Course Registration System – NoSQL Concepts

ECON 485 – FALL 2025

Ramazan Kaan Erdemir

22232810024

Atılım University

Instructor: Bora Güngören

This report extends the relational course registration system designed in earlier assignments by exploring how NoSQL technologies can address performance and modeling limitations that arise at scale. The three tasks focus on key-value and document databases in the context of seat availability lookups, prerequisite eligibility caching, and complex historical action logging. Each section explains the operational problem in SQL terms and then discusses how the assigned NoSQL technology can complement the existing relational design.

Task 1- Seat Availability Lookups with a Key-Value Store (Redis)

In a purely relational implementation, seat availability for each section is computed by joining the Sections and Registrations tables and running a GROUP BY aggregation. For example, for section 101 of CS101, the system may execute a query that reads all registrations for that section, counts the number of enrolled students, and subtracts this from the section capacity. During peak registration periods, thousands of students refresh the page that lists open sections every few seconds. Each refresh repeats the same JOIN and COUNT operations, which can create heavy CPU and I/O load on the SQL server.

A key-value database such as Redis can store the number of available seats for each section as a simple integer value keyed by SectionID. For instance, when section 101 of CS101 is created with a capacity of 60, the application can set a Redis key section:101:available_seats = 60. When a student successfully enrolls, the application performs an atomic decrement, and the value becomes 59. When a student drops the course, the application performs an atomic increment, returning the value to 60. Any time a student views the list of sections, the application can simply read the integer from Redis instead of recomputing it from the Registrations table.

Redis supports atomic operations such as INCR, DECR, and DECRBY on numeric values. These operations are executed as single, indivisible steps on the server. For example, if 10 students all click “Register” for section 101 in the same second, Redis processes each DECR command one by one and updates the counter safely from 60 down to 50. No two updates interfere in a way that causes the counter to be incorrect. This atomic behavior greatly reduces the chance that two students are told “a seat is available” when, in fact, there is only one seat left. In contrast, a naive SQL implementation might read the current count in separate transactions, leading to over-enrollment without careful locking.

Using Redis as a caching and counters layer reduces load on the underlying SQL database. Most read requests for seat availability can be satisfied directly from Redis, which keeps frequently accessed values in memory and is optimized for very high throughput. For example, a section list page might show the number of available seats for 100 sections by reading 100 keys from Redis in a few milliseconds. The relational database is then used mainly for less frequent tasks, such as storing definitive registration records or running administrative reports.

This approach is particularly beneficial when the ratio of reads to writes is very high and when the system needs low-latency responses during peak demand. Computing availability directly in

SQL remains appropriate in smaller systems, or when strong guarantees are needed that the count always matches the exact state of the registration table, even in the presence of temporary cache inconsistencies. There are also operational risks. For example, if some registration path forgets to update Redis when a student drops a course, the cache may show fewer available seats than actually exist. To mitigate this, systems often run periodic reconciliation jobs that recompute seat counts from SQL and correct Redis values, or they rebuild all counters from the relational database at startup. Another limitation is memory usage: if a university has tens of thousands of sections and multiple counters per section, the design must consider how many keys will be stored and whether any eviction policy is needed.

Task 2: Prerequisite Eligibility Caching with a Key-Value or Document Store

In the Homework 2 SQL design, prerequisite eligibility is checked by joining the Prerequisites table with a CompletedCourses or Registrations table to see whether a student has passed each required course with a sufficient grade. For example, to check if student 42 is eligible for CS201, the system might query all prerequisites for CS201 and then join them with the student's completed courses to compare grades. If the same student explores many course options in one session, the system repeatedly evaluates similar JOIN queries that almost always return the same result during the semester. This pattern creates unnecessary overhead on the SQL server.

A key-value store can cache the final eligibility result for each student–course combination. For instance, after checking eligibility for student 42 and course CS201 once, the application can store a key eligibility:student:42:course:CS201 with a value such as "ELIGIBLE" or "NOT_ELIGIBLE". The next time the student opens the CS201 page, the system first checks the cache. If the key exists, it returns the cached result immediately without touching the SQL database. This is especially effective when students frequently revisit the same course pages, or when an advising tool displays a list of many courses and marks them as eligible or not based on the precomputed flags.

A document store such as MongoDB can support a richer form of caching. Instead of caching a single flag per student and course, the application can store a document that describes the full prerequisite situation. For example, a document with a key student:42:course:CS201 could contain the target course ID, an array of prerequisite objects, each including the prerequisite course ID, minimum required grade, the student's grade, and a boolean indicating whether that prerequisite is satisfied. This document can also include a top-level eligibility flag and an explanation string, such as "Missing MATH101" or "Grade in CS101 below minimum B." When the front-end needs to explain why the student is not eligible, it can read this document and display a detailed message without re-running multiple SQL queries.

