

Egypt University of Informatics

Computer and Information Systems

Big Data Engineering Course

**Technical Report**

**Deliverable 5**

**Online Gaming System Project**

Submitted by:

Nada Ashraf 22-101043

Menna Azazy22-101003

Aly Zaki 22-101096

Ahmed Waleed 22-101058

Omar Bayoumi 22-101022

Submission Date: 26th May 2025

**I. Introduction**

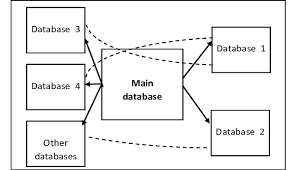
The rapid growth and complexity of online gaming platforms demands robust and scalable data management solutions. Modern games generate vast amounts of heterogeneous data, including player profiles, game state information, real-time events, social interactions, and analytics. Managing this data efficiently requires leveraging the strengths of multiple database management systems (DBMS), each optimized for specific data types and access patterns.

This project presents the design and implementation of a comprehensive online gaming platform built on a multi-DBMS architecture. The platform incorporates relational databases alongside several NoSQL databases including key-value stores, document stores, column family stores, and graph databases to meet the diverse data requirements.

**The multi-DBMS approach allows the platform to:**

* Handle real-time, low-latency updates for dynamic game state and player interactions.
* Support complex relationships and transactional consistency for core player data.
* Provide scalable storage and fast querying for time-series leaderboard and player statistics.
* Accommodate flexible and evolving schemas for game objects, world layouts, and configuration data.
* Facilitate efficient social graph traversals for friend lists and group memberships.

By mapping each type



**II. Summary of multi-DBMS:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Requirement** | **DBMS** | **Reason for Choice** | **Design Summary** |
| **Player Profiles & Inventory** | Relational DB | ACID, Complex relationships, Structured | Tables with normalized relations (covered separately) |
| **Leaderboard Data** | Cassandra | Scalable, high throughput, time-series | Partition by *game\_mode/season/snapshot,* clustering on score |
| **Player Statistics** | Cassandra | Time-series, flexible schema, scalable | Partition by *player/season,* clustering by match\_id |
| **In-Game Chat** | Cassandra | High write throughput, time-ordered logs | Partition by *game\_id/channel*, cluster by *message\_time* |
| **Object Catalog** | MongoDB | Schema flexibility, heterogeneous data | Unified collection with type field differentiation |
| **Game Catalog & World Layout** | MongoDB | Nested, flexible, evolving schema | Separate collections for games and world layouts |
| **Game Object Instances (Real-time)** | Redis | Low latency, high throughput | Key pattern with game/object/instance, JSON values |
| **Game State Data (Real-time)** | Redis | Ultra-low latency, event logging | Player states, event lists, world states as JSON |

**Data Requirements**

**Player Data**

**1. Player Profile Data**

Database Type: Relational Database

Structured details:

* + Player ID
  + Username
  + Email
  + Registration date
  + Last login timestamp
  + Account status

Justification:

* Player profile data requires a relational database due to its highly structured nature and the need for strong consistency. The fields are well-defined and benefit from enforced schema constraints and unique indices (e.g., for email and username) to ensure data integrity.

A screenshot of a computer program

AI-generated content may be incorrect.

**2. Achievement Catalog**

*Database Type: Relational DBMS*

Schema:

* Achievement ID (PK)
* Name
* Description

**Justification:**

* Achievements are structured, discrete entities with descriptive metadata. Using a relational database ensures data integrity and easy maintenance of the catalog.

Example:

* First blood: Kill your first enemy
* Sharp shooter: Achieve 5 headshots in a agme

**3. Player Achievements**

*Database Type: Relational DBMS*

Schema:

* Tracks which achievements a player has unlocked and when.
* Represents a many-to-many relationship between players and achievements catalog

Design:

* Player ID (FK)
* Achievement ID (FK)
* Unlocked at (Date)

**Justification:**

* The relational model supports the many-to-many relationship with a join table, enforcing referential integrity and enabling efficient queries about player progress and achievements.

A screenshot of a computer code

AI-generated content may be incorrect.

