Ringstats Dokumentation

Justin Rost 7208260
Michelle Reichelt 7216628
Yasin Ergün 7215184

7. Juli 2026

Inhaltsverzeichnis

1 Anwendungsszenario

In diesem Kapitel zeigen wir, worum es bei Ringstats eigentlich geht und in welchen Situationen die Anwendung dem Spieler im Spiel weiterhilft.

1.1 Projektidee und Zielsetzung

Die Idee von Ringstats ist die Bereitstellung einer Schnittstelle zwischen dem aktiven Spiel und einer Datenbank mit hilfreichen Spielinformationen. Während des Spiels erfasst die Desktop Anwendung in Echtzeit Parameter des Spielers und schickt diese an die Datenbank. Zu diesen Parametern zählen z.B. die aktuelle Position oder Ausrüstung. Auf Grundlage dieser Live Daten stellt unsere Anwendung dann hilfreiche Informationen über Bosse oder Gegner in der Region des Spielers bereit. Zudem kann auch eine regionenbasierte Checkliste angezeigt werden, auf der sich weitere benutzerdefinierte Aufgaben befinden. All diese Informationen werden auf In-Game Panels eingeblendet.

1.2 Kernfunktionen und Anwendungsfälle

Um den Nutzen von Ringstats zu verstehen, haben wir folgende Kernfunktionen definiert: Die wichtigste Funktion ist das Erfassen der Spielerdaten in Echtzeit. Während der Spieler spielt, liest die Desktop Anwendung laufend aus, wo der Spieler gerade steht, wie viel Leben er hat, wie schnell er ist und was er ausgerüstet hat. Diese Daten werden direkt an die Datenbank geschickt, damit immer der aktuelle Stand vorliegt. Sobald der Spieler einen Gegner oder Boss anvisiert, blendet die Anwendung ein kleines Panel im Spiel ein. Darin sieht der Spieler die Resistenzen des Gegners und welche Fähigkeiten dieser am häufigsten benutzt. So weiß man direkt, worauf man sich einstellen muss. Zusätzlich gibt es eine Checkliste, die sich nach der Region richtet, in der sich der Spieler gerade befindet. Diese lässt sich mit einem Hotkey ein und ausblenden. Auf der Liste stehen zum Beispiel Items, Dungeons und Bosse. Der Spieler kann die Einträge filtern und selbst festlegen, ob sie automatisch abgehakt werden, sobald eine Bedingung erfüllt ist, oder ob er das lieber von Hand macht. Alle diese Daten landen nicht nur im Spiel, sondern sind gleichzeitig über die Internetseite abrufbar. Der Spieler meldet sich auf der Seite an und verbindet sich mit der Desktop Anwendung. Dann kann er seine Informationen auch im Browser einsehen.

Damit das Ganze einen Sinn ergibt, hier ein paar typische Situationen aus dem Spiel:
Der Spieler startet einen Bosskampf. In einem kleinen Fenster sieht er sofort, welche Resistenzen der Boss hat und welche Fähigkeiten am häufigsten kommen.
Während der Spieler durch ein Gebiet läuft, werden seine Daten an den Server geschickt. Dadurch sieht er direkt, welche Truhen und Sammelobjekte in seiner Nähe sind.
Wechselt der Spieler in ein neues Gebiet, zeigt die Checkliste an, welche Aufgaben in diesem Gebiet noch offen sind.

2 Dokumentation der Datenmodellierung

In diesem Kapitel schauen wir uns an, wie wir die Daten von Ringstats aufbauen und wie das Ganze in den verschiedenen Datenbanken aussieht.

2.1 Das konzeptionelle ER-Modell

Die komplexe Struktur der Spielwelt bilden wir mit einem Klassischen ER-Modell ab. Das Modell veranschaulicht, wie der Spieler (Player) über seine Ausrüstung (Gear) mit Waffen (Weapon), Rüstungen (Armor) oder Talismanen (Talisman) etc. verbunden ist. Fast alle dieser Gegenstände haben einen Typen (Type) und Effekte (Item_Effect). Auch ist der Spieler (Player) immer in einer Region (Region), in der Gegenstände (Item) und Gegner (Enemy) aufzufinden sind.

Das Modell dient in den folgenden Kapiteln als Grundlage für die Modellierung und den Vergleich.

