# What is a multi-model database and why use it?

An ArangoDB White Paper (Updated August 2018)

When it comes to choosing the right technology for a new project, ongoing development or a full system upgrade, it can often be challenging to define the exact right tools that will match set-up criteria from start to finish. Especially, when it comes to choosing the right database.  It has been actively discussed and debated by many experts that “one size does not always fit all”. This idea suggests that one would use different data models for different parts of large software architectures. Meaning that one has to use multiple databases in the same project, potentially resulting in some operational friction, data consistency and duplication issues. This is where a native multi-model database with all its benefits and flexibility comes into play.

In this white paper we explain what a multi-model database is, and why and where it makes sense to use it, including a use case based on aircraft fleet management.

# Table of Contents

# 1 What is a multi-model database?

As multi-model databases and a NoSQL approach is becoming more and more trendy, many vendors label themselves as “multi-model”. Therefore, making it challenging to find a definition of what a multi-model database should be. It also makes it challenging for those searching for a multi-model solution to understand what is what. It is important to keep in mind that there is a difference between an added e.g. graph layer on top of a key/value or a document database and a full-blown native multi-model solution.

So what is a native multi-model database?

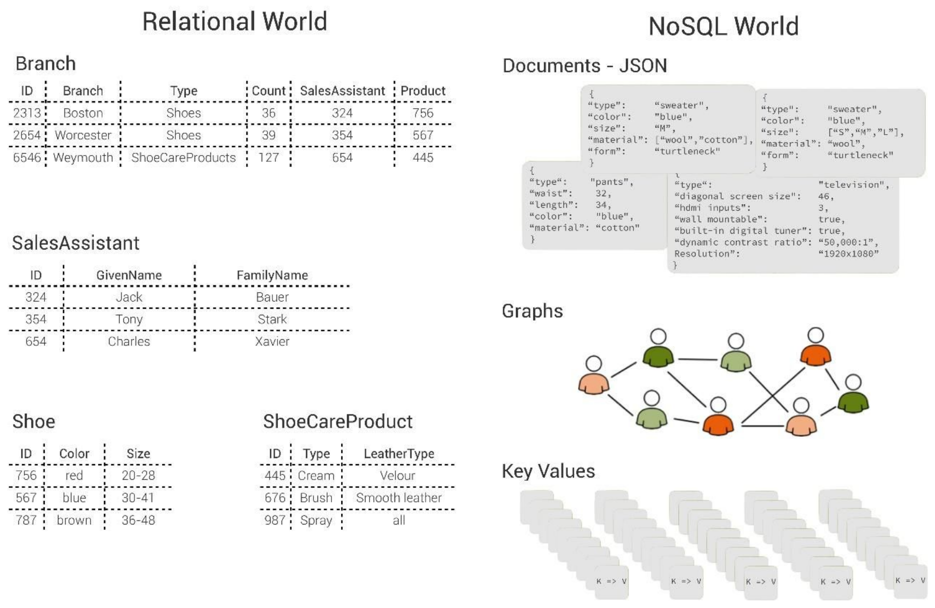
A native multi-model database is – simply put – a combination of several data stores in one. In a multi-model database data can be stored as key/value pairs, graphs or documents and can be accessed with one declarative query language. It is also possible to combine different models in a single query. With a native multi-model approach you can build high-performance applications and scale horizontally using all three data models to their full extent. In comparison to a “layered approach” many vendors adapt, a native multi-model solution leads to flexibility and performance advantages. In short, a native multi-model database has one core, one query language, but multiple data models.

*A native multi-model database has one core, one query language, but multiple data models.*

# 2 Why multi-model?

In recent years, the idea of “polyglot persistence” has become quite popular. However, as mentioned above, there is an ongoing debate between some industry experts that it can be beneficial to use a variety of appropriate data models for different parts of the persistence layer of larger software projects.

