형의 기술

데이터 모델이 무너지는 자리의 설계와 규율

ThakiCloud (Hyojung Han)

형의 기술

데이터 모델이 무너지는 자리의 설계와 규율

ThakiCloud (Hyojung Han)

Contents

1 제품의 형은 코드보다 먼저 정해진다

화요일 밤. 고객에게 메시지가 온다. 청구서에 다중 화폐를 지원해 주었으면 좋겠다고. 처음에는 작은 요청으로 들린다. 금액 옆에 통화 코드를 한 칸 넣는 작업. 모델을 열었다가 멈춘다. 금액은 정수 한 컬럼. 통화 코드 자리도, 발행 시점 환율도, 세금 기준도 없다. 청구 PDF는 그 컬럼에서 바로 렌더링된다. 돈을 읽는 모든 곳을 건드려야 하고 이미 발행한 과거 행을 전부 다시 써야 한다. 구 코드와 새 코드가 함께 도는 기간을 관리해야 한다. 이튿날 아침, 작은 요청은 일주일짜리 프로젝트가 되어 있다.

문제는 요청이 아니었다. 형이었다. 데이터의 형, 과거를 어떤 모양으로 저장했는지가 제품의 뼈대. 제품이 할 수 있는 기능도, 깨지지 않는 한계도, 결국 저장된 데이터의 형에서 나온다.

1.1 모든 기능은 형의 투사

구체적으로 보자. 제품은 워크스페이스를 가진다. 데이터베이스에는 workspaces 테이블 한 행이 있고, 그 멤버는 users 행에 workspace_id를 가진다. 이 형은 한 사람은 한 워크스페이스라고 말해 준다. 그러면 제품의 한계가 저절로 자란다.

소유권 이전이라는 한계. workspace_id 값을 바꾸면 되는 줄 알았는데, 워크스페이스 설정과 청구 계정과 팀 결제 정보가 전부 소유자 행에 붙어 있다. 새 컬럼을 만들고 관계를 옮기는 작업이 된다. 초대라는 한계. 초대돼는 중인데 아직 가입하지 않은 사람이라는 상태를 담을 자리가 형에 없다. 그래서 초대 플로우가 없거나, 밖에 임시 테이블로 붙여 놓는다. 청구라는 한계. 청구 계좌가 소유자에 묶여 있으니, 이 워크스페이스의 청구는 소유자만 낼 수 있다. 다른 멤버가 낼 수 있느냐는 질문 자체가 형에 답할 자리가 없어 성립하지 않는다.

LLM 앱도 다르지 않다. 채팅 제품은 메시지를 conversation_id와 순서로 저장해 두었다. 화면에 답장이 답장에 답하는, 줄이 지은 대화가 필요하다. 형에 줄의 개념이 없으니 새로운 줄을 다 다시 저장해야 한다. 사용자가 편집한 메시지와 모델이 낸 원본을 따로 남길 자리도 없다. 편집이 원본을 덮어 쓰니, 평가 데이터셋에 쓸 원본이 이미 사라져 있다. 기능의 아이디어는 회의실에서 나왔다. 형이 구현을 막는다.

이것이 데이터의 형이 제품의 형을 정하는 기작이다. 코드는 형의 투사를 화면에 그릴 뿐이다. 형에 자리가 없으면 코드가 만들어 내지 못한다. 새 기능이 들어올 때 어떤 행에 어떤 사실이 추가되는지 먼저 물어 보라. 한두 행으로 답이 나오면 기능은 싸다. 세 테이블에 퍼뜨려야 한다면, 형 바꾸기 작업이다. 이미 마이그레이션이다.

1.2 새 요청을 형으로 읽어 보기

기능 요청이 들어올 때, 화면이나 코드를 먼저 떠올리지 말고 형으로 읽어 보자. 두 개의 요청을 비교해 본다.

첫 번째 요청: “사용량 내역을 CSV로 내려받게 해 달라.” 사용량 내역이라는 사실은 이미 usage 이력 테이블에 있다. 그런데 내역을 내려받았다는 사실은 어디에 저장되는가. 누가, 어떤 기간을, 어디로, 어떤 상태로. 형에 그 자리가 없다. 이 요청은 exports라는 새 테이블이다. id, user_id, 기간, 파일 위치, 상태 컬럼으로. 새 요청이 새 엔티티를 낳는 순간, 모델 작업이다.