**4. Player Inventory**

*Database Type: Relational DBMS*

Schema:

* Item ID
* Name
* Type
* Stats

Example:

* AK-47, Weapon, Damage=35
* Med kit, Consumable, Heal=50

**5. Instance of items a player owns**

*Database Type: Relational DBMS Cached KVS:*

* Player ID
* Item ID
* Quantity

Example:

* Player 1 owns 2 AK-47s

**Justification:**

* The catalog requires structured data storage with referential integrity, making relational DB the ideal choice.
* Real-time inventory states are cached in a key-value store to improve game performance by reducing database load and latency.

A screenshot of a computer code

AI-generated content may be incorrect.

**Game Object Data:**

**1. Object Catalog**

*Database Type: Document Database*

*Tool Used: Mongo DB*

Introduction:

* Design and implement the Object catalog system for a large-scale online gaming platform using MongoDB, a document-oriented NoSQL database.
* The Object Catalog stores definitions of all in-game object types such as monsters, NPCs, projectiles, and collectibles.
* These objects are semi-static and used during game initialization, configuration, or logic handling.
* MongoDB provides a flexible, schema-less structure that allows storing varied object types in one collection, making it ideal for heterogeneous game data.

Database Selection & Justification

Why MongoDB?

* **Schema Flexibility**: Supports different object types with unique attributes without enforcing rigid schemas.
* **Document-Based Storage**: Objects are stored as JSON-like documents, naturally fitting the structure of game entities.
* **Ease of Evolution**: New object fields or types can be added without altering existing records.
* **Efficient Reads/Writes**: Suitable for semi-static data used during game setup or asset loading.
* **Developer Friendly**: Easy to query, modify, and interact with using tools like MongoDB Compass and Python.

Schema Design:

It stores different types of game objects in a unified structure. MongoDB’s document model allows each object to have different fields based on its type.

1. Monster Document

* **Purpose:**
  + Defines enemy characters (e.g., Zombie, Dragon) with combat stats and weapons. Used by the game engine to spawn and manage AI enemies.
* **Partition Key:** type = "monster"
  + Allows fast grouping and filtering of all monster objects using an index.
* **Clustering Columns:** name or \_id
  + Useful for identifying or retrieving a specific monster (e.g., "Zombie")

A screenshot of a computer program

AI-generated content may be incorrect.

2. NPC Document

* **Purpose:**
  + Represents non-hostile characters like shopkeepers, quest givers, and townsfolk. Used for world-building, dialogue, and non-combat interactions.
* **Partition Key:** type = "npc"
  + Separates NPCs from other object types.
* **Clustering Column:** dialogue\_id or name
  + Used to fetch the correct character for a specific conversation or location.

A screenshot of a computer

AI-generated content may be incorrect.

3. Projectile Document

* **Purpose:**
  + Represents any moving object fired or thrown in the game (e.g., arrows, fireballs, bullets). Used in combat and physics simulations.
* **Partition Key:** type = "projectile"
  + Group all projectiles together for loading/logic.
* **Clustering Column:** owner\_id or name
  + Often queried by those who fired it (e.g., to track performance or trajectory logic).

A screen shot of a computer

AI-generated content may be incorrect.

4. Collectible Database

* **Purpose:**
  + Defines items that can be picked up by players in the game world (e.g., coins, health potions, ammo). Used during level setup or rewards.
* **Partition Key:** type = "collectible"
  + Helps load all pickup items quickly or group them for spawning logic.
* **Clustering Column:** item\_id or name.
  + Uniquely identifies what effect the collectible has in-game.

A screenshot of a computer code

AI-generated content may be incorrect.