According to this, one would e.g. use a relational database to persist structured tabular data; a document store for unstructured object-like data; a key/value store for a hash table; and a graph database for highly linked referential data. This means that one has to use multiple databases in the same project, which leads to some operational friction (more complicated deployment, more frequent upgrades) as well as data consistency and duplication issues.



Branch

| ID | Branch | Type | Count | SalesAssistant | Product |
|---|---|---|---|---|---|
| 2313 | Boston | Shoes | 36 | 324 | 756 |
| 2654 | Worcester | Shoes | 39 | 354 | 567 |
| 6546 | Weymouth | ShoeCareProducts | 127 | 654 | 445 |

SalesAssistant

| ID | GivenName | FamilyName |
|---|---|---|
| 324 | Jack | Bauer |
| 354 | Tony | Stark |
| 654 | Charles | Xavier |

Shoe

| ID | Color | Size |
|---|---|---|
| 756 | red | 20-28 |
| 567 | blue | 30-41 |
| 787 | brown | 36-48 |

ShoeCareProduct

| ID | Type | LeatherType |
|---|---|---|
| 445 | Cream | Velour |
| 676 | Brush | Smooth leather |
| 987 | Spray | all |

```
{
“type":       “sweater",
“color":      "blue",
“size":       “M”,
“material”: [“wool”,”cotton”],
“form”:       “turtleneck"
}
```

```
{
“type":       "sweater",
“color":      "blue",
“size":       [“S”,“M”,”L”],
“material”: “wool”,
“form”:       “turtleneck"
}
```

```
{
“type“:     "pants",
“waist":     32,
“length”:    34,
“color":     "blue",
“material”: “cotton"
}
```

```
{
“type“:                         "television",
“diagonal screen size":        46,
“hdmi inputs":                 3,
“wall mountable":              true,
“built-in digital tuner": true,
“dynamic contrast ratio”: “50,000:1”,
Resolution”:                    “1920x1080”
}
```

Figure 1: tables, documents, graphs and key/value pairs: different data models.

This is the calamity that a multi-model database addresses. You can solve this problem by using a multi-model database that consists of a document store (JSON documents), a key/value store, and a graph database, all in one database engine. Such a database has a unifying query language and API that cover all three data models that allows mixing them in a single query. These three data models are chosen because an architecture like this can successfully compete with more specialized solutions on their own turf, with respect to both: query performance and memory usage. Such a combination allows you to follow the polyglot persistence approach without the need for multiple databases.

*You can solve this problem by using a multi-model database [...].*

So, what is the concept behind a native multi-model database? Documents in a document collection usually have a unique primary key that encodes document identity, which makes a document store into a key/value store, where the keys are strings and the values are JSON documents. The fact that the values are JSON does not impose a performance penalty, but offers a good amount of flexibility. The graph data model can be implemented by storing a JSON document for each vertex and a JSON document for each edge. The edges are kept in special edge collections that ensure that every edge has `from` and `to` attributes which reference the starting and ending vertices of the edge. Having unified the data for the three data models in this way, it remains to implement a common query language that allows users to express document queries, key/value lookups, “graphy queries,” and arbitrary mixtures of these. By “graphy queries,” I mean queries that involve the particular connectivity features coming from the edges, e.g. `ShortestPath`, `GraphTraversal`, and `PatternMatching`. A Pattern Matching query in a multi-model database identifies all paths that follow an arbitrary complex combination of conditions. These conditions are composed of conditions on each single document or edge and conditions on the overall layout created by these objects.

# 3 Data modeling with native multi-model databases

## 3.1 Aircraft fleet maintenance: A case study

One area where the flexibility of a native multi-model database is extremely well suited is the management of large amounts of hierarchical data, such as in an aircraft fleet. An aircraft fleet consists of several aircrafts, and a typical aircraft consists of several million parts: subcomponents, larger and smaller components. We get a whole hierarchy of "items". *To organize the maintenance of such a fleet, one has to store a multitude of data at different levels of this hierarchy*.