두 번째 요청: “로그인 화면에 회사 로고를 넣게 해 달라.” 로고는 파일 저장소에 저장하고 users 행에 경로 한 컬럼을 더하면 된다. 새 사실은 이미 있는 자리에 붙는다. 이 요청은 싸다.

이 차이가 요청 평가의 전부다. 요구서가 길어도 형에는 한 컬럼이 안 붙으면 싸고 요구서가 한 줄이어도 새 테이블이 필요하면 비싸다. 작업량 추정 전에

새 사실이 어디에 붙나를 묻는 습관만 들이면, 이 책의 절반은 끝난 것이다.

1.3 데이터 바꾸기가 비싼 이유

코드는 버릴 수 있다. 함수를 다시 쓰고 파일 지우고 배포 되돌린다. 코드에는 기억이 없다. 데이터는 다르다. 데이터는 제품의 기억이다. 과거의 모든 청구서와 모든 이벤트는 이미 세상 밖에 나갔다. 고객이 보고 세무서가 받아, 보고서로 떠났다.

그래서 형 바꾸기는 이사다. 비용은 세 덩어리로 온다.

첫째는 변환 비용이다. 기존 행을 새 형으로 다시 써야 한다. 행이 백만 개면, 세트로 나누어 조심스레 돌리면 몇 시간이다. 그 사이에도 서비스는 살아 있어야 한다. 백만 행을 초당 천 행씩 돌리면 십칠분, 서버 부하를 보정하면 한두 시간 [추정]. 배포 윈도우 밖에서 조용히 흘러야 한다는 점이 중요하다.

둘째는 이중 도는 기간이다. 마이그레이션 중에는 구 코드와 새 코드가 동시에 돌고 둘 다 같은 테이블을 읽어서 쓴다. 형은 두 버전이 같이 살아야 할 모양이어야 한다. 많은 마이그레이션이 먼저 넣고 나중에 지우는 두 단계인 이유는 여기에 있다. 배포 리듬이 이 기간의 길이를 정한다. 구 버전을 완전히 내리기까지 하루를 걸면, 형은 하루 동안 두 세계를 다 이해해야 한다.

셋째는 검증 비용이다. 구 형과 새 형이 같은 값을 가짐을 증명해야 한다. 행 수와 합계와 샘플 행을 대조한다. 청구 제품이므로 금액 합계가 한 원이라도 어긋나면 끝이다. 증명하지 못하면 밤을 설 수 없다.

검증의 모양도 미리 정해 두자. 다음 세 가지가 맞을 때, 마이그레이션이 끝났다고 말할 수 있다. 상태별 행 수의 합이 구 형과 같고, 열려 있는 청구의 금액 합계가 한 원의 차이도 없이 같고, 무작위로 뽑은 백 개 행이 컬럼마다 같은 값을 가진다. 이 세 가지 쿼리는 마이그레이션을 시작하기 전에 써 두면 좋다. 시작 전에는 당연히 같은 값이고 끝난 뒤에는 대조한다. 검증

쿼리를 나중에 쓰면, 쓰는 데만 두 시간이 걸리고, 대조할 때는 이미 잠이 와 있다 [추정].

코드 변경과 비교해 보자. 버그 수정은 테스트로 검증한다. 영향 범위는 함수다. 형 변경은 쿼리로 검증한다. 영향 범위는 제품 전체다. 그래서 기능 릴리스보다 코드 줄 수는 적어도, 마이그레이션 릴리스가 훨씬 무겁게 느껴진다.

이 책의 규율은 이 비용을 필요할 때에서 쌀 때로 옮기는 일이다. 미리 정한 형은 나누어 낸 선불이다. 설계 단계에서 컬럼 한 개를 더 두는 것은 무료에 가깝고, 운영 단계에서 컬럼 한 개를 더 만드는 것은 프로젝트다.