Object Catalog Attribute Definitions

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Description** |
| **type** | **string** | **The category of the object (monster, npc, projectile, collectible)** |
| **name** | **string** | **Display name of the object** |
| **base\_stats** | **object** | **Optional field for stats like HP, attack, speed (used by monster and npc)** |
| **weapon** | **object** | **Weapon information including type and damage range (for monster)** |
| **behavior** | **string** | **NPC behavior (shop, quest\_giver, etc.)** |
| **dialogue\_id** | **int** | **ID for predefined dialogues** |
| **velocity** | **object** | **For projectiles, defines movement direction/speed** |
| **owner\_id** | **int** | **ID of the entity who fired the projectile** |
| **item\_id** | **int** | **ID of a collectible item** |
| **effects** | **array** | **List of item effects (e.g., healing)** |

**2. Game Object Instances (Real-Time)**

Purpose

* Store and manage data about specific game object instances (e.g., monsters, NPCs, projectiles, and collectibles ) actively running within game sessions. This includes dynamic attributes such as position and health, which frequently update as the game progresses.

Why Redis (Key-Value Store)?

* *Low latency:* 
  + Data stored in-memory ensures millisecond-level access, critical for smooth real-time updates.
* *Simple key-value model:* 
  + Keys structured to encode game ID, object type, and instance ID allow efficient lookups.
* *High throughput:* 
  + Supports thousands of reads/writes per second, suitable for active game sessions with many objects.
* *Data isolation:* 
  + Separate keys per game instance and object make cleanup and scaling easier.

1. Monster

* **Definition:**  
  Any enemy in the game such as goblins, dragons, zombies, bosses, etc.
* **Example Attributes:**
  + Type: Always "monster" (allows filtering for enemies)
  + x, y, z: Current position in the 3D game world
    - x: left/right
    - y: forward/backward
    - z: up/down
  + hp: Hit points representing current health (0 means dead)
  + state: Current behavior: idle, aggro, or dead

2. NPC (Non-Player Character)

* **Definition:**
  + Characters not controlled by players and not enemies
  + Shopkeepers, quest givers, or townsfolk.
* **Example Attributes:**
  + type: "npc"
  + x, y, z: Current position in the game world
  + hp: Hit points, often max unless in dangerous areas
  + behavior: shopkeeper, quest giver, wandering, talking

3. Projectile

* **Definition:**
  + Moving objects shot or thrown during gameplay like arrows, bullets, fireballs, grenades.
* **Example Attributes:**
  + type: "projectile"
  + x, y, z: Current position
  + vx, vy, vz: Velocity vector components in 3D space
  + owner\_id: ID of player or monster who fired the projectile (used for crediting kills/hits)

4. Collectible

* **Definition:**
  + Items placed in the game world for players to pick up — health potions, coins, ammo, power-ups.
* **Example Attributes:**
  + type: "collectible"
  + x, y, z: Position in the game world
  + item\_id: Reference ID specifying the type of collectible and its effect

**CMD Redis Implementation**

A screenshot of a computer program

AI-generated content may be incorrect.

**Python CRUD:**

A computer screen with many small colored lines

AI-generated content may be incorrect.

**Game State Data (Real Time Updates)**

*Purpose*

* Real-time tracking of critical, rapidly changing data during active gameplay, including player locations, game events like item pickups or enemy defeats, and dynamic world state information.

*Data Requirements*

* **Track player locations:**
  + Continuously update and retrieve each player’s current position (x, y, z) and state.
* **Real-time game events:**
  + Capture time-stamped events such as item pickups, kills, and other actions.
* **Dynamic world state:**
  + Maintain global game properties like safe zones that evolve over time.
* **Very low latency reads and writes:**
  + Millisecond-level speed is essential to ensure smooth, responsive gameplay.
* **High throughput:**
  + The system must handle thousands of simultaneous read/write operations per second.

*Why Key-Value Stores (Redis)?*

* **In-memory storage:**
  + Data is stored in RAM, enabling ultra-fast access and updates.
* **Simple data model:**
  + Key-value pairs are ideal for storing individual player states and event logs.
* **List structures:**
  + Redis lists efficiently manage ordered event sequences.
* **Volatile data:**
  + Data is volatile and game-session bound, aligning with Redis strengths.
* **Easy scaling:**
  + Supports partitioning and replication for high availability and performance.