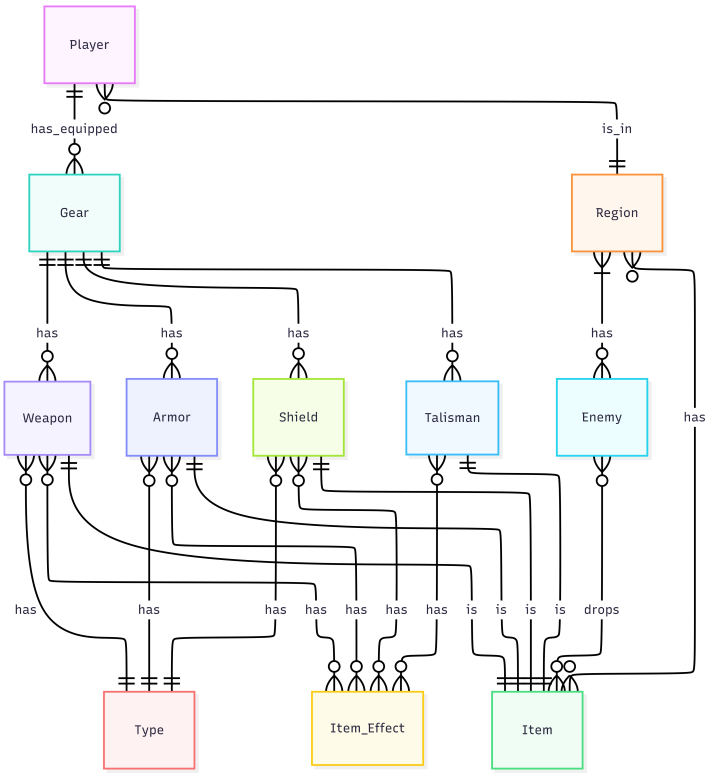

Abbildung 1: Konzeptionelles ER-Modell

2.2 Key-Value-Modellierung

Key-Value Datenbanken sind eine einfache und schnelle Lösung für die Speicherung von unstrukturierten Daten. Wir verwenden in diesem Kapitel Redis.

Schlüsselaufbau

Keys sind Strings, die eindeutig zum identifizieren von Objekten verwendet werden. Diese starten immer mit dem Namen der Entität, gefolgt von einem Doppelpunkt und der ID des Objekts. Dabei werden alle Entitäten als Redis-Datentyp Hash Map gespeichert um die Daten effizient zu verwalten.

Anhand dieses Systems bilden wir die Schlüssel wie folgt ab:

Datentyp	**Key-Struktur**	**Attribute**
`HMAP`	item_effect:{id}	id, name, value
`HMAP`	item_type:{id}	id, item_id, name
`HMAP`	weapon:{id}	id, item_effect_id, item_type_id, weight, damage
`HMAP`	armor:{id}	id, item_effect_id, item_type_id, weight, defense
`HMAP`	shield:{id}	id, item_effect_id, item_type_id, weight, damage, defense
`HMAP`	talisman:{id}	id, item_effect_id
`HMAP`	item:{id}	id, name, gear_type, gear_id
`HMAP`	player:{id}	id, name, health, stamina, level, region_id
`SET`	player:{id}:weapon	item_id
`STRING`	player:{id}:shield	item_id
`SET`	player:{id}:armor	item_id
`SET`	player:{id}:talisman	item_id
`HMAP`	region:{id}	id, name
`SET`	region:{id}:items	item_id
`SET`	region:{id}:enemies	enemy_id
`HMAP`	enemy:{id}	id, name, health, level, damage
`SET`	enemy:{id}:items	item_id

Tabelle 1: Übersicht der Schlüsselstrukturen in Redis

2.3 Dokumentenorientierte Modellierung

Dokumentenorientierte Datenbanken speichern alles, was zusammengehört, gebündelt in einem Dokument. Wir verwenden in diesem Kapitel MongoDB.

Aufbau der Dokumente

Jede Entität wird zu einem Dokument und liegt in einer eigenen Collection. Der Player hält von seiner Ausrüstung nur die ids fest, denn Weapon, Armor, Shield und Talisman liegen in eigenen Dokumenten und werden von vielen Player gleich gebraucht. So stehen ihre Grundwerte nur an einer Stelle und nicht bei jedem Player erneut. Genauso bekommen Item und Region ein eigenes Dokument, und Region und Enemy zeigen nur über die ids auf ihre Item und Enemy.

