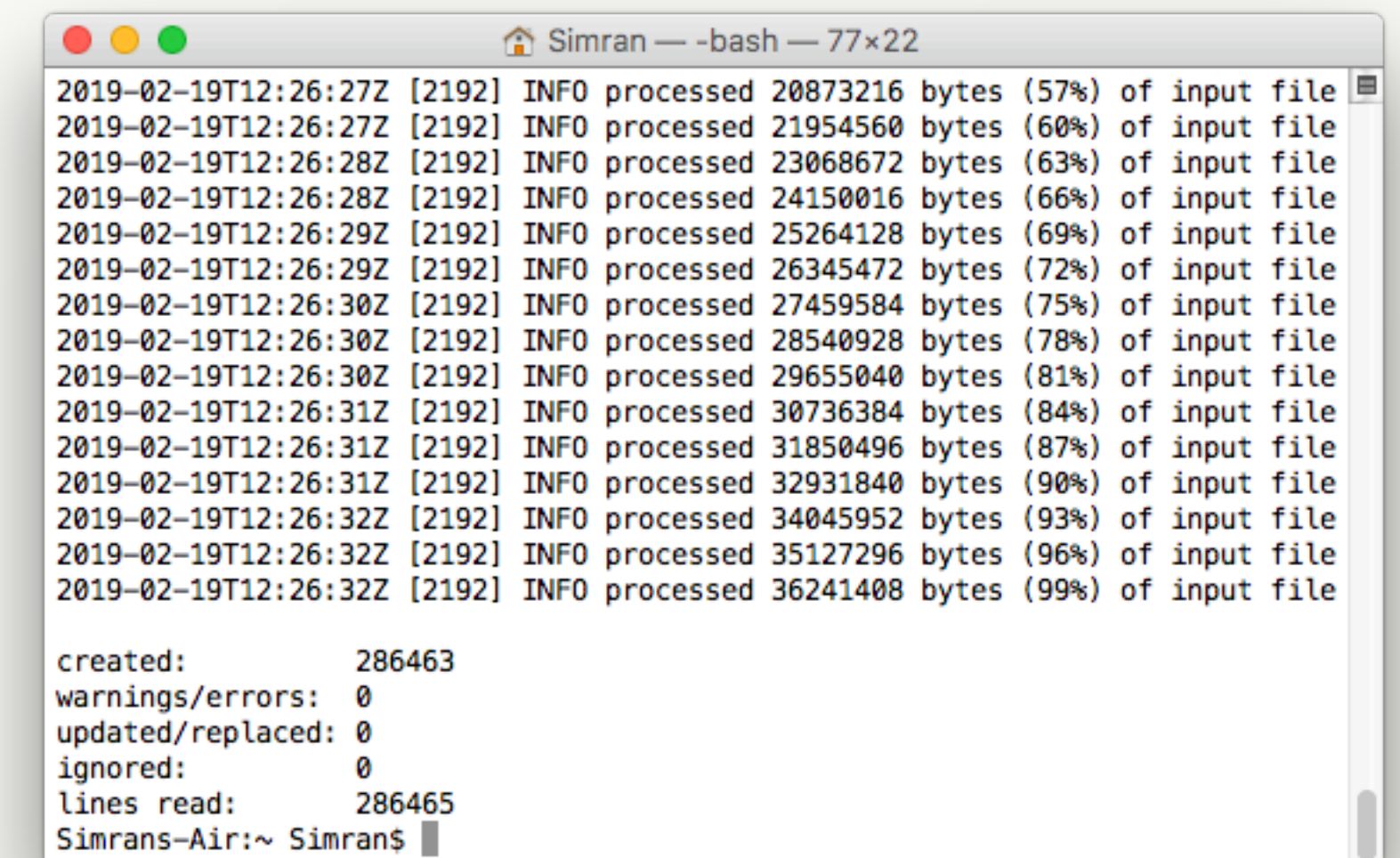
The imported *airports* are the vertices of our graph. To complete our graph dataset, we also need edges to connect the vertices. In our case the edges are *flights*.

- ▶ Import the flights into an edge collection with *arangoimport*.

Run the following in your command line (single line):

```
arangoimport --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. On a decent computer with at least 4 GB of memory and an SSD drive it should take less than a minute.



```
Simran - bash - 77x22  
2019-02-19T12:26:27Z [2192] INFO processed 20873216 bytes (57%) of input file  
2019-02-19T12:26:27Z [2192] INFO processed 21954560 bytes (60%) of input file  
2019-02-19T12:26:28Z [2192] INFO processed 23068672 bytes (63%) of input file  
2019-02-19T12:26:28Z [2192] INFO processed 24150016 bytes (66%) of input file  
2019-02-19T12:26:29Z [2192] INFO processed 25264128 bytes (69%) of input file  
2019-02-19T12:26:29Z [2192] INFO processed 26345472 bytes (72%) of input file  
2019-02-19T12:26:30Z [2192] INFO processed 27459584 bytes (75%) of input file  
2019-02-19T12:26:30Z [2192] INFO processed 28540928 bytes (78%) of input file  
2019-02-19T12:26:30Z [2192] INFO processed 29655040 bytes (81%) of input file  
2019-02-19T12:26:31Z [2192] INFO processed 30736384 bytes (84%) of input file  
2019-02-19T12:26:31Z [2192] INFO processed 31850496 bytes (87%) of input file  
2019-02-19T12:26:31Z [2192] INFO processed 32931840 bytes (90%) of input file  
2019-02-19T12:26:32Z [2192] INFO processed 34045952 bytes (93%) of input file  
2019-02-19T12:26:32Z [2192] INFO processed 35127296 bytes (96%) of input file  
2019-02-19T12:26:32Z [2192] INFO processed 36241408 bytes (99%) of input file  
  
created:          286463  
warnings/errors:  0  
updated/replaced: 0  
ignored:          0  
lines read:       286465  
Simrans-Air:~ Simran$
```

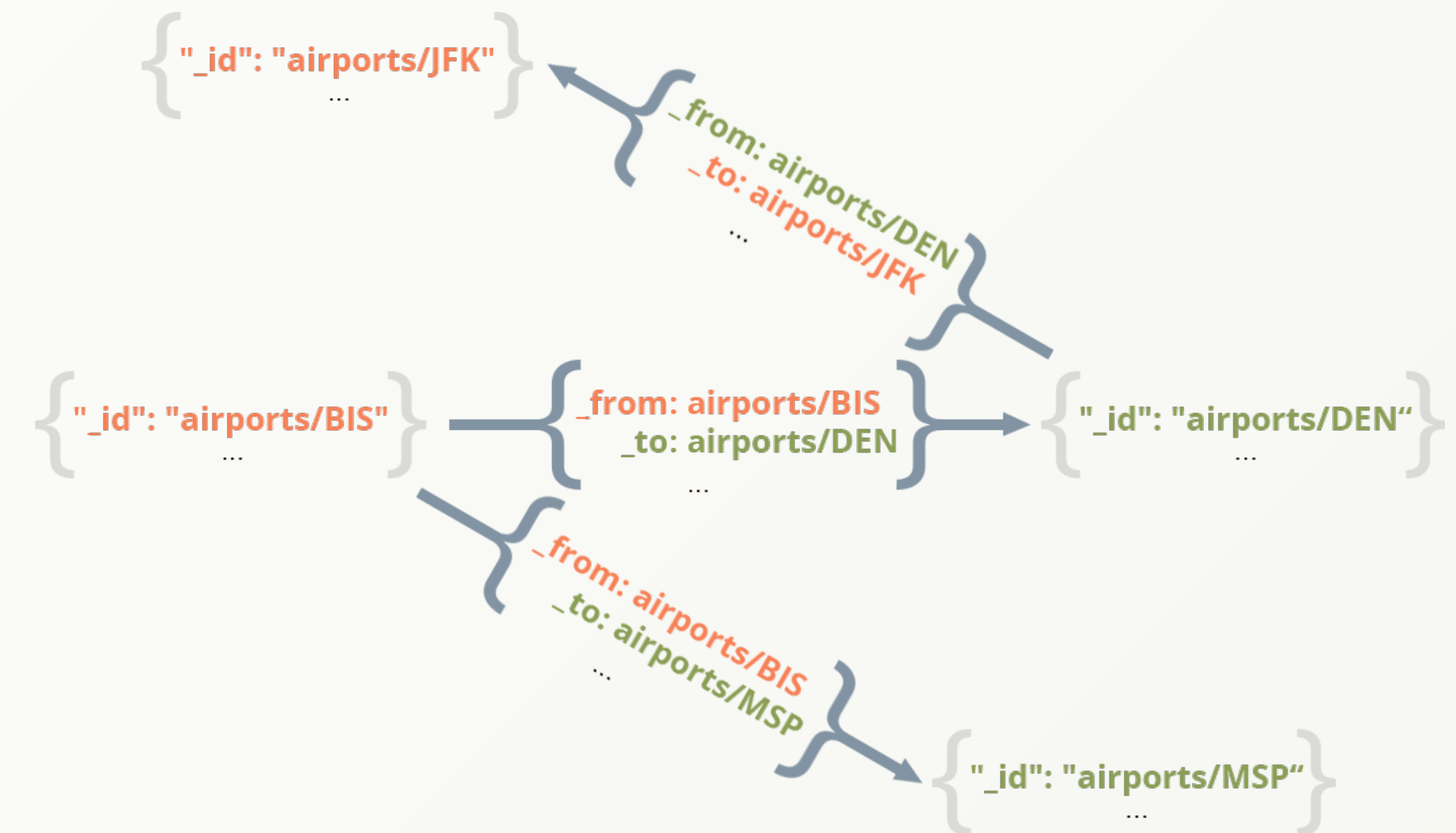
# Import the Dataset – Flights

What did *arangaimport* do?

- ▶ Created a new edge collection (**flights**) with a primary index on attribute `_key` and an edge index on `_from` and `_to`
- ▶ Created one edge document for each line of the CSV file (except the header and the last line)

Note:

- ▶ The `_from` and `_to` attributes form the graph by referencing document `_ids` of departure and arrival airports
- ▶ No `_key` is provided, thus it gets auto-generated



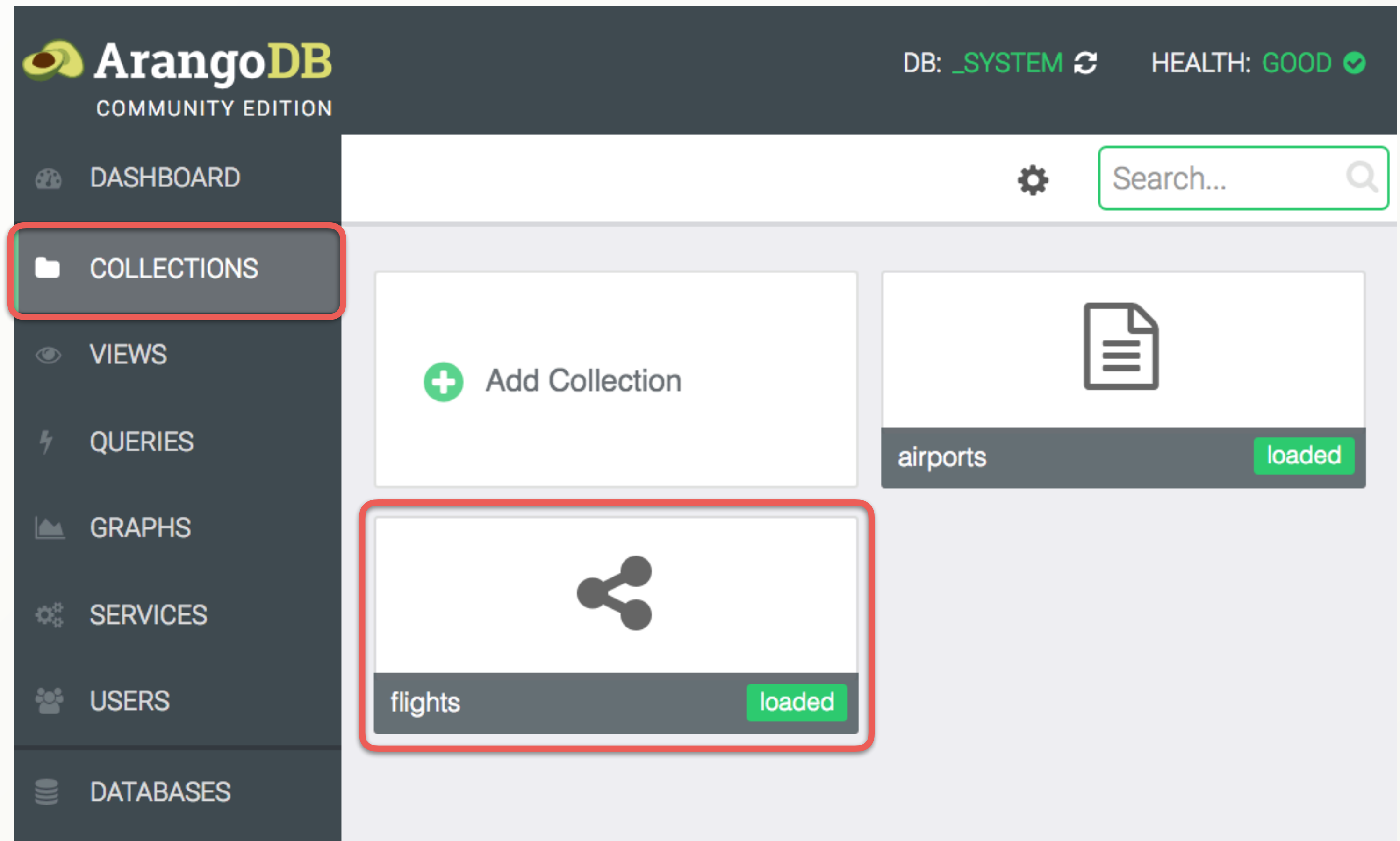
```