1인 팀에게는 이 계산이 더 무겁다. DBA도, 온콜도, 배포 리듬을 나눠 줄 동료도 없다. 이중 도는 기간은 내 배포 리듬 그대로다. 새벽 두 시에 마이그레이션이 멈추면, 그 새벽에 일어나서 계속하는 사람도 나다. 그래서 1인 개발자의 규율은 팀 규율보다 더 단순해야 한다. 작게, 자주, 낮에. 다음 장부터 그 설계와 진화의 방법을 하나씩 만들어 가자.

1.4 모델이 무너지는 네 가지 형

대부분의 모델은 드라마틱하게 무너지지 않는다. 네 가지 조용한 모양으로 무너진다.

숨은 1:N. 설계 시점에는 하나로 보인다. 사용자의 회사, 워크스페이스의 청구 계정, 제품의 가격표. 그래서 단일 필드로 만든다. 반년 뒤, 한 사람이 두 회사에 속할 수 있다고 제품은 말한다. 단일 필드가 테이블이 되어야 하고 그 필드를 전제로 한 모든 쿼리가 고쳐져야 한다. 방어법. 제품의 언어에 목적지, 계정, 소속 같은 단어가 보이면 관계가 커질 가능성이다. 성장 수평선이 일 년 이내라면 처음부터 조인 테이블을 두라.

동결된 열거형. 상태 컬럼에 값이 네 개 있고 코드가 그것으로 분기한다. 환불됐다는 다섯 번째 값이 들어온다. 값을 추가하는 것은 어렵지 않다.

어려운 것은 그 값을 보게 될 구 코드다. 네 값을 전부 처리하고 나머지는 오류로 던지던 분기는, 이제 합법적 상태를 오류로 삼킨다. 방어법. 열거형은 읽는 이와 맺은 계약이다. 값을 추가하기 전에 그 컬럼을 읽는 모든 곳을 나열하라. 상태 전이를 아는 곳이 한두 개도 아니면, 그것은 다음 버그의 씨다.

돈과 시간의 형. 금액이 float이고 타임스탬프에 지역 시간대가 있고 날짜가 문자열인 경우. 어느 것도 오늘 터지지 않는다. 다중 화폐와 여름 시간제와 월 단위가 넘는 보고서가 들어오는 날에 터진다. float의 금액은 반올림 오차가 청구 분쟁이 된다. 지역 시간의 날짜는 정산 기간의 경계에서 한 달이 엇나간다. 금액은 최소 단위와 통화 코드, 시간은 UTC. 나중에 한꺼번에 변환 비용을 치르지 않으려는 계산이다.

사라지는 이력. 모델이 현재 상태만 남긴다. 마지막으로 쓴 모델, 현재 잔액, 최근 로그인. 과거는 덮어 쓰인다. 그러다가 이력 화면이나 사용 보고서를 요구한다. 데이터가 없다. 저장하지 않은 것은 재구성할 방법이 없다. append만 하는 이력 테이블은 처음에 싸고 나중에는 대체 불가능하다. 제품에서 가장 비싼 쿼리는 존재하지 않는 데이터에 대한 쿼리다.

형	드러나는 자리	아픈 시점
숨은 1:N	설계 시 단일 필드	관계가 커질 때
동결된 열거형	상태 값 추가	구 코드가 새 상태 봄
돈과 시간 형	float, 지역 시간	다중 화폐, 월 보고서
사라지는 이력	덮어쓰는 컬럼	이력 화면 요구

네 형의 아픈 시점은 전부 나중에 온다. 이것이 함정이다. 형은 가장 쌀 때 정해지고 가장 비쌀 때 대가를 치른다. 그 간격이 이 책 전체다.

1.5 첫 모델을 정할 때의 형 체크

그러면 첫 모델을 정할 때 무엇을 하는가. 완벽을 찾지 않는다. 형 체크라는 짧은 의식을 치른다. 새 테이블마다 코드 작성 전에 다음 질문에 답한다.

첫째, 정체성이다. 이 행을 행으로 만드는 것은 무엇인가. id다,는 답은 답이 아니다. 답은 특정 청구서의 특정 날 특정 결제다, 하는 모양이어야 한다. 정체성이 기본 키를 정하고 기본 키가 모든 것을 정한다.

