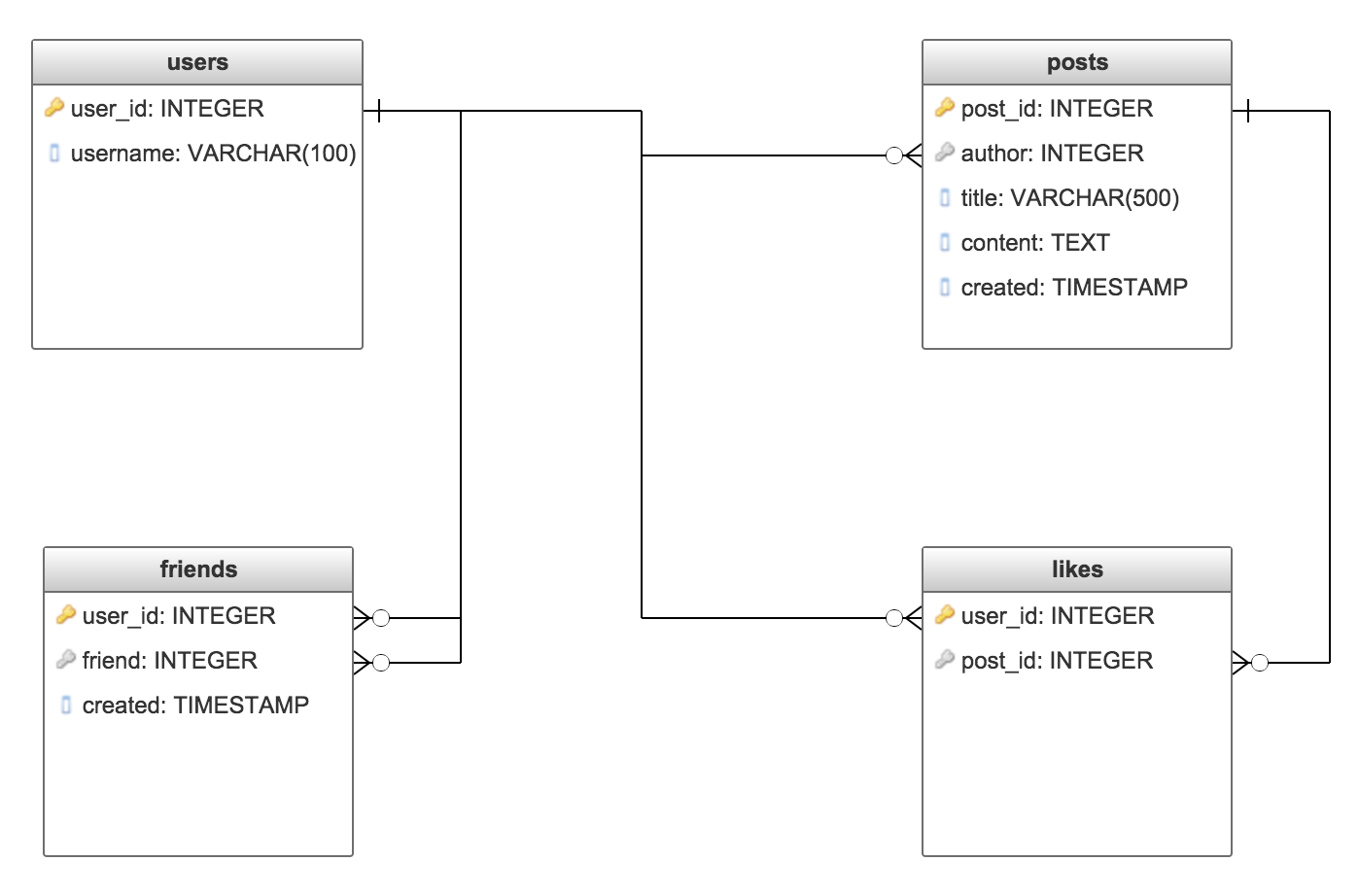
**Graph Databases are NOT partition tolerant hence they fall into CA category that is Available and Consistent.**

Other way to look is that the graph is highly connected and sharding it on some domain/key base might be complicated hence it doesn’t also look like scalable. Possible a Master-Slave arrangement where most of the CA categories fall into.

Something like RDBMS where it works on a single Node without any Partition failure hence Available and Consistent.

**Relational Database :**



We have ended up with 4 different tables, with 5 foreign key relationships between them.  
Two of these tables are actual data, and the other two are nothing more than links between entities in our system.

