NoSQL деректерді модельдеу принциптері

Реляциялық дерекқорларда (SQL) деректер көбіне кестелерге бөлінеді, нормализация жасалады, ал байланыстыру үшін JOIN қолданылады. Бұл тәсіл құрылымы қатаң жүйелерге өте ыңғайлы.

Ал MongoDB – **құжатқа бағытталған** дерекқор. Мұнда деректер **құжат** (document) түрінде сақталады (BSON/JSON форматына ұқсас). MongoDB-дегі ең маңызды сұрақ: “Қосымша бұл деректерді қалай оқиды және қалай жазады?”

Яғни MongoDB-де модельдеу көбіне:

- ✓ қай экран қандай деректерді көрсетеді,
- ✓ қандай сұраныс жиі орындалады,
- ✓ қандай деректер бірге өзгереді,
- ✓ қандай деректер уақыт өте өте үлкен болып өседі деген сұрақтарға жауап беруден басталады.

Query Patterns: модельдеудің бастауы

Query pattern – жүйеде ең жиі қайталанатын сұраныстардың “шаблоны”

Қалай анықтаймыз:

- қолданбадағы негізгі экрандар/есептер
- filter/sort/pagination талаптары
- оқу (read) жиілігі мен жазу (write) жиілігі

Мысалдар (университет):

- Студент профилі (бір студенттің толық ақпараты)
- Пән журналы (курс → студенттер → бағалар)
- Топ бойынша тізім + GPA сұрыптау
- Баға қосу, қатысуды белгілеу

Принцип 1: Query-driven design (сұранысқа негізделген дизайн)

❑ Нені ескеру керек:

•бір экранға мәлімет шығару үшін 1–2 сұраныс жеткілікті болуын;

•жиі filter/sort өрістері индекстелетіндей құрылым болуын;

•күрделі $lookup тұрақты қолданудан сақ болуды.

Бірге оқылады/бірге өзгереді бірге сақталады

Әрдайым жұмыс істейтін қағида:

- Егер деректер *әрдайым бірге көрсетілсе* → embedded ойлау;
- Егер деректер *бөлек қолданылса* немесе *көп жерде ортақ болса* → references ойлау.

Мысал:

- Студенттің телефон, email, адресі → профильде бірге көрсетіледі → embedded;
- Пән (Course) көптеген студентке ортақ → бөлек коллекция + reference.

Atomicity (құжат ішінде атомарлық)

Принцип 3: Мағынасы: бір құжатқа жасалған өзгеріс толық орындалады немесе орындалмайды

- **Неге маңызды:**

- профиль деректері бір құжатта болса → қауіпсіз жаңарту
- “бір операцияда бірнеше өріс өзгеруі тиіс” сценарийлері үшін ыңғайлы

16MB шектеуі және өсетін массивтер проблемасы

Қауіпті сценарийлер:

- студенттің барлық қатысу тарихы embedded массивте
- пайдаланушы әрекет логы бір құжатта
- чат хабарламалары бір құжатта

Шешімдер:

- бөлек коллекция (references)
- bucket/ракеттеу (күн/апта/ай бойынша топтау)
- subset (тек соңғы 10 жазбаны embedded)

Read-heavy vs Write-heavy балансы

- **Read-heavy (оқу көп):**
  - embedded және денормализация көбірек;
  - мақсат: бір сұраныспен тез көрсету.

- **Write-heavy (жазу көп):**
  - references көбірек;
  - мақсат: жаңарту оңай, қайталау аз.

Принцип 6: Индекстерді модельдің бір бөлігі ретінде қарау

❑ **Индекс керек болатын жағдайлар:**

•жиі filter: group, studentId, courseId, date;

•жиі sort: gpa, createdAt;

•пагинация және іздеу.

- **Қорытынды:**
- MongoDB модельдеу = сұранысқа бейімделген дизайн
- Embedded/Reference – “бірге оқылады/бірге өзгереді” қағидасымен таңдалады
- 16MB және өсетін массивтерді алдын ала жоспарлау керек
- Индекстерсіз модель толық емес
  Тапсырма (10 минут):
- 3 query pattern жазыңыз (“электронды журнал”)
- embedded қайда? reference қайда?
- 2 индекс ұсыныңыз және не үшін екенін түсіндіріңіз
- **Спикер мәтіні:**
  Келесі тақырыптарда біз осы принциптерді нақты құралдармен бекітеміз: embedded құрылымдар, references түрлері, денормализация және үлгі-шаблондар. Ал қазір шағын тапсырмамен практика жасайық.

Кірістірілген құжаттар (embedded documents)

- MongoDB – құжатқа бағытталған дерекқор, яғни жүйедегі ақпаратты кестелерге бөліп, кейін оларды JOIN арқылы құрастырудан гөрі, бір объектіге қатысты мәліметті бір құжаттың ішіне жинақтап сақтауға бейім. Embedded құжат дегеніміз – негізгі құжаттың ішінде қосымша ішкі объектілерді немесе объектілер массивін сақтау тәсілі. Бұл тәсілдің мәні өте қарапайым: егер ақпарат қолданбада әрдайым бірге көрсетілсе және көбіне бірге өзгертілсе, онда оны бөлек коллекцияларға бөліп, байланыс арқылы жинаудың орнына бір құжаттың ішінде ұстау тиімді болады.