There are names of parts or components, serial numbers, manufacturer information, maintenance intervals, maintenance dates, information about subcontractors, links to manuals and documentation, contact persons, warranty and service contract information, to name but a few. Every single piece of data is usually attached to a specific item in the above hierarchy.

This data is tracked in order to provide information and answer questions. Questions can include but are not limited to the following examples:

- **What are all the parts in a given component?**
- **Given a (broken) part, what is the smallest component of the aircraft that contains the part and for which there is a maintenance procedure?**
- **Which parts of this aircraft need maintenance next week?**

### 3.1.1 A data model for an aircraft fleet

So, how do we model the data about our aircraft fleet if we have a multi-model database at our disposal?

There are several possibilities, but one good option allowing quick query execution is the following: there is a JSON document for each item in our hierarchy. Due to the flexibility and recursive nature of JSON, we can store nearly arbitrary information about each item, and *since the document store is schemaless, it is no problem that the data about an aircraft is completely different* from the data about an engine or a small screw.

Furthermore, we store containment as a graph structure. That is, the fleet vertex has an edge to every single aircraft vertex, an aircraft vertex has an edge to every top-level component it consists of, component vertices have edges to the subcomponents they are made of, and so on, until a small component has edges to every single individual part it contains.

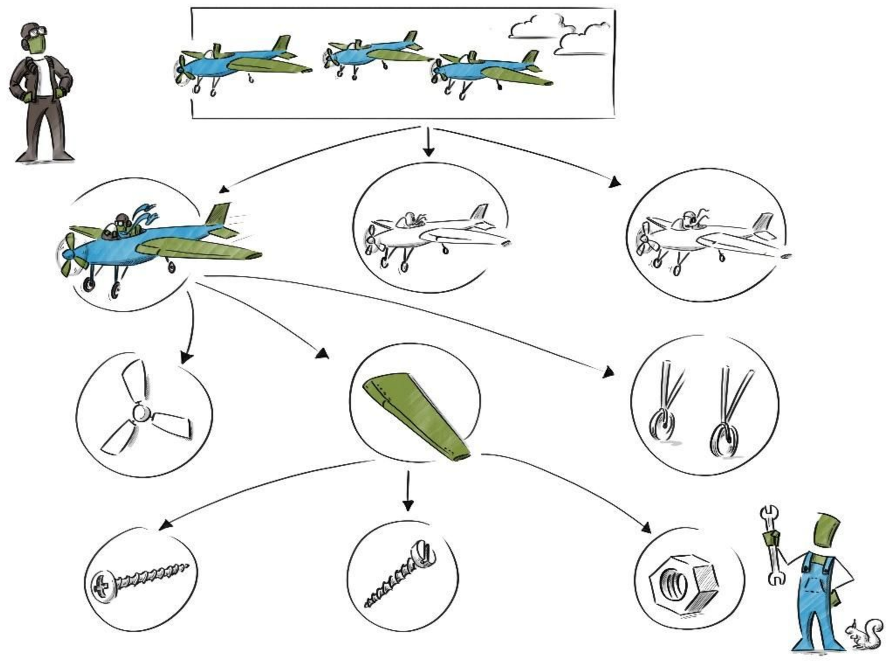


Figure 2: A tree of items.

We can either put all items in a single (vertex) collection or sort them into different ones — e.g. grouping aircraft, components, and individual parts respectively. For the graph, this does not matter, but when it comes to defining secondary indexes, multiple collections are probably better. We can ask the database for exactly those secondary indexes we need, such that the particular queries for our application are efficient.

### 3.1.2 Queries for aircraft fleet maintenance

We now come back to the typical questions we might ask of the data, and discuss which kinds of queries they might require. We will also look at concrete code examples for these queries using the ArangoDB Query Language (AQL).