flights.csv x
1  "_from","_to","Year","Month","Day","DayOfWeek","DepTime","ArrTime","DepTimeUTC","ArrTimeUTC","UniqueCarrier","FlightNum","TailNum","Distance"
2  "airports/ATL","airports/CHS",2008,1,1,2,2,57,"2008-01-01T05:02:00.000Z","2008-01-01T05:57:00.000Z","FL",579,"N937AT",259
3  "airports/CLE","airports/SAT",2008,1,1,2,3,230,"2008-01-01T05:03:00.000Z","2008-01-01T08:30:00.000Z","XE",2895,"N14158",1241
4  "airports/IAD","airports/CLE",2008,1,1,2,5,132,"2008-01-01T05:05:00.000Z","2008-01-01T06:32:00.000Z","YV",7185,"N592ML",288
5  "airports/JFK","airports/PBI",2008,1,1,2,8,332,"2008-01-01T05:08:00.000Z","2008-01-01T08:32:00.000Z","B6",859,"N505JB",1028
  
```



# Import the Dataset – Flights

- ▶ Go to ArangoDB WebUI and click on 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 on the flights collection to browse its edge documents



# Starting with the dataset

---

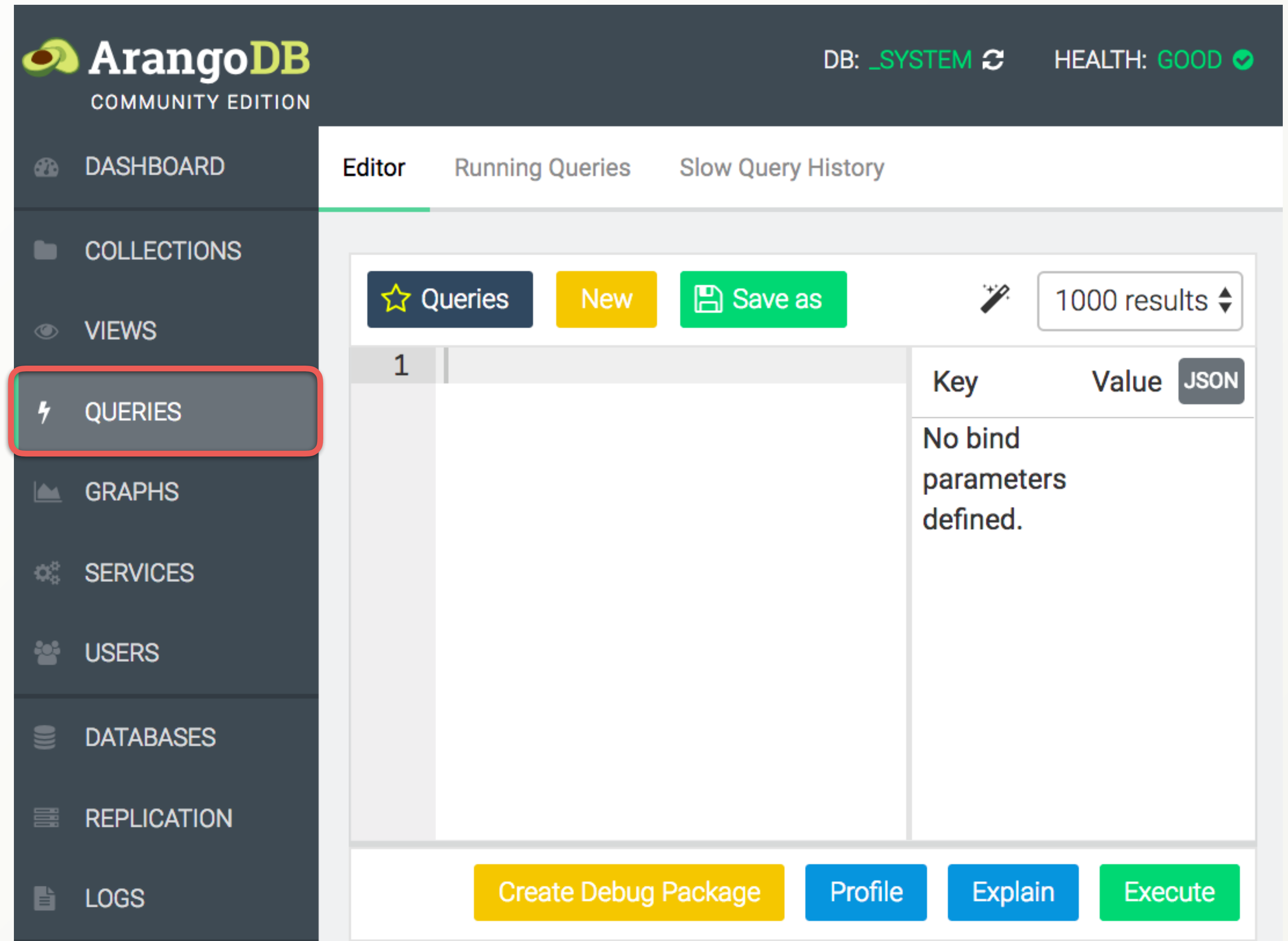
## **AQL Query Editor & First AQL Queries**



# ArangoDB Query Editor

Now that we have demo data in ArangoDB, let us start to write AQL queries!

- ▶ Click on *QUERIES* in the ArangoDB WebUI
- ▶ It brings up the AQL query editor to write, execute and profile queries
- ▶ It supports syntax highlighting and allows you to save and manage queries



The screenshot shows the ArangoDB Query Editor interface. The left sidebar contains a navigation menu with the following items: DASHBOARD, COLLECTIONS, VIEWS, **QUERIES** (highlighted with a red box), GRAPHS, SERVICES, USERS, DATABASES, REPLICATION, and LOGS. The main area is titled 'Editor' and shows a 'Queries' tab with a 'New' button and a 'Save as' button. A dropdown menu shows '1000 results'. The main editor area is empty, and the right sidebar shows a table with columns 'Key' and 'Value', and a 'JSON' button. The table contains one row with the text 'No bind parameters defined.' At the bottom, there are buttons for 'Create Debug Package', 'Profile', 'Explain', and 'Execute'.

ArangoDB COMMUNITY EDITION

DB: **\_SYSTEM** HEALTH: **GOOD**

DASHBOARD COLLECTIONS VIEWS **QUERIES** GRAPHS SERVICES USERS DATABASES REPLICATION LOGS

Editor Running Queries Slow Query History

Queries New Save as 1000 results

1	Key	Value
		No bind parameters defined.

Create Debug Package Profile Explain Execute

# ArangoDB Query Editor

Set limit for results shown

1000 results

Run query

Write queries here

Remove all results Create Debug Package Profile Explain Execute

Switch result view mode

JSON Table

Query 7 elements 5.261 ms

_key	_id	_rev	name	city	state	country	lat	long	vip
AMA	airports/AMA	_Y0008JS-_x	Amarillo International	Amarillo	TX	USA	35.2193725	-101.7059272	true
ATL	airports/ATL	_Y0008JW-f	William B Hartsfield-Atlanta Intl	Atlanta	GA	USA	33.64044444	-84.42694444	true
DFW	airports/DFW	_Y0008Jm-A_	Dallas-Fort Worth International	Dallas-Fort Worth	TX	USA	32.89595056	-97.0372	true
JFK	airports/JFK	_Y0008KG-_T	John F Kennedy Intl	New York	NY	USA	40.63975111	-73.77892556	true
LAX	airports/LAX	_Y0008KK-AI	Los Angeles International	Los Angeles	CA	USA	33.94253611	-118.4080744	true
ORD	airports/ORD	_Y0008Ki-A_	Chicago O'Hare International	Chicago	IL	USA	41.979595	-87.90446417	true
SFO	airports/SFO	_Y0008K2-_f	San Francisco International	San Francisco	CA	USA	37.61900194	-122.3748433	true

Download CSV Copy To Editor

3.4.0

# First AQL Queries – Hands on

- ▶ Fetch John F. Kennedy airport by `_id` using the [DOCUMENT\(\) function](#), which will look up the document utilizing the primary index:

```
RETURN DOCUMENT( "airports/JFK" )
```

- ▶ Use a [FOR loop](#) to iterate over the airports collection, filter by `_key` and return the Kennedy airport document. This pattern gets optimized automatically to utilize the primary index as well:

```
FOR airport IN airports
  FILTER airport._key == "JFK"
  RETURN airport
```

- ▶ This construct can be used for complex filter criteria. Various [operators](#) are available.

```
FOR airport IN airports
  FILTER airport.city == "New York"
  AND airport.state == "NY"
  RETURN airport
```

- ▶ You can [SORT](#) the results by one or multiple conditions in ascending (default) and descending order (DESC), as well as offset and [LIMIT](#) the number of results. Note: The order of such high-level operations influences the output!