**Key Naming Patterns**  
*Player State:*

* game:{game\_id}:player:{player\_id}:state

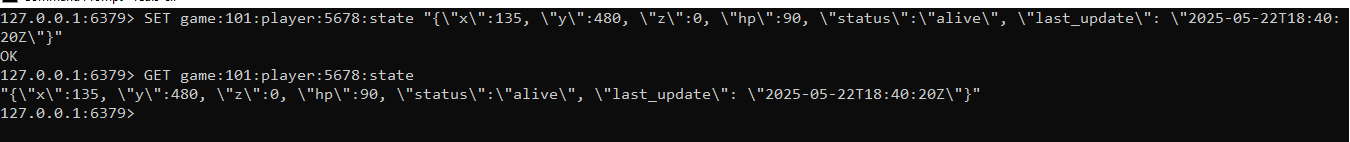
*Game Events List:*

* game:{game\_id}:events

*World State:*

* game:{game\_id}:world:state

Player State:

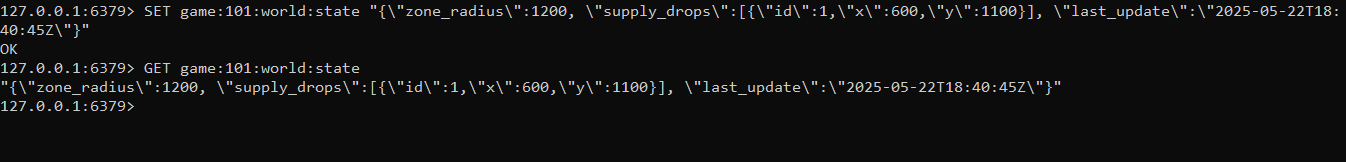


Event State:

A black background with many small lights

AI-generated content may be incorrect.

World State:



Python CRUD:

A computer screen with many small colored lines

AI-generated content may be incorrect.

Python CRUD User Input (Redis)

A black screen with many small colored lights

AI-generated content may be incorrect.

Validation in Command Prompt (Redis)



Game State GUI:

Add Player State:

A screenshot of a computer

AI-generated content may be incorrect.

Get Player State:

A screenshot of a computer

AI-generated content may be incorrect.

Add World State

A screenshot of a computer

AI-generated content may be incorrect.

Get World State

A screenshot of a computer

AI-generated content may be incorrect.

**Leaderboard Data:**

*Introduction*

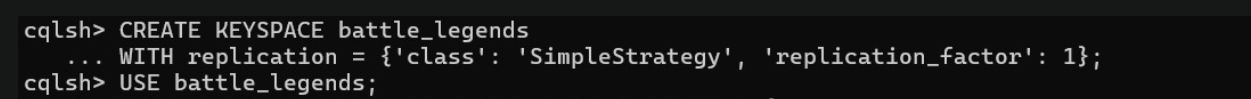
* In this project, we design and implement a scalable online gaming leaderboard system using Apache Cassandra, a column family NoSQL database.
* The leaderboard supports real-time score submissions, historical snapshots, and complex configuration management, making it suitable for massive concurrent users in modern online games.
* The system is intended to handle high volumes of data efficiently, provide fast top N queries for leaderboards, and maintain historical data for auditing and tournament replay purposes.

*Why Cassandra?*

* **Horizontal Scalability:**
  + Cassandra can easily scale out by adding more nodes to the cluster, handling millions of players and massive data volumes without performance degradation.
* **High Throughput:**
  + It supports high-speed writes and reads, which is essential for real-time score submissions and frequent leaderboard queries.
* **Flexible Schema:**
  + Cassandra’s column family data model allows for flexible, evolving schemas without downtime, enabling the addition of new fields like complex configurations and detailed match data.
* **High Availability and Fault Tolerance:**
  + With no single point of failure, Cassandra ensures the leaderboard system remains available 24/7, even during node failures or network issues.
* **Efficient Time-Series and Historical Data Handling:**
  + Cassandra naturally supports wide rows and clustering columns, making it ideal for storing historical scores and leaderboard snapshots efficiently.
* **Industry Proven:**
  + Many large-scale gaming companies use Cassandra for their leaderboard and statistics systems, proving its reliability at scale.

**Schema Design:**

*Keyspace Creation*



This creates a keyspace named battle\_legends with a simple replication strategy suitable for development and testing.

