Player

* Players get added (low frequency, no performance needed)
* Players log in (high frequency, performance needed)
* Report most popular skin by target audience (low frequency, no performance needed)
* Report most popular class by target audience (low frequency, no performance needed) Characters get updated when a new character gets created. (high frequency, high performance needed)

Pet

* Pets to buy for a player get shown (high frequency, performance needed)
* Pets get added (low frequency, no performance needed)
* Show all pets (high frequency, performance needed)

GoldOffer

Player 1:M GoldOffer

* Get the aggregated gold amount from successful gold offers (high frequency, performance needed)
* Updated when a new gold offer gets inserted (high frequency, performance needed)

Player M:N Pet

* Pets to buy for a player get shown (high frequency, performance needed)
* Bought pets get shown (high frequency, performance needed)
* Pet gets bought (high frequency, performance needed)
* Aggregate spent gold on pets (high frequency, performance needed)

Player 1:M Character

* All characters of a player get shown. (high frequency, performance needed)
* Aggregate quest reward gold of all characters, aggregate monster loot gold of all characters (high frequency, performance needed)
* Aggregate spent gold on skins (high frequency, performance needed)

Character

* Character gets created (high frequency, high performance needed)

Character Class

* Character class gets added (low frequency, no performance needed)
* Character classes get shown (high frequency, performance needed)

Skin

* Skin gets added (low frequency, no performance needed)
* Show available skins to buy for character (high frequency, performance needed)

Character class 1:M Character

* Character gets added with a class (low frequency, no performance needed)
* Report most popular class by target audience (low frequency, no performance needed)
* Get bonus stats for fights (high frequency, performance needed)

Character class 1:M Skin

* Skin gets added for character class (low frequency, no performance needed)
* Show all skins for class (high frequency, performance needed)
* Aggregate spent gold for skins (high frequency, performance needed)

Character M:N Skin

* Player buys skin for character (high frequency, performance needed)
* Show bought skins (high frequency, performance needed)
* Show available skins to buy for character (high frequency, performance needed)
* Report most popular skin by target audience (low frequency, no performance needed)

Quest

* Quest gets added (low frequency, no performance needed)

Character M:N Quest

* Character completes quest (high frequency, performance needed)
* Aggregate quest rewards of character (high frequency, performance needed)

Quest reward

* Quest reward gets added when a quest gets added (low frequency, no performance needed)

Quest 1:1 Quest reward

* Embed quest reward in quest

Monster

* Add new monster (low frequency, no performance needed)

Monster M:M Monster

* Add allied monster (low frequency, no performance needed)
* Get possible allies of monsters (for encounters; optional) (high frequency, performance needed)

Character 1:M Monster

* Aggregate monster loot of character (high frequency, performance needed)
* Character slays monster (increase amount or insert new entry) (high frequency, performance needed)

Monster loot

* Add monster loot when a new monster gets added (low frequency, no performance needed)

Monster 1:1 Monster loot

* Embed monster loot in monster

Reports Character Class

1. Filtern der relevanten Charaktere über PlayerAge (embedded in Character)
2. Count Character Class entry amount grouped by class id
3. Order by amount

db.character.aggregate([

{"$group" : {\_id:"$characterClass.classId", className: {$first : “$characterClass.className”} , amount:{$sum:1}}}

])

Report bought skins

1. Filtern der relevanten Charaktere über PlayerAge (embedded in Character)
2. Count bought skin entry amount grouped by skin id
3. Order by amount

db.character.aggregate([ {"$unwind" : "$boughtSkins"} , { "$group" : {"\_id" : {"characterId" : "characterId"} , { "skin" : {"$first" : "$boughtSkins"}}}} , { "$group" : { "\_id" : { "$skin.skinId" } , { "skinName" : { "$first" : "$skin.skinName"}} , { "count" : {"$sum" : 1}}}} ])

Formularbeginn

db.character.aggregate([

{“$unwind”: “$boughtSkins”},

{"$group": {\_id:"$boughtSkins.skinId", skinName: {$first : “$boughtSkins.skinName”} , amount:{$sum:1}}}

])

https://stackoverflow.com/questions/13055462/group-by-specific-element-of-array-with-mongo-aggregation-framework

# indizes:

Soweit ich das mit den indizes jetzt verstanden habe will man damit die queries beschleunigen inklusive sortieren.

Das was mir jetzt dazu einfällt sind folgende indexmöglichkeiten:

1) nach speziellen Einträgen über die ID suchen wie bei character, player, pet, quest, skin, monster, ... Können wir generell mit der Begründung sagen dass es für den User angenehmer ist, wenn Einträge generell in derselben Reihenfolge aufscheinen und wir deswegen einen aufsteigenden uniquen index nehmen.  
EDIT: jede collection muss beim inserten als primary key \_id specifien oder es werden automatisch objekte erstellt. Es wird automatisch ein index über die \_id fields gemacht.

2) nach player age filtern in character; index wird auch bei der aggregation pipeline am anfang für die "match"-stage verwendet was wir machen.

3) player beim login mit username? Könnten da den index darauf setzen und eigentlich auch die id löschen wenn der username eindeutig ist

4) successful bei gold offer

5) "Multi key indexes for querying arrays": useful für uns überall wo wir arrays haben schätze ich, da wir bei jedem array während dem spiel einträge adden wollen/werden. Beispielsweise monster slayen, pet kaufen, skin kaufen, char erstellen, quest completen, ...

Hab ich überlegt aber denke dass es nichts bringt:

- nach amount sortieren wobei wir da denke ich keinen index machen können; es ist ja ein result von ner temporären collection.

# Index notes:

Source: https://www.mongodb.com/blog/post/performance-best-practices-indexing

**Use Compound Indexes**

Compound indexes are indexes composed of several different fields. For example, instead of having one index on "Last name" and another on "First name", it is typically most efficient to create an index that includes both "Last name" and "First name" if you query against both of the names. Our compound index can still be used to filter queries that specify the last name only.

**Follow the ESR rule**

For compound indexes, this rule of thumb is helpful in deciding the order of fields in the index:

* First, add those fields against which **Equality** queries are run.
* The next fields to be indexed should reflect the **Sort** order of the query.
* The last fields represent the **Range** of data to be accessed.

**Use Partial Indexes**

Reduce the size and performance overhead of indexes by only including documents that will be accessed through the index. For example, create a [partial index](https://docs.mongodb.com/manual/core/index-partial/) on the orderID field that only includes order documents with an orderStatus of "In progress", or only indexes the emailAddress field for documents where it exists.

**Take Advantage of Multi-Key Indexes for Querying Arrays**

If your query patterns require accessing individual array elements, use a [multi-key index](https://docs.mongodb.com/manual/core/index-multikey/). MongoDB creates an index key for each element in the array and can be constructed over arrays that hold both scalar values and nested documents.