Switching from Relational Databases to ArangoDB

Comparing RDBMS to a Native Multi-Model Database System

This whitepaper compares relational database management systems (RDBMS) to native multi-model database systems — in particular, MySQL and ArangoDB. It describes their key concepts, contrasts both at the end of each section, and concludes by explaining what sets ArangoDB apart.

This is not a migration guide, nor does it explain how to port software applications using an RDBMS to ArangoDB. Instead, it's a high-level overview of features and differences. It can, though, serve as a starting point to understand the mindset of both types of systems, for a reader to bridge the gap from what they know about databases like MySQL, to what they may not yet understand in ArangoDB.

This white paper will also familiarize the reader with principles in ArangoDB, such as document stores and graph databases. These are combined in ArangoDB in the same database core, accessible with one query language. The goal of this white paper is especially to help the reader appreciate the comparable ease of the transition to greater storage flexibility with multiple data models that ArangoDB provides.

An ArangoDB White Paper – August 2018

ArangoDB

Table of Contents

1. Introduction

ArangoDB is a native multi-model NoSQL database. It supports three popular data models, namely key-value, document and graph. ArangoDB is written in modern C++ and is available as open-source version (Community Edition) and also comes with a commercial version including additional features for performance and security (Enterprise Edition).

Since many developers with a relational database background are considering moving to NoSQL for new projects, we created this white paper to make the reader familiar with the differences and similarities of both database systems, in particular ArangoDB.

Moving relational data to ArangoDB is simple: export the stored data to a format such as CSV, and then use `arangoimp` to import the data into ArangoDB. Once that's done, you may continue working with the same data model that you used previously with the relational database. Because ArangoDB supports real join operations, there is no need for nested data to circumvent normalization and joins. Hence, you don't have to rework the data model.

Still, there are some differences between both storage approaches. This paper will describe the difference between each database concept, as well as the data modeling possibilities and query options available with each. In the final sections, this paper will also introduce the cluster architecture of ArangoDB and the options that are available for scaling horizontally – even with complex queries.

2. Database Concepts

Before delving into data modeling and more advanced concepts, let's start with a general examination of the database concepts behind relational database systems and contrast that with the basic concepts of a database system like ArangoDB.

2.1 Relational Database System Concepts

The data model of a relational database management system (i.e., RDBMS) utilizes the controlling metaphor of a table, much like ones found in inventory ledger books used for centuries in business. Records are stored in tables, with each record being a row in that table. The column heads define the available fields and the values for each of the columns per record in individual cells. Such tables can be organized in databases (e.g., for multi tenancy use). There are a few core databases (e.g., `information_schema`) that contain tables which store system configuration variables and other data for internal use by the system.

Data in a relational database system can be queried and retrieved in a tabular format. It can be full rows or subsets of columns. Behind the scenes, a binary transport protocol is used to send data. Although one can choose between different storage engines – InnoDB is the most popular – with different features and performance properties, how data is actually stored is invisible to the user.

Table and Schema Definition

Before data can be inserted into a table, a developer must first consider the data it will contain, how it will be organized and the nature of that data – often times without having been given more than a small sample of the data in advance. She must decide on the number of columns and recognizable names for each column. More importantly, she must decide on the the data type (e.g., `INT`, `CHAR`, `DATETIME`) appropriate to the type of data anticipated. Unless she chooses to accept the default settings for each column, she may want to set some restrictions on the data accepted (e.g., `NOT NULL`). Plus, she may want to take a guess at which column should be indexed without any usage information, and to what extent an index should be used, as in the case of multi-column indexes.

The definition of a table with all of its column settings is called a schema. Schemas can be modified in most relational database systems after the initial definition, but in some systems, especially with clusters, it can be an expensive operation because the data needs to be re-organized on schema changes. Although a skilled developer can anticipate potential table usage and the needs of its users, designing a schema can be much like a written one: making changes later to the organization of the data can be cumbersome – even when you used a pencil instead of a pen.

Insertion and Identification of Records

When inserting records into a table, the value of all of the fields can be provided in the same order of the table's columns based on its schema, or a subset of the columns can be given with the name and values of each field given. Fields not included will be set generally to a `NULL` value, which expresses the absence of a proper value – unless a default value was specified in the schema. If column definition has no default value and it was defined as `NOT NULL`, an error will occur when insert a row without giving a value for the column. Depending on the SQL mode used, this will prevent the row from being inserted. If it's part of a multiple row insert, it will prevent all rows from being inserted.

Each row in a table can usually be uniquely identified by an automatically incremented number. The column for this identifier is defined as the primary key. It's acceptable to use a different data type, or forego a primary key altogether and identify rows by other means, even by a combination of values in certain columns. However, this is not a common practice. If a primary key is referenced by another table, it is called a foreign key.

Indexing

The primary key index for fast access of records is usually a BTREE, the typical index type used in RDBMS. There are some idiosyncrasies related to this index type. The primary key must be unique. The user can create secondary indexes using one or more fields, with settings like unique or fulltext indexes. Data queries executed on a huge dataset, without using indexes can be significantly slower. Therefore, users are dependent on the database designer to define well and properly indexes for the best performance based on the data content and how it will be used.

Transactions

A key strength of RDBMS is that they support transactional guarantees. In SQL queries, transactions are started with explicit `BEGIN` or `START TRANSACTION` command. Following a series of data retrieval or modification operations, an SQL transaction is completed with a `COMMIT` command, or rolled back with a `ROLLBACK` command or if the session is terminated. There may be client communications with the server between the start and the commitment or rollback of an SQL transaction.

2.2 Concepts in ArangoDB

At its core ArangoDB is a transactional document store, in the sense of JSON objects as documents.

Internally, data is stored in a binary form (VelocyPack), but what is entered and returned from the system is typically JSON. By default, JSON data is sent and received over the well-known HTTP protocol by the server, which provides a RESTful API. Other options are VelocyPack over HTTP and VelocyStream, a binary transport protocol.

How data is actually persisted is not visible to the user. A developer can choose between two storage engines with different characteristics according to the particular use case:

MMFiles (memory-mapped files)	**RocksDB** (since v3.2)
default	optional
dataset needs to fit into memory	work with as much data as fits on disk
indexes in memory	hot set in memory, data and indexes on disk
slow restart due to index rebuilding	fast startup (no rebuilding of indexes)
volatile collections (only in memory, optional)	collection data always persisted
collection level locking (writes block reads)	Document level locks, concurrent reads and writes

JSON Data Types

JSON supports a few primitive and compound data types that can be combined to complex data structures. The primitive types are: null, boolean (true, false), number (integer and floating point numbers), and string. These are fairly straightforward and the same in any database system.

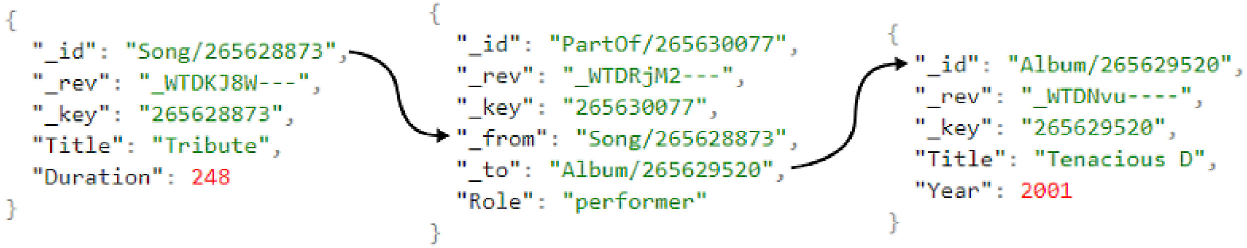
The compound types are more complicated. They are two types: array and object. An array is an ordered collection of elements, each identified by an index starting at 0; arrays can be multidimensional. An object is a hashmap which maps string keys (i.e., attribute keys) to values of arbitrary types (i.e., attribute values). Objects can contain primitive types or nested compound types, with support for arbitrary deep nesting, if desired. Each document is essentially a JSON object at the top level, with arbitrary named attributes that can have primitive values or be nested arrays and objects.