Кіріктіру тәсілінің негізгі мақсаты – қосымшаның ең жиі орындалатын оқу сценарийлерін жылдамдату. Қолданба белгілі бір экранды ашқанда, көбіне бір объектіге қатысты толық мәліметті көрсетеді: мысалы, студент профилі, клиент карточкасы, тапсырыс мазмұны. Егер осы ақпарат бірнеше коллекцияға бөлініп кетсе, жүйе бір объектіні көрсету үшін бірнеше сұраныс жасауы мүмкін.

Ал embedded қолданылса, бір сұраныспен-ақ құжат толық келеді. Бұл желі арқылы берілетін сұраныстар санын азайтады, кодты қарапайым етеді, күту уақытын төмендетеді және жалпы өнімділікті арттырады.

Embedded-тің басты артықшылығы: бір сұраныспен толық нәтиже

- Кірістірілген құжаттарды қолданғанда деректерге қол жеткізу жылдамырақ болады, себебі “объект + оған қажет қосымша мәлімет” бір құжатта орналасады. Мысалы, студенттің контактілері, адресі, оқу тобы сияқты деректер профильді ашқанда бірден керек. Егер олар бөлек коллекцияда болса, қосымша сұраныс қажет болады немесе агрегаттау арқылы біріктіру керек болады.

“Бірге оқылады және бірге өзгереді” қағидасы

- Embedded таңдаудың негізгі критерийі – деректердің қосымшада қалай қолданылатыны. Егер екі немесе бірнеше дерек жиынтығы әрдайым бірге көрсетілсе және көбіне бірге жаңартылса, оларды бір құжатқа біріктіру орынды. Мысалы, студенттің email-і өзгерсе, ол студент профилінің бөлігі ретінде қарастырылады; адресі жаңарса да солай. Мұндай деректер тәуелсіз объект емес, негізгі объектінің құрамдас бөлігі. Сондықтан embedded тәсілі деректерді “агрегат” ретінде сақтауға мүмкіндік береді: бір объект – бір құжат, оның құрамындағы бөлшектер – ішкі құжаттар.

1:1 сценарийі: embedded-тің ең дұрыс қолданылуы

- Embedded ең “таза” және қауіпсіз түрде 1:1 қатынаста қолданылады. Бұл жағдайда ішкі деректер негізгі объектіден бөлек өмір сүрмейді, басқа жерде қайта қолданылмайды және көлемі шағын болады. Мысалы, contacts (телефон, email), address (қала, көше), preferences (тіл, хабарлама баптаулары) сияқты мәліметтер бір адамға ғана тиесілі және басқа объектілермен ортақ емес. Мұндай деректерді бөлек коллекцияға шығару жүйені күрделендіріп, оқу кезінде қосымша сұраныстарды көбейтеді. Сондықтан 1:1 үшін embedded – әдетте ең тиімді модель.

1:N сценарийі: “шағын N” болғанда embedded қолайлы

- Embedded тәсілі 1:N қатынасында да тиімді болуы мүмкін, бірақ мұнда ең маңызды сұрақ – “N өсіп кете ме, жоқ па?”. Егер элементтер саны табиғи түрде аз және шектелетін болса, массив ретінде ішке сақтау өте ыңғайлы. Мысалы, студент білетін тілдер, қысқа тізім түріндегі тегтер, бірнеше байланыс тәсілі немесе бірнеше шағын сертификат. Мұндай деректер профильді ашқанда қатар көрсетіледі және жүйеге қосымша жүктеме түсірмейді. Ал егер N жүздеген, мыңдаған элементке дейін өсуі мүмкін болса, онда embedded қауіпті шешімге айналады.

Embedded массивтер: сұраныс пен жаңарту ерекшелігі

- Ішкі массивтермен жұмыс MongoDB-де өте белсенді қолданылады. Құжат ішінде массив элементін іздеу, белгілі бір шартқа сай ішкі объектілерді табу, массивке элемент қосу немесе жою сияқты әрекеттер жиі орындалады. Бұл жерде маңыздысы – массив құрылымын сұраныстарға ыңғайлап жасау. Егер сіз массив ішінен жиі іздей беретін болсаңыз, индекстеу мәселесі пайда болады: кей жағдайларда MongoDB массив өрістерге мульти-кілтті индекс құра алады, бірақ күрделі ішкі объектілерде дұрыс индекстеу алдын ала ойластырылуы керек. Яғни embedded тек сақтау емес, сұраныс логикасымен бірге қарастырылатын құрылым.

Негізгі қауіп: құжаттың “ұлғаюы” және 16MB лимиті

- MongoDB құжатының максималды өлшемі 16MB. Бұл шектеу embedded модельдеуді жоспарлауда шешуші рөл атқарады. Егер сіз құжат ішіне уақыт өте үздіксіз қосыла беретін деректерді жинай берсеңіз, құжат бір күні лимитке тірелуі мүмкін. Мысалы, студенттің қатысу тарихын, барлық бағаларын, барлық әрекет журналын немесе чат хабарламаларын бір құжатта сақтау бастапқыда өте ыңғайлы көрінеді. Бірақ дерек жиналған сайын құжат көлемі өседі, оқу/жазу операциялары баяулайды, ал лимитке жеткенде жүйе қате қайтарады. Сондықтан embedded қолданғанда “өсуі шектелмейтін деректер” деген сигналды ерте танып үйрену керек.

Embedded бойынша жиі кездесетін қателер

- Embedded қолданудағы ең жиі қателік – өсетін деректі ішке жинау.
- Екінші қате – “бәрін бір құжатқа тықпалау”, яғни өте үлкен “god document” жасау.
- Үшінші қате – массивтерге индекстерді дұрыс жоспарламау, нәтижесінде іздеу баяулайды.
- Тағы бір қате – бірдей ақпаратты әр жерде қайталап, кейін оны синхрондауды ойламау.