```
FOR a IN airports
  FILTER a.vip
  SORT a.state, a.name DESC
  LIMIT 5
  RETURN a
```

- ▶ You don't have to [RETURN](#) full documents, you can also return just parts of them (see the [KEEP\(\)](#) and [UNSET\(\)](#) functions for instance) or construct the query result as you desire:

```
FOR a IN airports
  FILTER a._key IN [ "JFK", "LAX" ]
  RETURN { fullName: a.name }
```



# First AQL Queries – Hands on

- ▶ Count all documents in the collection:  
**RETURN** COUNT(**airports**)
- ▶ Count how many V.I.P. airports there are.  
Below we use [COLLECT](#) to group the intermediate results without condition, which means all filtered documents are grouped together. COLLECT has a syntax variation which allows us to count the number of documents efficiently. We return this number as result:

```
FOR airport IN airports
FILTER airport.vip
COLLECT WITH COUNT INTO count
RETURN count
```

Feel free to experiment further. You can do a lot more with AQL, but that is beyond the scope of this course. Find the full [AQL documentation](#) online and see the [Training Center](#) on our website!

## Exercises A: Document Queries

Here are some challenges if you want to practice your AQL skills. Example solutions can be found at the end of this course.

1. Retrieve the airport document of **Los Angeles International (LAX)**.
2. Retrieve all airport documents of the **city Los Angeles**.
3. Find all airports of the **state North Dakota (ND)** and return the **name** attribute only.
4. Retrieve multiple airports via their primary key (**\_key**), for example **BIS**, **DEN** and **JFK**. Return an object for each match: **RETURN** {airport: a.name}
5. Count the airports in the **state New York (NY)** which are **not vip**.



# First AQL Queries – Hands on

Now that you are familiar with the dataset and AQL, try out the following graph queries before we go into the details of graph traversal.

- ▶ Return the names of all airports one can reach directly (1 step) from Los Angeles International (LAX) following the `flights` edges:

```
FOR airport IN 1..1 OUTBOUND
  'airports/LAX' flights
  RETURN DISTINCT airport.name
```

- ▶ Return any 10 flight documents with the flight **departing** at LAX and the destination airport documents like `{"airport": {...}, "flight": {...}}`

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

- ▶ Return 10 flight numbers with the plane **landing** in Bismarck Municipal airport (BIS):

```
FOR airport, flight IN INBOUND
  'airports/BIS' flights
  LIMIT 10
  RETURN flight.FlightNum
```

- ▶ Find all connections which **depart from or land at** BIS on January 5th and 7th and return the destination city and the arrival time in universal time (UTC):

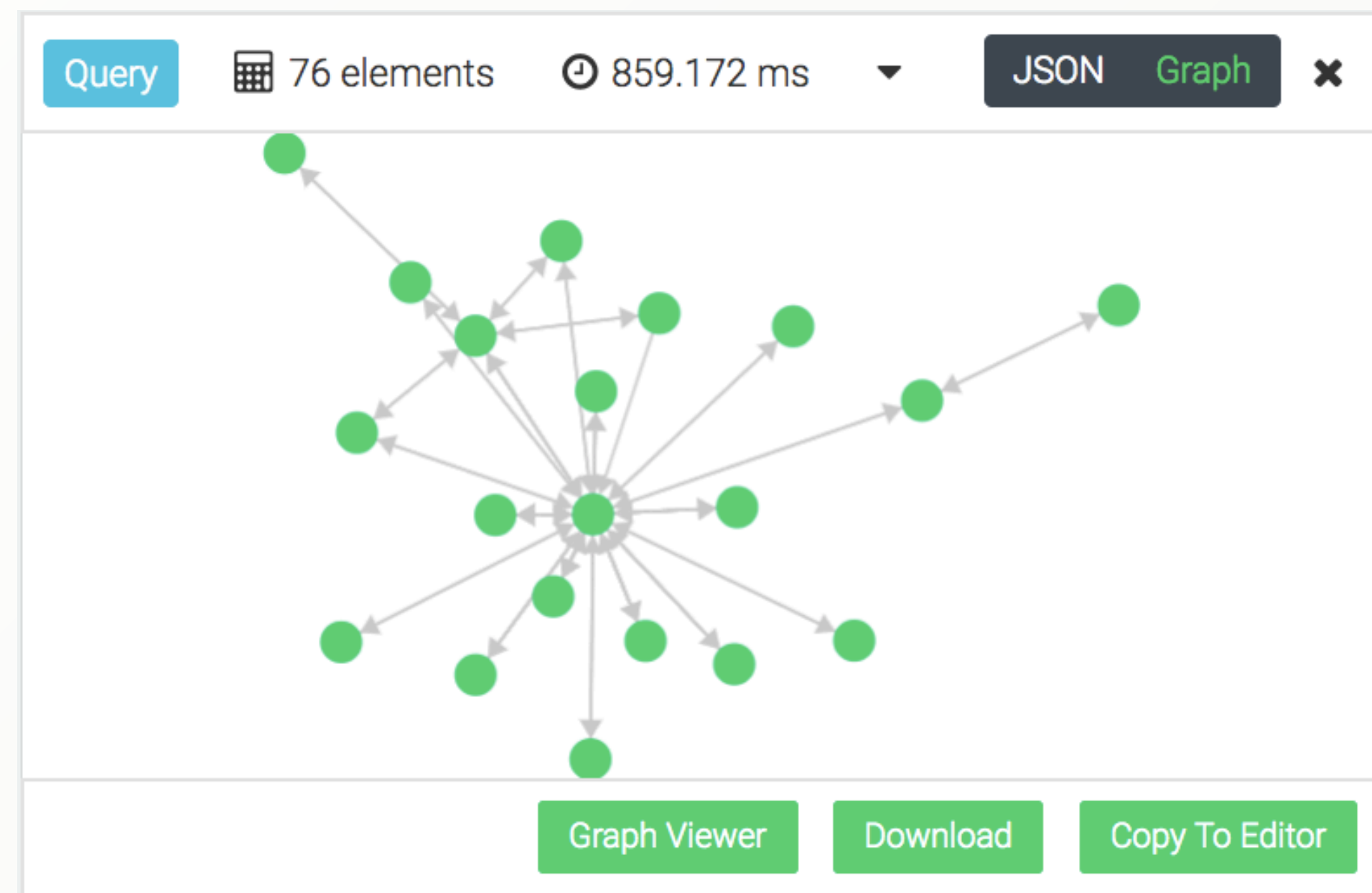
```
FOR airport, flight IN ANY
  'airports/BIS' flights
  FILTER flight.Month == 1
  AND flight.Day >= 5
  AND flight.Day <= 7
  RETURN { city: airport.city,
            time: flight.ArrTimeUTC }
```

# First AQL Queries – Hands on

- Edges can also be accessed without using graph traversals – they are just documents:

```
FOR flight IN flights
  FILTER flight.TailNum == "N238JB"
  RETURN flight
```

If there are `_from`, `_to` and `_id` attributes in the response, the WebUI will try to display the result in *Graph* view mode:



## Exercises B: Graph Queries

- Find all flights with `FlightNum 860` (number) on January 5th and return the `_from` and `_to` attributes only (you may use `KEEP()` for this).
- Find all flights departing or arriving at `JFK` with `FlightNum 859` or `860` and return objects with flight numbers and airport names where the flights go to or come from respectively.
- Combine a **FOR** loop and a traversal like:

```
FOR orig IN airports
  FILTER orig._key IN ["JFK", "PBI"]
  FOR dest IN OUTBOUND orig flights
```

...

to do multiple traversals with different starting points. Filter by flight numbers `859` and `860`. Return `orig.name`, `dest.name`, `FlightNum` and `Day`. Name the attributes appropriately.

# Graph Traversals

---

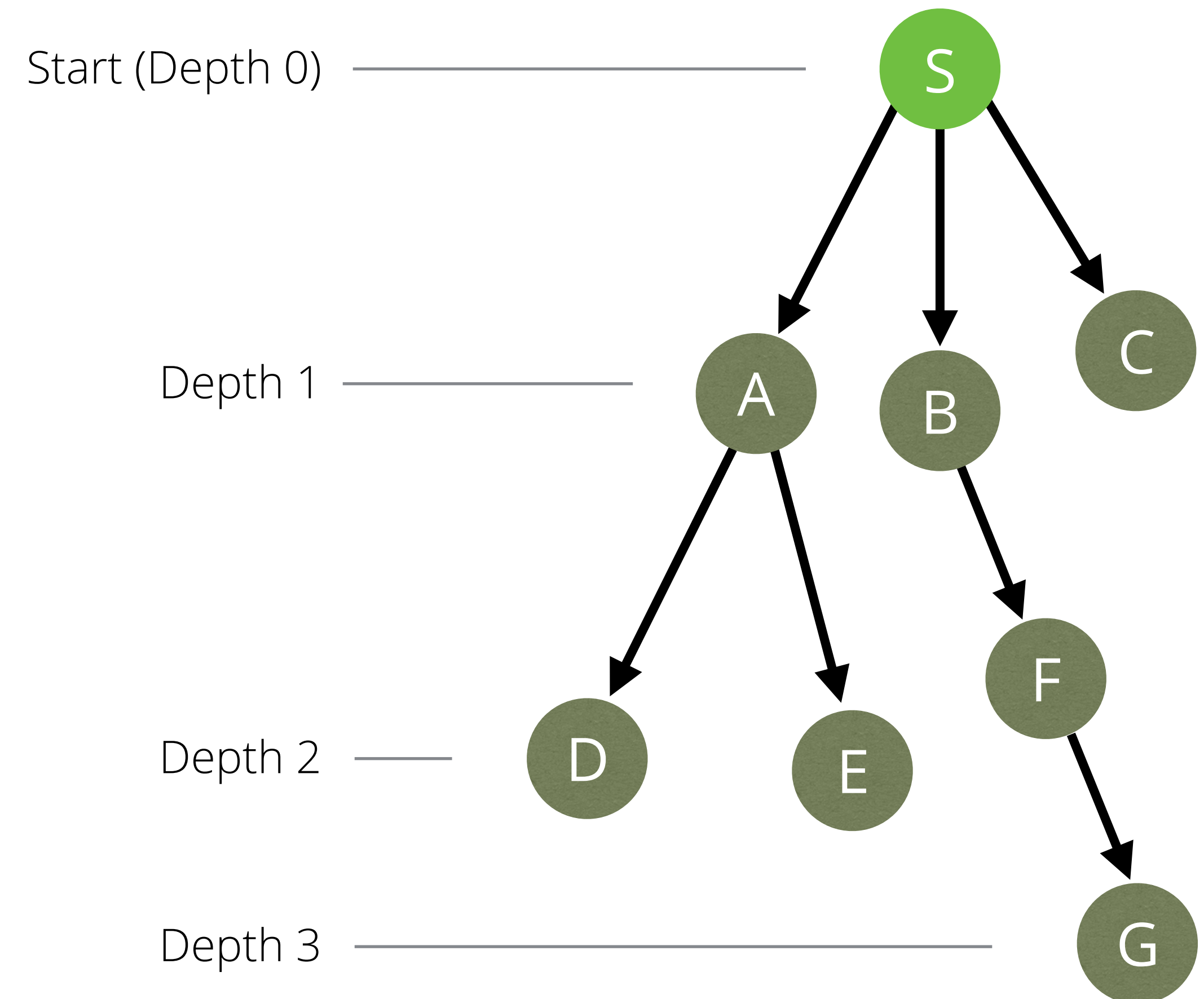
## **Traversals explained & Graph Traversal Syntax**

# Traversals explained

Traversal means to walk along edges of a graph in certain ways, optionally with some filters. Traversing is very efficient in graph databases. In ArangoDB, this is achieved by a [hybrid index type](#) which you already heard of: the edge index.

How many steps to go in a traversal is known as traversal **depth**:

- ▶ 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).

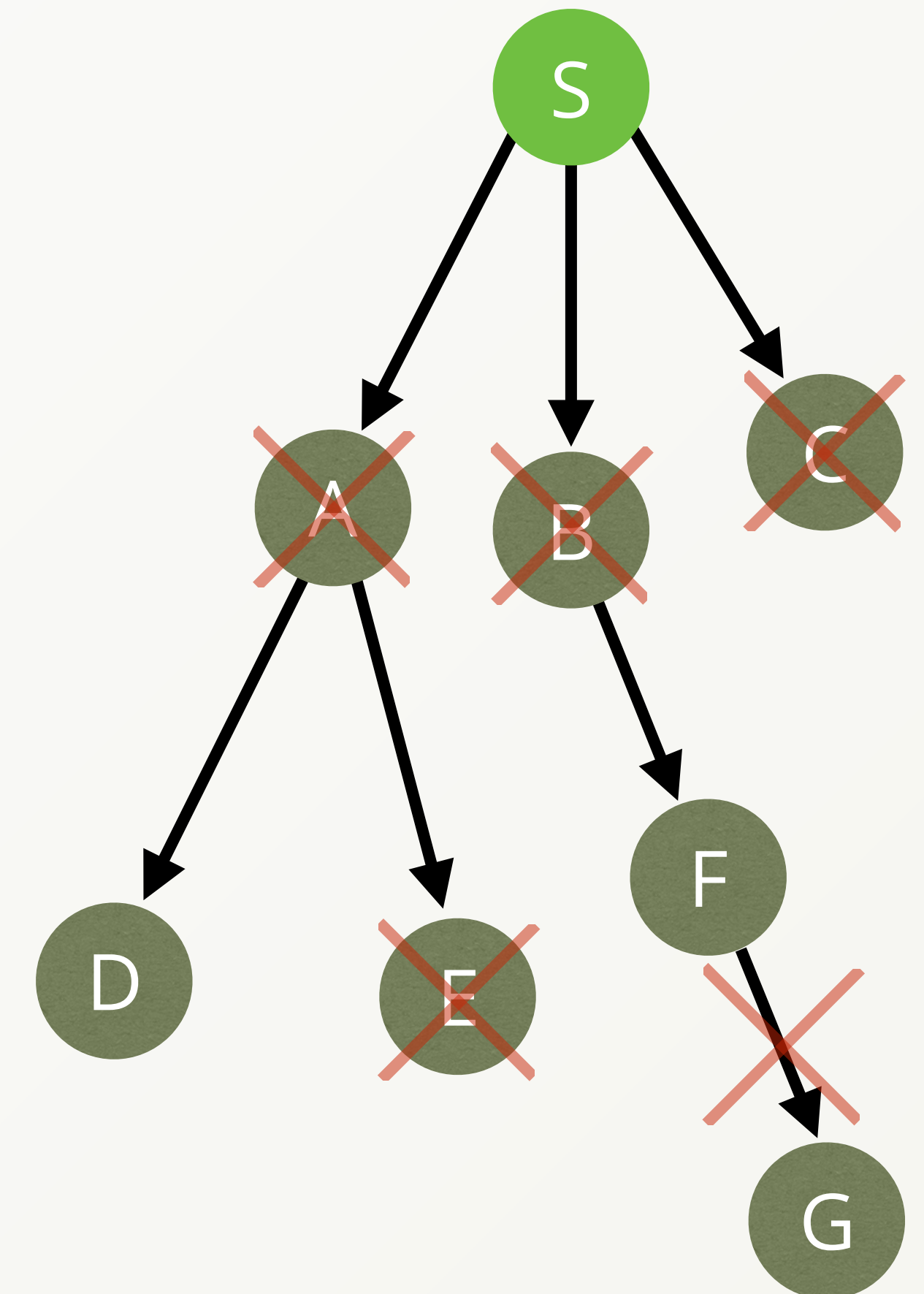




# Traversals explained

A traversal in OUTBOUND direction with a minimal and maximal depth of 2 might look like the following:

- ▶ We start the traversal at a vertex (**S**)
- ▶ The traverser walks down the first outgoing edge to **A**, but we are only at depth 1 (we defined a minimum and maximum of 2) – **A** is ignored.
- ▶ It continues down from **A** to **D**, the depth is 2 as required, so it returns **D**
- ▶ It also follows the other outgoing edge of **A** down to **E**. Depth is 2, but some filter condition we put in place is not met, so the path is discarded
- ▶ There are no more edges to follow from **A**, therefore the traversal continues with the second outgoing edge of **S** down to **B**. Depth is only 1, so **B** is ignored, but the traverser will continue from here
- ▶ It follows the edge from **B** down to **F**, depth is 2, filter conditions are met, **F** is returned
- ▶ There is an edge from **F** to **G**, but the maximal depth is reached already
- ▶ The traversal ends with the last outgoing edge of **S** to **C**, which has no edges to follow, depth is 1, hence **C** is ignored



# Graph Traversal Syntax

Before we do more 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:

## Query Syntax

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

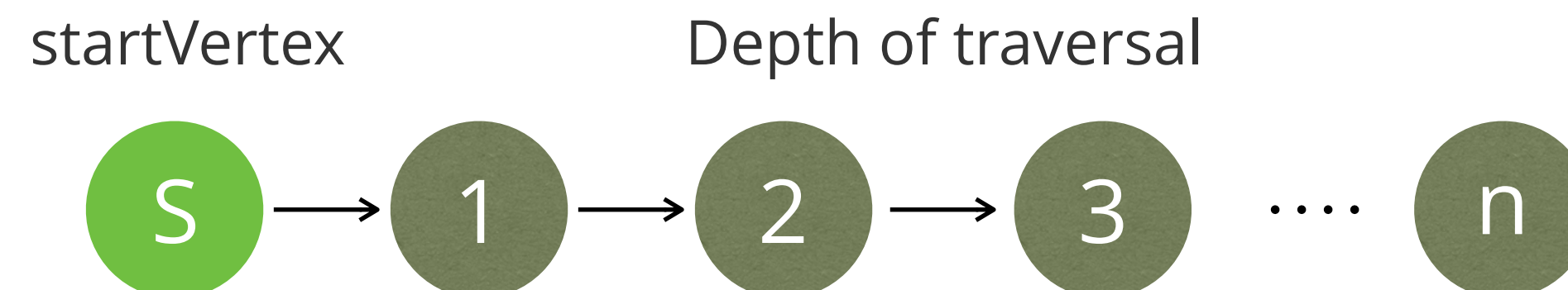
By the way: Keywords like **FOR**, **IN** and **ANY** are written all upper case in the code examples, but it is merely a convention. You may also write them all lower case or in mixed case. Names of variables, attributes and collections are case-sensitive however!

## 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 **min** defaults to **1** and **max** defaults to **min**



Traversal in AQL documentation

# Graph Traversal Syntax

Before we do more 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:

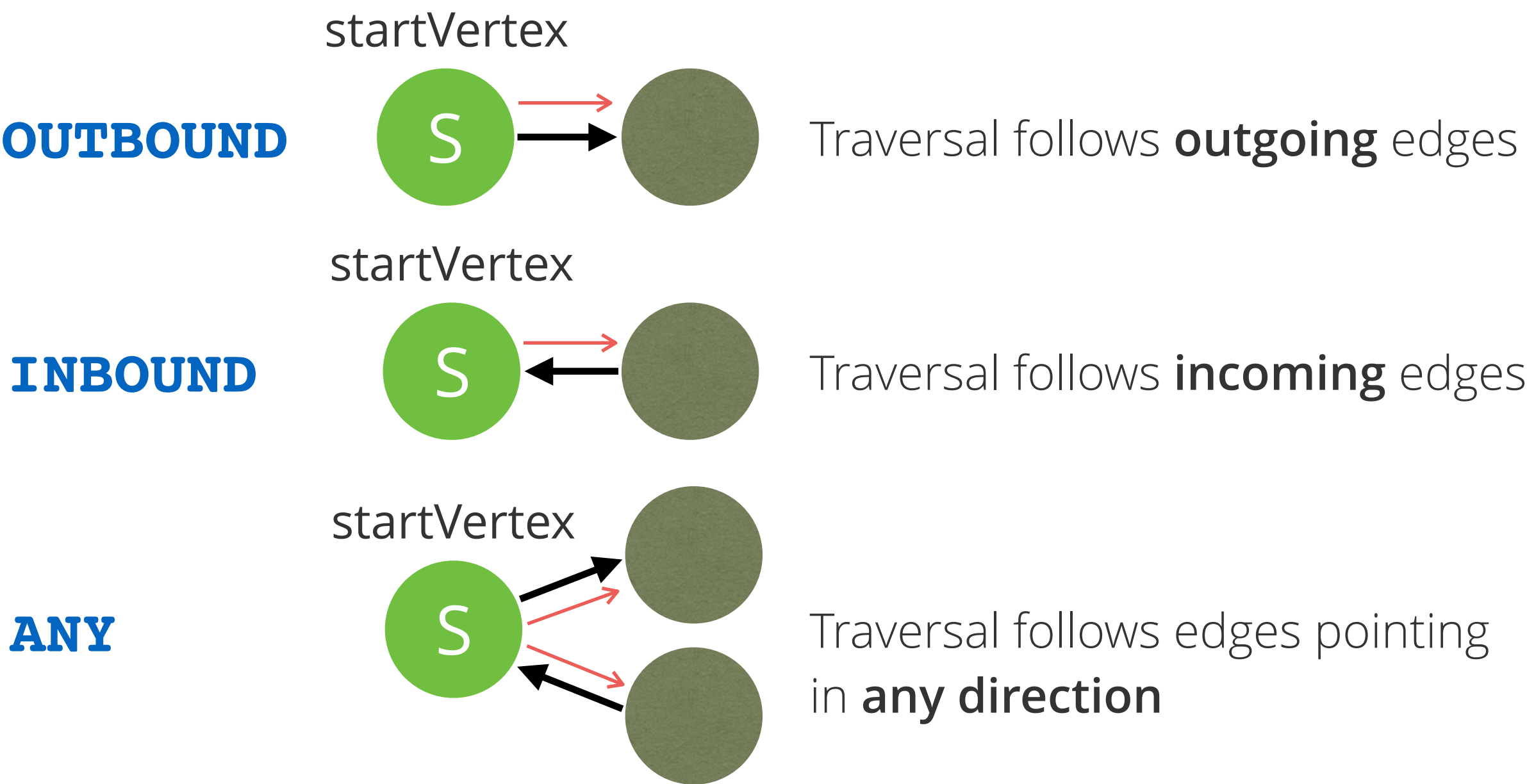
## Query Syntax

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



## Explanation

**OUTBOUND/INBOUND/ANY** defines the direction of your search



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



Traversal in AQL documentation



# Traversal Options

---

## **Depth vs. Breadth First Search & Uniqueness Options**

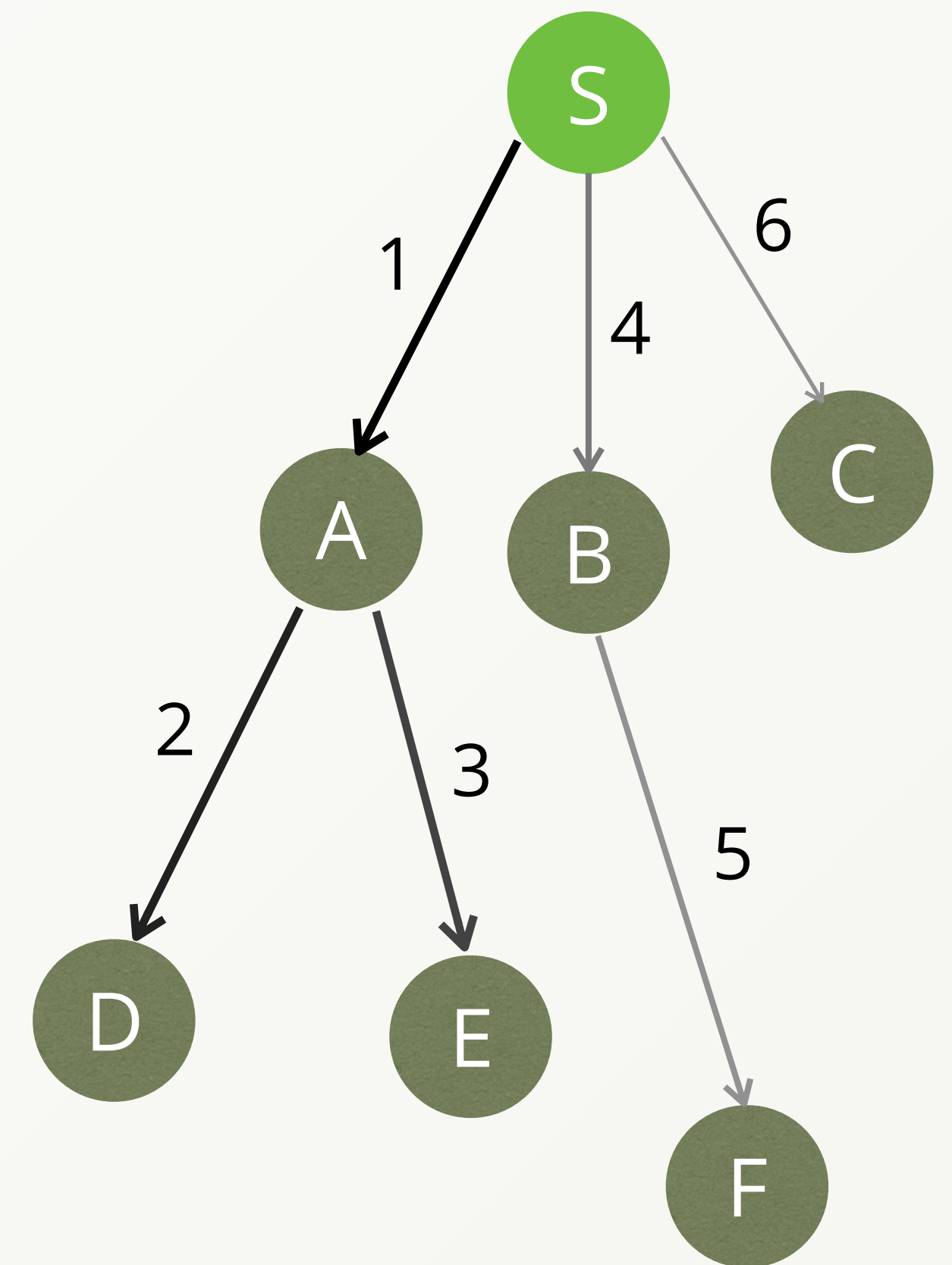
# Depth vs. Breadth First Search

Everybody who already took a closer look into the documentation about traversals, saw that there are also **OPTIONS** to control the traversal behavior.

For traversals with a minimum depth greater than or equal to 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 vs. Breadth First Search

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

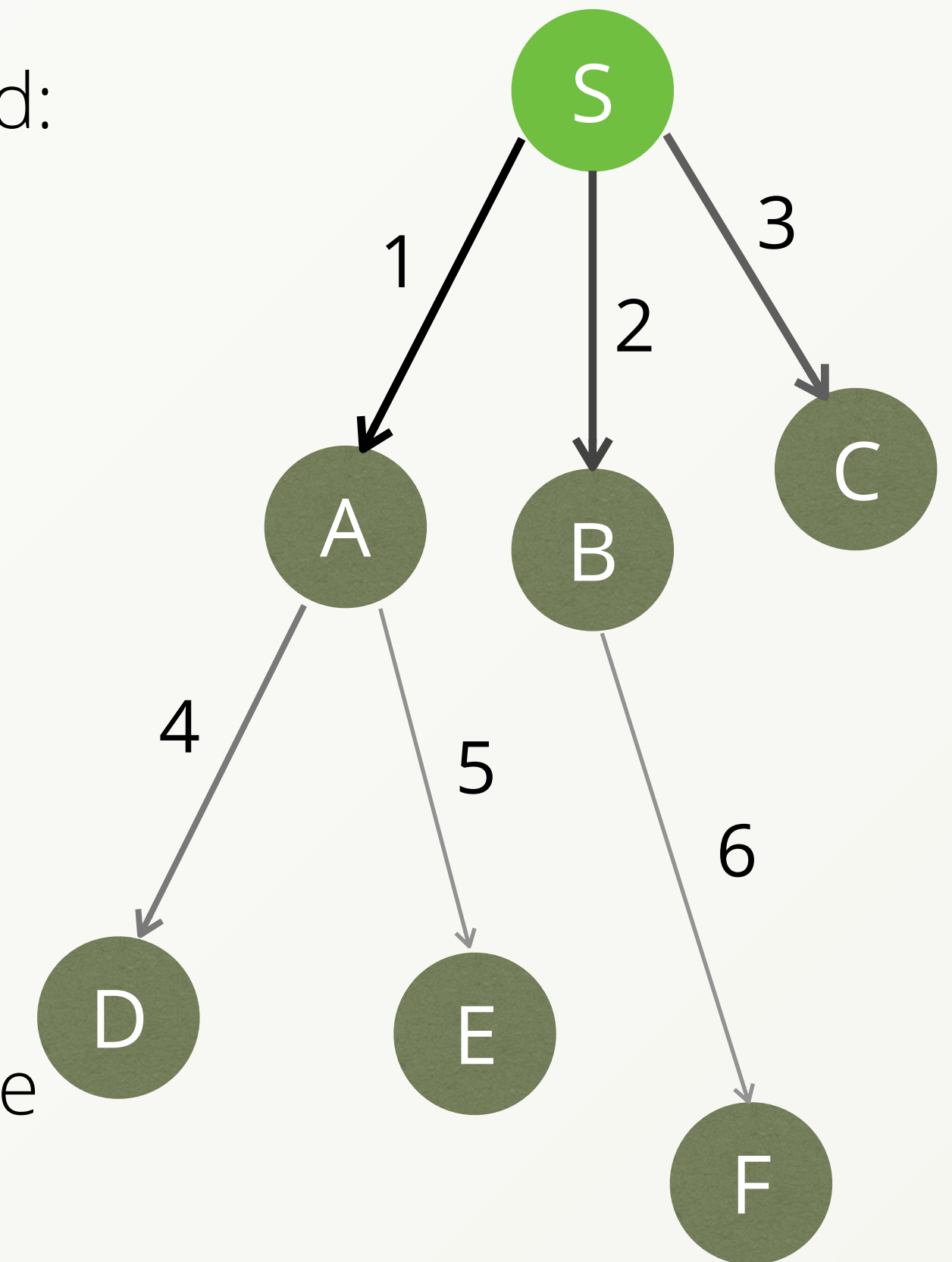
$S \rightarrow A$   
 $S \rightarrow B$   
 $S \rightarrow C$   


---

 $S \rightarrow A \rightarrow D$   
 $S \rightarrow A \rightarrow E$   
 $S \rightarrow B \rightarrow F$

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





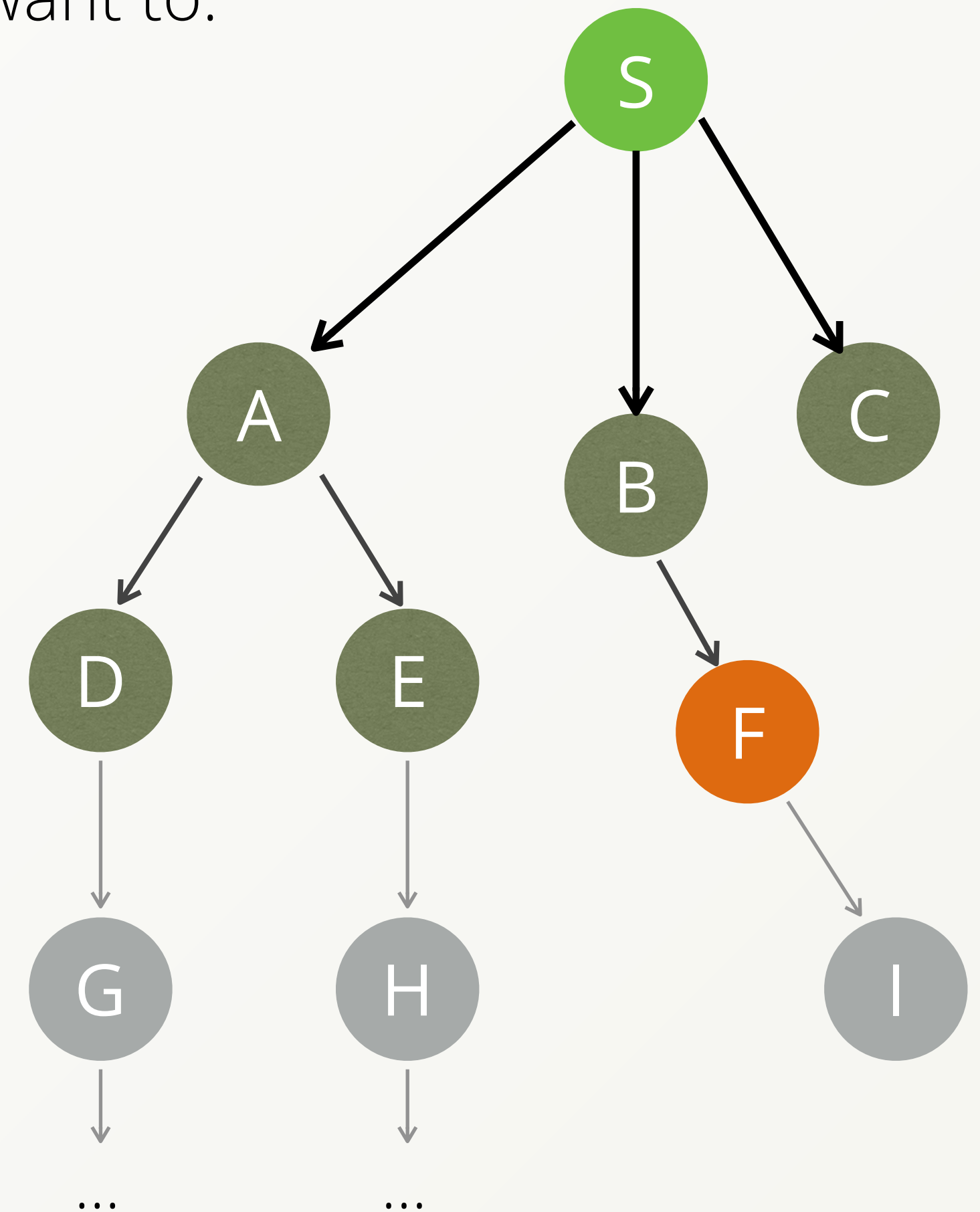
# Depth vs. Breadth First Search

**Breadth-first search** can significantly improve performance if used together with filters and limits by stopping before the maximal depth is reached.

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
```