Documents are stored in collections. Collections have names, which ideally describe what kind of information the documents contain. Collections can be organized in multiple databases (e.g., for multi-tenancy use). The default database in ArangoDB is called `_system`. It contains hidden collections for internal purposes, but the user may also add collections to the database.

Living without a Schema

ArangoDB is schema-free. The system does not require you to declare or define field attributes. Nor is it necessary to specify in advance the data types of fields. In case a schema validation is necessary it can be integrated by a JOI validation in a ArangoDB Foxx microservice.

Documents can have an arbitrary structure and can use any supported data type. In principle, each document in a collection can be structured differently. However, there will usually be some fields that all documents have in common in the sense of attribute keys. The attribute values may use different data types, nonetheless, even if the attribute keys match.



```
{
  "_id": "Song/265628873",
  "_rev": "_WTDKJ8W---",
  "_key": "265628873",
  "Title": "Tribute",
  "Duration": 248
}
```

```
{
  "_id": "PartOf/265630077",
  "_rev": "_WTDRjM2---",
  "_key": "265630077",
  "_from": "Song/265628873",
  "_to": "Album/265629520",
  "Role": "performer"
}
```

```
{
  "_id": "Album/265629520",
  "_rev": "_WTDNvu----",
  "_key": "265629520",
  "Title": "Tenacious D",
  "Year": 2001
}
```

There is no such thing as filling in a subset of fields in a schema-free system, because for every document, what shall be persisted is defined by the document itself. It is self-contained. A document can have as few or as many attributes including nested attributes as needed. Both, attribute keys and values, have to be explicitly specified all the time.

Document Keys and Indexes

Each document requires a key that uniquely identifies it within a collection. It can be assigned by the user upon creation, or ArangoDB will generate one. The data type is always string. The automatically generated[1] keys are increasing numbers (with gaps), but converted to a non-numeric character sequence. Document keys cannot be modified after document creation (immutable). There is a virtual attribute (i.e., the document ID) which is comprised of the collection name, a forward slash and the document key to identify a document within the database. Document keys and document identifiers are always indexed. The index on the `_key` attribute is called a primary index. It exists for each collection and can't be removed. It's a hash index with the options non-sparse[2] and unique. This means it is always non-empty and that there are no duplicate keys in the collection.

Additional indexes can be defined by the user over one or multiple fields, as well as the individual elements of arrays. Available index types:

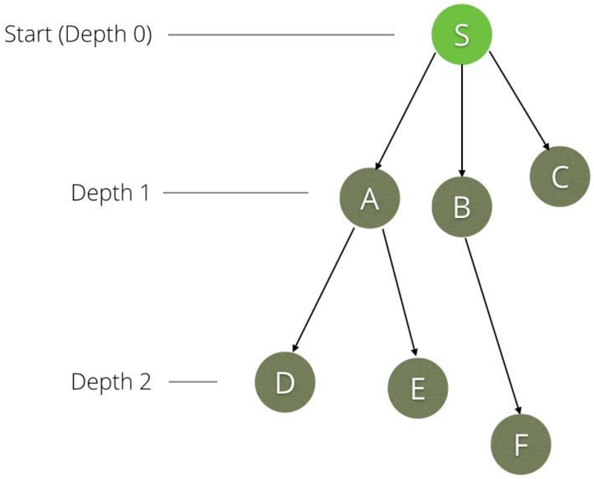
- **Hash** - constant lookup time (super fast), only applicable if an exact value is searched
- **Skiplist** - sorted index, slower than hash index, but suitable for range queries
- **Geospatial** - can index 2D coordinates for fast retrieval of documents near a reference point (i.e., closest distance first), full GeoJSON support will be added in ArangoDB 3.4
- **Fulltext** - full word and prefix matching in strings with a reverse lookup index (complete text search and ranking engine including inverted indexes will be integrated in ArangoDB 3.4)
- **Persistent**: With MMfiles storage engine, persistent skiplist indexes can be created via this type. With RocksDB, all indexes are persisted.

ArangoDB can be used as a key-value store by retrieving documents via their document keys or IDs. The primary index is hash-based with a constant lookup time and has a selectivity of 100% because each key must be unique in a collection, which enables quick retrieval via the key. The document as a whole is returned as a value in the key-value access pattern.

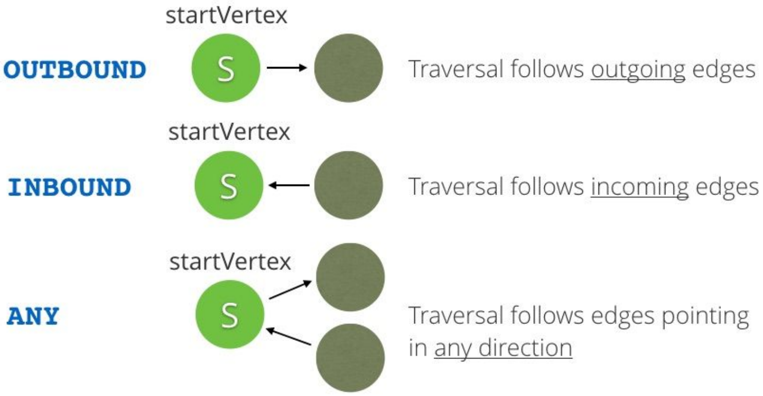
If secondary indexes are involved, or a document is returned partially (i.e., only certain attributes), it is not used strictly speaking as key-value store. Instead, it's used like a document store. There is no real distinction, however, between them in ArangoDB.

Graphs

The third data model supported by ArangoDB is graphs. Graphs are like networks consisting of nodes that are linked together, which can express many-to-many relationships. There can be various topologies, like a graph forming a tree (e.g., corporate hierarchy, electricity grid) rather than a mesh (e.g., social network, fraud networks).



There are two types of collections in ArangoDB: *vertex* and *edge* collections. Both store documents as previously described. Documents in an edge collection have two additional attributes, `_from` and `_to`. Both have to be assigned document IDs to link together documents. The document that links them is called an edge and the linked documents are called vertices in the graph model. Edges are always directed in ArangoDB. The edge is leading from the source (i.e., `_from`) to the target (i.e., `_to`). Edges do have a direction in ArangoDB but can be ignored by the `ANY` statement.


OUTBOUND
startVertex
S
Traversal follows outgoing edges
INBOUND
startVertex
S
Traversal follows incoming edges
ANY
startVertex
S
Traversal follows edges pointing in any direction

Additionally, there is a special edge index that can receive all of the edges of a vertex in constant time. Since receiving edges is the most used operation in any graph query, this constant time lookup is a requirement for graph database to operate performantly. This index cannot be removed. Users may define additional indexes for edge collections in general, and indexes over the `_from` and `_to` fields in particular, to create vertex-centric indexes to speed up certain graph queries.

ArangoDB can also efficiently handle other types of data, such as geo-spatial and text. With ArangoDB 3.4, the data types text and geo-spatial will receive a much richer featureset and new indexes to extend the number of possible use cases. These data types are intentionally not called data models, as the processing of the two is more a matter of computation than storage.

Transactions

In ArangoDB, a transaction is always a server-side operation. It is executed on the server in one go, without any client interaction. All operations to be executed within a transaction need to be known by the server when the transaction is started.

There are no individual `BEGIN`, `COMMIT` or `ROLLBACK` statements in ArangoDB's query language AQL. Instead, a transaction in ArangoDB is started by providing a description of the transaction written as JavaScript function. This function will then automatically start a transaction, execute all required retrieval and modification operations, and automatically commit transactions at the end. If an error occurs during the execution of a transaction, the transaction is automatically aborted, and all changes are rolled back.

Transactions in ArangoDB can be multi-document and multi-collection transactions in a single instance. For cluster scenarios, ArangoDB already supports atomic operations. In every document, ArangoDB stores automatically the so called revision key, `_rev` in all documents. For a single instance, this is being used to support full Multi Version Concurrency Control (MVCC, RocksDB storage engine) and in a cluster setting to support atomic operations like Compare-and-Swap. ArangoDB will soon support more options for transactional guarantees in cluster settings on our path to cluster-wide MVCC.