Мұндай қателер көбіне жүйе “кішкентай кезде” байқалмайды, бірақ дерек көлемі ұлғайғанда өнімділік пен қолдауға қиындық туғызады.

Өсетін деректерге шешімдер: reference, bucket, subset

Егер дерек өсетіні анық болса, оны embedded ретінде сақтауға болмайды. Мұндай жағдайда үш негізгі стратегия қолданылады.

- Біріншісі – деректі бөлек коллекцияға шығару, ал негізгі құжатта тек идентификатор немесе байланыс ақпаратын сақтау.
- Екіншісі – bucket тәсілі: мысалы, қатысу жазбаларын апта/ай бойынша топтап, әр құжатта белгілі бір уақыт аралығына қатысты элементтер ғана болатындай ұйымдастыру.
- Үшіншісі – subset тәсілі: негізгі құжатта тек соңғы 10–20 жазбаны сақтау, ал толық архив бөлек коллекцияда тұрады. Бұл әдістер embedded-тің ыңғайын сақтай отырып, құжаттың бақылаусыз өсуін тоқтатуға көмектеседі.

Өсетін деректерге шешімдер: reference, bucket, subset

Егер дерек өсетіні анық болса, оны embedded ретінде сақтауға болмайды. Мұндай жағдайда үш негізгі стратегия қолданылады.

- Біріншісі – деректі бөлек коллекцияға шығару, ал негізгі құжатта тек идентификатор немесе байланыс ақпаратын сақтау.
- Екіншісі – bucket тәсілі: мысалы, қатысу жазбаларын апта/ай бойынша топтап, әр құжатта белгілі бір уақыт аралығына қатысты элементтер ғана болатындай ұйымдастыру.
- Үшіншісі – subset тәсілі: негізгі құжатта тек соңғы 10–20 жазбаны сақтау, ал толық архив бөлек коллекцияда тұрады. Бұл әдістер embedded-тің ыңғайын сақтай отырып, құжаттың бақылаусыз өсуін тоқтатуға көмектеседі.

Құжаттар арасындағы байланыстар (references)

Құжаттар арасындағы байланыстар: негізгі ұғым

MongoDB-де байланыс (reference) дегеніміз – бір құжаттың басқа құжатты тікелей “ішіне салмай”, тек оның идентификаторы арқылы көрсету тәсілі. Яғни негізгі құжаттың ішінде басқа объектінің толық сипаттамасы емес, сол объектіге апаратын ObjectId немесе басқа бір тұрақты идентификатор сақталады.

Бұл тәсіл реляциялық БД-дегі сыртқы кілтке ұқсайды, бірақ MongoDB-де байланыс автоматты JOIN арқылы емес, қолданбаның логикасы немесе aggregation құралдары арқылы іске асырылады. References қолдану арқылы біз деректерді қайталамаймыз, ортақ объектілерді бір жерде сақтап, оларды көптеген құжаттарға “сілтеме” ретінде пайдаланамыз.

- Reference тәсілінің басты мақсаты – деректердің көлемі үлкейгенде және объектілер бір-бірімен ортақ болғанда жүйені басқаруды жеңілдету. Егер бір объект көптеген құжаттарда кездессе (мысалы пән, оқытушы, өнім, бөлім), оны әр құжаттың ішіне көшіре беру деректердің қайталануына әкеледі. Мұндай қайталау кейін синхрондау мәселесін тудырады: бір жерде өзгеріс болса, барлық көшірмелерді жаңарту керек болады. Reference бұл проблеманы азайтады, себебі “ақиқат көзі” бір коллекцияда сақталады, ал басқа құжаттар тек сол объектінің id-сына сүйенеді. Сонымен қатар өте үлкен массивтерді негізгі құжаттың ішіне салмай, бөлек коллекцияға шығару арқылы 16MB лимиті мен “құжаттың ұлғаюы” мәселесінен құтыламыз.

One-to-many

- One-to-many байланыс дегеніміз – бір объект көптеген басқа объектілермен байланысты. Университет мысалында “бір студенттің бағалары көп”, “бір пәнге көптеген студент тіркелген”, “бір оқытушы бірнеше пән жүргізеді”. MongoDB-де one-to-many байланысты жасаудың екі жиі тәсілі бар. Біріншісі – “many” жақтағы құжаттарда “one” жақтың идентификаторын сақтау, яғни баға жазбаларында studentId және courseId сақтау сияқты. Екіншісі – “one” жақта байланысты id-ларды массив түрінде сақтау, мысалы студент құжатында courseIds: [...] деп көрсету. Қайсысы дұрыс болатыны деректің көлеміне және сұраныс үлгілеріне байланысты: егер курстар саны шектелсе, id-массив ыңғайлы; ал егер байланыс өте көп және үнемі өссе, онда “many” жақта id сақтау тұрақтырақ шешім.

