Handout - Ringstats

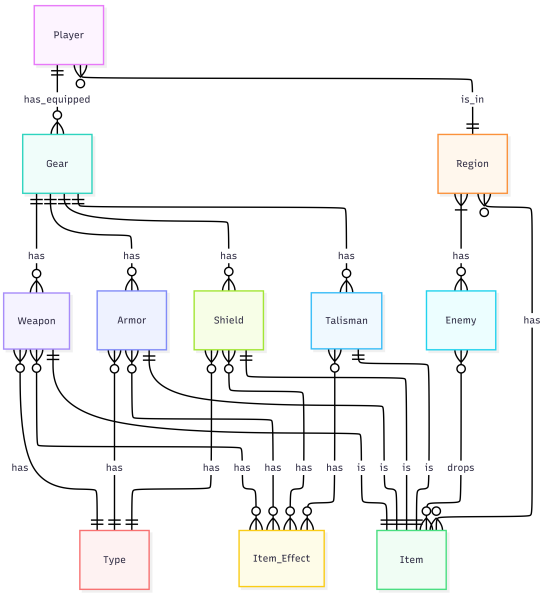
von Justin Rost, Michelle Reichelt, Yasin Ergün

Anwendungsszenario

Ringstats verbindet das laufende Spiel (Elden Ring) mit einer Datenbank voller hilfreicher Spielinformationen. Eine Eldenring-Mod liest in Echtzeit aus, wo der Spieler gerade steht, wie viel Leben er hat und was er ausgerüstet hat, und schickt das laufend an die Datenbank. Im Spiel tauchen die passenden Infos dann von selbst auf: Visiert der Spieler einen Boss an, öffnet sich ein kleines Panel mit dessen Resistenzen und häufigsten Fähigkeiten. Beim Gebietswechsel zeigt eine Checkliste an, was in der Region noch offen ist (Truhen oder Bosse). Im Browser kann man dann noch mehr Informationen lesen.

Modellierung des Anwendungsszenarios

Grundlage ist unser ER-Modell. Der Player hängt über sein Gear an Weapon, Armor, Shield und Talisman und steht immer in einer Region, in der Items und Enemies aufzufinden sind.



Das Modell oben haben wir in verschiedenen Datenbanken modelliert. Im **Key-Value-Modell** bekommt alles einen Key nach dem Muster `entität:{id}` und wird als Hash Map in der Datenbank gespeichert. Beziehungen wie die Ausrüstung speichern wir in Sets, z.B. `player:{id}:armor`. **Dokumentenorientiert** speichert zu jeder Entität ein Dokument in einer eigenen Collection. Und im **Graph-Modell** ist jede Entität ein Knoten. Die Verbindungen aus dem ER-Modell werden zu Kanten wie `has_equipped` oder `drops`.

Entschieden haben wir uns für das Key-Value-Modell. Ringstats schreibt die Daten des Spieler laufend und liest sie sofort wieder aus, viel mehr passiert da nicht. Die komplexen Abfragen, bei denen Dokumentenorientiert oder Graph stark wären, kommen bei uns kaum vor, und tief verschachtelt sind unsere Daten auch nicht.

Marktrecherche und Nutzwertanalyse

Für das Key-Value-Modell haben wir uns sechs verschiedene Datenbanken angeschaut: Redis, Memcached, ArangoDB (multimodal), Amazon DynamoDB, etcd und Riak KV. Jedes Kriterium bekommt pro Datenbank 0 bis 2 Punkte, multipliziert mit dem Gewicht.

Kriterium	**Gewicht**	**Redis**	**Memc.**	**Arango**	**Dynamo**	**etcd**	**Riak**
Hohe Perfomanz	5	2	2	1	1	1	1
Daten-Streaming	3	2	0	0	2	1	0
Niedrige Latenz	3	2	2	1	1	1	1
Skalierbarkeit	1	1	1	1	2	0	2
Keine Lizenzkosten	5	2	2	2	0	2	2
Indexing	2	1	0	1	1	0	1
Vektorrechnungen	1	1	0	1	0	0	0
Gewichtete Summe		**36**	**27**	**22**	**18**	**21**	**22**

Redis hat als einziger bei Performanz, Latenz und Daten-Streaming die volle Punktzahl. Memcached hält mit, hat aber weder Sets noch Persistenz. DynamoDB scheitert mit seinen laufenden Kosten am Muss-Kriterium „keine Lizenzkosten“, etcd ist eher für Konfigurationsdaten gedacht und an Riak KV wird kaum noch entwickelt.

Am ende haben wir davon 3 Stück umgesetzt: Redis, Memcached und ArangoDB.

Benchmarks

Gemessen haben wir zwei typische Anwendungsfälle. Beim Live-Player-Sync werden Spielerdaten gelesen und geschrieben. Beim Gebietswechsel sind es für alle Entitäten im Gebiet mehr oder weniger Anfragen. Die Ergebnise in Operationen pro Sekunde:

Ops/s	**Spieler Schreiben**	**Spieler Lesen**	**Region Lesen**	**Region Schreiben**
Redis	74,407	75,108	81,416	78,688
Memcached	65,254	62,512	72,065	69,745
ArangoDB	2,568	7,546	2,365	717

Redis und Memcached liegen nah beieinander. Die multimodale Flexibilität kostet spürbar Geschwindigkeit, und genau die brauchen wir für unser schreiblastiges Live-Szenario.

Fazit

Darum haben wir uns für Redis entschieden. Genauso schnell wie Memcached, aber mit Hashes und Sets. Und im gegensatz zu Memcached ist Redis sogar Persistenz. Genau das brauchen wir bei der Anwendung.