- **What are all the parts in a given component?**

This involves starting at a particular vertex in the graph and finding all vertices "below" — all vertices that can be reached by following edges in the forward directions. This is a graph traversal, which is a typical graphy query.

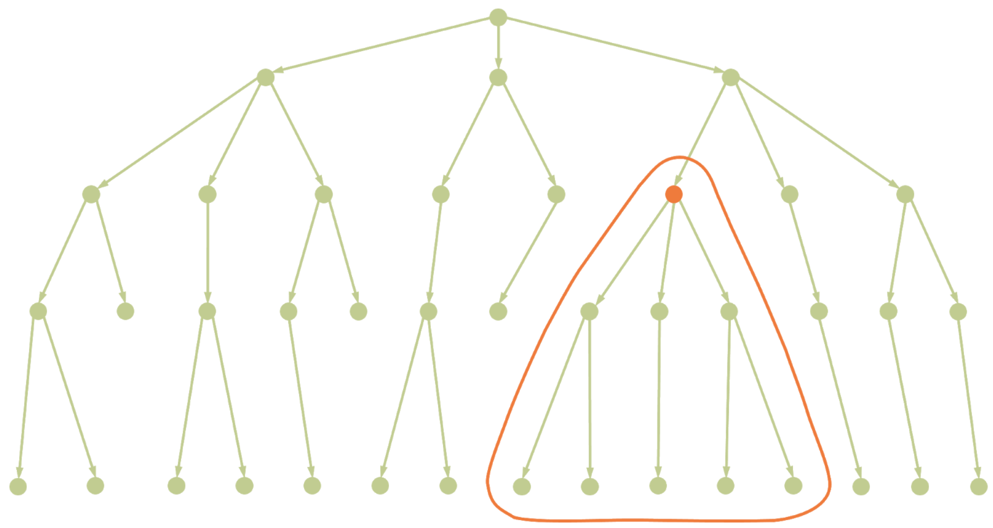


Figure 3: Finding all parts in a component.

Here is an example of this type of query, which finds all vertices that can be reached from “`components/Engine765`” in 4 steps by doing a graph traversal:

```
FOR part IN 1..4 OUTBOUND "components/Engine765" GRAPH "FleetGraph"
  RETURN part
```

In ArangoDB, one can define graphs by giving them a name and by specifying which document collections contain the vertices and which edge collections contain the edges. Documents, regardless of whether they are vertices or edges, are uniquely identified by their `_id` attribute – a string that consists of the collection name, a slash “/” character and then the primary key. The traversal only needs the graph name "`FleetGraph`", the starting vertex, and `OUTBOUND` for the direction of the edges to be followed. You can specify further options, but that is not relevant here. AQL directly supports this type of graphy query.

- **Given a (broken) part, what is the smallest component of the aircraft that contains the part and for which there is a maintenance procedure?**

This involves starting at a leaf vertex and searching upward in the tree until a component is found for which there is a maintenance procedure. That can be read off the corresponding JSON document. This is again a typical graphy query since the number of steps to go is not known a *priori*. This particular case is relatively easy since there is always a unique edge going upward.

