Task 1 – Seat Availability with a Key-Value Store

In the relational implementation of the registration system, the remaining seats in each section are typically obtained by joining the sections table with the registrations table and running an aggregate to count how many students are currently enrolled. When many students are online at the same time, the database ends up performing the same join and GROUP BYcomputation repeatedly for identical queries, which wastes resources and can slow down other operations.

A key-value database such as Redis offers an alternative pattern: instead of recomputing availability from base tables every time, the system keeps a running counter for each section directly in memory. Each section can be associated with a key like section:3001:available_seats, and the value stored for that key is simply an integer representing the number of seats left. At the start of the term, the application initializes these counters using the authoritative SQL data, for example by setting the value to capacity − current registrations for each section.

Once initialized, the application updates these counters whenever a registration or drop is confirmed. A successful registration triggers a decrement on the corresponding key, and a successful drop triggers an increment. Redis provides built-in counter commands such as INCR, DECR, and INCRBY that execute atomically on the server. This means that concurrent updates cannot interleave in a way that corrupts the stored value; even if multiple students try to take the last seat at the same time, only one decrement will transition the counter from 1 to 0, while others will see the updated value and can be rejected by the application.

By placing Redis in front of the SQL database as a caching layer, the system turns most seat availability checks into inexpensive in-memory reads. The relational database still holds the official registration records and enforces constraints, but it no longer needs to run heavy aggregations for every lookup. This combined design is especially effective when seat information is read much more frequently than it is updated, when registration load comes in short but intense bursts, and when low latency is important for the user experience.

However, there are operational trade-offs to consider. Redis usually stores data in memory, so persistence and backup must be configured to avoid losing counters in the event of a crash. The application code must also carefully ensure that every successful add and drop is reflected both in Redis and in the SQL database; if one of the updates fails or is forgotten, the cached seat counts will diverge from reality. Running and monitoring a separate Redis deployment adds complexity compared with a pure SQL solution, so this pattern makes the most sense in systems where caching is already used or where the performance benefits clearly justify the overhead.

Task 2 – Caching Prerequisite Eligibility with a Document Store

In the basic SQL design, checking whether a student satisfies the prerequisites for a course requires combining prerequisite rules with the student’s completed courses and grades. Practically, this means joining a prerequisites table with a completed-courses table for a given student and target course and then evaluating each condition. Since course grades are relatively stable during a semester, running the same joins repeatedly for the same student–course pairs is inefficient.

Caching can significantly reduce this redundant work. One straightforward strategy is to treat each (student, course)combination as a unique key and store the outcome of the prerequisite check as the associated value. A key could look like eligibility:student:<StudentID>:course:<CourseID>, and the value could be something simple such as "ELIGIBLE" or "NOT_ELIGIBLE". When the application needs to know whether a student can take a course, it first looks up this key in the cache; only if there is no entry or the entry is considered stale does it fall back to executing the full SQL query.

A pure key-value store works well for this minimalist approach because it stores only the final decision and keeps each lookup extremely fast and compact. The downside is that it does not contain any explanation. If the user interface needs to show which prerequisite course is missing or which grade is too low, the system must either query SQL again or store more detailed information somewhere else.

A document database such as MongoDB can keep both the eligibility result and the reasoning in a single document. For example, one document per (student, course) pair might hold the student identifier, the target course, an overall eligibility flag, and an array describing each prerequisite. Each element in the array could include the prerequisite course code, the minimum required grade, the student’s actual grade, and a status field indicating whether the condition is satisfied. With this model, a single read operation provides enough data both to decide whether the student may register and to build a human-readable explanation for the interface.

Caching prerequisite outcomes reduces the frequency of joins between the prerequisites and completed-courses tables, since the relational database is only consulted when there is no cached entry or when underlying information changes. To keep the cache accurate after grades are updated or prerequisite rules are modified, the system needs an invalidation strategy. A simple method is to assign a time-to-live to each cached record so that it expires automatically after a certain period. A more targeted approach is to delete or refresh all cached entries related to a student whenever new grades are posted or when prerequisite definitions for a course change. This way, subsequent eligibility checks will recompute and store fresh results.

Task 3 – Representing Complex Historical Actions in a Document Database

Appendix 1 describes that a real registration system maintains a detailed history of various actions, such as add attempts, drop attempts, withdrawal requests, overrides, and time-conflict approvals. Each of these action types may require storing different kinds of data: some need instructor names and approval timestamps, others involve conflict details, reasons typed by the student, or multiple related sections. Modeling all these variations in a strictly relational schema often leads to many optional columns or to a large set of specialized tables linked together by foreign keys.

A document database like MongoDB can handle this diversity more naturally. One option is to store each student's entire registration history as a single document containing a list of event objects. Each event can have a common core—such as an action type, course identifier, and timestamp—but may also carry extra fields that only apply to that event type. For example, an "ADD" event might include section information and the channel used (web or mobile), while an "OVERRIDE" event might include the approving advisor, a free-text justification, and details of the conflict. Because the document store does not enforce a rigid column structure, each event can include exactly the fields it needs without requiring schema migrations when a new action type appears.

Another approach is to represent the history as an event collection, where each document corresponds to a single action. In this case, documents share some basic fields but can still differ in terms of optional or nested attributes depending on the action. Document databases support indexing on nested and optional fields, which makes it possible to execute queries like "find all overrides in the last year" or "list all actions involving time-conflict approvals for a given course” efficiently even though not every document has the same shape.

Document stores are particularly suitable for history logs because this workload is typically append-heavy and read relatively infrequently. New events keep being appended as students interact with the system, but full historical reports are generated only when needed for audits, troubleshooting, or analytics. Storing these logs in a document database reduces the need for complex joins between multiple relational tables every time a long history needs to be reconstructed.

There are trade-offs, however. As a student accumulates more actions, the corresponding history document can grow large, and the system may need to split records or adopt an event-per-document strategy to avoid hitting size limits. In addition, the flexibility of a document schema can make it harder to guarantee strict consistency and to enforce simple constraints compared with relational tables. For these reasons, many systems keep core transactional data such as active registrations and grades in a relational database while using a document store as a complementary system for rich, semi-structured historical logs. This hybrid approach takes advantage of the strengths of both models: strong structure where it matters and flexibility where the data is irregular.