**Document Store:**

#### Users

{

"user\_id": "u1",

"username": "grahamcox",

"friends": {

"u2": "2017-04-25T06:41:11Z",

"u3": "2017-04-25T06:41:11Z"

}

}

#### Posts

{

"post\_id": "p1",

"author": "u1",

"title": "My first post",

"content": "This is my first post",

"created": "2017-04-25T06:41:11Z",

"likes": [

"u2"

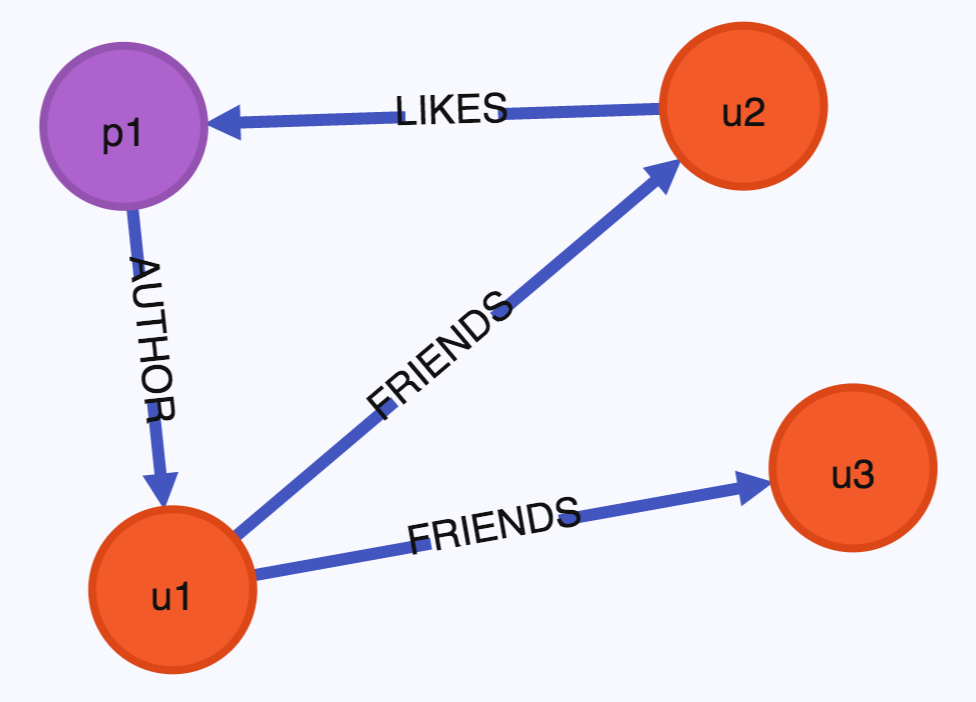
]

}

Straight away we've reduced the number of entities we are modeling down to two - which is correct from our original data modeling. We've also made it so that we get some of the related data about an entity all in one go - a Post and all of the Likes, for example. However, the cross-links from Post to User and from User to User are harder to manage in this setup. Also, remember that most Document Databases don't support relational integrity so these cross-links need to be maintained by the software, and support needs to be built in for when they are broken.

**Graph Database :**

In a Graph Database, we can choose to model the Entities as our Nodes, and the Relationships as our Edges. This gets us closer to the Document Store model - where we only have two types of Entity - but with the power of the Relational Model - where we don't have to handle links between Entities manually, and where we can easily traverse these links inside the database itself. This might look something like this:



Here we have two different types of Node, and three different types of Edge.  
Whilst not visible in the diagram, the Nodes and Edges can each contain data, similar to the Relational model.  
For example, the "FRIENDS" Edge would contain the date when the Relationship was created, allowing us to list all Friends in time order.

This very quickly shows us that we have all of the power that we are used to from the Relational model, but with the flexibility we are used to from the Document Store model.

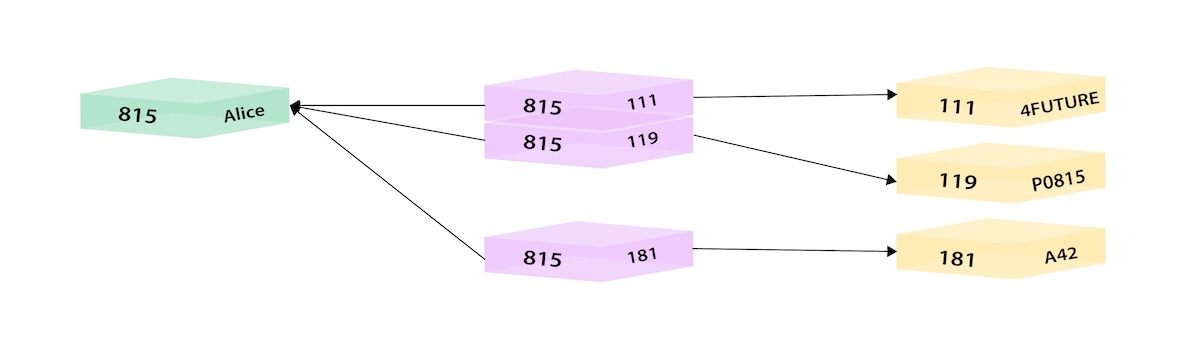
## Should I use a Graph Database?

Obviously, a Graph Database will not always be the best fit for your needs. Every situation is different and you need to evaluate the requirements every time. The most important thing you need to do is evaluate your data model. It's very likely that it is highly relational. Most real world data models are. In this case, a Graph Database is already likely to be a good fit for your needs.

Next, determine the type of relationships that your data has. If it contains a number of Many-To-Many relationships then a Graph Database will probably work better for your needs than a traditional Relational Database. Even if it contains a number of One-To-One or One-To-Many relationships though, a Graph Database may make this easier to represent.