Many-to-many

- Many-to-many дегеніміз – екі жақ та көптеген байланыстарға ие, мысалы “студенттер көптеген пән алады, пәнде көптеген студент бар”. Мұны MongoDB-де көбіне екі тәсілмен шешеді. Бірінші тәсіл – екі жақта да id массив сақтау: студентте courseIds, ал курста studentIds. Бірақ мұнда массив өсімі мен жаңарту қиындығы пайда болады, сондықтан бұл тәсіл тек байланыс саны аз және тұрақты болғанда тиімді. Екінші, кең таралған тәсіл – бөлек “байланыстырушы коллекция” құру, мысалы enrollments коллекциясы. Онда studentId, courseId, және қосымша өрістер (семестр, статус, тіркелген күні) сақталады. Бұл тәсіл масштабтау үшін әлдеқайда қауіпсіз: байланыс саны миллионға жетсе де, негізгі құжаттар ауырламайды.

MongoDB-де reference болғанда деректі біріктірудің екі негізгі жолы бар.

- Бірінші жол – қолданба деңгейінде екі сұраныс жасау: алдымен негізгі құжатты алу, кейін id бойынша байланысты құжаттарды алу.
- Екінші жол – aggregation pipeline ішіндегі $lookup операциясы. $lookup SQL-дегі JOIN сияқты әсер береді: бір коллекциядан екіншісіне сәйкестендіріп қосады.

- Reference тәсілі деректерді басқаруға қолайлы, себебі “ортақ объект” бір ғана жерде сақталады. Бұл өзгерістер енгізуді жеңілдетеді: мысалы пән атауы, оқытушы дерегі немесе факультет туралы ақпарат өзгерсе, оны бір коллекцияда жаңартсаңыз жеткілікті. Reference сондай-ақ үлкен тарихты бөлек сақтауға мүмкіндік береді, мысалы бағалар тарихы, қатысу логтары, транзакциялар. Мұнда негізгі объект құжаты жеңіл болып қалады, ал тарих бөлек коллекцияда өсіп отыра береді. Бұл тәсіл масштабталатын жүйелерде өте маңызды, себебі үлкен көлемнің бәрін бір құжатқа салмай, деректерді жауапкершілік аймағына қарай бөлеміз.

- Reference қолданғанда ең жиі проблема – оқу кезінде қосымша әрекет қажет болуы. Бір экранды ашу үшін екі-үш сұраныс жасауға тура келуі мүмкін немесе $lookup қолдану керек болады. Бұл latency-ді арттыруы мүмкін, сондықтан query patterns дұрыс жоспарлануы керек. Тағы бір қауіп – байланыстарды дұрыс қолдамасаңыз, “жетім” сілтемелер пайда болуы мүмкін: мысалы courseId бар, бірақ course құжаты өшірілген. MongoDB-де SQL сияқты толық referential integrity автоматты түрде сақталмайтындықтан, мұны қолданба логикасы немесе бизнес ережелері арқылы бақылау қажет. Сонымен қатар байланыс көп болғанда индекстерді дұрыс жасамау өнімділікті бірден түсіреді.

- Reference моделінде ең жиі қателік – many-to-many байланысын екі жақта да шексіз массивпен сақтап қою. Бұл уақыт өте 16MB лимитіне немесе жаңарту қиындығына алып келеді. Тағы бір қателік – индекстерді елемеу: studentId, courseId, date секілді өрістерге индекс болмаса, үлкен көлемде сұраныс баяулайды. Үшінші қателік – referential integrity-ді мүлдем бақыламау: өшірілген құжатқа сілтеме қалып қоюы мүмкін. Соңғы қателік – $lookup-ты күнделікті ауыр жүктемелі интерфейсте шектен тыс қолдану, себебі ол кейде жүйені ресурстық тұрғыдан қымбаттатуы мүмкін.

- References – MongoDB-де үлкен және ортақ деректерді басқарудың негізгі тәсілі. Ол деректердің қайталануын азайтады, үлкен тарихты бөлек сақтауға мүмкіндік береді және жүйені масштабтауға көмектеседі. Бірақ reference моделі оқу кезінде қосымша сұраныс немесе $lookup қажеттігін тудыруы мүмкін, сондықтан query patterns пен индекстер алдын ала жоспарлануы тиіс. Тапсырма ретінде “электронды журнал” жүйесінде студент, пән, баға, қатысу үшін қандай коллекциялар керек екенін және many-to-many байланысты қалай ұйымдастыру тиімді екенін жазып шығу ұсынылады. Бұл тапсырма келесі бөлім – денормализация және модельдеу шаблондарына өтуді жеңілдетеді.

Денормализация ұғымы

Денормализация – деректерді толық “таза” түрде бөлек коллекцияларда сақтап, байланыстар арқылы біріктірудің орнына, оқу жылдамдығын арттыру үшін кейбір мәліметті әдейі қайталап сақтау тәсілі. Яғни бір объектіге қатысты ақпараттың бір бөлігі басқа құжаттың ішінде көшірме ретінде тұруы мүмкін.

Бұл SQL әлеміндегі қатаң нормализацияға қарсы сияқты көрінгенімен, MongoDB сияқты құжаттық дерекқорларда өте жиі қолданылатын практикалық әдіс. Денормализацияның негізгі себебі – қосымшаның ең жиі орындалатын сұраныстарын жылдамдату және экранға деректерді бір-екі сұраныспен шығаруға мүмкіндік беру.

Reference моделін қолданғанда, толық ақпарат алу үшін көбіне бірнеше сұраныс жасауға немесе $lookup арқылы біріктіруге тура келеді. Егер жүйеде белгілі бір экран күн сайын мыңдаған рет ашылса, қосымша сұраныстар саны көбейеді, жауап беру уақыты ұлғаяды және серверге түсетін жүктеме артады.