So sieht das Dokument eines Player aus:

```
player {
  id: 1,
  name: "Jonas",
  health: 100,
  stamina: 100,
  level: 1,
  region_id: 1,
  weapon: [ 12 ],
  shield: 30,
  armor: [ 41, 42 ],
  talisman: [ 7 ]
}
```

Die id beim Player ist dabei die id des Item. Das Item zeigt über gear_id auf die eigentliche Weapon, die in einem eigenen Dokument liegt und ihren item_type und item_effect direkt mitbringt:

```
weapon {
  id: 213,
  item_type: { id: 1, name: "Dolch" },
  item_effect: { id: 4, name: "Stärke", value: 5 },
  weight: 2,
  damage: 70
}
```

Und so ein Enemy, der seine Item als id behält:

```
enemy {
  id: 1,
  region_id: 1,
  name: "Tree_Sentinel",
  health: 30000,
  level: 20,
  damage: 70,
  items: [ 5, 10 ]
}
```

Das Item liegt in einem eigenen Dokument und hält die Grundwerte fest:

```
item {
  id: 12,
  name: "Dolch",
  gear_type: "weapon",
  gear_id: 213
}
```

Und die Region zeigt über die ids auf ihre Item und Enemy:

```
region {
  id: 1,
  name: "Limgrave",
  items: [ 12, 19, 248 ],
  enemies: [ 1, 245 ]
}
```

2.4 Graph-Datenbank-Modellierung

Graph-Datenbanken legen den Fokus auf die Beziehungen zwischen den Objekten. Wir verwenden in diesem Kapitel Neo4j.

Aufbau aus Knoten und Kanten

Jede Entität wird zu einem Knoten, der seine eigenen Werte direkt mitträgt. Die Verbindungen aus dem ER-Modell werden zu Kanten zwischen diesen Knoten. So lässt sich direkt verfolgen, welcher Player welche Weapon trägt oder welches Item ein Enemy fallen lässt.

Die Knoten und ihre Werte bilden wir wie folgt ab:

Knoten	**Attribute**
Player	id, name, health, stamina, level
Weapon	id, weight, damage
Armor	id, weight, defense
Shield	id, weight, damage, defense
Talisman	id
Type	id, name
item_effect	id, name, value
Item	id, name
Region	id, name
Enemy	id, name, health, level, damage

Tabelle 2: Übersicht der Knoten in Neo4j

Die Kanten verbinden die Knoten so, wie es schon das ER-Modell vorgibt:

Von	**Kante**	**Nach**
Player	has_equipped	Weapon, Armor, Shield, Talisman
Player	is_in	Region
Weapon, Armor, Shield	has	Type
Weapon, Armor, Shield, Talisman	has	item_effect
Weapon, Armor, Shield, Talisman	is	Item
Region	has	Item, Enemy
Enemy	drops	Item

Tabelle 3: Übersicht der Kanten in Neo4j

3 Vergleich der Datenmodellierung

3.1 Vor- und Nachteile des Key-Value-Modells

Ein klarer Vorteil ist, dass Redis extrem schnell ist. Genau das brauchen wir, weil wir die Daten des Player laufend schreiben und gleichzeitig im Spiel und im Browser wieder auslesen. Über die Keys kommen wir direkt an ein Objekt, ohne lange zu suchen.

Ein klarer Nachteil ist, dass komplexe Abfragen schwierig sind. Wollen wir zum Beispiel alle Weapon mit einem bestimmten item_effect finden, müssen wir das in der Anwendung selbst zusammenbauen, weil Redis dafür keine fertige Suche mitbringt.

3.2 Vor- und Nachteile des dokumentenorientierten Modells

Ein klarer Vorteil ist, dass das Modell extrem leserlich und einfach zu modellieren ist. Jedes Dokument ist für sich verständlich und man sieht direkt, welche Daten zu einem Objekt gehören.

Ein klarer Nachteil ist bei uns, dass das meiste über ids referenziert wird und nur wenig wirklich verschachtelt ist. Zwar bringt eine Weapon ihren item_type und item_effect direkt mit, aber Player, Item, Region und Enemy zeigen größtenteils nur über ids aufeinander. Dadurch nutzen wir die eigentliche Stärke des Modells kaum und haben am Ende vor allem viele einzelne Dokumente, die sich gegenseitig referenzieren.