둘째, 시간축이다. 이 행에 무슨 일이 있었는가. 최소한 created_at과 updated_at. 유효 기간이 있는 행이면 valid_from과 valid_to. 시간이 없는 행은 추론할 수 없다. 언제 생긴 값인지 모르면 그 값은 증거가 되지 못한다.

셋째, 소유자다. 누구의 데이터인가. 사용자의, 조직의, 플랫폼의. 소유자가 없으면 접근 경계가 없고 멀티 테넌트 제품은 여기서부터 흐트러진다.

넷째, 상태다. 상태가 있는가. 있으면 상태 목록은 어디에, 전이는 누가 기록하는가. 전이 기록 없는 상태는 다음 장의 동결된 열거형이다.

다섯째, 확장 여지다. 어떤 컬럼이 일 년 뒤 바뀔 것 같은가. 그 자리에 미리 여지를 두라. nullable 컬럼 한두 개는 거의 무료다. 조인 테이블 하나도 일 년 안쪽에서는 싸다.

여섯째, 삭제다. 이 행은 사라질 수 있는가. 사라진다면 물리 삭제인지 플래그인지 미리 정해라. 청구 제품의 사라진 행은 사라지지 않는다. 그게 채무다. 삭제한 행을 다시 보여 달라는 요청은 늘 온다.

형 체크는 테이블당 한 시간 안에 끝난다 [추정]. 모델이 맞다는 보장은 못 해 준다. 보장이 되는 것은 모델이 우연으로 정해지지 않았다는 점이다. 우연으로 정한 형은 새벽 두 시에 값을 치른다. 의식으로 정한 형은 설계 그날, 감당 가능한 금액으로 값을 치른다.

제품의 형은 화면에서 태어나지 않는다. 테이블에서 태어난다. 다음 장에서는 실제로 테이블을 그리기 시작한다. 엔티티부터.

2 엔티티에서 테이블까지

형 체크는 의식이었으니까. 이 장은 일 자체다. 실제 제품의 모델을 그려 보자. 제품은 사용량 과금 SaaS다. 팀이 가입하고 API 키를 발급하고 시스템이 각 호출을 기록하고 말미에 청구서를 발행한다. 이 제품을 고른 이유는 가장 많이 무너진다고 했던 형을 축소 버전으로 전부 담고 있기 때문이다. 돈, 시간, 소유, 이벤트 이력, 사람과 조직의 관계.

2.1 엔티티 찾기: 명사와 그 한계

첫 단계는 엔티티를 찾는 일이다. 흔한 기법은 제품 스펙을 읽으며 명사를 동그라미 치는 일이다. 조직. 사용자. API 키. 호출 기록. 청구서. 결제. 명사 목록은 좋은 시작이다. 답은 아니다.

이유는 모든 명사가 엔티티가 아니기 때문이다. 엔티티는 자기만의 정체성을 가진 것이다. 내용이 같아도 따로 행이 될 수 있는 것. 시험 질문을 던져 보자. 같은 내용의 것이 둘 있으면, 그것은 하나인가 둘인가. 두 장의 청구서에 적힌 서울은 하나의 것이다. 데이터지, 엔티티가 아니다. 3월 15일의 호출과 3월 16일의 호출은 내용이 같아도 둘이다.

역방향도 필요하다. 어떤 엔티티는 동사나 수식어로 숨어 있다. 청구는 제품의 언어에서 동사다. 그런데 제품이 누가 얼마를 언제 냈는지를 기억한다. 그래서 payments 행이 존재한다. 오래 남는 것을 묻는 질문으로 숨은 엔티티를 찾는다. 제품이 오래 기억해야 하는 것은 전부, 엔티티이거나 엔티티의 행에 산다.

세 번째 범주도 있다. 속성인 척 하는 엔티티, 엔티티인 척 하는 속성. 사용자의 주소. 주소가 프로필에 표시만 되면 속성이다. 가로와 동네 두 컬럼이면 충분하다. 그런데 제품이 나중에 종이 청구서를 발송지로 보내

준다고 하면, 주소는 자기 생애주기를 가진 엔티티가 된다. 배송지와는 다르고 이력이 있다. 제품의 수평선이 판단 기준이다. 첫 모델에서는 속성으로 두되, 엔티티가 되는 날을 위한 마이그레이션 노트를 남겨 두라. 노트가 틀린 테이블보다 싸다.