Осындай жағдайда денормализация “оқуды арзандатады”: ең жиі керек болатын атрибуттар негізгі құжатта дайын тұрады. Мысалы, тапсырыс құжаты ішінде клиенттің аты-жөні немесе пән журналы жазбасында пән атауы мен оқытушының аты “көшірме” ретінде сақталуы мүмкін. Бұл жағдайда жүйе әр жолды көрсету үшін бөлек коллекцияға қайта-қайта бармайды.

Денормализация әрқашан бір компромисс береді: оқу (read) жылдамдайды, бірақ жазу (write) және деректердің келісімділігі (consistency) қиындайды. Өйткені бір ақпарат екі немесе бірнеше жерде қайталанып тұрса, ол өзгерген кезде барлық көшірмелерді де жаңарту қажет болады. Осы жерде ең маңызды сұрақ пайда болады: “Біз оқуды қаншалықты жылдамдатқымыз келеді және ол үшін қаншалықты күрделілікке дайынбыз?”

Егер жүйе read-heavy болса, яғни оқу операциялары басым болса, денормализация жиі ақталады. Ал егер write-heavy болса, яғни жаңарту өте көп болса, денормализация керісінше мәселе туғызуы мүмкін.

Денормализацияны әдетте мынадай жағдайларда қолданған дұрыс.

- Біріншіден, қосымшада белгілі бір деректер өте жиі оқылады және әр оқуда бірнеше коллекцияға бару қажет болады.
- Екіншіден, денормализацияланатын өрістер сирек өзгереді немесе өзгерген жағдайда оны синхрондау механизмін ұйымдастыру оңай.
- Үшіншіден, экранда “бірден көрсету” маңызды болғандықтан, дерек дайын болуы керек.

Мысалы, “тапсырыстар тізімі” экранында әр тапсырыс үшін клиенттің аты-жөнін шығару керек болса, клиент коллекциясына әр жол үшін бару тиімсіз болуы мүмкін. Мұнда customerName тәрізді өрісті тапсырыстың ішінде сақтау жүйені жеңілдетеді.

Денормализация қауіпті болатын негізгі жағдай – дерек жиі өзгеретін болса. Егер сіз көшірмесін сақтап қойған ақпарат күніне ондаған рет өзгерсе, әр өзгерісте барлық құжаттарды жаңарту керек болады. Бұл уақыт бойынша да, жүйелік ресурс бойынша да қымбат операцияға айналады.

Екінші қауіпті жағдай – көшірме деректердің нақты қай жерде “ақиқат” екенін анықтамау. Егер “source of truth” анық болмаса, әр жерде әртүрлі мән қалып, жүйеде сәйкессіздік пайда болады.

Үшінші қауіпті жағдай – денормализация көлемді деректерді қажетсіз көбейтіп, сақтау орнын шамадан тыс арттыруы. Сол себепті денормализацияда “қажетті минимум” қағидасын ұстану керек: тек шынымен жиі керек бірнеше өрісті ғана қайталау.

Денормализация жасағанда міндетті түрде “ақиқат көзі” анықталады, яғни негізгі әрі дұрыс мән қай коллекцияда сақталатыны белгіленеді. Мысалы, клиенттің нақты аты customers коллекциясында сақталса, orders коллекциясындағы customerName тек көрсетуге арналған көшірме болып есептеледі. Бұл қағида жүйеде кім “басты” екенін нақтылайды және синхрондау стратегиясын дұрыс құруға көмектеседі.

Егер customerName өзгерсе, біз оны customers ішінде өзгертеміз, ал orders ішіндегі көшірме қалай жаңарады – бөлек механизм арқылы шешіледі. Осы механизм болмаса, көшірме ескіріп, деректердің сенімділігі төмендейді.

- Университет жүйесінде “пән журналы” экранын алайық. Егер журналда әр студенттің аты-жөні, тобы, пән атауы, оқытушы аты қатар көрсетілсе, онда әр жол үшін бірнеше коллекцияға бару күту уақытын арттырады.
- Мұнда денормализация арқылы журнал жазбаларына пән атауын немесе оқытушының қысқа атын сақтап қою экранды жылдамдатады. Бірақ бұл өрістер өзгерсе не болады? Пән атауы жиі өзгермесе, денормализация қауіпсіздеу.
- Ал студенттің аты өзгеруі мүмкін болса, көшірме қаншалықты маңызды? Егер ол тек визуалды мақсатта болса және кейде ескіріп қалуына рұқсат етілсе, проблема аз. Егер қатаң дәлдік керек болса, синхрондау міндетті.

Денормализациядағы негізгі қателік – “көп оқылады” деп барлық деректі көшіріп тастау. Бұл сақтау көлемін өсіреді, жаңартуды қиындатады және бақылауды жоғалтады.

Екінші қателік – source of truth анықтамау, нәтижесінде әр жерде әртүрлі мән қалады.

Үшінші қателік – синхрондау стратегиясын жоспарламау: өзгеріс болғанда көшірме жаңармай қалады. Төртінші қателік – жиі өзгеретін өрістерді денормализациялау. Мұндай жағдайда жүйе “әр өзгерісте мың құжатты update ететін” қымбат режимге өтіп кетеді.

- Денормализация – MongoDB-де өнімділікті арттыру үшін жиі қолданылатын тәсіл, әсіресе read-heavy жүйелерде. Ол экрандарды ашуды жылдамдатады, сұраныстар санын азайтады және $lookup тәуелділігін төмендетеді. Бірақ денормализация әрдайым келісімділік пен жаңарту күрделілігін бірге әкеледі. Сондықтан “ақиқат көзін” анықтау және синхрондау стратегиясын жоспарлау міндетті.