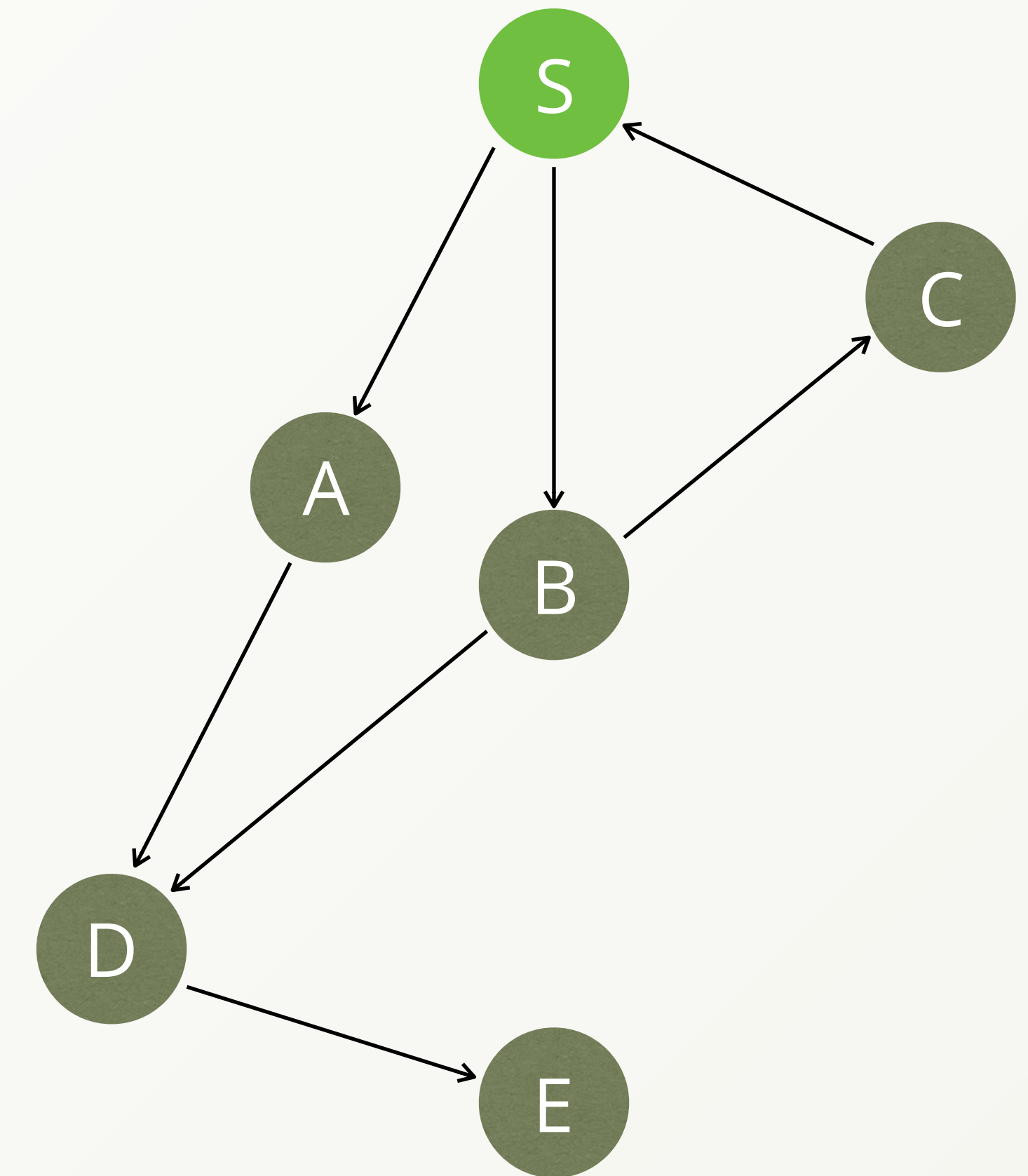


# Uniqueness Options

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

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

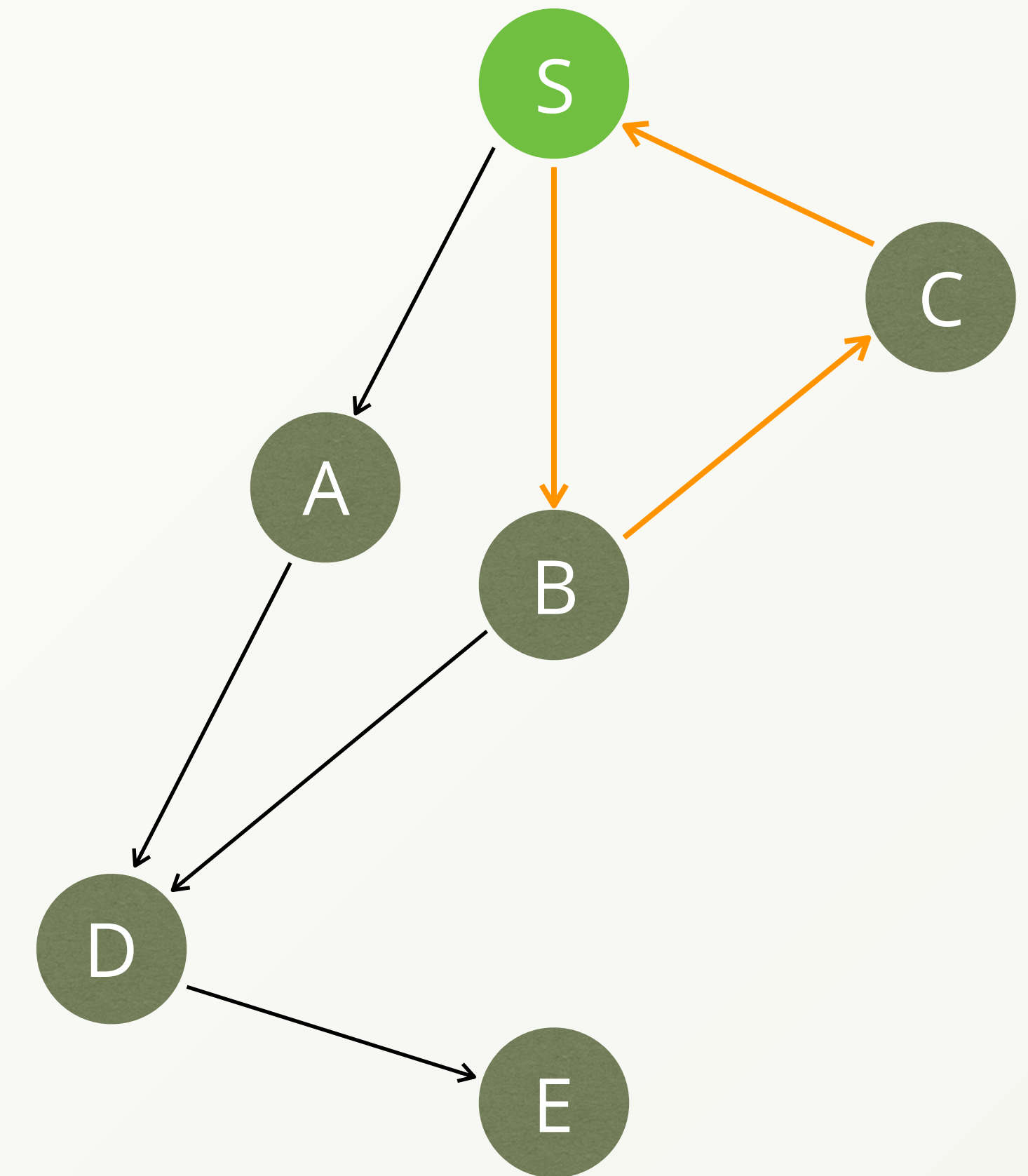
```
FOR v, e, p IN 1..5 OUTBOUND 'verts/S' edges
  OPTIONS {
    uniqueVertices: 'none',
    uniqueEdges: 'path'
  }
RETURN CONCAT_SEPARATOR('-', p.vertices[*]._key)
```