2.2 이름 짓기

테이블과 컬럼의 이름은 계약이다. 이름이 나르면 형이 바뀔 때마다 쿼리와 노트를 전부 다시 쓰게 된다. 실용 규칙 몇 가지.

테이블 이름은 단수, snake_case. users이지 user_list가 아니다. organization_members이지 tbl_org_mem_01이 아니다. 테이블 이름은 문장의 명사처럼 읽혀야 한다. invoices를 users와 join한다, 는 문장이 안 읽히면 이름이 틀린 것이다.

타임스탬프는 접두어에 주목. created_at, paid_at, occurred_at처럼 모든 순간은 _at. 기간은 period_start_date처럼 _date. 순간과 날이 섞인 컬럼은 타입과 이름이 항상 일치해야 한다.

줄임말 금지. 1인 팀에 cust_no 같은 다섯 글자 절약은 없어야 한다. 가독성은 매 읽음마다 이자를 붙여 주는 자산이다.

한 컬럼, 한 사실. amount에는 금액만 든다. 통화는 currency 컬럼의 일이다. 컬럼 이름 안에 괄호로 통화를 붙이는 순간, 그 컬럼은 두 사실을 담고 있다.

2.3 관계의 세 가지

엔티티는 연결된다. 연결의 형이 관계이고 종류는 셋이다.

하나 대 하나. 사용자와 프로필. 워크스페이스와 설정 페이지. 따로 테이블을 만들지는 한쪽이 정한다. 프로필이 자기 생애주기를 가지면,

선택적 가입이거나 따로 로드되는 경로가 있으면, 따로 만든다. 없으면 같은 테이블이다. 로드 편의의 결정인 경우가 대부분이다.

하나 대 여러 개. 조직과 그 청구서들. 사용자와 그 호출들. 외래키는 many가 있는 쪽에 간다. invoice 행에 organization_id를 두고, organization 행에 invoice_id를 두지 않는다. 이 규칙은 기계적으로 들리지만 이유가 있다. 많은 쪽의 행은 태어나면서 부모를 안다. 하나 쪽의 행은 자식이 늘어날 때마다 자기 형을 바꿀 필요가 없어야 한다.

여러 개 대 여러 개. 사용자와 조직. 한 사람이 두 조직에 속하고 한 조직에 열 사람이 속한다. 외래키 하나로 표현이 안 된다. 가운데 테이블, 조인 테이블이라고 부르는 것이 답이다. membership. user_id와 organization_id를 갖고, 자기만의 속성도 가진다. role, joined_at, invited_by.

조인 테이블은 타협이 아니다. 여러 대 여러 개의 유일한 올바른 형이다. 그리고 관계의 속성이 사는 곳이다. 멤버의 조직 내 역할은 user 행에도 organization 행에도 속하지 않는다. 관계 자체에 속한다. 역할 컬럼이 필요하다고 느낀 날은 이미 조인 테이블이 필요했던 날이다.

여기서 규칙 하나 더. 오늘 하나 대 여러 개인 관계가 일 년 뒤 여러 대 여러 개가 될 수 있다. 조직의 청구 계정. 오늘은 한 조직, 한 계정. 그런데 제품의 언어에 목적지라는 단어가 있다. 프로젝트별로 청구서를 쪼개면 좋겠다고. 수평선이 일 년 이내라면 처음부터 조인 테이블이 싸다. 수평선이 멀다면 하나 대 여러 개에 마이그레이션 노트면 충분하다. 앞 장의 형 체크가 바로 이 수평선을 정하는 질문이었다.

2.4 정규화를 어디까지 밀 것인가

이제 테이블에 컬럼을 넣는다. 모든 설계자가 묻는 질문이 있다. 얼마나 정규화할 것인가. 정규화는 사실을 어디에 둘 것인가, 라는 규칙 묶음이다. 각 사실을 한 번씩만, 그것이 결정되는 자리에 둔다.