MongoDB үшін модельдеу шаблондары

Кіріспе

MongoDB-де бір ғана “дұрыс схема” болмайды, себебі әр жүйенің сұраныстары, дерек көлемі және жаңарту логикасы әртүрлі. Сондықтан тәжірибеде қалыптасқан модельдеу шаблондары (patterns) қолданылады: олар қайталана беретін мәселелерге дайын шешім сияқты қызмет етеді. Шаблондар деректерді қалай ұйымдастыру керегін “ережемен” емес, дәлелденген тәжірибемен көрсетеді. Мақсаты – жүйені масштабталатын, өнімді және қолдауға жеңіл ету. Әсіресе үлкен деректер өсімі бар жүйелерде шаблондар құжаттың семіруін, шексіз массивтерді, артық $lookup-ты және денормализациядағы хаосты алдын ала болдырмауға көмектеседі.

Шаблон таңдаудың ортақ логикасы

Шаблон таңдағанда біз әрдайым бірдей сұрақтарға сүйенеміз.

- Біріншісі – деректер қалай оқылады: бір экранға қандай мәлімет бірге керек?
- Екіншісі – деректер қалай өседі: бұл массив немесе байланыс уақыт өте шексіз ұлғая ма?
- Үшіншісі – деректер қаншалықты жиі өзгереді және өзгеріс қай жерде “ақиқат” ретінде сақталады?
- Төртіншісі – қандай индекстер керек және олар сұраныстарды қалай жылдамдатады? Осы сұрақтарға жауап бере отырып, біз embedded, references және денормализацияны комбинациялап, нақты шаблонды таңдаймыз.

Embedded (бір агрегат – бір құжат)

- Embedded шаблоны объектіні толық сипаттайтын шағын деректерді бір құжатта сақтауға негізделеді. Мұнда негізгі мақсат – бір сұраныспен толық нәтиже алу және атомарлы жаңартуды сақтау. Бұл шаблон профиль типіндегі объектілерге өте тиімді: студент профилі, қолданушы профилі, ұйым карточкасы. Бірақ embedded шаблоны тек деректердің көлемі бақыланатын және шексіз өспейтін жағдайда қолданылады. Егер тарих, лог немесе транзакция сияқты дерек үнемі өсетін болса, оны осы шаблонмен бір құжатқа салу қателікке әкеледі. Сондықтан embedded шаблоны көбіне “статикалық немесе баяу өсетін” деректер үшін дұрыс.

Шаблон 2: Reference (нормализацияланған модель)

- Reference шаблоны ортақ немесе үлкен деректерді бөлек коллекцияда сақтап, негізгі құжатта тек идентификатор арқылы сілтеме жасауды ұсынады. Бұл тәсіл пән, оқытушы, ұйым бөлімдері сияқты көптеген жерде қолданылатын объектілерге және шексіз өсетін тарихқа ыңғайлы. Reference шаблоны деректер қайталануын азайтады және өзгерістерді бір жерден басқаруға мүмкіндік береді. Бірақ оқу кезінде бірнеше сұраныс немесе $lookup қажет болуы мүмкін, сондықтан бұл шаблонды таңдағанда query patterns пен индекстер алдын ала жоспарлануы тиіс.

Extended Reference (сілтеме + қысқа көшірме)

- Extended Reference шаблоны reference пен денормализацияны бірге қолданады. Негізгі құжатта объектінің id-сы сақталады, бірақ ең жиі көрсетілетін бірнеше өрісі де қосымша көшірме ретінде тұрады. Мақсат – $lookup-ты азайту және экранды жылдам ашу. Мысалы, студенттің баға жазбасында studentId және studentName қатар сақталуы мүмкін немесе журнал жазбасында courseId және courseTitle бірге тұрады. Мұнда “ақиқат көзі” міндетті түрде анықталады: атау өзгерсе, көшірме қандай механизммен жаңарады? Егер атау сирек өзгерсе, бұл шаблон өте тиімді; ал атау жиі өзгерсе, синхрондау қымбатқа түсуі мүмкін.

Subset Pattern (негізгі құжатта тек ең керек бөлік)

- Subset pattern үлкен деректің барлығын бір құжатқа салмай, ең жиі керек болатын шағын бөлігін ғана embedded ретінде сақтауға негізделеді. Толық дерек бөлек коллекцияда тұрады. Бұл тәсіл “бірден көрсету” қажет болатын интерфейстер үшін пайдалы: мысалы профильде соңғы 5 бағасын немесе соңғы 10 қатысу жазбасын көрсету, ал толық тарих “толығырақ” бетінде бөлек сұраныспен шығарылады. Subset pattern қолданғанда құжаттың семіруі тежеледі, бірақ негізгі экран жылдам болып қалады. Бұл шаблон көбіне “жылдам көру + толық архив бөлек” логикасымен түсіндіріледі.

