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

In the relational version of the registration system, the number of remaining seats in each section is recomputed by joining the sections table with the registrations table and counting rows for every lookup. When many students are online at the same time, this means the database repeatedly reads the same rows and performs the same GROUP BY aggregations, which is wasteful and can slow down the entire system.

A key-value database such as Redis offers a very different pattern: instead of calculating availability from scratch, the system keeps a running "available seats" counter in memory for each section. Each section can be mapped to a key like section:2001:available_seats, whose value is a single integer. When a new term opens, the application initializes these counters based on the relational data as capacity − current registered students. Afterwards, every successful registration decrements the relevant counter, and every successful drop increments it again.

The key to making this safe under heavy concurrency is Redis's atomic counter operations such as INCR, DECR, and INCRBY. These commands run entirely on the server and guarantee that two overlapping updates cannot interleave in a way that corrupts the value. For example, if two students both try to claim the last seat, only one DECR will bring the counter from 1 to 0; the other will see the updated value and can be rejected at the application layer. This removes the race conditions that commonly occur when using a manual SELECT-then-UPDATE sequence in SQL.

By placing Redis in front of the relational database as a cache, the system turns most availability checks into extremely cheap in-memory reads. Only the authoritative registration state remains in the SQL tables, which still handle constraints, foreign keys, and long-term persistence. This hybrid architecture works especially well when seat availability is read far more often than it is updated, when registration load spikes are short but intense, and when low latency is important for the user experience.

There are trade-offs and operational risks. Redis is typically memory-resident, so the deployment must be configured with persistence options and backup strategies to prevent losing counters after crashes. The application must also be carefully designed so that every successful add or drop operation updates both Redis and the underlying database; any mismatch will cause the cached availability to drift away from reality. Finally, running a separate Redis cluster increases operational complexity compared with a pure SQL solution, so this approach makes the most sense in environments that already rely on caching infrastructure or can justify the added overhead.

Task 2 – Caching Prerequisite Eligibility with a Document Store

Checking whether a student is allowed to register for a course normally involves combining prerequisite definitions with the student's completed courses and grades. In SQL terms, this means joining a prerequisite table with a completed-courses table for a specific student and target course. Because grades change rarely within a semester, running the same JOINs repeatedly for the same student and course pair is inefficient.

Caching offers a way to avoid unnecessary work. One strategy is to treat each student–course combination as a distinct key and store the result of the prerequisite check as a value. For example, a key could be eligibility:student:<StudentID>:course:<CourseID>, and the value could be a simple flag such as "ELIGIBLE" or "NOT_ELIGIBLE". When the application needs to know if a student can take a course, it first checks the cache and only falls back to the full SQL JOIN if no entry exists or if the data has expired.

A key-value store implements this pattern in the simplest way: it maintains only the precomputed decision. This is fast and compact, but it does not preserve any context. If the user interface needs to show why a student is ineligible—such as which prerequisite is missing or which grade is too low—the system must still query the relational tables or compute that explanation separately.

A document store like MongoDB can capture both the decision and the detailed reasoning in a single document. One document per student–course pair might contain the student ID, the target course, an overall eligibility boolean, and an array of objects describing each prerequisite: prerequisite course code, required minimum grade, the student's grade, and a status field. With this model, the application can load everything it needs with a single document read: whether the student is allowed to register and a human-readable breakdown of each prerequisite.

Caching prerequisite results reduces repeated JOIN operations because the relational database only needs to be consulted when the underlying information changes or when there is no cached entry yet. To keep the cache consistent after grade changes or rule updates, the system needs explicit invalidation. One common approach is to assign a time-to-live to each cached document so that it expires automatically after a set period. A more precise solution is to delete or update cache entries whenever new grades are

posted or prerequisite rules are modified; for instance, the grade-entry process can remove all cached eligibility documents for the affected student. In scenarios where only a yes/no answer is needed and storage must stay minimal, key-value caching is enough. When detailed explanations, historical checks, or more complex eligibility reporting are required, a document store provides a richer and more convenient representation.

Task 3 – Using a Document Database for Complex Historical Actions

A full registration system needs more than a snapshot of who is currently enrolled; it must record how that state came to be. This includes successful and failed add attempts, drops, withdrawals, overrides, changes between sections, and approvals for time-conflict exceptions. Trying to capture every variation of these events in a single relational log table often leads to many nullable columns and auxiliary tables, because each action type may need different attributes—such as an approval officer name, a reason for an override, or IDs of conflicting sections.

Document databases such as MongoDB handle this kind of irregular, evolving data more naturally. One pattern is to maintain a dedicated collection of event documents, where each document represents a single action with fields for student identifier, event type, course and section, timestamp, and an embedded details object containing arbitrary metadata. An override event might store an advisor's name and justification, whereas a time conflict approval event might store schedule information and the IDs of both sections involved. Because the schema is flexible, the application can begin recording new kinds of metadata without altering existing table definitions or performing schema migrations.

Another pattern is to organize history around the student, keeping a separate document per student with an array of events. Each event element in the array captures the type of operation, when it occurred, and any relevant attributes. This student-centric structure allows the system to retrieve a complete timeline of a student's registration activity in a single query, which is useful for advising sessions and appeals. Since historical writes are append-only in most cases, such workloads match document stores well: writes are frequent, but reads are less common and are usually scoped to a subset of students or time frames.

Document databases support indexing on top-level fields like student ID and event type, and many of them also allow indexes on nested fields inside embedded documents. This enables efficient queries such as “find all override events approved by a specific advisor” or “list all time conflict approvals for ECON230 this year” without restructuring the schema. The trade-off compared with a purely relational implementation is that cross-document joins and strict normalization are less central; instead, each document is more self-contained. For a registration history log, this is usually acceptable because the log is read occasionally and used primarily for auditing and analysis. A common architectural choice is to keep the main registration state in a relational database and offload the complex, heterogeneous history data to a document store, combining the strengths of both models.