Thirdly, determine the schema of your data.  
Graph Databases are generally much more flexible in the way that they allow you to store data, allowing for much more fluidity of the data present in each location. If your data needs are such that the schema is not absolutely rigid then a Graph Database may be a better fit, even if a Relational Database fits your needs otherwise.

Finally, determine what you want to do with your data.  
If you want to do complex data analysis, or potentially expensive queries spanning multiple types of data, then a Graph Database may make this easier to achieve and will possibly make the queries run more efficiently.

**Example of RDBMS and Graph Database**

Three tables here , Employee, Departments and Emp\_Dept to store relationship. The higher Normalization leads to higher Consistency so one can break it into De-Normalized data with a NoSQl database but that will lead to InConsistency.

Joins are complex and slow down with increased Data with any RDBMS.

**SELECT** name **FROM** Person

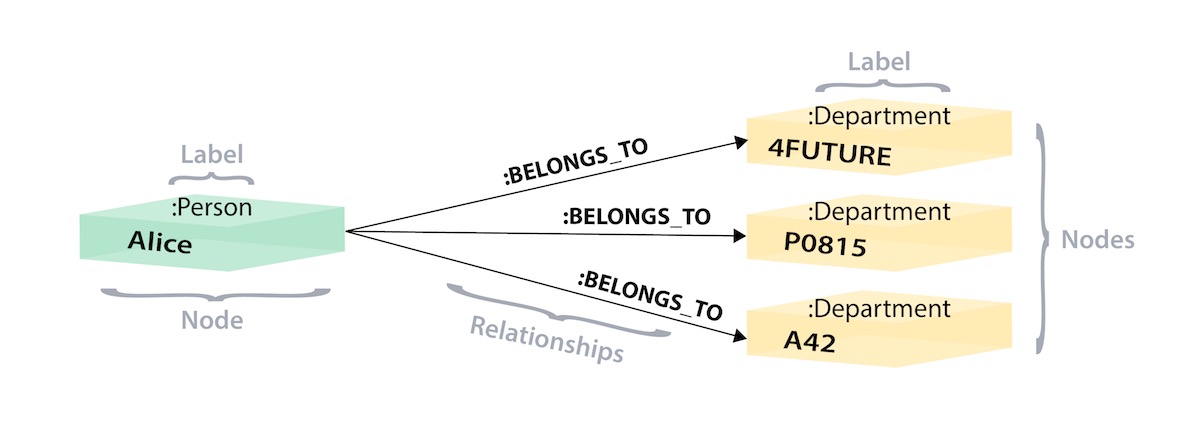
LEFT **JOIN** Person\_Department

**ON** Person.Id = Person\_Department.PersonId

LEFT **JOIN** Department

**ON** Department.Id = Person\_Department.DepartmentId

**WHERE** Department.name = "IT Department"



This is a Graph Database with two Nodes connected with a relationship Edge. The relationship itself can have properties but in this case has none.

Its easy to understand and write meaningful queries.

**MATCH** (p:Person)-[:WORKS\_AT]->(d:Dept)

**WHERE** d.name = "IT Department"

**RETURN** p.name

### **The ACID Consistency Model**

Many developers are familiar with ACID transactions from working with [relational databases](https://neo4j.com/whitepapers/rdbms-developers-graph-databases-ebook/?ref=blog). As such, the ACID consistency model has been the norm for some time.  
  
The key ACID guarantee is that it provides a safe environment in which to operate on your data. The ACID acronym stands for:  
  
**Atomic**

* + All operations in a transaction succeed or every operation is rolled back.

**Consistent**

* + On the completion of a transaction, the database is structurally sound.

**Isolated**

* + Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

**Durable**

* + The results of applying a transaction are permanent, even in the presence of failures.

### **The BASE Consistency Model**

For many domains and use cases, ACID transactions are far more pessimistic (i.e., they’re more worried about data safety) than the domain actually requires.  
  
In the NoSQL database world, ACID transactions are less fashionable as some databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like scale and resilience.  
  
(Notably, the .NET-based [RavenDB](https://ravendb.net/) has bucked the trend among aggregate stores in supporting ACID transactions.)  
  
Here’s how the BASE acronym breaks down:  
  
**Basic Availability**

* + The database appears to work most of the time.

**Soft-state**

* + Stores don’t have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

**Eventual consistency**

* + Stores exhibit consistency at some later point (e.g., lazily at read time).

BASE properties are much looser than ACID guarantees, but there isn’t a direct one-for-one mapping between the two consistency models (a point that probably can’t be overstated).  
  
A BASE data store values availability (since that’s important for scale), but it doesn’t offer guaranteed consistency of replicated data at write time. Overall, the BASE consistency model provides a less strict assurance than ACID: data will be consistent in the future, either at read time (e.g., [Riak](http://basho.com/products/)) or it will always be consistent, but only for certain processed past snapshots (e.g., [Datomic](https://www.datomic.com/)).  
  
The BASE consistency model is primarily used by [aggregate stores](https://neo4j.com/blog/aggregate-stores-tour/?ref=blog), including column family, key-value and [document stores](https://neo4j.com/developer/graph-db-vs-nosql/?ref=blog).