3.3 Vor- und Nachteile des Graph-Modells

Ein klarer Vorteil ist, dass sich Beziehungen sehr leicht abfragen lassen. Wir können zum Beispiel direkt verfolgen, welcher Player welche Weapon trägt oder welches Item ein Enemy fallen lässt, ohne die Verbindungen umständlich selbst zusammenzubauen.

Ein klarer Nachteil ist, dass Neo4j für unseren Fall überdimensioniert ist. Bei uns geht es vor allem darum, die Daten des Player schnell zu schreiben und wieder auszulesen, und nicht darum, ständig tiefe Beziehungen abzufragen. Für dieses einfache Schreiben und Lesen bringt der Graph mehr Aufwand mit, als er uns am Ende Vorteil bringt.

3.4 Fazit

Wir haben die gleichen Daten aus unserem ER-Modell in drei Datenbanken abgebildet und dabei gesehen, wie verschieden das aussieht. In Redis haben wir alles über Keys und Hash Maps abgelegt und kommen so direkt an ein einzelne Objekte. In MongoDB liegt jede Entität als Dokument in ihrer Collection, und die Weapon bringt ihren item_type und item_effect gleich mit. In Neo4j ist jede Entität ein Knoten und die Verbindungen aus dem ER-Modell werden zu Kanten, also zum Beispiel has_equipped oder drops.

Aufgefallen ist uns, dass wir bei Redis und MongoDB mehrere Anfragen brauchen, weil vieles nur über ids aufeinander zeigt. Bei Redis ist das allerdings kein Problem, da diese extrem schnell im Vergleich zu den anderen beiden sind. Wollen wir zum Beispiel zu einem Player seine Weapon, holen wir erst den Player, dann das Item und dann die Weapon. So konnten wir gut vergleichen, was jedes Modell bei unseren Daten gut kann und was eher umständlich ist.

Für Ringstats fällt unsere Wahl auf das Key-Value-Modell mit Redis. Der Grund ist einfach: Bei uns geht es vor allem darum, die Daten des Player laufend im Spiel und Browser zu schreiben und sofort wieder auszulesen. Genau dafür ist Redis gemacht. Über die Keys kommen wir direkt an ein Objekt und müssen nicht lange suchen. Die komplexen Abfragen, bei denen MongoDB oder Neo4j stark wären, brauchen wir kaum, und unsere Daten sind ohnehin nicht tief verschachtelt. Deshalb passt die schnelle und einfache Lösung am besten zu dem, was Ringstats braucht.

4 Marktrecherche

In der Zwischenpräsentation haben wir uns für das Key-Value-Modell entschieden. In diesem Kapitel schauen wir uns den Markt für Key-Value-Datenbanken genauer an. Wir stellen sechs Kandidaten vor, darunter ArangoDB eine multimodale Datenbank, und führen anschließend eine Nutzwertanalyse durch.

4.1 Betrachtete Datenbanken

4.1.1 Redis

Redis ist ein In-Memory Key-Value Store und unsere Wahl aus der Zwischenpräsentation. Alle Daten liegen im Arbeitsspeicher, wodurch Lese- und Schreibzugriffe im Bereich von Mikrosekunden liegen. Neben einfachen Strings bringt Redis Datentypen wie Hashes und Sets mit, auf denen unser Schlüsselaufbau aus Kapitel 2.2 direkt aufsetzt. Seit Version 8 steht Redis wieder unter einer Open-Source-Lizenz (AGPLv3) und kostet uns damit nichts.

4.1.2 Memcached

Memcached ist eine verteilte In-Memory-Cache Datenbank und in Sachen Geschwindigkeit vergleichbar mit Redis. Dafür ist der Funktionsumfang bewusst klein gehalten: Es gibt nur einfache Key-Value-Paare, keine Datentypen wie Sets oder Hashes, und keine Persistenz. Unser Schlüsselaufbau mit SETs für Ausrüstung und Regionen lässt sich also nicht direkt abbilden Memcached steht unter der BSD-Lizenz.