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





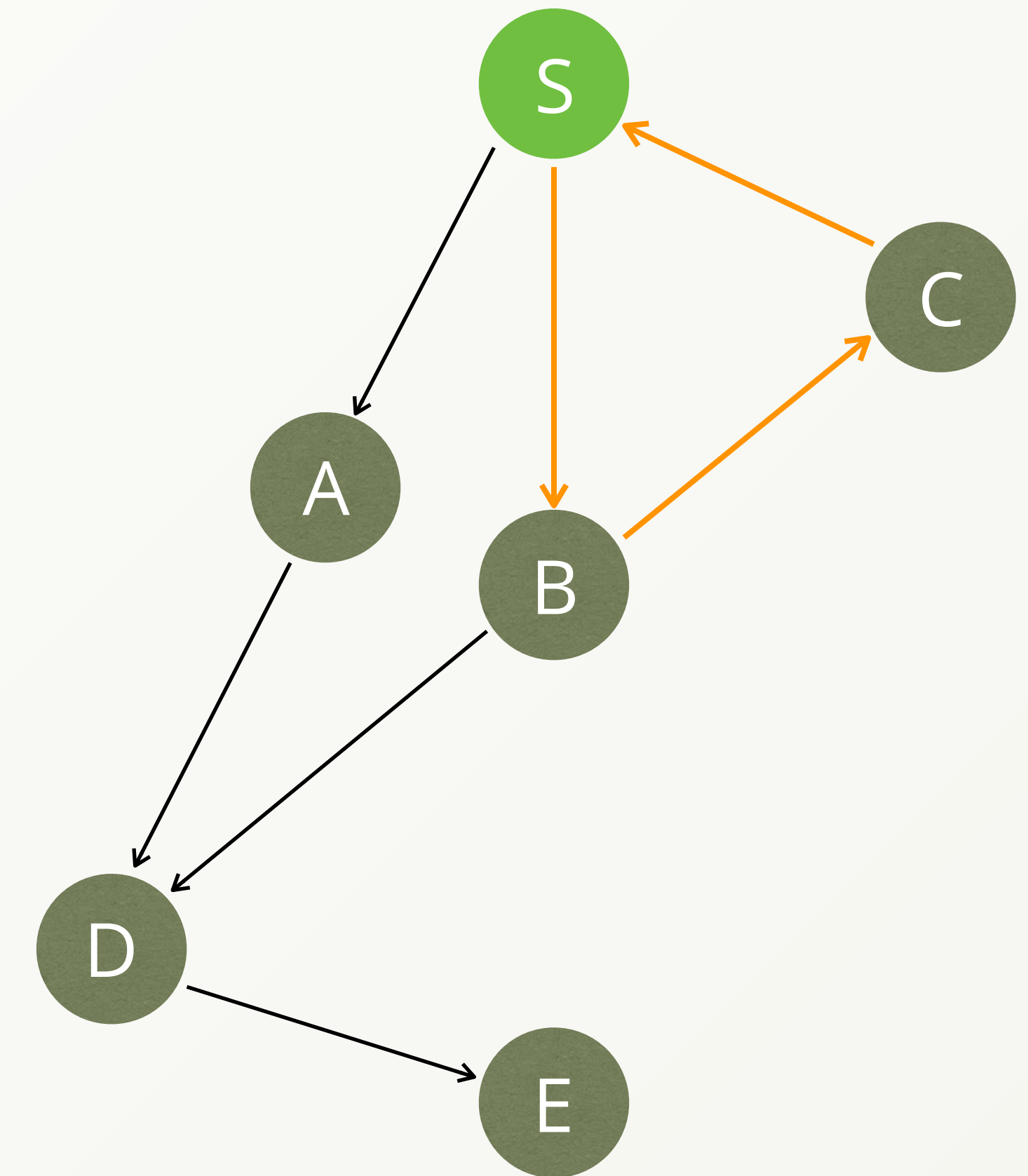
# Uniqueness Options

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 to C to S, and from S to B to C again. It would only stop there, because the maximum depth of 5 is reached at that point. If the maximum depth of the query was higher, then the traversal would run very long, producing a high amount of paths because of the loop.

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



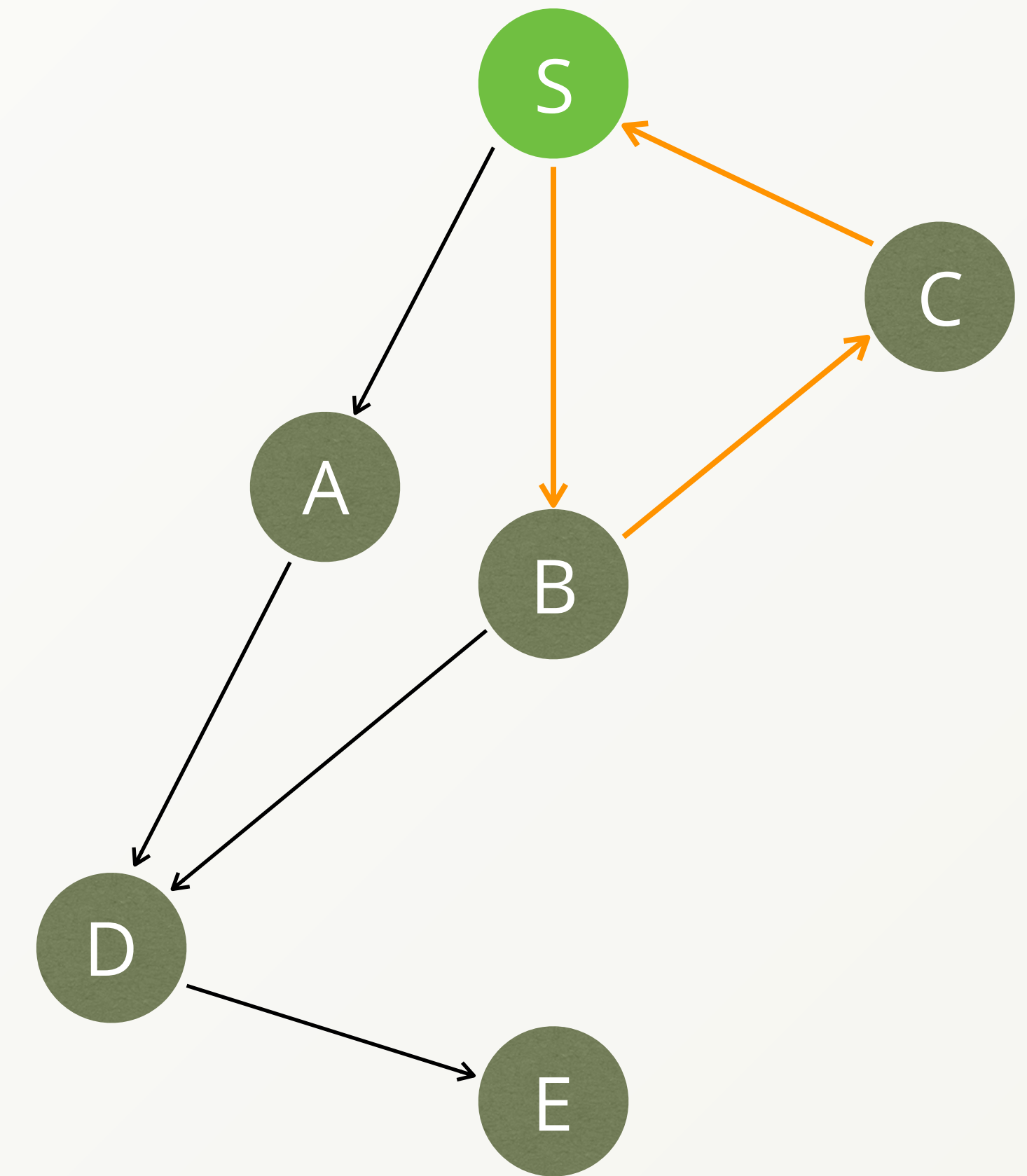
# Uniqueness Options

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

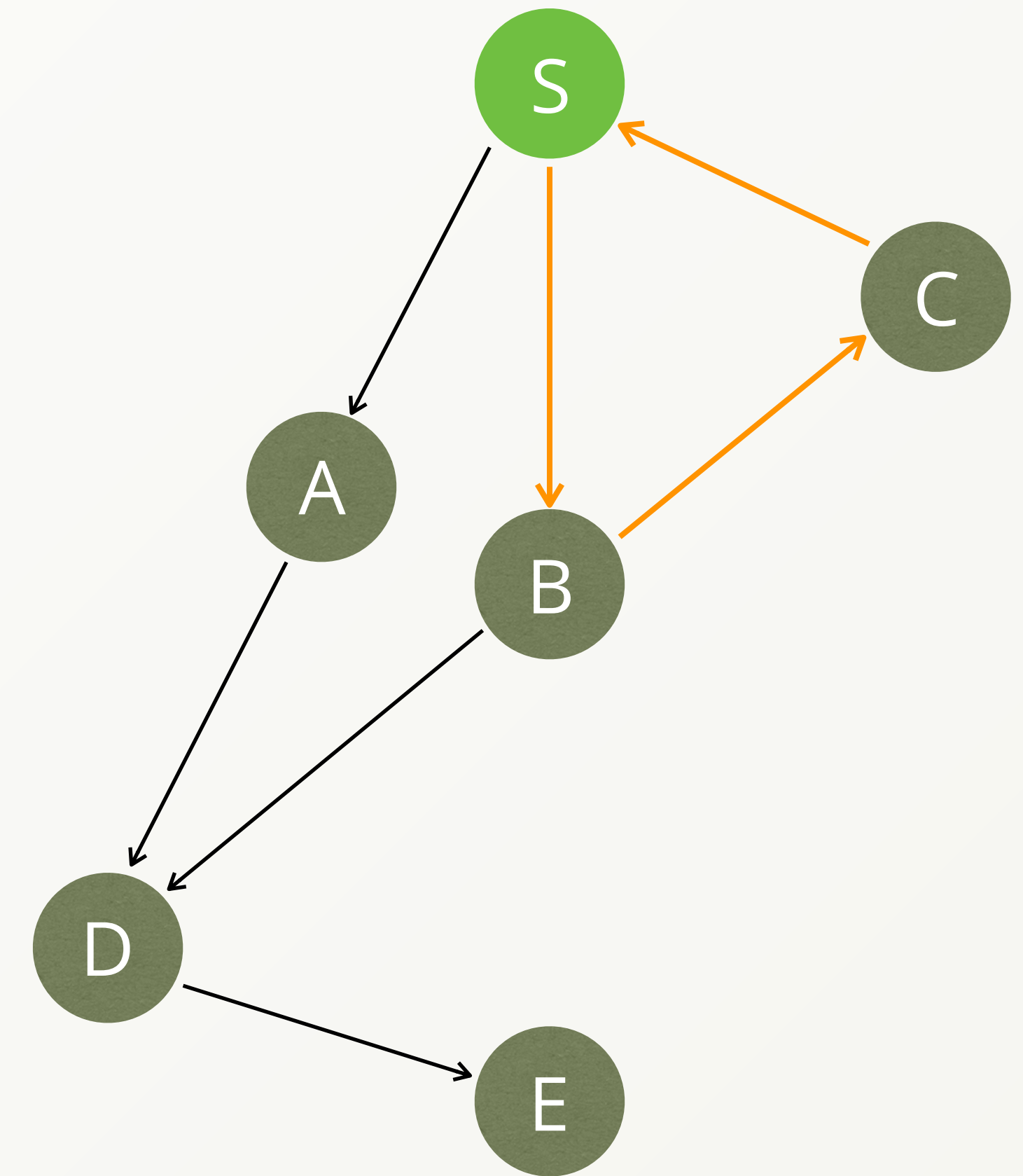
```
FOR v IN 0..5 OUTBOUND 'verts/S' edges
  OPTIONS {
    bfs: true,
    uniqueVertices: 'global'
  }
RETURN v._key
```

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  $E$





# Traversal Options – Hands on

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.

# Advanced Graph Queries

---

## **Shortest Path & Pattern Matching**

# Shortest Path – Hands on

A shortest path query finds a connection between two given vertices with the fewest amount of edges. With our domestic flights dataset we could search for a connection between two airports with the fewest stops for example.

- ▶ Find a shortest path between Bismarck Municipal airport and John F. Kennedy airport and return the airport names on the route:

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

We defined **BIS** as our start vertex and **JFK** as our target vertex.