*Table: leaderboard scores*

A screen shot of a computer

AI-generated content may be incorrect.

**Purpose:** Stores individual player score submissions with detailed metadata.

**Partition Key:** (game\_mode, season\_id, snapshot\_id) groups scores by game mode, season, and leaderboard snapshot.

**Clustering Columns:** Orders scores descending to efficiently query top scores, with tie-breakers on player ID and submission time.

*Table: leaderboard snapshots*

A computer screen with white text

AI-generated content may be incorrect.

**Purpose:** Tracks metadata about leaderboard snapshots, such as creation time and configuration.

**Partition Key:** (game\_mode, season\_id) groups snapshots by game mode and season.

**Clustering Column:** snapshot\_id uniquely identifies each snapshot.

A screenshot of a computer

AI-generated content may be incorrect.

*Attribute Definition:*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| **game\_mode** | text | Identifier for the game mode, e.g., "BattleRoyale" or "TeamDeathmatch". Used to separate leaderboards by mode. |
| **season\_id** | text | Identifier for the current season or tournament, e.g., "Season2025". Allows periodic resets and fair competition. |
| **snapshot\_id** | text | Unique ID for a leaderboard snapshot, marking a specific point in time, e.g., "Snapshot\_01". Enables historical views. |
| **player\_id** | int | Unique identifier for a player. Enables ranking and player-specific queries. |
| **score** | int | The player’s score for a particular submission. |
| **submission\_time** | timestamp | The exact time the score was submitted, enabling historical and most recent queries. |
| **details** | text | Additional information stored as JSON or text, such as match details, kills, or special rules. |
| **created\_at** | timestamp | Time when a leaderboard snapshot was created. Useful for auditing and historical analysis. |
| **snapshot\_name** | text | Human-readable name for the snapshot, e.g., "End of May Tournament". Improves usability. |
| **config** | text | Configuration settings for the leaderboard snapshot stored in JSON or text, e.g., scoring rules. |

Python CRUD

A screenshot of a computer

AI-generated content may be incorrect.

**Game Data**

**1. Game Catalog**

*Database Used: Document Database*

*Tools Used: Mongo DB*

**Why MongoDB?**

* **Document Model:** 
  + Naturally represents game catalog entries and nested world layouts.
* **Flexible Schema:** 
  + Evolve game or world structure without downtime.
* **Rich Queries & Indexing:** 
  + Fast access to nested arrays and fields.
* **Atlas & Community Editions:** 
  + Simple local setup and scalable cloud option.

**Schema Design:**

* Collection Name: game\_catalog
* Document Structure

A screenshot of a computer

AI-generated content may be incorrect.

Attribute Definitions:

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Description** |
| **game\_id** | int | Unique game identifier |
| **name** | string | Human-readable game name |
| **type** | string | Abbreviated mode |
| **description** | string | Textual summary |
| **config** | object | Settings sub-document |
| **max\_players** | int | Max concurrent players |
| **map** | string | Default map or mode configuration |

A screen shot of a computer program

AI-generated content may be incorrect.

**2. World Layout Details**

* Collection Name: world\_layouts

A screenshot of a computer

AI-generated content may be incorrect.

*Attribute Definitions*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Description** |
| **world\_id** | int | Unique world identifier |
| **name** | string | World name |
| **levels** | array of obj | Sub-areas within the world |
| **level\_id** | int | Unique level identifier |
| **name** | string | Level name |
| **spawn\_points** | array | Coordinates for entry/spawn points |
| **terrain** | object | Terrain details (type & obstacles) |

**In-Game Chat:**

Database Type: Column Family

Tools Used: Cassandra

Introduction:

* The chat system supports **real-time message logging**, player communication across teams and channels and efficient retrieval of conversation history.
* The solution is optimized for high-throughput insertions and time-ordered querying essential for in-game chat during fast-paced multiplayer sessions.