4.1.3 ArangoDB (multimodal)

ArangoDB ist eine multimodale Datenbank und vereint Dokumente, Graphen und Key-Value-Zugriffe in einem System mit der gemeinsamen Abfragesprache AQL. Damit hätten wir unsere drei Modellierungen aus Kapitel 2 theoretisch in einer einzigen Datenbank abbilden können. Der Nachteil ist der Langsame zugriff auf die Daten. Seit Version 3.12 steht ArangoDB unter der Business Source License. Für unser Uni-Projekt ist das kostenlos, für einen kommerziellen Betrieb gäbe es Einschränkungen.

4.1.4 Amazon DynamoDB

DynamoDB ist der verwaltete Key-Value- und Dokument-Store von AWS. Die Datenbank skaliert beliebig. Der große Haken ist das Preismodell: DynamoDB läuft nur in der AWS-Cloud und wird pro Anfrage und Speicher abgerechnet. Bei unseren dauerhaften Schreibzugriffen von bis zu 10 Anfragen pro Sekunde würden laufende Kosten entstehen, womit unser Muss-Kriterium „keine Lizenzkosten“ verletzt ist. Zudem ist die Latenz deutlich höher durch den Netzwerkweg in die Cloud.

4.1.5 etcd

etcd ist ein Key-Value Store, der hauptsächlich zum Speichern von Einstellungen und Konfigurationen benutzt wird, zum Beispiel intern von Kubernetes, ist also nicht dafür gedacht, laufend Spielerdaten zu speichern wie bei uns. Die Lizenz ist Apache 2.0 und damit kostenlos.

4.1.6 Riak KV

Riak KV ist ein verteilter Key-Value Store. Seine Stärke liegt in der Ausfallsicherheit: Daten werden automatisch über mehrere Knoten repliziert, und der Cluster bleibt auch beim Ausfall einzelner Knoten verfügbar. Riak KV steht unter der Apache-2.0-Lizenz.

4.2 Nutzwertanalyse

Wir bewerten alle sechs Datenbanken mit den Auswahlkriterien und Gewichten. Jedes Kriterium wird pro Datenbank mit 0 bis 2 Punkten bewertet und mit dem Gewicht multipliziert.

Kriterium (Gewicht)		**Redis**	**Memc.**	**Arango**	**Dynamo**	**etcd**	**Riak**
Hohe Performanz	(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
Präferenz	(1)	1	0	0	0	0	0
Gewichtete Summe		**37**	**27**	**22**	**18**	**21**	**22**

Tabelle 4: Nutzwertanalyse der sechs Key-Value-Datenbanken

Redis liegtklar vorne, weil es als einziger Kandidat sowohl bei Performanz und Latenz als auch beim Daten-Streaming die volle Punktzahl erreicht. Memcached kommt bei der reinen Geschwindigkeit zwar mit, scheitert aber am Muss-Kriterium Daten-Streaming. DynamoDB fällt aber wegen der laufenden Kosten am Muss-Kriterium „keine Lizenzkosten“ durch. ArangoDB ist als multimodale Datenbank flexibel, aber nicht auf unser schreiblastiges Live-Szenario zugeschnitten, etcd ist für Konfigurationsdaten statt Anwendungsdaten gebaut, und bei Riak KV kommt zur fehlenden Streaming-Funktion die kaum noch aktive Entwicklung hinzu.

4.3 Datenbankauswahl

Aus den sechs betrachteten Datenbanken wählen wir drei aus, die wir umsetzen und miteinander vergleichen, jeweils eine pro Gruppenmitglied:

- **Redis** (Justin Rost) — der Sieger unserer Nutzwertanalyse und unsere Wahl aus der Zwischenpräsentation, bisher aber noch nicht umgesetzt. Redis dient im Vergleich als Referenz, an der sich die beiden anderen Datenbanken messen müssen.
- **Memcached** (Yasin Ergün) — der direkteste Konkurrent, da ebenfalls In-Memory und ähnlich schnell. Der Vergleich zeigt, welche Redis-Funktionen wir tatsächlich brauchen und was uns bei einem reinen Cache fehlen würde.
- **ArangoDB** (Michelle Reichelt) — die multimodale Datenbank im Feld. Der Vergleich zeigt, was uns eine einzige Datenbank für alle drei Modellierungen aus Kapitel 2 gebracht hätte und was sie an Performanz kostet.