Caching eligibility results in either a key-value or document-based form reduces repeated JOIN operations against the Prerequisites and CompletedCourses tables. The first time the result is computed, the system performs the necessary SQL query, but subsequent checks use the cache. For example, when generating a "recommended courses" page, the application can fetch eligibility for each listed course from the cache rather than assembling several JOIN queries per course. This reduces CPU usage and latency on the SQL server.

Keeping the cache accurate requires an invalidation or expiration strategy. When new grades are posted at the end of a semester, many eligibility results may change—for example, a student who failed a prerequisite becomes permanently "Not Eligible," while another who passed becomes eligible for several advanced courses. A common approach is to set a time-to-live (TTL) on each cache entry so that entries expire automatically after a certain number of hours or days. Another approach is event-based invalidation: whenever a grade is updated in SQL, the application deletes or updates any cache keys or documents related to that student's eligibility. For example, when student 42 receives a final grade in CS101, the system can clear all eligibility:student:42:* keys so that the next check will recompute them.

A key-value cache is ideal when the system only needs a simple, final verdict such as "Eligible" or "Not Eligible" and when storing a small amount of data per key is sufficient. It is easy to implement and efficient in terms of memory and performance. A document store is preferable when the system needs richer, structured information about the prerequisites, such as which specific course is missing or which grade is too low. For example, a student portal that displays a detailed checklist of prerequisite courses and current status ("passed", "in progress", "not taken") benefits from a document that captures the entire prerequisite chain in one place. The decision between key-value and document caching depends on whether the main focus is minimal latency for a simple flag or flexible querying and detailed explanations based on more complex data.

Task 3: Storing Complex Historical Actions in a Document Database

Appendix 1 notes that a realistic course registration system maintains a complete history of many different actions: add attempts, drop attempts, withdrawal requests, overrides, time conflict approvals, and more. Each action can have different associated metadata. For example, an override may include the instructor who approved it, the reason for the override, and the date of approval. A time conflict approval might store the conflicting section identifiers and an explanation of how the conflict will be resolved. Modeling all of these variations in a relational schema usually requires either many optional columns in a central ActionLog table or multiple specialized tables for each action type, which can become complex and difficult to evolve.

A document database such as MongoDB can represent this history more flexibly. One option is to store a single document per student that contains an array of action entries. For example, a student document might have a field actions, where each element is a JSON-like object with

fields such as actionType, timestamp, courseID, sectionID, and a nested details object. The details for an override action might include approvedBy, reason, and overrideType, while the details for a withdrawal request might include requestedOn, processedOn, and outcome. Because documents do not require a fixed schema, new fields can be added for future action types without changing table definitions or migrating existing rows.

Another design is to store each action as its own document in a dedicated collection, such as registration_history. Each document contains the student ID and the metadata for a single event. For example, an add attempt for CS101 section 01 might be a document with fields { studentID: 42, actionType: "ADD", courseID: "CS101", sectionID: "01", timestamp: "...", status: "SUCCESS" }. A time conflict approval might use a more elaborate structure with nested arrays of conflicting sections. The database does not enforce a rigid schema, so these documents can vary in shape as needed. This event-based model is close to event sourcing practices, where every change is recorded as an immutable event.

Document databases are well suited for this type of history because the workload is mostly append-heavy and read-occasionally. Each time a student performs an action, the system simply appends a new document or pushes an item into an array within a student document. Reads occur when a student views their full history or when a staff member investigates a particular issue, which is relatively infrequent compared to the number of write operations. Indexes can be created on fields such as studentID, actionType, and nested fields like details.courseID or actions.courseID. For instance, if administrators want to find all override actions for CS101, they can query an index on actionType = "OVERRIDE" and details.courseID = "CS101".

There are trade-offs compared to a purely relational design. A document store provides flexibility but offers fewer built-in tools for complex joins across collections. If an institution needs to run heavy analytical queries across millions of history records, a relational data warehouse might still be preferable. However, for operational logging of irregular actions with diverse metadata, a document database keeps the model simpler and more adaptable. It avoids schema churn when new action types or fields are introduced and reduces the need for multiple tables linked by foreign keys. A hybrid approach is often effective: the core registration data (students, courses, active registrations) remains in a relational system, while a document database is used as a history log optimized for append operations and occasional, focused reads.

REFERENCES

Redis Ltd. (2024). *Redis documentation*.

MongoDB Inc. (2023). *MongoDB manual*.

Google Cloud. (2023). *Key-value database overview*.