Task 1 – Seat Availability Lookups with a Key-Value Database

In a relational course registration system, seat availability is usually computed by joining the sections table with the enrollments table and counting how many students occupy each section. During busy registration windows, thousands of learners refresh the same screens, which means the database must repeatedly scan the registrations, perform GROUP BY operations, and calculate remaining capacity. This pattern stresses the SQL server because it mixes heavy read traffic with aggregation under high concurrency.

A key-value database such as Redis can offload this work by keeping the current number of available seats as an in-memory counter instead of recalculating it for every request. In this design, each section has a dedicated key like seats:section:<SectionKey> and the value is a single integer representing remaining seats. When sections are first created, the application can compute the initial value as MaxSeats - current enrollments from the relational tables and then store it in Redis. After that, every successful add or drop operation updates the counter directly, so most user-facing seat lookups read from Redis instead of running a SQL aggregation.

Atomic increment and decrement operations are central to this solution. Redis commands such as INCR, DECR, and INCRBY update integer values in a single, indivisible step on the server. When two students attempt to claim the last seat at the same time, Redis serializes the updates so that one decrement succeeds and the next one sees the updated value, avoiding race conditions that would otherwise allow overbooking. Under high concurrency, this property is much more reliable than trying to emulate counters with ad-hoc SELECT and UPDATE statements in SQL.

Using Redis as a caching layer significantly reduces load on the relational database. Instead of performing repeated joins and counts, the application serves the majority of seat availability requests from cheap in-memory lookups, reserving the SQL backend for core transactional operations such as finalizing enrollments and persisting long-term data. This approach is particularly attractive when the same sections are viewed many times, the number of seat changes is relatively small compared to the number of reads, and the system must respond with very low latency.

However, the key-value approach introduces its own operational risks and limitations. Redis typically keeps data in memory, so administrators must configure persistence (for example, snapshots or append-only logs) and recovery strategies to avoid losing counters after a crash. The application must also update Redis consistently whenever an enrollment is created or removed; otherwise, the cache will drift away from the relational truth. In addition, operating a Redis cluster adds complexity to deployment, monitoring, and scaling. The design works best when the organization is comfortable maintaining both a relational database and a dedicated cache infrastructure.

Task 2 – Prerequisite Eligibility Caching with a Document Store

Prerequisite eligibility is often calculated by joining a table of prerequisite rules with the table of completed courses for a particular learner and target course. Because grades are mostly static during a registration term, many students repeatedly trigger the same eligibility checks for the same combinations of courses. Running identical JOIN queries over and over produces unnecessary overhead on the SQL server.

One way to reduce this cost is to store eligibility results as cached records keyed by learner and target course. For example, the system can treat the pair (LearnerID, CourseKey) as a logical key and store whether the learner is currently “Eligible” or “Not Eligible” to register. Instead of recalculating the full JOIN every time, the application first checks the cache; if an entry exists and is fresh, it uses that decision directly. Only when there is no cache entry, or when the data may be outdated, does the application run the full SQL query and then write a new cached result.

A key-value store would represent this with a simple entry such as eligibility:<LearnerID>:<CourseKey> = "ELIGIBLE". This design is straightforward, fast, and memory-efficient, but it stores only the final decision. If the user interface needs to show details—such as which prerequisite was missing or what grade was below the threshold—the application still has to query the relational database or compute explanations separately.

A document store such as MongoDB can store richer eligibility data within a single document. For instance, one document per learner–course pair might contain fields for the learner ID, target course, an overall eligible flag, and an array of prerequisite details, including each required course, the learner’s grade, the minimum acceptable grade, and a status like “OK” or “Below Minimum.” This structure allows an application to retrieve both the high-level decision and the full explanation in one read operation, without additional joins. It also supports storing additional metadata, such as when the check was performed or which term it applies to.

Caching eligibility results reduces repeated JOIN operations because the relational query is needed only when eligibility must be recomputed. To keep the cache accurate after grade changes or rule updates, the system needs an expiration or invalidation strategy. A simple approach is to assign time-to-live values so that entries automatically expire after a certain duration, forcing a fresh check when needed. A more precise approach is event-based invalidation, where any update to a learner's grades or to prerequisite definitions triggers deletion of related cache entries. In scenarios where only a yes/no answer is required and storage needs to stay minimal, a key-value store is an efficient choice. When the system must present detailed prerequisite chains, reasons for ineligibility, or historical eligibility checks, a document database is preferable because it models hierarchical data more naturally and avoids extra queries for explanations.

Task 3 – Storing Complex Historical Actions in a Document Database

A realistic registration system must keep a complete history of events such as add attempts, drop attempts, withdrawals, overrides, and time conflict approvals. In a purely relational model, designers tend to create a generic action log table plus additional tables or columns for different action types. Over time, this leads to many optional fields, sparsely populated columns, or a proliferation of specialized log tables, because different actions require different metadata—for example, the name of an approving instructor, the conflicting section, or the reason for an override.

A document store such as MongoDB addresses this problem by allowing flexible, nested documents without a rigid schema. One common pattern is to use an event collection where each document represents a single registration event with fields like learner identifier, action type, course and section information, timestamp, and a metadata subdocument. The metadata can hold any key-value pairs relevant to that specific action, such as {"advisorApproved": true, "overrideReason": "Graduating student"} for an override, or {"conflictWithSection": 1002, "conflictType": "time overlap"} for a time conflict approval. When new action types appear, the system can simply start adding new metadata fields without altering table definitions or migrating existing rows.

Another option is to store a learner’s entire registration history in a single document that contains an array of event objects. In this learner-centric model, one document includes the learner’s ID and a chronologically ordered list of actions, each with its own type, timestamp, and metadata. This design makes it easy to load the complete history needed for audits or advising in a single query. It also matches the append-heavy nature of these logs, because most operations only add new events rather than updating existing ones.

Document databases are well-suited for workloads where writes are frequent but reads are less common and often focused on specific learners or time periods. They can efficiently append new events, and they provide indexing on top-level fields such as learner ID or action type as well as on nested metadata fields. For example, an index on events.actionType or metadata.advisorApproved enables fast queries like “find all overrides approved by a given advisor” or “list all time conflict approvals this semester.” The main trade-off compared with a relational approach is that document stores sacrifice some aspects of rigid normalization and cross-document constraints. Relational databases excel at enforcing strict referential integrity and running complex joins across many tables, but modeling heterogeneous, evolving event structures in SQL becomes cumbersome. Using a document database for historical logs, while preserving relational tables for core transactional data, strikes a balance between flexibility and consistency that fits the behavior of real registration histories.