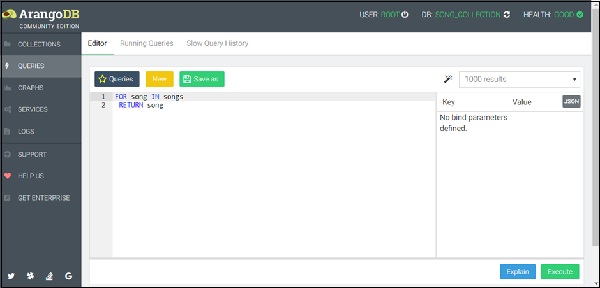
In this chapter, we will discuss how to query the data with AQL. We have already discussed in our previous chapters that ArangoDB has developed its own query language and that it goes by the name AQL.

Let us now start interacting with AQL. As shown in the image below, in the web interface, press the **AQL Editor** tab placed at the top of the navigation bar. A blank query editor will appear.

When need, you can switch to the editor from the result view and vice-versa, by clicking the Query or the Result tabs in the top right corner as shown in the image below −



Among other things, the editor has syntax highlighting, undo/redo functionality, and query saving. For a detailed reference, one can see the official documentation. We will highlight few basic and commonly-used features of the AQL query editor.

AQL Fundamentals

In AQL, a query represents the end result to be achieved, but not the process through which the end result is to be achieved. This feature is commonly known as a declarative property of the language. Moreover, AQL can query as well modify the data, and thus complex queries can be created by combining both the processes.

Please note that AQL is entirely ACID-compliant. Reading or modifying queries will either conclude in whole or not at all. Even reading a document's data will finish with a consistent unit of the data.

We add two new **songs** to the songs collection we have already created. Instead of typing, you can copy the following query, and paste it in the AQL editor −

FOR song IN [

{

title: "Air-Minded Executive", lyricist: "Johnny Mercer",

composer: "Bernie Hanighen", Year: 1940, \_key: "Air-Minded"

},

{

title: "All Mucked Up", lyricist: "Johnny Mercer", composer:

"Andre Previn", Year: 1974, \_key: "All\_Mucked"

}

]

INSERT song IN songs

Press the Execute button at the lower left.

It will write two new documents in the **songs** collection.

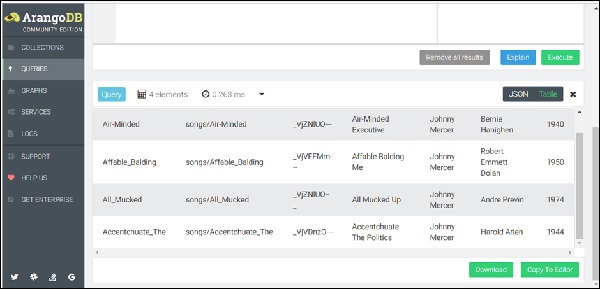
This query describes how the FOR loop works in AQL; it iterates over the list of JSON encoded documents, performing the coded operations on each one of the documents in the collection. The different operations can be creating new structures, filtering, selecting documents, modifying, or inserting documents into the database (refer the instantaneous example). In essence, AQL can perform the CRUD operations efficiently.

To find all the songs in our database, let us once again run the following query, equivalent to a **SELECT \* FROM songs** of an SQL-type database (because the editor memorizes the last query, press the **\*New\*** button to clean the editor) −

FOR song IN songs

RETURN song

The result set will show the list of songs so far saved in the **songs** collection as shown in the screenshot below.



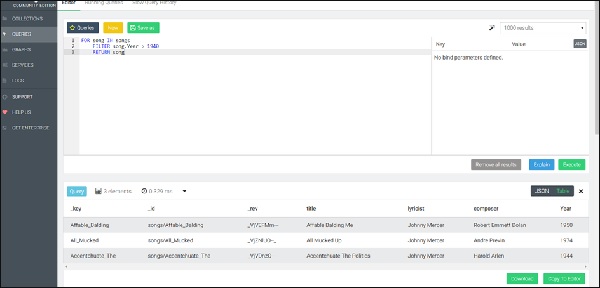
Operations like **FILTER, SORT** and **LIMIT** can be added to the **For loop** body to narrow and order the result.

FOR song IN songs

FILTER song.Year > 1940

RETURN song

The above query will give songs created after the year 1940 in the Result tab (see the image below).



The document key is used in this example, but any other attribute can also be used as an equivalent for filtering. Since the document key is guaranteed to be unique, no more than a single document will match this filter. For other attributes this may not be the case. To return a subset of active users (determined by an attribute called status), sorted by name in ascending order, we use the following syntax −

FOR song IN songs

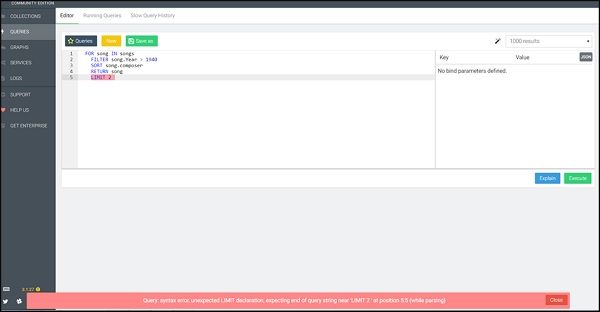
FILTER song.Year > 1940

SORT song.composer

RETURN song

LIMIT 2

We have deliberately included this example. Here, we observe a query syntax error message highlighted in red by AQL. This syntax highlights the errors and is helpful in debugging your queries as shown in the screenshot below.