Bucket Pattern (уақыт бойынша пакеттеу)

- Bucket pattern үздіксіз келетін деректерді (лог, өлшемдер, қатысу, оқиға жазбалары) бір құжатқа шексіз жинаудың орнына, оларды уақыт немесе көлем бойынша “шелекке” (bucket) топтап сақтауды ұсынады. Мысалы, қатысу деректерін апта бойынша бір құжатқа жинауға болады немесе жүйелік логтарды күн бойынша бөлуге болады. Бұл тәсіл жазуды да, оқуды да жеңілдетеді: соңғы бір апта/бір күн дерегін алу өте оңай, ал құжат көлемі бақылауда болады. Bucket pattern әсіресе IoT, streaming, мониторинг, журнал жүргізу сияқты сценарийлерде негізгі шаблон болып есептеледі

Outlier Pattern (сирек “аса үлкен” деректерді бөлек шығару)

- Outlier pattern “көпшілік объектілер шағын, бірақ кейбіреуі ерекше үлкен” болатын жүйелерде пайдалы. Мысалы, көп студенттің қосымша мәліметі аз, бірақ кейбір студентте өте көп сертификат немесе өте көп қосымша жазба болуы мүмкін. Егер біз бәрін бірдей embedded жасасақ, ерекше үлкен объектілер бүкіл жүйеге проблемасын тигізеді. Outlier pattern мұндай “аса үлкен” жағдайларды бөлек коллекцияға шығарып, негізгі құжатты жеңіл қалдыруға көмектеседі. Яғни біз “орташа жағдайға” емес, “шеткі жағдайға” (outlier) дайын боламыз.

Polymorphic Pattern (әртүрлі типті объектілерге бір схема)

- Polymorphic pattern әртүрлі типті объектілерді бір коллекцияда сақтау қажет болғанда қолданылады. Мысалы, хабарландырулар әртүрлі болуы мүмкін: студентке қатысты, оқытушыға қатысты, әкімшілікке қатысты. Мұнда жалпы өрістер ортақ, ал типке байланысты өрістер бөлек болуы мүмкін. Бірақ бұл шаблонды қолданғанда тип бойынша фильтрлеу, индекстеу және құжат құрылымын басқару маңызды болады. Әйтпесе коллекция ішінде “әркім әртүрлі жазған” хаос пайда болады.

Шаблондарды таңдауда жиі қателесетін жерлер

- Көбісі “бір шаблонды бәріне қолдануға болады” деп ойлайды, бірақ бұл қате. Embedded барлық жерде қолданылса, құжат семіреді; reference барлық жерде қолданылса, оқу кезінде сұраныстар саны өседі. Extended reference дұрыс бақылаусыз қолданылса, синхрондау мәселесі туындайды. Bucket дұрыс жобаланбаса, кейін сұраныстар қиын болып кетеді (мысалы, қандай уақытта қандай bucket-та тұрғанын дұрыс есептемеу). Сондықтан шаблон таңдау “интуициямен” емес, query patterns және өсу логикасы арқылы дәлелденуі керек.

Қорытынды

- MongoDB модельдеу шаблондары – тәжірибеде ең жиі кездесетін мәселелерді шешуге арналған дайын тәсілдер. Embedded, reference, extended reference, subset, bucket, outlier және polymorphic сияқты шаблондар арқылы біз жүйені масштабталатын әрі өнімді ете аламыз. Бірақ кез келген шаблон тек сұраныстар мен деректердің өсімін ескере отырып таңдалғанда ғана тиімді.

Жиі кездесетін қателер мен антипаттерндер

Қателер мен антипаттерндер не үшін керек?

- MongoDB-де схема икемді болғандықтан, бастапқыда бәрі оңай сияқты көрінеді: кез келген өрісті қосып, кез келген құрылымды жасай салуға болады. Бірақ жүйе дерек көлемі өскенде, сұраныстар көбейгенде және команда бірнеше адамнан тұрған кезде дәл осы икемділік үлкен проблемаға айналады. Антипаттерндер – тәжірибеде жиі қайталанатын қате шешімдер. Олар “жүйе кіші кезде” байқалмайды, бірақ уақыт өте өнімділікті түсіріп, деректердің келісімділігін бұзып, қолдауды күрделендіреді. Сондықтан бұл бөлімнің мақсаты – студенттерге MongoDB моделін салғанда қай жерде тоқтау керегін және қандай белгілер “қауіпті” екенін үйрету.

Антипаттерн 1: SQL сияқты ойлау және бәрін бөлшектеп жіберу

Көп студент MongoDB-ні SQL сияқты модельдейді: әр логикалық нәрсені бөлек коллекцияға шығара береді, тіпті профильдің шағын бөліктерін де бөледі. Нәтижесінде қарапайым экранды ашу үшін бірнеше сұраныс керек болады немесе $lookup үнемі қолданылып кетеді. Бұл әсіресе read-heavy жүйеде latency-ді арттырады және жүйе жүктемесін өсіреді. MongoDB-нің күші – бірге оқылатын деректерді бірге сақтау, сондықтан “бәрін нормализациялап тастау” MongoDB философиясына қайшы келеді. Дұрыс бағыт – query patterns-ті қарап, бірге көрсетілетін және бірге жаңартылатын деректерді embedded ретінде біріктіру, ал тек ортақ немесе өте үлкен деректерді ғана reference-ке шығару.

Антипаттерн 2: “God document” – бәрін бір құжатқа тығу

Екінші шектік қате – керісінше, барлық деректі бір құжатқа тықпалау. Мұндай құжат уақыт өте “семіреді”, ішінде үлкен массивтер, көптеген ішкі объектілер пайда болады, ал сұраныстар күрделене бастайды. Жаңарту қымбаттайды, құжаттың көлемі өседі, 16MB лимитіне жақындайды. “God document” көбіне “бір сұраныспен бәрін аламын” деген жақсы ниеттен туады, бірақ ұзақ мерзімде жүйені тиімсіз етеді. Дұрыс шешім – subset, bucket немесе history-ді бөлек коллекцияға шығару сияқты шаблондарды қолдану және негізгі құжатта тек жиі керек “ядро” деректерді қалдыру.