2.3 Comparison of Terminology

Having examined and compared the key concepts and methods of both RDBMS and ArangoDB, let's consider and compare the terminology. There are some terms used in an RDBMS that are identical to its counterpart in ArangoDB, although viewed a little differently. There are others, though, for which a different term is used to reflect more clearly the different perspective of ArangoDB. Below is a list of these key terms for both systems:

RDBMS	ArangoDB
database	database
table	collection
row / record	document
column	attribute key / attribute name
cell / field	attribute (value) / field
table joins	collection joins
primary key *(column(s) specified by user)*	primary key *(automatically present on _key attribute)*
index	index

In an RDBMS, entities of the same type are stored in tables. Collections are roughly the equivalent of tables in document stores, as they usually hold documents of similar entities. There normally will be some or many common attributes. Because of this varying nature of data structures, *collection* as is a more accurate term than *table*, which has a fixed structure. Documents are also not tabular, because unneeded attributes (i.e., fields) can be omitted and it's possible to nest data, which allows for a tree-like structure.

The document structure can differ between any two documents in a single collection. For example, as new features are added to a product that uses ArangoDB as a backend, additional attributes may become necessary. They can be added progressively on-the-fly for newly created documents, without the need to update all previously existing documents. In SQL, the table structure (i.e., schema) would need to be updated with additional columns. This is can be a significant operation in some systems and requires planning.

While there are a few primitive data types in ArangoDB, the system does not impose any restrictions such as having to define what data types must be used for which attributes — it is schema-free. If you wish to validate document structures on the server-side, ArangoDB provides joi object schema validation via Foxx.

There are many more primitive data types in relational database systems. In ArangoDB, there is only a single primitive type for numbers, which covers integers as well as floating point numbers. There's also only one data type for character sequences. There are no limits to the length of strings and strings are stored in normalized UTF-8 encoding. There aren't data types such as enums or sets, which require the defining acceptable values. Sets can be emulated by using a function to restrict an array to unique elements. A missing or unavailable value in an RDBMS is be represented by `NULL`. ArangoDB also has a null value, but it's rarely needed. The absence of a value can simply be expressed by the lack of the attribute.

The way ArangoDB implements graphs is different from many other graph databases, which use “index-free adjacency”. Edges stored in a separate collection with a special index on the `_from` and `_to` attributes are not too different from cross tables for storing many-to-many relationships in an RDBMS. The performance guarantees for traversals are, nonetheless, on par with other graph databases, but there is more potential to scale graphs. Combined with the Enterprise Edition features (i.e., SmartGraphs and SatelliteCollections), graph performance can be further improved.

Exkursion – The Foxx Framework for data-centric Microservices

Foxx is a JavaScript framework for writing data-centric HTTP microservices that run within ArangoDB. Foxx is based on Google's V8 engine, which is used in applications such as. Google Chrome. By directly integrating Foxx into the ArangoDB core, the framework has full access to all core functions on a C++ level. ArangoDB Foxx allows application developers to write their data access and domain logic as microservices, and operate directly within a database with native access to in-memory data. Foxx can handle anything from optimized REST endpoints performing complex data access, to standalone applications running directly inside the database. It might sound strange to let business logic run in the database, but moving data-intensive parts of the business logic closer to the data has two key advantages: improved performance by reducing network overhead; and better data security by isolating sensitive information and ensuring that it never has to leave the database. More details about the Foxx Framework can be found on our Foxx page.

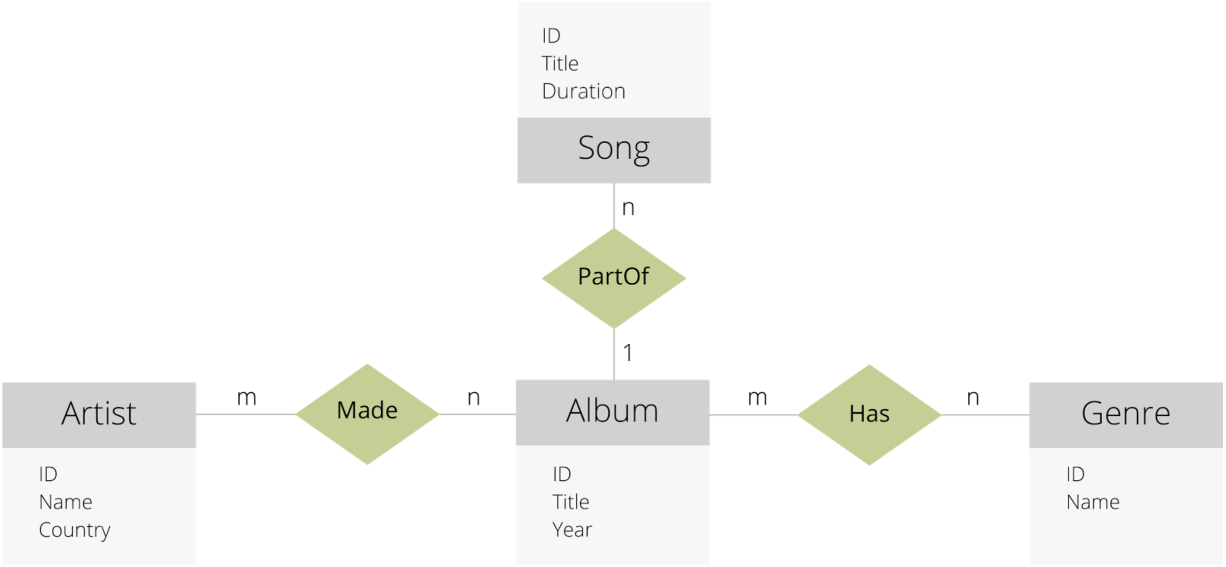
3. Data Modeling

Modeling data for a database system starts with defining what data needs to be stored to solve the problem. This usually requires being clear about the questions which should be answered, what entities there are needed, and what are their relationships to create a logical model. This model can be iterated upon until it addresses the entire problem. The model needs to be translated to a design that the chosen database systems allows and ideally supports in an easy to use and performant way.

3.1 Data Modeling in Relational Systems

The first step in creating a logical model is to determine the entities. For example, consider a simple database about music. Suppose we chose to store songs and albums by artist together with genres.

The next step is to define the properties of these entities and their relationships. An entity-relationship-model (ERM) is the standard way of representing all of this.

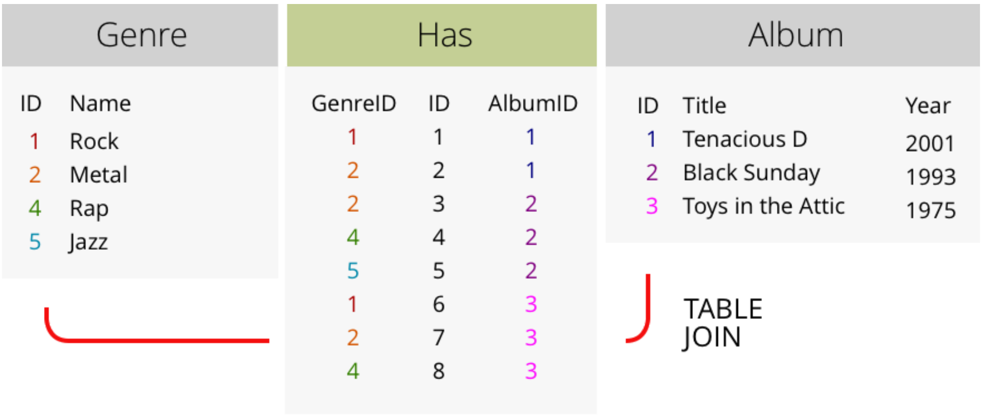


It's easy to cast this into a relational database model. For every type of entity, one table is defined with the entity properties translated to the columns.

Relationships between records can be expressed with matching values between tables. They can be one-to-one relations (i.e, 1:1), in which each record in table A matches one record in table B. That is to say, a field in A and another in B have the same value — usually by way of a primary key. Not every records needs a counterpart; there may also be one-to-zero relations, but there can't be more than one record on either side. It's common to integrate the fields of table B into table A in such a situation, unless there are special considerations to hold the data in different tables (e.g., access permission).

In case of one-to-many relations (i.e., 1:n), each record in table A matches with one or more records in table B. Applying this to our example, albums and their respective songs would be stored separately in this model. Each song record can only be linked to one album in this simple model. In a reality, one would probably allow the same songs to be linked to multiple releases.