첫째 정형은 셀에 관한 규칙이다. 한 셀, 한 값. A, B, C가 한 컬럼에 있는 것은 위반이다. 사용자 id 목록이 문자열인 것은 위반이다. 편리해 보이는 배열 컬럼은 회색 지대이고 다음 장에서 다룬다.

둘째 정형은 키의 완전성에 관한 규칙이다. 모든 속성이 기본 키 전체에 의존해야 하고 일부에 의존하면 안 된다. 예를 들어 보자. 청구서 라인 테이블의 키가 (invoice_id, line_number)이다. 여기에 고객 세율이라는 컬럼을 놓으면, 라인마다 세율이 반복되고 그 진실은 고객에게 있다. 둘째 정형 위반이다. 세율은 고객에게, 또는 세율 규칙 테이블에 간다.

셋째 정형은 전이적 의존에 관한 규칙이다. 키가 아닌 다른 속성에 의존하는 속성의 문제다. 예를 들어 주문 행에 창고라는 컬럼이 있고 창고는 공급자가 결정한다. 공급자 행에 창고 컬럼이 있다. 공급자가 창고를 옮기면, 주문은 바뀌지 않았는데 주문 행의 창고 컬럼이 이제 옛날 창고가 된다. 그 사실은 공급자의 것인데, 주문은 공급자를 참조해 읽을 때 물을 수도 있고, 쓸 때 스냅샷을 남길 수도 있다. 어느 쪽을 택하느냐는 다음 장의 판단이다.

그러나 밀어 넣는 데에는 한도가 있다. 하나의 사건이 여러 자리에 잘못된 값을 쓰지 못하게 하는 것이 정규화의 목적이다. 공급자가 창고를 옮기는 사건이 업데이트해야 하는 행은 한 행이어야지, 천 개 주문 행이어서는 안 된다. 업데이트 이상이 사라지는 선에서 멈추라. 그보다 멀리 가면 일관성을 더 사지 못하고 join만 사게 된다.

실용적인 기준선을 하나 준다. 모든 컬럼에 두 가지를 물어 보라. 이 컬럼을 쓰는 사건은 무엇인가. 그 사건은 몇 행을 건드리는가. 한 사건이 많은 행을 건드린다면, 그 컬럼은 잘못된 자리에 있다. 사건이 자연히 소유하는 쪽으로 옮기면 된다. 이 질문은 제일, 둘째, 셋째 정형을 외우는 것보다 믿을 만하다.

2.5 개별 컬럼의 형

테이블은 잡혔다. 이제 컬럼이다. 대부분의 아픔을 만드는 컬럼은 네 가지다.

id 컬럼. 순차 정수냐, UUID 같은 무작위 값이냐. 순차 id는 규모를 누출한다. 테이블에 몇 행이 있느냐가 바깥에서 id를 보면 안다. id로 사람을 초대하는 제품은 열거 공격에 여미. UUID는 누출하지 않지만 길고 인덱스에 느리다. 무작위 삽입은 페이지를 흩어 넣고 쓰기 속도를 떨어뜨린다. 규모는 데이터마다 다르다 [추정]. 실용선은 이것이다. 내부 기본 키는 순차, 외부에 노출하는 것은 따로 token이나 hash를 만든다. 기본 키의 형과 대외 id의 형은 별개의 문제다.

돈 컬럼. float을 쓰지 않는다. 반올림 오차는 청구에서는 책임이다. 원이나 센트 같은 최소 단위의 정수, 또는 고정 소수점을 쓴다. 그리고 통화 코드 컬럼을 처음부터 두라. 오늘의 단통화 제품이 내일의 다중 화폐 제품이다. 나중에 추가하는 통화 컬럼은 마이그레이션이고 처음부터 둔 통화 컬럼은 컬럼 하나다. 발행 시점 환율은 사건의 데이터다.

시간 컬럼. UTC timestamp. 지역 시간대는 화면의 결정이지 데이터베이스의 결정이 아니다. 모든 테이블에 created_at과 updated_at. 유효 기간이 있는 행에는 valid_from과 valid_to. 날은 순간과 다른 개념이다. 일일 rollup의 날짜는 date 컬럼이지 timestamp가 아니다.