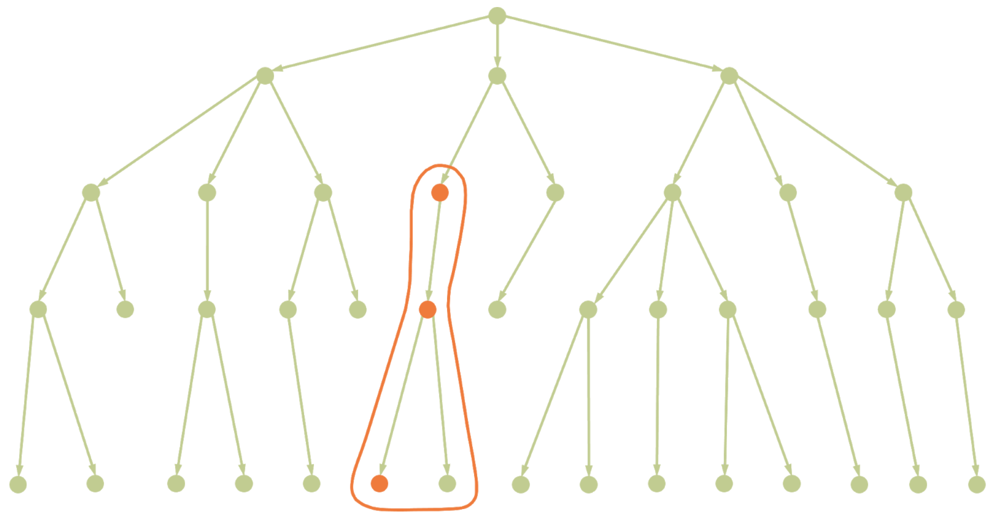


Figure 4: Finding the smallest maintainable component.

For example, the following is an AQL query that finds the shortest path from "`parts/Screw56744`" to a vertex whose `isMaintainable` attribute has the Boolean value true, following the edges in the "`inbound`" direction:

```
FOR component IN 0..4 INBOUND "parts/Screw56744" GRAPH "FleetGraph"
  FILTER component.isMaintainable == true
  LIMIT 1
  RETURN component
```

Note that here, we specify the graph name, the `_id` of the start vertex and filter on the target vertex. We are only interested in the first vertex matching the filter, hence we apply a `LIMIT 1`. We see again that AQL directly supports this type of graphy query.

- **Which parts of this aircraft need maintenance next week?**

This is a query that does not involve the graph structure at all: rather, the result tends to be nearly orthogonal to the graph structure. Nevertheless, the document data model with the right secondary index is a perfect fit for this query.

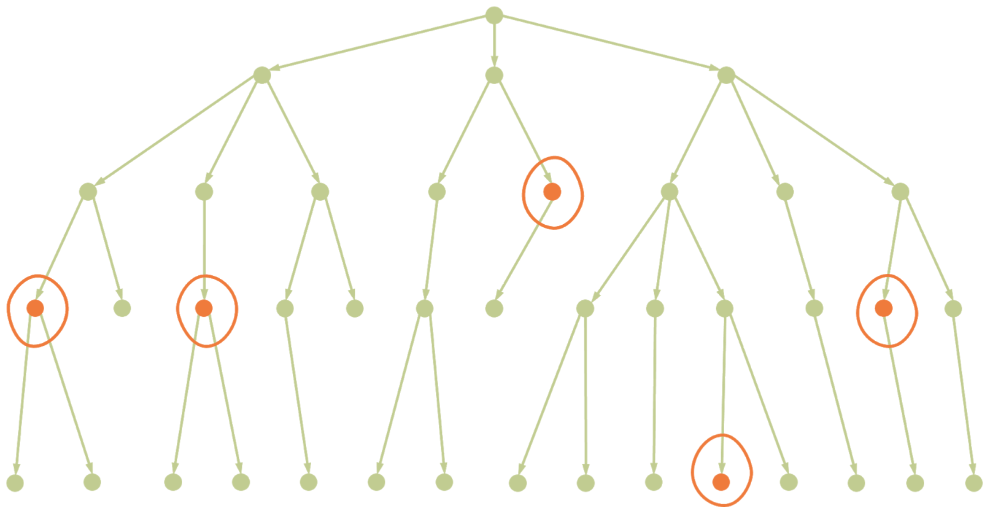


Figure 5: Query whose result is orthogonal to the graph structure.

With a pure graph database, we would be in trouble rather quickly for such a query. That is because we cannot use the graph structure in any sensible way, so we have to rely on secondary indexes — here e.g. on the attribute storing the date of the next maintenance.