The relationship between albums and genres is different: a many-to-many relation (i.e., n:m). A single album can have multiple genres, and multiple albums can have the same genre. Albums and genres can be stored in two tables, and a third table known as a cross table be introduced to link the records of the two former tables. A record in the cross table stores its own primary key, and one album ID and one genre ID, a foreign key.



Genre

ID	Name
1	Rock
2	Metal
4	Rap
5	Jazz

Has

GenreID	ID	AlbumID
1	1	1
2	2	1
2	3	2
4	4	2
5	5	2
1	6	3
2	7	3
4	8	3

Album

ID	Title	Year
1	Tenacious D	2001
2	Black Sunday	1993
3	Toys in the Attic	1975

To make use of stored relations, to resolve foreign keys to fields from another table, the tables — or rather subsets of their records — are joined on the fields which hold the record keys. These fields should be indexed, as there can be a high computational complexity if all of the combinations of records need to be checked. The result is often a database with many tables and many cross tables, always requiring joins to query the data, thus impacting performance.

3.2 Data Modeling in ArangoDB

As described in the Database Concepts section above, ArangoDB is a document store at its core. A JSON document containing multiple entities embedded in a single document could look like the example on the right.

Curly brackets surround JSON objects, which can contain attributes. Attributes are given in pairs, key/value pairs. Each key and value is separated by a colon. Attribute pairs are separated by commas.

Attribute keys are enclosed in double quote marks, which signifies a string. Keys are always string data types in JSON. They are depicted in teal blue here.

The values can be of any type. In the example here, name, country and other nested attribute values are also strings, colored in green. The album year and song duration values are numbers and highlighted in blue.

Square brackets surround value lists. In this example, albums, genres and songs are such arrays. Regarding genres, the arrays contain multiple strings with genre names.

The array for albums contains two nested objects, one per album. Each object has four attributes: Title, Year, Genres and Songs. The Songs array holds even more objects, one per track of each album. Each object has a title and a duration attribute. By the way, the dashed lines denote omissions, to fit the example document on this page.

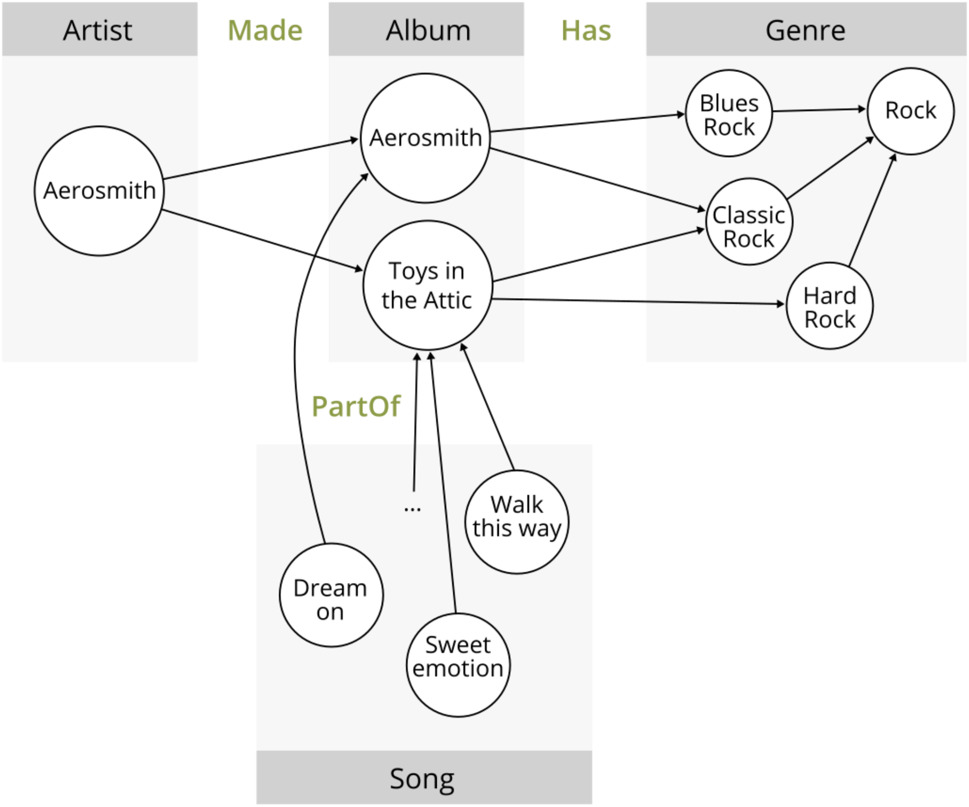
This is a contrived example with all of the data about an artist in a single document, to demonstrate information nesting.

```
{
  "_id": "Artist/265645002",
  "_rev": "_WTFii72---",
  "_key": "265645002",
  "Name": "Aerosmith",
  "Country": "USA",
  "Albums": [
    {
      "Title": "Aerosmith",
      "Year": 1973,
      "Genres": [
        "Rock",
        "Classic Rock",
        "Blues Rock"
      ],
      "Songs": [
        {
          "Title": "Make It",
          "Duration": 225
        },
- - - - - - - - - - - - - - - - - - - - - -
        {
          "Title": "Walkin' The Dog",
          "Duration": 192
        }
      ]
    },
    {
      "Title": "Toys in the Attic",
      "Year": 1975,
      "Genres": [
        "Rock",
        "Metal",
        "Rap"
      ],
      "Songs": [
        {
          "Title": "Toys In The Attic",
          "Duration": 185
        },
        {
          "Title": "Uncle Salty",
          "Duration": 248
        },
        {
          "Title": "Adam's Apple",
          "Duration": 274
        },
        {
          "Title": "Walk This Way",
          "Duration": 219
        },
- - - - - - - - - - - - - - - - - - - - - -
        {
          "Title": "You See Me Crying",
          "Duration": 312
        }
      ]
    }
  ]
}
```

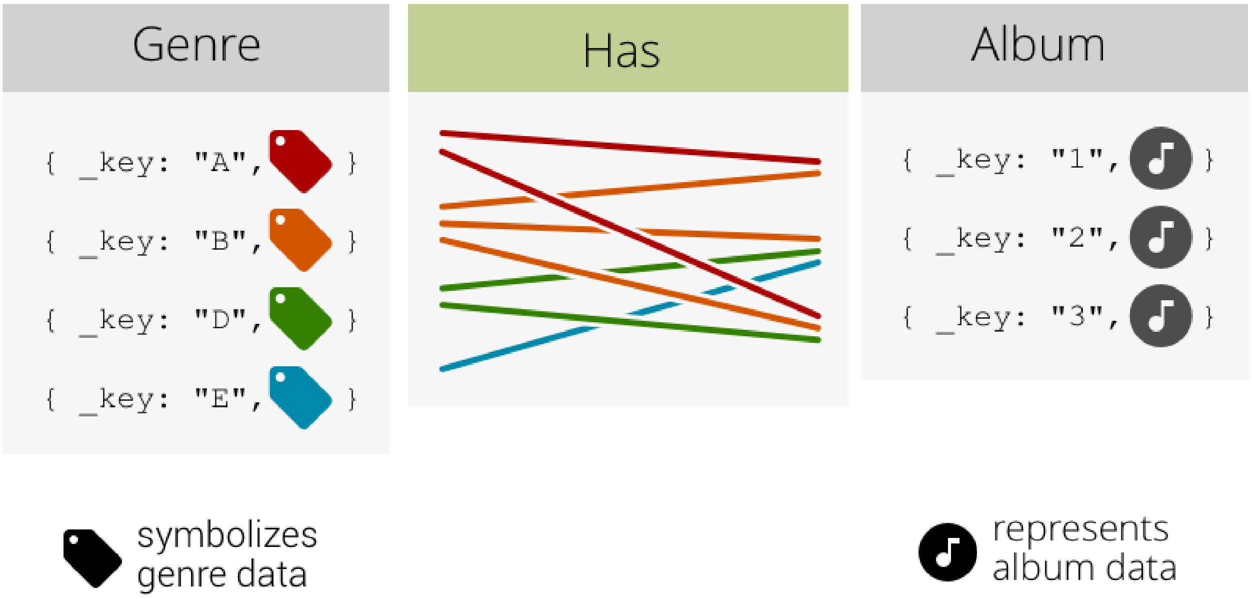
Using multiple levels of nesting can be very powerful, but it's not the only way to structure data. Based on the data model draft and ERM diagram from the beginning of this chapter, there are four entities in the normalized model: Artist; Album; Song; and Genre.

