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

In a traditional SQL-based course registration system, seat availability for each section is typically computed by joining the Sections table with the Registrations table and performing a GROUP BY aggregation to count registered students. During peak registration periods, thousands of students repeatedly query the same availability information, which forces the SQL server to execute many read and aggregation operations on the same data. This can create significant CPU and I/O load, especially when sections and registrations grow large.

A key-value database such as Redis can be used to maintain seat availability as an in-memory counter instead of recomputing it with every SQL query. In this approach, each course section is represented by a key like seats:section:<SectionID> and the value is the current number of available seats. The initial value can be computed once from the relational database as capacity - current registrations and then written into Redis. After that, all add and drop actions update the Redis counter directly rather than recalculating availability through SQL joins.

Atomic increment and decrement operations are a core feature of Redis and similar key-value stores. When a student successfully adds a section, the application issues an atomic decrement, such as DECR seats:section:3, and when a student drops a section, it issues an atomic increment like INCR seats:section:3. Because these operations are atomic, Redis guarantees that concurrent updates from many clients will be applied one at a time, preventing race conditions where two students might see the same last seat as available. This ensures that the seat counter remains internally consistent even under high concurrency.

Placing Redis as a caching layer in front of the SQL database significantly reduces the load on the relational system. Instead of querying the Sections and Registrations tables and performing GROUP BY calculations on every page load, the application can serve seat availability directly from Redis with a simple key lookup. The SQL database is still the system of record for persistent registration data, but it no longer needs to handle the high-frequency, read-heavy workload of repeated availability checks.

This approach is preferred over computing availability directly in SQL when the following conditions hold: registration traffic is high, the same sections are queried repeatedly, and latency requirements are strict. In such scenarios, computing seat counts dynamically with SQL aggregations is unnecessarily expensive compared to reading a cached integer from memory. However, using a key-value store also introduces operational risks and limitations. Redis is primarily memory-based, so administrators must ensure persistence and recovery strategies in case of crashes. If application logic forgets to update the Redis counter on certain add or drop operations, the cache can diverge from the underlying SQL truth. There is also additional operational complexity, because the system must deploy, monitor, and scale a separate Redis cluster alongside the relational database.

Task 2 – Prerequisite Eligibility Caching with Key-Value or Document Stores

Prerequisite eligibility in the relational system is typically determined by joining the Prerequisites and CompletedCourses tables for a given student and target course. The query checks which courses are required, what minimum grades are needed, and what grades the student has actually earned. Because students often check eligibility for multiple potential courses and their completed grades rarely change during a registration period, repeatedly running the same joins can create unnecessary overhead on the SQL server.

A caching strategy can store the eligibility result for each student and course as a precomputed value. In the simplest key-value design, the system uses a key such as eligibility:<StudentID>:<CourseID> and a value that indicates whether the student is currently eligible, for example "ELIGIBLE" or "NOT_ELIGIBLE". When a student first checks eligibility for a target course, the application runs the full SQL query with joins between Prerequisites and CompletedCourses, computes the result, and then writes it into the cache. Subsequent eligibility checks for the same student and course can be served directly from the cache without hitting the relational database again.

Using a key-value store like Redis for eligibility caching keeps the model very simple. Each key corresponds to one student–course pair, and the value is a single flag or small object. Lookups are extremely fast, and the memory footprint is small. However, this design only stores the final decision and does not preserve detailed information about which prerequisite courses were passed or failed. If the user interface needs to explain “you are not eligible because you did not pass MATH101 with a sufficient grade,” the application must still run a SQL query or perform a secondary lookup.

A document store such as MongoDB can store richer eligibility information in a single document. For example, a document might contain the student ID, the target course ID, the overall eligibility flag, and an array of details describing each prerequisite: the prerequisite course, the student's grade, the minimum required grade, and a status indicating whether the requirement is satisfied. This allows the application to retrieve both the summary decision and detailed explanations with a single document read, without further joins. It also enables the system to store historical eligibility evaluations over time if needed.

Caching eligibility results reduces repeated JOIN operations against CompletedCourses and Prerequisites because the relational query is only executed on the first check or when the underlying data changes. To keep cached eligibility accurate after grade updates, the system must implement expiration or invalidation strategies. One approach is time-based expiration, where keys are given a time-to-live and automatically expire after a registration period or a fixed number of hours. Another approach is event-based invalidation: whenever a student's grade record changes, the application deletes all cached eligibility entries for that student so that they will be recomputed on the next access.

A key-value cache is preferable when the system only needs a fast yes/no eligibility answer and when storage simplicity is more important than rich detail. A document store is preferable when the application needs to display detailed reasons for eligibility decisions, support more complex reporting, or store multiple related pieces of eligibility information together. In such cases, MongoDB or a similar document database provides more expressive modeling at the cost of slightly higher storage and retrieval complexity.

Task 3 – Storing Complex Historical Actions in a Document Database

Appendix 1 describes that a real course registration system must keep a complete history of actions such as add attempts, drop attempts, withdrawal requests, overrides, and time conflict approvals. In a purely relational design, modeling this history often requires a main ActionLog table plus additional tables or columns to capture action-specific metadata. Different action types require different fields, for example approver names for overrides, conflicting section identifiers for time conflict approvals, or free-form reasons for withdrawals. Over time, the relational schema can become cluttered with many nullable columns or a large set of specialized tables.

A document store such as MongoDB offers a more flexible way to represent registration history by allowing variable and nested structures. One common approach is the event collection model, where each historical action is stored as a separate document in a collection like registration_events. Each document includes standard fields such as student identifier, action type, course and section identifiers, timestamp, and a nested metadata object that can hold arbitrary key–value pairs. For example, an override event might include metadata fields for advisorApproved and reason, while a time conflict approval might store conflictingSectionId and conflictMinutes. The schema does not need to be changed when new types of actions or metadata are introduced.

Another approach is a student-centric model, in which each student has a document in a students_history collection that contains an array of embedded event objects. Each event entry records the type of action, timing, and any relevant details. This model makes it easy to retrieve the entire history of a single student with one query, which is useful for audits, appeals, or advising. Both models benefit from MongoDB's ability to store variable-length arrays and deeply nested objects without predefined rigid schemas.

Document databases are particularly well suited to append-heavy, read-occasionally workloads like registration history logs. Most operations involve appending new events as students perform actions, while read access is less frequent and often scoped to a specific student or case. In a relational system, this pattern can create large, heavily indexed tables that are costly to maintain and query with multiple joins. In contrast, MongoDB can handle large collections of documents with efficient appends and flexible indexing on fields such as studentId, actionType, and timestamp. Indexes can also be defined on nested metadata fields, enabling fast queries like "find all overrides approved by a specific advisor" or "find all time conflict approvals for a particular course."

There are trade-offs between using a document store versus maintaining multiple relational tables for historical actions. Relational databases provide strong consistency guarantees, normalized schemas, and well-understood SQL query capabilities, which are valuable for core transactional data. However, they handle evolving, heterogeneous metadata less gracefully and often require schema changes or complex join graphs as new action types are introduced. Document databases relax schema rigidity in exchange for greater modeling flexibility and simpler write patterns, making them an attractive choice for storing registration history as a secondary system optimized for audit, logging, and analysis rather than for primary transactional updates.