**Database Relationships**

***Relationships –***

1. **SQL (via Foreign Keys)**

* one to one
* one to many
* many to many

***Mongo Relationships –***

**One to many / Approach 1 (one to few)**

Store the child document inside parent.

Eg.,

addresses: [

{ location: ‘shivdhara’ , city: ‘Darbhanga’ },

{ location: ‘sarai’ , ‘Laheriyasarai’ }

]

**One to many / Approach 2**

Store a reference to the child document inside parent.

Eg.,

orders: [

objectId(“101”),

objectId(“102”),

objectId(“103”)

]

***One to Many (POPULATE)***

**One to Many / Approach 3 (one to squillions)**

Store a reference to the parent document inside child.

***Database denormalization rules of thumb: Your guide through the rainbow***

* **One:** Favor embedding unless there is a compelling reason not to.
* **Two:** Needing to access an object on its own is a compelling reason not to embed it.
* **Three:** Arrays should not grow without bound. If there are more than a couple of hundred documents on the “many” side, don’t embed them; if there are more than a few thousand documents on the “many” side, don’t use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
* **Four:** Don’t be afraid of application-level joins: If you index correctly and use the projection specifier, then application-level joins are barely more expensive than server-side joins in a relational database.
* **Five:** Consider the read-to-write ratio with denormalization. A field that will mostly be read and only seldom updated is a good candidate for denormalization. If you denormalize a field that is updated frequently then the extra work of finding and updating all the instances of redundant data is likely to overwhelm the savings that you get from denormalization.
* **Six:** As always with MongoDB, how you model your data depends entirely on your particular application’s data access patterns. You want to structure your data to match the ways that your application queries and updates it.