Each type of entity can be stored in a collection: all of the artist documents can be stored in an Artist collection; and album documents can be stored in an Album collection. Entities or documents can then be linked in multiple ways.

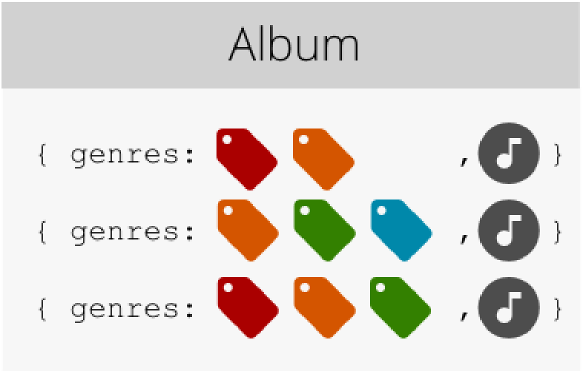
An additional collection could be created, to store an album document key and a song document key, similar to a cross table, to be joined in a query based on matching keys. It's a permitted method, but not recommended. Instead, cross tables can be translated to edge collections almost unedited. Instead of establishing a connection with a pair of foreign keys, the `_from` and `_to` attributes of an edge can be used to express the relation and open the possibility for graph traversals.



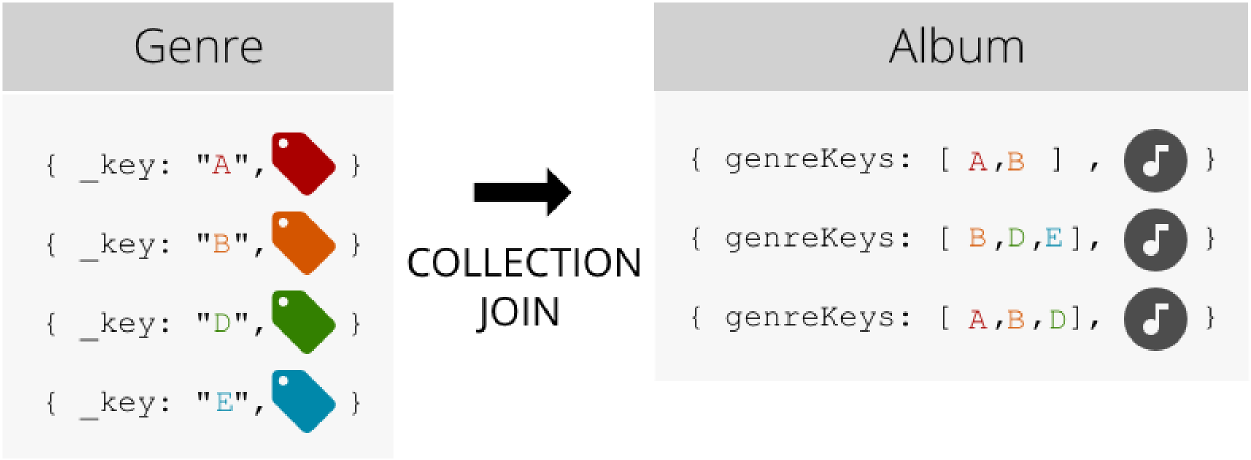
It's not necessary to use the graph model for everything. It may not be the ideal solution related to performance to do this for very limited, fixed depth traversals — especially a one step traversal as we would here for albums and the associated genres.



Embedding genre data into the album documents is a possible solution. Only a single document read would be needed to retrieve the album information, including the genre data. On the other hand, it becomes harder to maintain the data. For example, renaming a genre without introducing inconsistencies would be tedious. All album documents containing a to-be-renamed genre would need to be modified. Furthermore, there wouldn't be an efficient way to retrieve a list of all genres.



A viable middle way is to have two collections, one for albums and one for genres, and store an array of genre document keys in the album documents for many-to-many relations. This way there is a central place to retrieve all available genres and an individual genre. There would be only a single document to maintain all of the information related to a genre in the Genre collection. The link between albums and genres is established via the genreKeys attribute in each Album document and can be easily resolved in a query, without an additional collection. A lookup or collection join of genres leads to the same results as with the graph model, but performs better than a full graph traversal. If performance is less a concern, though, you may as well use the graph approach since it resembles more closely the way we think of relations.



Another consideration should be the number of links between documents. Chances are that at least some of the genres will be referenced by thousands or even millions of albums. Using a graph model would result in some genre vertices, having a huge amount of edges connected to it. Such vertices are called *super nodes*. They should be avoided. Traversals running into super nodes will have to follow all of these edges. The collection join model scales much better. The desired approach, though, can be chosen by the user, depending on the use case.

3.3 Comparison of Data Models

Data is modeled as tables for an RDBMS. It’s common practice to normalize an initial model so that one arrives at a model without redundancies with atomic values, in the sense that pieces of information aren’t being split into separate tables and columns. For example, you wouldn’t have a single field containing "Toys in the Attic (1975)", with both an album title and the release year. Instead, you would have two fields, one containing the title and the other the year. You would also have distinct tables for each entity type. The normalization process involves the addition of cross tables to store relations between records.

While there is a plethora of data types from which to choose before storing data in relational systems, there are only a few basic data types in ArangoDB that can be combined to structure data in many ways. They allow for simple and complex alike, independent for each document. Everything expressible with tables can be translated to JSON documents, without the need to specify in advance the structure of documents.

A direct conversion of relational records to documents would result in JSON documents with top-level attributes only (i.e., no nested objects). The possibility of having sub-attributes and arrays, however, enables quite a different data modeling approach. This approach permits the embedding of data in documents and eliminates some extra tables or collections. Translating cross tables to edge collections is a great way to store many-to-many relationships in ArangoDB, to gain graph features such as shortest path computation and graph traversal. Distributed graph processing based on the Pregel computing model is also available starting in version 3.2 of ArangoDB. If you don't have a need for real graph, a model with collection joins can be chosen instead. Multiple models can be combined freely. ArangoDB provides flexibility, allowing the developer to strike a balance between an understandable data model, fast development iterations, needed features and performance.

4. Query Concepts

Storing data is only part of the function of a database system. The ability to retrieve data in various ways from a database is where an electronic database is most useful. Otherwise, you may as well use a paper based system. To facilitate the retrieval of data, a query language is provided. Relational database systems like MySQL use a Structured Query Language (i.e., SQL), while ArangoDB provides Arango Query Language (i.e., AQL).

4.1 SQL

SQL can not only retrieve records from the database system, it can also insert, update and delete records. Tables and databases can be created, altered and dropped, and user permissions can be managed with SQL statements. These uses are grouped and classified based on their similarities: data retrieval queries as DQL (i.e., data query language); inserting, updating and deleting data as DML (i.e., data manipulation language); structure and schema statement as DDL (i.e., data definition language); and creating users and setting permissions as DCL (i.e., data control language). There are also procedural elements in the form of stored procedures for more complex logic.

SQL became a standard in 1986, with several revisions over the years up to the current SQL:2016 standard. Most RDBMS implement a subset of these standards or a variant with additional vendor-specific features with a dialect of the query language. Thus, other than the most basic SQL statements, SQL code that works in one system may not work in another system, without modifications because of syntax and feature differences.

SQL comes with several language constructs to interact with databases. Each construct or SQL statement, reads like an English sentence with a fixed order of clauses. Many clauses are optional. A clause is started by a keyword, followed by an expression. For example, to retrieve records from a database, a `SELECT` statement is used and specifies what to return:

```
SELECT col1, MAX(col2)
FROM table1
WHERE col2 > 20
GROUP BY col1
HAVING col1 > 100
ORDER BY col2
```

Each clause must appear in the designated place if it is used. For example, it would be a syntax error if the `WHERE` clause was entered before the `FROM` clause. The `WHERE` clause can be omitted to return unconditionally the desired fields of all records. `HAVING` allows more conditions to be added, but can only be used after `GROUP BY` as a post-filter. The

WHERE clause describes filter conditions applied before the aggregation in the combination of both. The ORDER BY clause is used to sort the matching records by one or more fields in ascending or descending order.