Let us now run the correct query (note the correction) −

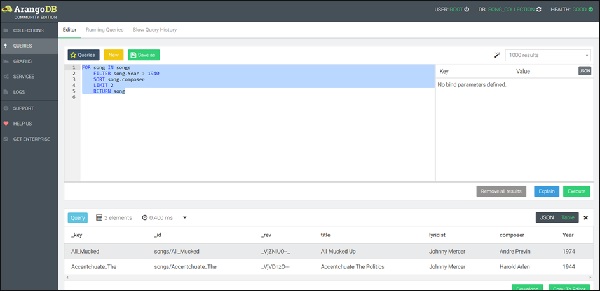
FOR song IN songs

FILTER song.Year > 1940

SORT song.composer

LIMIT 2

RETURN song



Complex Query in AQL

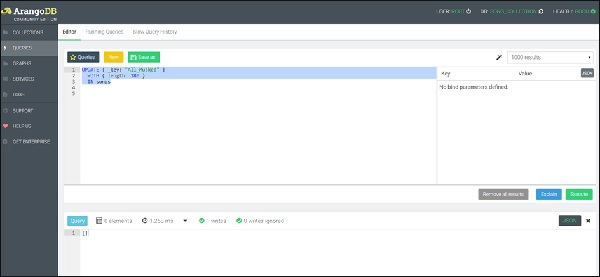
AQL is equipped with multiple functions for all supported data types. Variable assignment within a query allows to build very complex nested constructs. This way data-intensive operations move closer to the data at the backend than on to the client (such as browser). To understand this, let us first add the arbitrary durations (length) to songs.

Let us start with the first function, i.e., the Update function −

UPDATE { \_key: "All\_Mucked" }

WITH { length: 180 }

IN songs



We can see one document has been written as shown in the above screenshot.

Let us now update other documents (songs) too.

UPDATE { \_key: "Affable\_Balding" }

WITH { length: 200 }

IN songs

We can now check that all our songs have a new attribute **length** −

FOR song IN songs

RETURN song

Output

[

{

"\_key": "Air-Minded",

"\_id": "songs/Air-Minded",

"\_rev": "\_VkC5lbS---",

"title": "Air-Minded Executive",

"lyricist": "Johnny Mercer",

"composer": "Bernie Hanighen",

"Year": 1940,

"length": 210

},

{

"\_key": "Affable\_Balding",

"\_id": "songs/Affable\_Balding",

"\_rev": "\_VkC4eM2---",

"title": "Affable Balding Me",

"lyricist": "Johnny Mercer",

"composer": "Robert Emmett Dolan",

"Year": 1950,

"length": 200

},

{

"\_key": "All\_Mucked",

"\_id": "songs/All\_Mucked",

"\_rev": "\_Vjah9Pu---",

"title": "All Mucked Up",

"lyricist": "Johnny Mercer",

"composer": "Andre Previn",

"Year": 1974,

"length": 180

},

{

"\_key": "Accentchuate\_The",

"\_id": "songs/Accentchuate\_The",

"\_rev": "\_VkC3WzW---",

"title": "Accentchuate The Politics",

"lyricist": "Johnny Mercer",

"composer": "Harold Arlen",

"Year": 1944,

"length": 190

}

]

To illustrate the use of other keywords of AQL such as LET, FILTER, SORT, etc., we now format the song's durations in the **mm:ss** format.

Query

FOR song IN songs

FILTER song.length > 150

LET seconds = song.length % 60

LET minutes = FLOOR(song.length / 60)

SORT song.composer

RETURN

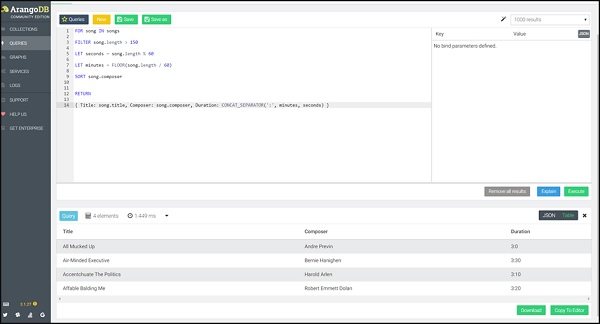
{

Title: song.title,

Composer: song.composer,

Duration: CONCAT\_SEPARATOR(':',minutes, seconds)

}



This time we will return the song title together with the duration. The **Return** function lets you create a new JSON object to return for each input document.

We will now talk about the ‘Joins’ feature of AQL database.