**Why Cassandra?**

* **Horizontal Scalability:** 
  + Cassandra can easily scale out by adding more nodes to the  
    cluster, handling millions of players and massive data volumes without performance degradation.
* **High Throughput:** 
  + It supports high-speed writes and reads, which is essential for real-time score submissions and frequent leaderboard queries.
* **Flexible Schema:** 
  + Cassandra’s column family data model allows for flexible,  
    evolving schemas without downtime, enabling the addition of new fields like  
    complex configurations and detailed match data.
* **High Availability and Fault Tolerance:** 
  + With no single point of failure, Cassandra  
    ensures the leaderboard system remains available 24/7, even during node failures  
    or network issues.
* **Efficient Time-Series and Historical Data Handling:** 
  + Cassandra naturally supports  
    wide rows and clustering columns, making it ideal for storing historical scores and leaderboard snapshots efficiently.

**Schema Design:**

Keyspace Creation  


Table: in game chat

A screen shot of a computer program

AI-generated content may be incorrect.

**Purpose:**

* stores **real-time chat messages** exchanged between players during gameplay.

**Partition Key:**

* (game\_id, channel) Groups all messages for the same game session (game\_id) and chat channel (channel)

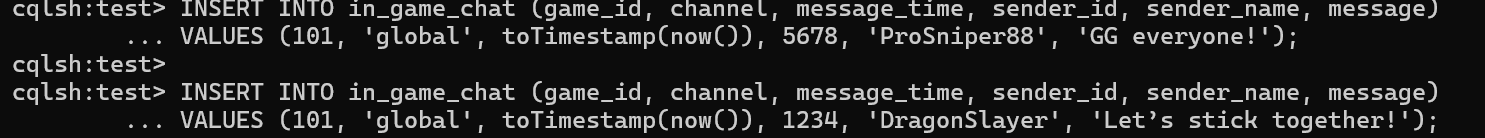
**Clustering Columns:**

* Orders messages chronologically within each game and channel

**Attribute Definition:**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| game\_id | int | ID of the game session or match |
| channel | text | Chat channel (e.g., team\_alpha, global) |
| message\_time | timestamp | Time the message was sent |
| sender\_id | int | ID of the player who sent the message |
| sender\_name | text | Player username or display name |
| message | text | The content of the chat message |

Implementation



A screen shot of a computer

AI-generated content may be incorrect.

Python CRUD:

A screen shot of a computer program

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

**Player Statistics**

Database Used: Column Family

Tools Used: Cassandra

Purpose:

* The player statistics capture comprehensive gameplay data per player, season, and match to enable flexible queries and analytics.

**Why Cassandra?**

* High scalability & availability:
  + Handles large volumes of game data distributed globally.
* Column-family model:
  + Efficiently stores denormalized time-series and hierarchical player  
    data for fast reads.
* Flexible schema:
  + Easily accommodates evolving player statistics fields without downtime.
* High write throughput:
  + Suitable for real-time ingestion of gameplay stats from thousands  
    of players.

Attribute Definitions

A table with text and numbers

AI-generated content may be incorrect.

**Primary Key and Clustering**- Partition Key: (player\_id, season\_id) allows queries scoped per player per season.  
- Clustering Key: match\_id — uniquely identifies each match per player and orders stats.  
This design supports queries like fetching all matches for a player in a specific season and  
facilitates aggregation or filtering.



A computer screen shot of white text

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

**Game Analytics:**

**Database Used: Column Family Cassandra**

**Why?**

* **High Write Throughput:** Analytics data often involves huge volumes of write-heavy logs and events, which Cassandra handles very efficiently.
* **Time-Series Data:** Player behaviors and game events naturally fit time-series data models, which column family DBs are optimized for.
* **Scalability:** Analytics data can grow very large, requiring horizontal scaling.
* **Flexible Queries:** While Cassandra is optimized for known query patterns, many teams build analytics pipelines on top and export data into OLAP solutions

**Example Cassandra Table for Game Analytics**

CREATE TABLE player\_event\_aggregates (

player\_id INT,

event\_date DATE,

event\_type TEXT,

event\_count COUNTER,

PRIMARY KEY (player\_id, event\_date, event\_type)

);

* Increment counters for each event type per player daily for fast aggregation.

**Friend List:**

Database Used: Graph Database

Tools Used: Neo4j