We did not include a LIMIT clause in this generic example to define an optional offset and a maximum number of records to return. The LIMIT clause is commonly used with MySQL, but it's actually non-standard. TSQL, used in Microsoft SQL, supports the syntax SELECT TOP 10, for instance. There are many more variations and some support for multiple syntaxes in various systems. The SQL:2008 standard provides for a FETCH clause (e.g., FETCH FIRST 10 ROWS ONLY), with varying support across systems.

The most common joining of tables in SQL is the inner join, which returns the intersection of two sets. A column of the first set is used to match the values of a table with a specified column of another table, the second set. For instance, if there's a record with a specific value in a particular column in one table, and a record with the same value in a corresponding column in another table, those records with the selected columns are returned.

```
SELECT emp_id, name, address, telephone
FROM employees
INNER JOIN emp_contact_info
ON employees.emp_id = emp_contact_info.emp_id;
```

Unless emp_id is defined so as to allow only unique values, the same employee identification number may occur multiple times in either table. This will produce additional results in SELECT statements. A JOIN is used to express the other table you want to join. An INNER JOIN says only to return records in which the values in the given columns are equal. It can also be expressed like this:

```
SELECT emp_id, name, address, telephone
FROM employees,  emp_contact_info
WHERE employees.emp_id = emp_contact_info.emp_id;
```

If you want all of the records from the `employees` table are supposed to be returned, regardless of a match in the `emp_contact_info` table, a left join can be used like so:

```
SELECT emp_id, name, address, telephone
FROM employees
LEFT JOIN emp_contact_info
ON employees.emp_id = emp_contact_info.emp_id;
```

Basically, the table being selected is the table on the left, and the one being joined the table on the right. A left join will return every row in the left table (i.e., `employees` here) that meet the conditions of the `WHERE` clause, if there were one, as well as the matching data in the right table. However, if there isn't a matching row in the right table, NULL values are displayed for fields related to it.

4.2 AQL

AQL is classified as DQL and DML, which means it can create, read, update and delete documents. It cannot be used to create collections and databases, manage permissions or define schemas. As such, AQL has some similarities to SQL, but also distinct differences, mainly because it's designed for combining multiple models and graph traversals in particular.

AQL was invented because when ArangoDB was created, there was no SQL standard which covered its multi-model needs. A standard does still not exist which covers all that is ArangoDB, in particular graphs. This is probably because NoSQL systems involve a wide range of data concepts and practical goals, with vastly different technological approaches. ArangoDB supports multiple data models with its single query language, AQL. It's a custom tailored fit. Because it's not led by a standard, it can improve quickly and freely, based on market and developer needs.

FOR Loop Construct

AQL has a language construct for looping through data, typical in programming languages more than database query languages. This allows developers to both query and program in one versatile language, rather than have to learn and use two. It's the central building block for document retrieval. The most common form is the `FOR` loop used with a collection of documents, much like `SELECT` in SQL. It can also be used to iterate through arrays, such as an array attribute, for intermediate results from a subquery or a variable defined inline.

A `FOR` loop can be nested in different ways. Joins can be implemented by processing two collections, usually combined with a condition such as the equality of an attribute between both collections.

A syntactical variant of the `FOR` loop with three emitted variables and some extra clauses is available to perform graph traversals. There is also a `SHORTEST_PATH` variant of the `FOR` loop to return a path from a start vertex to an end vertex with the minimum amount of hops (i.e., lowest traversal depth).

High-Level Operations

Other essential high-level operations are `FILTER`, `SORT` and `LIMIT`. They can be entered in different locations, usually with varying results based on the processing order described by the query. You can define what a query should return in a `RETURN` statement. One can chose which parts of a document to return, or the document as a whole, optionally with additional or altered attributes computed.

Data modification queries (i.e., `INSERT`, `UPDATE`, `REPLACE`, `DELETE`) don't require a `RETURN` statement. It's mandatory only in data access queries. It can be used nonetheless in modification queries, such as for returning old or new state of documents. Modification queries come in a few syntax variants, but the predominant forms are like the following:

```
INSERT { _key: "123", Title: "Walk This Way" } INTO Song
UPDATE "123" WITH { Duration: 219} IN Song
REPLACE "123" WITH { Title: "Uncle Salty", Duration: 248 } IN Song
REMOVE "123" IN Song
```

The difference between `UPDATE` and `REPLACE` is that `UPDATE` allows for partial modifications, such as changing existing attributes or adding new ones, whereas `REPLACE` retains only the document key, and switches the rest of the document with the provided new content.

Query Optimization in AQL

In order to execute the above described queries, ArangoDB uses a two-phase query optimization technique. In the first phase, the query is parsed and transformed into an internal representation, an abstract syntax tree (i.e., AST). Then optimizer rules transform the AST without changing the result. For instance, the optimizer will try to use indexes wherever possible.

There are often several possible ways to execute the same query. For example, in a join it could start the search in any of the involved collections. The optimizer estimates what's required to solve the query for each way possible, and will execute the one that requires the lowest amount of operations. If you are curious to see what ArangoDB creates from your query you can check the explain output. This tool can help to improve query performance.

Combine All of the Models

Now let's see where we can make use of the multi-model approach and combine a graph search with joins. Let's assume that we have a very large dataset of songs from several years and several genres. In this case, some genres will most likely be super-nodes, vertices with many connected edges (e.g., "Pop"). It's always a bad idea related to performance to iterate through them within a query.

Consider the follower user request: "I just listened to a song called, *Tribute* and I liked it very much. I suspect that there may be other songs of the same genre as this song that I might enjoy. So, I want to find all of the albums of the same genre that were released in the same year".

Let's break this into logical steps, utilizing the names we have assigned to components of our database. We'll do this in the pure graph way, in this order:

1. Start with Song, 'Tribute'
2. One step (PartOf) to find the Album
3. One step (Has) to find the Genre
4. One step back (Has) to find all otherAlbum's of this Genre
5. Year of otherAlbum is identical to year of Album

Written in AQL, the query would look like this:

```
FOR s IN Song FILTER s.Title == "Tribute"
  // We want to find a Song called Tribute
  FOR album IN 1 INBOUND s PartOf
    // Now we have the Album this Song is released on
    FOR genre IN 1 OUTBOUND album Has
      // Now we have the genre of this Album
      FOR otherAlbum IN 1 INBOUND genre Has
        // All other Albums with this genre
        FILTER otherAlbum.year == album.year
          // Only keep those where the year is identical
          RETURN otherAlbum
```

This query is straightforward and will find the solution. However, it has the drawback that is most-likely traversing over a super-node in Genre. Also the list of genres is unlikely to grow rapidly. Although we may add many new songs, we seldomly add new genres. Therefore it would be better first to select all Albums of the same year and then validate that the genre is identical. This way we get a limited set of Albums, and each has only one genre. That query would be resilient to data growth.

So let us modify our query to the following:

1. Start with Song, 'Tribute'
2. One step (PartOf) to find the Album
3. One Step (Has) to find the Genre
4. Join all Albums with identical Year
5. For each otherAlbum, one step to find its genre
6. Filter all otherAlbum where the genre is different

These steps can be written in AQL like this:

```
FOR s IN Song
  FILTER s.Title == "Tribute"
  // We want to find a Song called Tribute
  FOR album IN 1 INBOUND s PartOf
    // Now we have the Album this Song is released on
    FOR genre IN 1 OUTBOUND album Has
      // Get the genres of this Album
      FOR otherAlbum IN Album
        // Now we want all other Albums of the same year
        FILTER otherAlbum.Year == album.Year
        // So here we join album with album based on identical year
        FOR otherGenre IN 1 OUTBOUND otherAlbum Has
          FILTER otherGenre == genre
          // Validate that the other album's genre is identical
          // to the genre of the original album
          RETURN otherAlbum
          // Finally return all albums of the same year
          // with the same genre
```

4.3 Query Language Comparison

At the core, SQL and AQL are both declarative languages, with functions that can be called for additional operations. The key difference between SQL and AQL is that in SQL you describe the result you want, and in AQL you describe the process to get there. The concept behind AQL is closer to coding, and therefore easier than switching between a coding language and querying language.