To get our answer, we turn to a document query, which does not consider the graph structure. Here is one that finds the components that are due for maintenance:

```
FOR c IN components
  FILTER c.nextMaintenance <= "2016-12-15"
  RETURN {id: c._id,
          nextMaintenance: c.nextMaintenance}
```

What looks like a loop is AQL's way to describe an iteration over the `components` collection. The query optimizer recognizes the presence of a secondary index for the `nextMaintenance` attribute, such that the execution engine does not have to perform a full collection scan to satisfy the `FILTER` condition. Note AQL's way to specify projections by simply forming a new JSON document in the `RETURN` statement from known data. We see that the very same language (AQL) also supports queries that are usually found in a document store.

### 3.1.3 Using multi-model querying

To illustrate the potential of the multi-model approach, I'll finally present an AQL query that mixes the three data models. The following query starts by finding parts with maintenance due, runs the above shortest path computation for each of them, and then performs a JOIN operation with the `contacts` collection to add concrete contact information to the result:

```
FOR p IN parts
  FILTER p.nextMaintenance <= "2016-12-15"
  FOR c IN 0..4 INBOUND p GRAPH "FleetGraph"
    FILTER c.isMaintainable == true
    LIMIT 1
    FOR person IN contacts
      FILTER person._key == c.contact
      RETURN {part: p._id, component: c, contact: person}
```

Finally, we can see AQL's formulation for a JOIN. The second `FOR` statement brings the `contacts` collection into play. The query optimizer recognizes that the `FILTER` statement can be satisfied best by doing a JOIN, which in turn is very efficient because it can use the primary index of the `contacts` collection for a fast hash lookup.

This is a prime example for the potential of the multi-model approach. The query needs all three data models: documents with secondary indexes, graphy queries, and a JOIN powered by fast key/value lookup. Imagine the hoops through which we would have to jump if the three data models would not reside in the same database engine, or if it would not be possible to mix them in the same query.

Even more importantly, this case study shows that the three different data models were indeed necessary to achieve good performance for all queries arising from the application. Without a graph database, the queries of a graphy nature with path lengths, which are not a *priori* known, notoriously lead to nasty, inefficient multiple JOIN operations. However, a pure graph database cannot satisfy our needs for the document queries that we got efficiently by using the right secondary indexes. The key/value lookups complement the picture by allowing interesting JOIN operations that give us further flexibility in the data modeling. For example, in the above situation, we did not have to embed the whole contact information with every single path, simply because we could perform the JOIN operation in the last query.

## 3.2 Lessons learned for data modeling

- **JSON is very versatile for unstructured and structured data.** The recursive nature of JSON allows embedding of subdocuments and variable length lists. You can even store the rows of a table as JSON documents. Modern data stores are so good at compressing data that there is no memory overhead in comparison to relational databases. For structured data, schema validation can be implemented as needed using an extensible HTTP API.

- **Graphs are a good data model for relations.** In many real world cases, a graph is a natural data model. It captures relations and can hold label information with each edge and with each vertex. JSON documents are a natural fit to store this type of vertex and edge data.

- **A graph database is particularly good for graphy queries.** The crucial thing here is that the query language must implement routines like "shortest path" and "graph traversal". The fundamental capability for these is to access the list of all outgoing or incoming edges of a vertex rapidly.

- **A multi-model database can compete with specialised solutions.** The particular choice of the three data models allows us to combine them in a coherent engine. This combination is no compromise, it can – as a document store – be as efficient as a specialised solution, and it can – as a graph database – be as efficient as a specialised solution.

- **A multi-model database allows you to choose different data models with less operational overhead.** Having multiple data models in a single database engine alleviates several challenges of using different data models at the same time. It means less operational overhead and less data synchronisation, allowing for a huge leap in data modeling flexibility. You have an option to keep related data together in the same data store, even if it needs different data models. Mixing different data models within a single query increases options for application design and performance optimizations. And if you choose to split the persistence layer into several different database instances, you still have the benefit of only having to deploy a single technology. Furthermore, a data model lock-in is prevented.