상태 컬럼. text enum, 정수가 아니다. 정수 상태는 쿼리에서 사람이 읽을 수 없고 값을 하나 추가하면 구 코드의 번호가 바뀐다. text 상태는 쿼리에서 그대로 읽힌다. 상태 목록은 마이그레이션 파일의 주석에 쓰고 전이를 기록하는 자리는 형 체크에서 정해 둔다. 전이 기록 없는 상태는 다음 사고의 씨다.

JSON 컬럼. 가장 남용되는 것. JSON 컬럼은 모델이 아니다. 아직 컬럼 자격이 안 되는 속성의 백이다. 흔한 쿼리 경로에는 컬럼을, 나머지는 백에

둔다. 백 안의 key에 WHERE를 쓰는 날이 오면, 백은 더 이상 백이 아니다. 테이블이다. 그날 꺼내면 된다. 마이그레이션 노트 하나면 충분하다.

2.6 워크스루: 사용량 과금 SaaS의 테이블

전체를 그려 보자. 테이블 일곱 개.

organizations. id, name, created_at, updated_at. 모든 것의 소유자. organization_id가 없는 user 행은 아직 팀을 안 만든 개인이다.

users. id, email, name, created_at. email은 unique. 사용자는 membership을 통해 조직에 속한다.

memberships. id, user_id, organization_id, role, invited_at, joined_at, created_at. 조인 테이블. role은 관계의 속성. invited_at과 joined_at은 별개다. 초대는 가입보다 먼저 온 사실이다. (user_id, organization_id)에 unique 제약이 들어가면 중복 소속이 막힌다.

api_keys. id, organization_id, name, key_hash, created_at, revoked_at. 키의 평문은 저장하지 않는다. hash만. 모델에서 평문 컬럼을 발견한 날은 이미 사고가 난 날이다.

usage_events. id, api_key_id, organization_id, occurred_at, request_id, tokens_in, tokens_out, cost_minor, currency. append만 하는 테이블. UPDATE 없다. DELETE 없다. 이 테이블이 청구의 source of truth다. organization_id를 api_key_id에서 유도할 수 있음에도 여기 다시 둔 이유는 쿼리 패턴이다. 이 조직의 이번 달 사용량은 가장 뜨거운 쿼리이고 수천만 행 테이블에서 join을 매번 치르는 비용을 치르기 싫다. 이것이 이 모델의 첫 번째 비정규화 결정이고, 다음 장에서 정당화한다.

invoices. id, organization_id, period_start, period_end, amount_minor, currency, status, issued_at, created_at. status는 pending, issued,

paid, void. 기간은 date. 청구서는 usage_events의 한 기간의 스냅샷이다.

invoice_lines. id, invoice_id, description, amount_minor, currency, created_at. 청구서 하나, 라인 여러 개. 라인은 금액의 분해다. 라인의 합은 청구서 금액과 같아야 한다. 이 불변식은 테스트다.

payments. id, invoice_id, amount_minor, currency, method, paid_at, created_at. 청구서 하나, 결제 여러 개. 부분 결제와 환불이 존재한다. payments가 invoice의 컬럼이었으면, 부분 결제가 들어온 날이 마이그레이션의 날이었을 것이다.

테이블	담는 사실	관계
organizations	팀이 있다	모든 것의 소유자
users	사람이 있다	membership을 경유
memberships	사람이 팀에 있다	role의 자리
api_keys	키가 발급됐다	org에서 1:N
usage_events	호출이 일어났다	append만
invoices	청구가 발행됐다	org에서 1:N
payments	돈이 들어왔다	invoice에서 1:N

테이블 일곱 개를 보면 결정들이 보인다. 읽기를 위한 usage_events의 비정규화. 부분 결제를 위한 invoice와 payments의 분리. role을 위한 membership의 분리. 그 당시의 진실을 위한 invoice의 스냅샷. 각 결정에는 이유가 있고 각 이유는 제품이 말해 준 이야기다. 모델은 다이어그램이 아니다. 모델은 결정들의 퇴적이다.