Let us begin by creating a collection **composer\_dob**. Further, we will create the four documents with the hypothetical date of births of the composers by running the following query in the query box −

FOR dob IN [

{composer: "Bernie Hanighen", Year: 1909}

,

{composer: "Robert Emmett Dolan", Year: 1922}

,

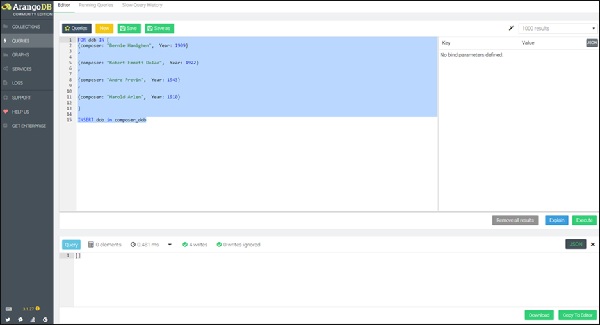
{composer: "Andre Previn", Year: 1943}

,

{composer: "Harold Arlen", Year: 1910}

]

INSERT dob in composer\_dob



To highlight the similarity with SQL, we present a nested FOR-loop query in AQL, leading to the REPLACE operation, iterating first in the inner loop, over all the composers’ dob and then on all the associated songs, creating a new document containing attribute **song\_with\_composer\_key** instead of the **song** attribute.

Here goes the query −

FOR s IN songs

FOR c IN composer\_dob

FILTER s.composer == c.composer

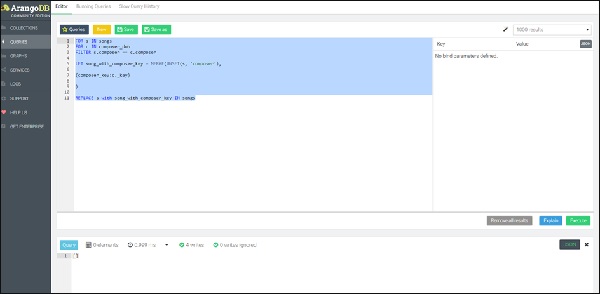
LET song\_with\_composer\_key = MERGE(

UNSET(s, 'composer'),

{composer\_key:c.\_key}

)

REPLACE s with song\_with\_composer\_key IN songs



Let us now run the query **FOR song IN songs RETURN song** again to see how the song collection has changed.

Output

[

{

"\_key": "Air-Minded",

"\_id": "songs/Air-Minded",

"\_rev": "\_Vk8kFoK---",

"Year": 1940,

"composer\_key": "5501",

"length": 210,

"lyricist": "Johnny Mercer",

"title": "Air-Minded Executive"

},

{

"\_key": "Affable\_Balding",

"\_id": "songs/Affable\_Balding",

"\_rev": "\_Vk8kFoK--\_",

"Year": 1950,

"composer\_key": "5505",

"length": 200,

"lyricist": "Johnny Mercer",

"title": "Affable Balding Me"

},

{

"\_key": "All\_Mucked",

"\_id": "songs/All\_Mucked",

"\_rev": "\_Vk8kFoK--A",

"Year": 1974,

"composer\_key": "5507",

"length": 180,

"lyricist": "Johnny Mercer",

"title": "All Mucked Up"

},

{

"\_key": "Accentchuate\_The",

"\_id": "songs/Accentchuate\_The",

"\_rev": "\_Vk8kFoK--B",

"Year": 1944,

"composer\_key": "5509",

"length": 190,

"lyricist": "Johnny Mercer",

"title": "Accentchuate The Politics"

}

]

The above query completes the data migration process, adding the **composer\_key** to each song.

Now the next query is again a nested FOR-loop query, but this time leading to the Join operation, adding the associated composer's name (picking with the help of `composer\_key`) to each song −

FOR s IN songs

FOR c IN composer\_dob

FILTER c.\_key == s.composer\_key

RETURN MERGE(s,

{ composer: c.composer }

)

Output

[

{

"Year": 1940,

"\_id": "songs/Air-Minded",

"\_key": "Air-Minded",

"\_rev": "\_Vk8kFoK---",

"composer\_key": "5501",

"length": 210,

"lyricist": "Johnny Mercer",

"title": "Air-Minded Executive",

"composer": "Bernie Hanighen"

},

{

"Year": 1950,

"\_id": "songs/Affable\_Balding",

"\_key": "Affable\_Balding",

"\_rev": "\_Vk8kFoK--\_",

"composer\_key": "5505",

"length": 200,

"lyricist": "Johnny Mercer",

"title": "Affable Balding Me",

"composer": "Robert Emmett Dolan"

},

{

"Year": 1974,

"\_id": "songs/All\_Mucked",

"\_key": "All\_Mucked",

"\_rev": "\_Vk8kFoK--A",

"composer\_key": "5507",

"length": 180,

"lyricist": "Johnny Mercer",

"title": "All Mucked Up",

"composer": "Andre Previn"

},

{

"Year": 1944,

"\_id": "songs/Accentchuate\_The",

"\_key": "Accentchuate\_The",

"\_rev": "\_Vk8kFoK--B",

"composer\_key": "5509",

"length": 190,

"lyricist": "Johnny Mercer",

"title": "Accentchuate The Politics",

"composer": "Harold Arlen"

}

]