- **Multi-model has a larger solution space than relational.** Considering all these possibilities for queries, the flexibility in data modeling and the benefits of polyglot persistence without the ensuing friction, the multi-model approach covers a solution space that is larger than that of the relational model.

# 4 Further use cases for multi-model databases

## Content management

Content can have a very inhomogeneous structure, which makes a document store a good data model. However, frequently there are links and connections between different pieces of content, which are most naturally described by a graph structure.

## Complex, user-defined data structures

Any application that deals with complex, user-defined data structures benefits dramatically from the flexibility of a document store and has often good applications for graph data as well.

## E-commerce systems

E-commerce systems need to store customer and product data (JSON), shopping carts (key/value), orders and sales (JSON or graph) and data for recommendations (graph), and need a multitude of queries featuring all of these data items.

## Enterprise hierarchies

Enterprise hierarchies come naturally as graph data and rights management typically needs a mixture of graphy and document queries.

## Fraud Detection

In this use case one usually stores a huge amount of log data which involves connections between different entities like accounts, IP addresses, machines and the like. In many cases this can sensibly be modelled by a graph structure. Detecting fraud now involves complicated pattern matching the often considers the graph structure (e.g. an unusual amount of connections to a single host or account), but sometimes also other data which is best accessed orthogonally to the graph structure using secondary indexes.

## Identity and Access Management

Like in the above case about enterprise hierarchies, identity and access management often involves data that has a hierarchical structure, and usually, people or entities higher up in the hierarchy have their own access rights plus in addition all the access rights of their subordinates. This data is best described by a tree or a directed acyclic graph. Deciding access rights often involves the graph structure, but there are also a lot of queries about the identities which completely ignore the hierarchy.

## Internet of things

The IoT produces a very high volume of status data, geo location information, sensor data and the like. At the same time, the actual things in the IoT typically come in a hierarchical structure. For example, all home devices in the same house would report up to the house, which in turn would aggregate some data and report to higher levels. This means that the data about the devices is naturally modelled by a graph and the high volume of sensor data has a diverse structure and often needs to be joined to the much smaller set of thing data.

## Knowledge graphs

Knowledge graphs are enormous data collections, most queries from expert systems use only the edges and graphy queries, but often enough you needs “orthogonal” queries only considering the vertex data.

## Logistics

In logistics a lot of data occurs: geo locations, tasks, dependencies of tasks, resources needed for tasks. The data is both of a rather diverse structure and highly connected. Queries involve both graphy queries considering dependencies and standard index backed queries ignoring dependencies.

## Network and IT Operations

Computer networks and the associated hosts themselves form a graph, and management of such infrastructure frequently involves queries about this very graph structure, but also queries about the set of hosts or similar things.

## Real-time Recommendation Engine

Coming up with sensible and effective real-time recommendations for customers in e-commerce is essentially path pattern matching in graphs, since one would like to recommend things to a customer A that have been bought by another customer B who is linked to A in some way, for example by both having bought similar products. At the same time, queries also use secondary indexes on the produce catalog, for example to take sales rank and things into account.

## Social networks

Social networks are the prime example for large, highly connected graphs and typical queries are graphy, nevertheless, actual applications need additionally queries which totally ignore the social relationship and thus need secondary indexes and possibly JOINs with key lookups.

## Traffic Management

Street networks are naturally modelled as a graph. Traffic flow data produces a high volume of time based data which is closely related to the street network. Finding good decisions about traffic management involves querying all this data and running intelligent algorithms using aggregations, graph traversals and joins.

## Version management applications

Version management applications usually work with a directed acyclic graph, but also need graphy queries and others.

## Workflow management software

Workflow management software often models the dependencies between tasks with a graph, some queries need these dependencies; others ignore them and only look at the remaining data.