SQL syntax has a fixed structure, whereas high level operations in AQL, such as filters, can be put in various places for different purposes and results. The flow is more logical than adhering to SQL statement order. Plus, high-level operations like `FILTER` and `SORT` are more versatile. `FILTER` equals `WHERE` and `HAVING` in SQL, depending on where it appears. `SORT` is a direct equivalent of `ORDER BY` in SQL. `LIMIT` is the same as in MySQL, although there are some things to consider regarding scope in conjunction with subqueries. `RETURN` doesn't exist in SQL, but you can specify which fields to return in `SELECT`. In SQL what is to be returned is given early in the statement, whereas in AQL, `RETURN` is placed at the end of a query and of subqueries.

Aggregation with `COLLECT` isn't too different from `GROUP BY` in SQL, but the existence of compound types in AQL makes it very different. While in AQL there is a direct alternative to the construct of a `GROUP BY` clause in SQL, with AQL there are many more twists and ways to aggregate data. For example, all documents that fall into a group based on the grouping criteria can be kept and further processed. With the `COLLECT AGGREGATE` clause, you can efficiently compute statistical figures while the data is being grouped. Grouping is fast even without an index, using a hash-based approach.

`FOR` loops in all their variants are a key concept of AQL. They makes AQL feel more like a programming language than a query language. It's a major building block and adds a sense of flow to queries.

Joins, which are heavily used in most applications based on an RDBMS, are also possible in AQL. Joins are expressed with nested `FOR` loops combined with `FILTER`, instead of a separate syntax for each JOIN. It's quite similar to the syntax for inner joins in SQL, but without a `JOIN` statement. Multiple types of joins can be carried out with AQL, although it may not be necessary to do so. An array of document keys together with the `DOCUMENT()` function is a viable alternative to resolve one-to-many relationships.

Recursion is not possible in SQL, except with a stored procedure — which is basically procedural programming and not strictly part of the SQL standards; they are largely vendor-specific. Using nested `FOR` loops in AQL allows for the traversing of multiple levels of attributes within a document. The graph traversal variant of the `FOR` loops allows a query to traverse a graph with a definable minimum and maximum depth with a few simple lines of code. This same task would be extremely complex in a relational database system with a stored procedure.

5. Cluster and Scalability

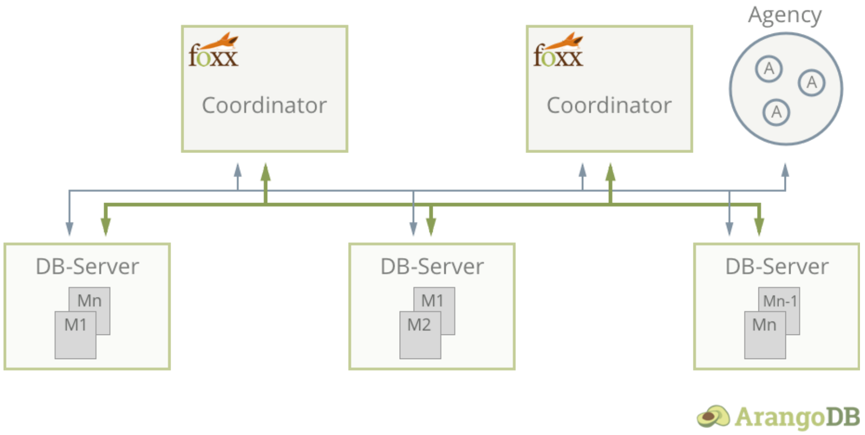
Vertical and horizontal scalability is a recurring topic and increasing in importance. This paper does not deal with the tiring discussion that relational databases can't scale horizontally. Teams at Facebook, Uber and LinkedIn already proved that claim is incorrect. Discussing the strategies for relational databases and ArangoDB would is beyond the scope of this paper. Therefore, the paper instead introduces the options available with ArangoDB and especially how the database handles complex queries at scale, including joins and graph traversals.

5.1 Cluster Architectures with ArangoDB

When ArangoDB is started in cluster mode, each instance can play different and multiple roles. Below is a list of the roles permitted and an explanation of each:

- **Coordinators** are stateless and responsible for query handling, query processing, result set build-up, and the location of Foxx microservices.
- **DB Servers** are stateful and where the data is actually stored.
- **Agency** is a highly available and resilient key/value store kept on an odd number of ArangoDB instances, running the Raft Consensus Protocol.
- **Agents** hold the current cluster configuration in the Agency.

Each role can be scaled independently. They can serve, for example, higher query processing or storage needs.



Following the CAP theorem for distributed data stores, clusters in ArangoDB use the CP master/master model with no single point of failure. When a cluster encounters a network partition, ArangoDB prefers to maintain its internal consistency over availability. If no problem occurs, then ArangoDB serves all parts of the CAP theorem, of course.

Clients experience the same view of the database regardless of the node to which they connect. And, the cluster continues to serve requests even when one machine fails.

ArangoDB provides three different data models and related access patterns, like joins or graph traversals. The specific data models and especially their query techniques pose individual characteristics in terms of scalability. In the following section, these characteristics and their challenges, with related query techniques, are discussed.

5.2 Data Models and Horizontal Scalability

ArangoDB can scale vertically, but also horizontally to a cluster with all data models already in the Community Edition (open-source under Apache 2 license). Because of the very nature of the data models and their access patterns, the models differ by the simplicity to scale and by the extend.

Scaling horizontally with simple key/value pairs is the easiest method. Absent secondary indexes, the collections always behave as a simple key/value store. The only sensible operations in this context are single key lookups and key/value pair insertions and updates. If the key attribute is the only sharding attribute, sharding is done with the primary key and all operations will scale linearly.

Using the document model is similar to key/value usage, since the index for sharded collections is the same as the local index for each shard. Each shard holds only the parts of the index that it needs.

AQL allows for complex queries using multiple collections, secondary indexes as well as joins. Data necessary for the join operation can reside on different machines. This can lead to performance drains due to network hops during join operations. With the ArangoDB Enterprise feature called Satellite Collections, these join operations can achieve performance similar to a single instance in some situations. You can read more about it in the following section.

As discussed earlier, a graph dataset consists of nodes connected by edges. Graph traversals are especially good for searches of unknown or varying depths. But because of this high connectivity and depth of search, the graph data model is the most difficult one to scale to a cluster. When the vertices and edges along the path are distributed across a cluster, the query requires more communication between the servers and performance is reduced, compared to a path which is limited to a single server. With SmartGraphs, ArangoDB provides a feature to reduce to a minimum the network hops during a graph traversal, achieving near the same traversal performance as it would with a single instance.

How SmartGraphs and Satellite Collections work and why we developed these features, is described briefly in the following section.

5.3 Performing Complex Queries at Scale

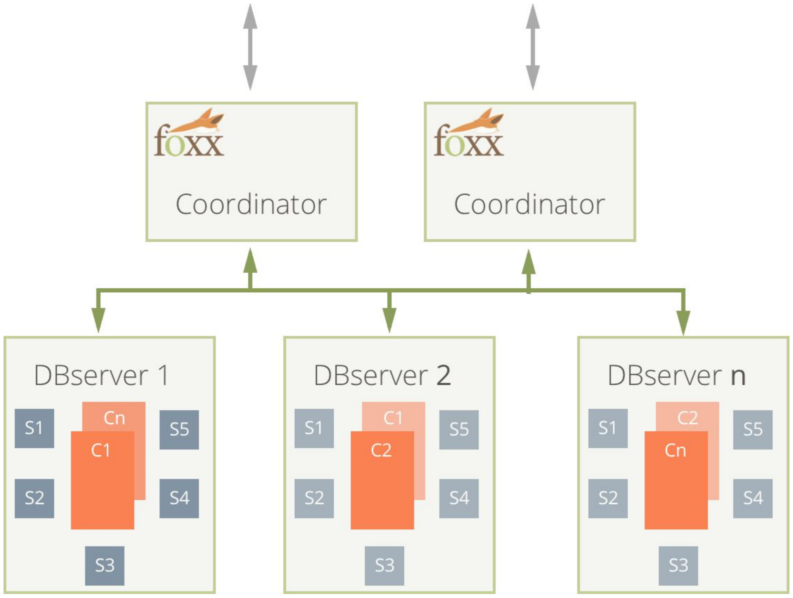
With the possibility of combining different access patterns like joins and traversals with sorting, aggregations, filtering and many other functions in a single query, AQL can express a large range and great depth of queries on the data. This capability is also possible, when data is sharded to a cluster.