Антипаттерн 3: Шексіз өсетін массивті embedded сақтау

MongoDB-де ең классикалық қателік – уақыт өте үнемі өсетін деректерді құжат ішіндегі массивке жинай беру. Логтар, әрекет тарихы, бағалар тарихы, қатысу деректері, чат хабарламалары – бұлардың бәрі “шексіз өсетін” категорияға жатады. Басында 10–20 элемент болғанда бәрі жақсы, бірақ кейін жүздеген, мыңдаған элемент жиналғанда құжат ауырлап, оқу мен жаңарту баяулайды. Соңында 16MB лимитіне жеткенде жүйе техникалық шектеуге тіреледі. Мұның дұрыс баламасы – деректі бөлек коллекцияға шығару немесе bucket pattern арқылы уақыт бойынша пакеттеп сақтау, ал негізгі құжатта тек соңғы бірнеше элементті preview ретінде қалдыру.

Антипаттерн 4: Many-to-many байланысты екі жақта да “үлкен массивпен” сақтау

Many-to-many байланыста студенттер жиі екі жақта да id массив сақтайды: мысалы студентте courseIds, курста studentIds. Бұл байланыс саны аз кезде жұмыс істеуі мүмкін, бірақ жүйе үлкейген сайын массивтер өседі, жаңарту күрделенеді және құжат көлемі бақылаудан шығады. Сонымен бірге бір студентті өшіргенде немесе пәннен шығарғанда екі жақтағы массивті синхрондау керек болады. Мұндай модельде келісімділік жиі бұзылады. Дұрыс шешім – бөлек “байланыстырушы коллекция” (enrollments) жасау: онда studentId, courseId, семестр, статус, тіркелу күні сияқты өрістер тұрады. Бұл масштабталатын және басқаруға жеңіл тәсіл.

Антипаттерн 5: Индекстерді ойламау немесе қате индекс стратегиясы

MongoDB-де көптеген өнімділік проблемасы индекстердің болмауынан немесе дұрыс қойылмауынан шығады. Әдетте жүйе кішкентай кезде толық скан байқалмайды, бірақ дерек көбейгенде сұраныстар күрт баяулайды. Жиі фильтрленетін және байланыс жасалатын өрістер (studentId, courseId, date, group) индекстелмесе, жүйе әр сұраныста көп құжатты қарап шығады. Тағы бір қате – индекстерді “көп болса жақсы” деп шамадан тыс көбейту: әр индекс жазуды баяулатады және сақтау көлемін арттырады. Дұрыс тәсіл – query patterns-ке сүйеніп, нақты қажетті индекстерді таңдау, әрі жазу/оқу балансын сақтау.

Антипаттерн 6: Денормализация жасап, синхрондауды жоспарламау

Денормализация оқу жылдамдығын арттыра алады, бірақ ол міндетті түрде көшірмелерді синхрондау мәселесін әкеледі. Көп жағдайда customerName, courseTitle сияқты өрістерді әр жерде көшіріп сақтап, кейін олар өзгергенде жаңартуды ұмытып кетеді. Нәтижесінде жүйеде әртүрлі мәндер пайда болады: бір жерде пән атауы ескі, бір жерде жаңа. Бұл пайдаланушы сенімін төмендетеді және есеп беруде қателік тудырады. Дұрыс шешім – source of truth анықтау және синхрондау стратегиясын нақтылау: өзгеріс болғанда бірден update жасаймыз ба, әлде жоспарлы түрде пакетпен жаңартамыз ба, әлде кешігуді қабылдаймыз ба – осының бәрі алдын ала жазылуы керек.

Қателердің “ерте белгілері” және тексеру сұрақтары

Антипаттерндер көбіне бірден байқалмайды, бірақ бірнеше белгі ерте ескерту береді. Егер бір экранды ашу үшін 5–7 сұраныс керек болып кетсе – модельді қайта қарау керек. Егер негізгі құжаттың ішінде тарих массиві күн сайын өсіп жатса – бұл шексіз өсудің белгісі. Егер дерек әр жерде қайталанып тұрса және оны жаңарту “қиын” болса – денормализация бақылаусыз кеткен. Егер сұраныстарда жиі толық скан байқалса – индекстер дұрыс емес. Студенттерге осы белгілерді “диагностика” ретінде үйрету пайдалы: жүйе өсер алдында қателікті түзетуді жеңілдетеді.

Қорытынды

MongoDB-дегі қателердің көпшілігі екі шектен шығады: бәрін тым бөлшектеу немесе бәрін бір құжатқа тықпалау. Сонымен қатар өсетін массивтер, many-to-many массивтері, индекстерді елемеу, денормализацияны бақылаусыз қолдану, типтердің араласуы және $lookup-ты орынсыз қолдану жүйені тез әлсіретеді. Тапсырма ретінде студенттерге университет кейсінде екі модель ұсыну сұралады: біреуі қате (антипаттерндері бар), екіншісі дұрыс (subset/bucket/enrollment, дұрыс индекстер). Әрқайсысында қай жерде қандай тәуекел бар екенін мәтінмен дәлелдеп жазуы қажет. Бұл тапсырма студенттің модельдеуді “тек схема емес, жүйе мінез-құлқы” ретінде түсінгенін көрсетеді.