[Shortest\\_Path in AQL documentation](#)





# Shortest Path



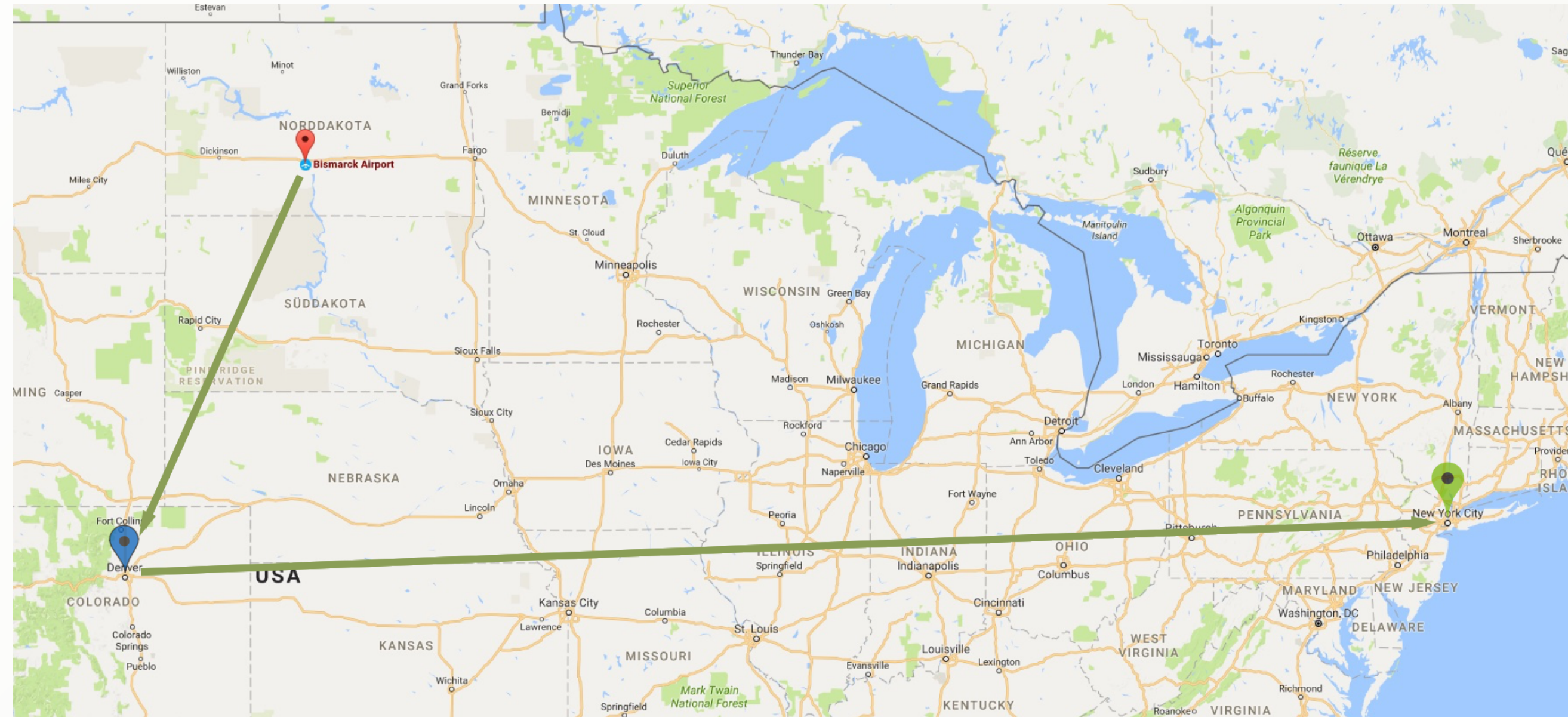
Source: Google Maps

We found a route via Denver International airport:

Query		3 elements	22.669 ms
1	[		
2	"Bismarck Municipal",		
3	"Denver Intl",		
4	"John F Kennedy Intl"		
5	]		



# Shortest Path



Source: Google Maps

The result of the previous shortest path query shows that you have to change in Denver (DEN) for example to get to JFK. There is apparently no direct flight.

**Note:** A Shortest\_Path query can return different results. It just finds and returns one of possibly multiple shortest paths. In this case it found: BIS→DEN→JFK

# Shortest Path – Hands on

Sometimes you just want the length of the shortest path. To achieve this you can use [LET](#).

- ▶ 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.

Note:

- ▶ We placed a **-1** at the end of the query to not count the end vertex as a step!
- ▶ **Using the shortest path algorithm one can not apply filters.**  
We need to resort to *pattern matching* instead to do so.



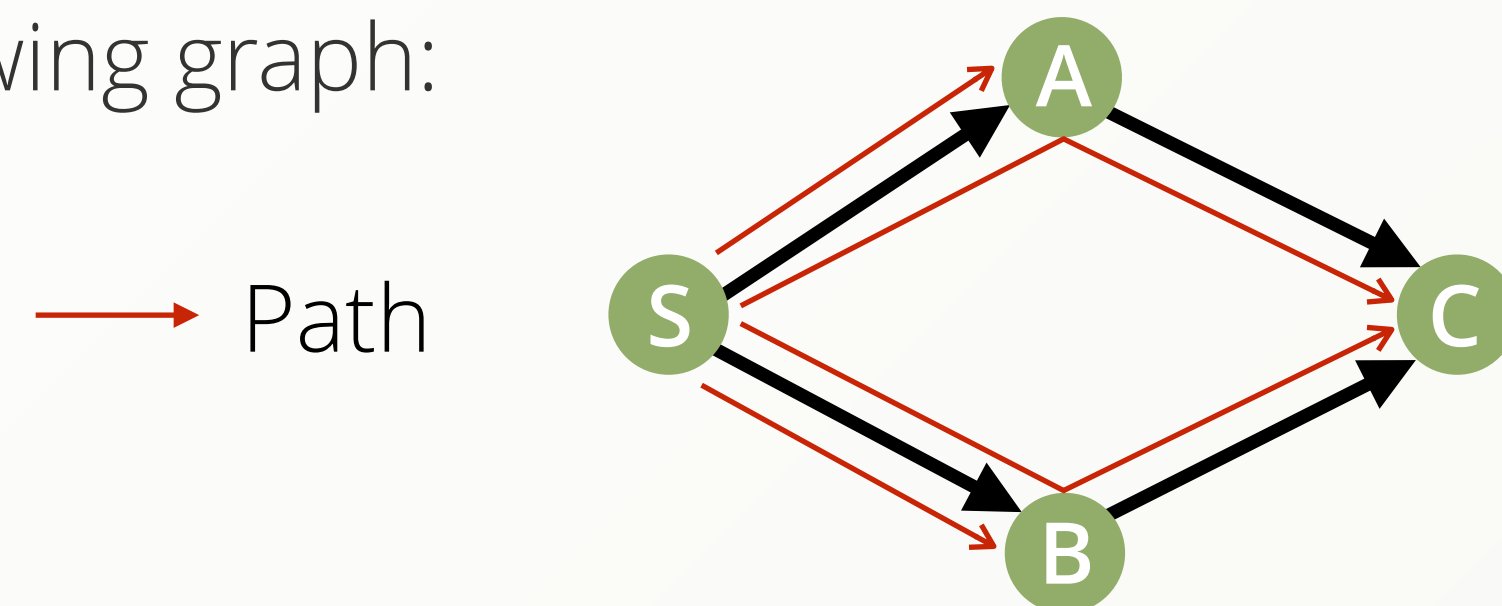
# Pattern Matching

We adventured pretty deep into the graph jungle already. Exploring pattern matching in detail is beyond the scope of this course, but let us take a quick look at it nonetheless.

We can easily add filter conditions for the end vertex and/or the edge which leads to it. Both are emitted by the traversal as we know:

**FOR** endVertex, edgeToVertex **IN** ...

With a variable traversal depth of 1..2 and the default traversal options, there are 4 paths in the following graph:



If we return the emitted end vertex, then the result will contain the vertices A, B, C and C again.

We could also return the edges and would end up with four edges in total. However, for the paths  $S \rightarrow A \rightarrow C$  and  $S \rightarrow B \rightarrow C$  we may want to choose one over the other based on certain criteria. Full paths can be optionally emitted as third variable:

**FOR** vertex, edge, path **IN** ...

The path variable can then be used to apply filter conditions on intermediate or all vertices and/or edges on the path. This allows for queries like:

*What are the best connections between the airports A and B determined by the lowest total travel time?*

It can be used to apply complex filter conditions in traversals taking the entire path into account. In other words, it lets you discover specific patterns – combinations of vertices and edges in graphs – and is therefore called *pattern matching*.

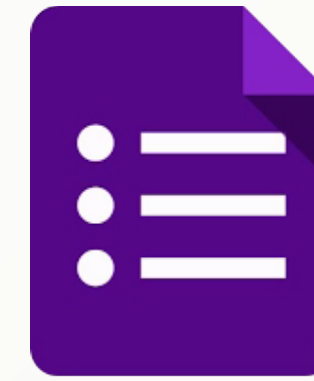
# Landing

---

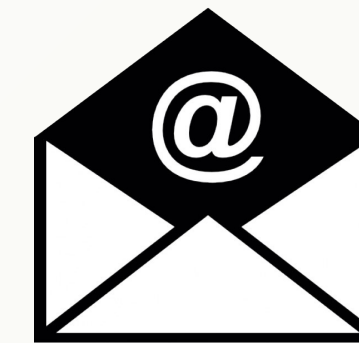
**Survey and Support  
&  
Exercise Solutions**

## What would you like to learn next?

Tell us with 3 clicks:



Survey



Feedback to  
the course



Star us on  
GitHub



Slack  
Community



StackOverflow

## Feeling stuck? Not for long.

Join the ArangoDB community to get help,  
challenge ideas or discuss new features!



# Exercises A – Solutions

There are often multiple ways in AQL to retrieve the same result. If your solution is different to below queries but produces the correct result then you did very well :)

1. Retrieve the airport document of Los Angeles International (LAX).

```
RETURN DOCUMENT( "airports/LAX" )
```

2. Retrieve all airport documents of the city Los Angeles.

```
FOR a IN airports
  FILTER a.city == "Los Angeles"
RETURN a
```

3. Find all airports of the state North Dakota (ND) and return the name attribute only.

```
FOR airport IN airports
  FILTER airport.state == "ND"
RETURN airport.name
```

4. Retrieve multiple airports via their primary key (\_key), for example BIS, DEN and JFK. Return an object for each match: **RETURN** {airport: a.name}

```
FOR a IN airports
  FILTER a._key IN [ "BIS", "DEN", "JFK" ]
RETURN { airport: a.name }
```

5. Count the airports in the state New York (NY) which are not vip.

```
FOR airport IN airports
  FILTER airport.state == "NY"
  AND NOT airport.vip
COLLECT WITH COUNT INTO count
RETURN count
```

# Exercises B – Solutions

1. Find all flights with **FlightNum** 860 (number) on January 5th and return the *\_from* and *\_to* attributes only (you may use [KEEP\(\)](#) for this).

```
FOR f IN flights
  FILTER f.FlightNum == 860
  AND f.Month == 1
  AND f.Day == 5
  RETURN KEEP(f, "_from", "_to")
```

2. Find all flights departing or arriving at **JFK** with **FlightNum** 859 or 860 and return objects with flight numbers and airport names where the flights go to or come from respectively.

```
FOR a,f IN ANY
  "airports/JFK" flights
  FILTER f.FlightNum IN [859,860]
  RETURN { airport: a.name,
           flight: f.FlightNum }
```

3. Combine a **FOR** loop and a traversal like:

```
FOR orig IN airports
  FILTER orig._key IN ["JFK", "PBI"]
  FOR dest IN OUTBOUND orig flights
```

...

to do multiple traversals with different starting points. Filter by flight numbers 859 and 860. Return *orig.name*, *dest.name*, **FlightNum** and **Day**. Name the attributes appropriately.

```
FOR orig IN airports
  FILTER orig._key IN ["JFK", "PBI"]
  FOR dest, flight IN
    OUTBOUND orig flights
    FILTER dest.FlightNum IN [859,860]
    RETURN { from: orig.name,
             to: dest.name,
             number: f.FlightNum,
             day: f.Day }
```



---

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



**Simran**

Documentation Manager

AQL and data modeling  
enthusiast with a passion  
for technical writing



**Jan**

Head of Communications

Makes complex things  
easier to digest. Big fan  
of community support