For high performance needs on large datasets, ArangoDB provides currently two features within the Enterprise Edition: Satelite Collections for joins at scale; and SmartGraphs for handling an extraordinary number of edges.

Satellite Collections — Joins At Scale

The goal of Satellite Collections is to enable local join operations for large, sharded collections in a cluster setup.

To understand this better, imagine an IoT use case with plenty of sensor data from a variety of devices. Imagine further that the set of individual events is too large for a single server, but the number of devices is small enough to be stored in a collection which can be replicated on each machine. These smaller collections are called satellites, indicated with the S prefix in the image below. They are replicated to each database server (i.e., DBserver) and orbit all shards of the large events collection, indicated with the C prefix.

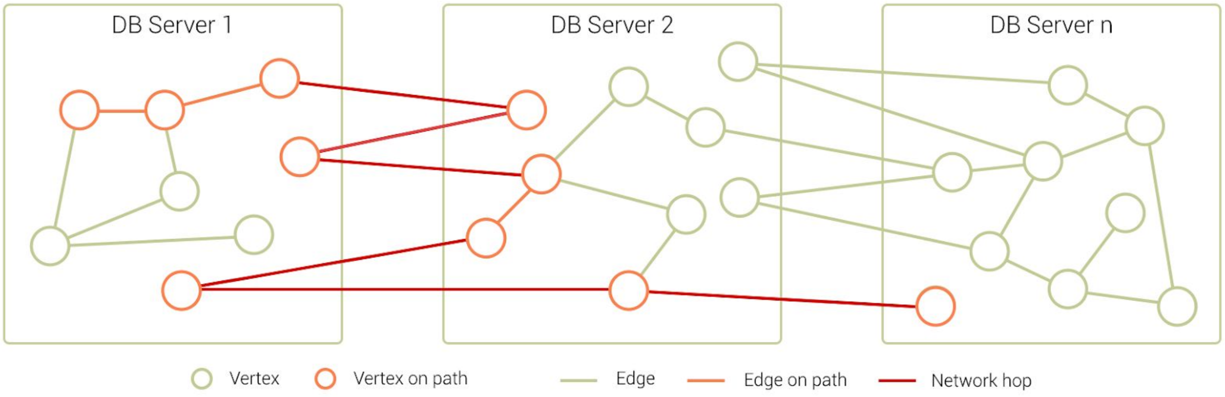


As for join operation between a collection and a satellite, the query optimizer of ArangoDB knows where each shard participating in the join operation is located and sends the request to the right machine where the query will be processed locally on the DBserver. In this way, network hops can be reduced to a minimum. This is because the DBserver sends only the result back to the Coordinator where the final result set will be assembled.

More information with query examples and load calculations can be found on our Satellite Collection page.

SmartGraphs — Handling Billions of Edges

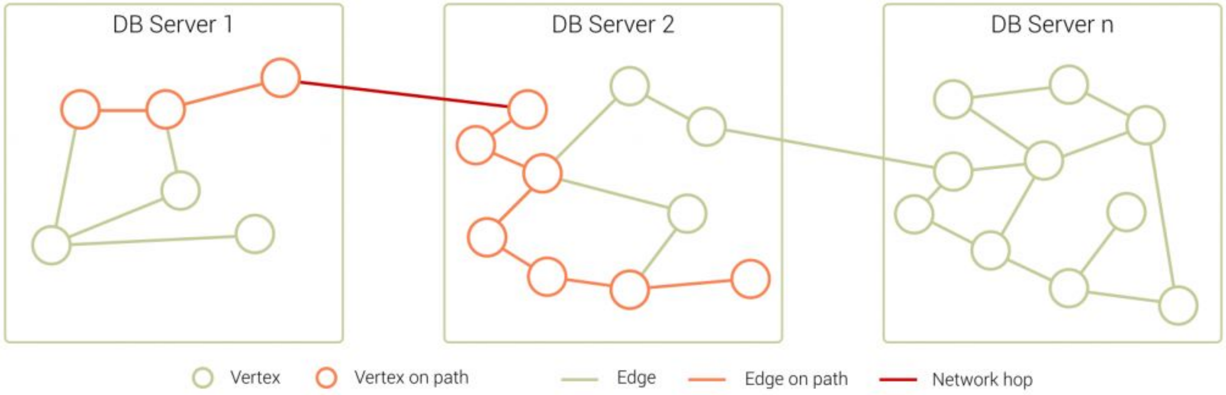
When the dataset for a graph exceeds the limits of what you can be hosted in a single instance of ArangoDB, you need to scale out. However, sharding a graph to multiple machines introduces new concerns. During a traversal, many network hops between database servers can occur. In the image below, you can see a schema of a graph dataset distributed on a few machines. As edges carry the traversal onto other machines, performance takes a hit caused by network latency.



Even with a good network, the lookup of edges and vertices is much more expensive compared to in-memory computing. The solution is utilize domain knowledge of your graph, which will allow for the use of ArangoDB SmartGraphs.

A common characteristic within large graph datasets is that they either form communities (i.e., areas of very high connectivity), or they can be easily split into subgraphs (e.g., a hierarchy tree). Both characteristics can be exploited to use the ArangoDB SmartGraph feature and shard data efficiently to each machine.

Once this sharding is done, graph traversals using SmartGraphs can be performed more efficiently because vertices and the connecting edges reside on the same machine and traversals can be executed locally with a minimum number of network hops. The same query path as shown in the image above will include much fewer network hops and best case none. In the “smartified” traversal shown below we could reduce the network hops to one.



SmartGraphs and Satellite Collections use similar optimizations to reduce network hops during query processing. The performance gains by SmartGraphs depend on the individual use case and the structure of a graph dataset.

There is a huge set of options to use and optimize both features. We are happy to support projects in which SmartGraphs or Satellite Collections could be an option. For more information, please contact us via learn@arangodb.com.

6. Conclusion

Storage technology has come a long way and the decrease of storage pricing has allowed for better optimized database architectures. People store more data than ever before, but this isn’t the only challenge. We believe that the versatility of data and efficient access to it is the main challenge that many companies are facing — and many more companies in the near future. Companies need to know the context of their data, and therefore may have to combine multiple data models to gain insights from that data. The “Customer360” buzzword is just one example.

A few years ago, investing in learning, mastering and maintaining multiple storage technologies was the only way to cope with the growing versatility. Today, native multi-model has matured enough to be the much better choice. The flexibility to use ArangoDB as a specialized document or graph store, the ease to switch between the models by just rewriting a query, and the possibility to combine models to harness the computational advantages of joins and graph traversals in the same query, is just what modern software development needs.

We hope we have demonstrated in this white paper how easy the transition from relational databases to ArangoDB is. It's not a drop-in replacement, but joins enable the direct transition of your data model to ArangoDB. No nesting necessary in ArangoDB but possible if needed. In combination with ArangoDBs schemaless nature, you can save plenty of time on development, especially in new and evolving projects and many options to optimize as you go. If you prefer, though, you can rely on schema validation via Foxx.

SQL and AQL are both declarative languages and capable of expressing a very broad spectrum of queries. We think that the `FOR` loop concept of writing queries gives AQL the feel of coding, which makes it economical, while retaining the advantages of a declarative query language.

With native multi-model, users have more freedom in decision making with regards to data modeling, data organization and available access patterns to their data — and at scale. We think the real power of native multi-model is, that it can multiply your capabilities.

Our hope is to assist you in improving your skills in this area, and to help you to stay on course as you grow into a new era of database management. You can start your ArangoDB journey today with one of these tutorials:

- First Steps with AQL
- Graph Course: From Zero Knowledge about ArangoDB to Complex Graph Queries
- Performance Course: Data Modeling and Query Tuning Options with ArangoDB

If you have any feedback on this white paper, please email us via learn@arangodb.com.