다음 장에서 가장 큰 결정에 정면으로 맞는다. 한 사실을 한 번 저장할 것인가, 두 번 저장할 것인가.

3 정규화 대 비정규화

앞 장에서 usage_events에 비정규화 하나를 남겼다. organization_id는 api_key_id에서 유도할 수 있음에도, 모든 행에 복사해 뒀다. 왜 그랬는가. 이 장 전체가 그 질문에 답한다.

정규화는 사실을 한 번만 저장하는 규율. 비정규화는 그것을 고의로 깨는 일. 둘은 옳고 그름의 반대편이 아니다. 예산의 반대편이다. 하나의 예산은 쓰기 비용이고 다른 하나는 읽기 비용. 모델은 그 예산의 배분이다.

3.1 정규화가 사는 것, 치르는 것

먼저 편익. 한 번 저장된 사실은 불일치할 수 없다. 조직 이름은 organizations 테이블에 있고, 청구서는 organization_id를 참조한다. 조직이 이름을 바꾸는 날, 한 행만 업데이트되면, 이름을 읽는 모든 화면이 새 이름을 본다. 단일 진실 출처의 가치.

두 번째 편익은 짧은 쓰기. usage_events에 호출 하나를 넣는 것은 INSERT 하나. 더 엄격하게 정규화해 조직의 현재 월 사용량 합계를 organizations 테이블에 두었다면, INSERT마다 organizations 행의 합계도 갱신해야 한다. 두 번의 쓰기, 하나의 트랜잭션, 그리고 고속 제품에서는 organizations 행에 대한 락 충돌. 쓰기 경로를 짧게 유지하는 것.

비용은 읽기 쪽에 있다. 조직 목록에 현재 월 사용량을 붙이는 조회는 organizations에 usage_events의 group by를 join한 쿼리. 매 페이지 로드마다 이 쿼리를 도는 뜨거운 화면은 매번 join을 치른다. 애플리케이션 코드도 불어난다. 화면 하나를 채우기 위해 테이블 세 개를 가져와 메모리에서 조립한다. 행 하나마다 조회 하나를 보내는 N+1 쿼리는 이 비용의 흔한 모양.

그러니까 질문은 정규화할 것인가, 는 아니다. 질문은 이 사실의 비용을 누가 치르느냐, 쓰기 쪽이냐 읽기 쪽이냐, 그리고 얼마나 자주 치르느냐.

3.2 비정규화가 사는 것, 치르는 것

비정규화는 유도돼야 할 사실을 저장하는 곳으로 옮기는 일. organizations 행의 member_count. usage_events 행의 organization_id. users 행의 last_login_at.

편익은 싼 읽기. join이 컬럼 참조가 된다. group by가 SELECT가 된다. 뜨거운 화면이 쿼리 하나로 줄어든다. 읽기가 쓰기의 천 배인 제품에서 이것은 큰 이익.

비용은 잘못된 값. 저장된 유도 값은 복사본. 복사본은 부패할 수 있다. 멤버가 가입했는데, 코드가 member_count 갱신을 놓쳤다. 화면은 틀린 숫자를 보고 그 숫자가 언제부터 틀렸는지는 아무도 모른다. 고객이 이 카운트가 왜 틀린지를 물을 때, 답이 알 수 없다는 말이 나오는 날, 데이터 문제가 가장 나쁜 모양을 가진다. 속도보다 정확성이 더 중요하다.

두 번째 비용은 이중 쓰기. 소스를 바꾸는 모든 이벤트가 복사본도 바꿔야 한다. 쓰기 경로가 길어지고 그 중간에 실패하면 소스와 복사본이 다른 모양이 된다. 트랜잭션이 해결책이다만, 테이블 여러 개를 오가는 트랜잭션은 그 자체로 락 충돌의 원인이 된다.

그러니 비정규화는 무료의 최적화가 아니다. 이것은 약속. 복사본이 소스를 따라갈 것이다, 라는 약속. 약속은 지킬 자가 필요하다.

3.3 판단 절차

네 가지 질문으로 판단한다.

첫 번째 질문. 얼마나 자주 읽히나. 월 1회에 백오피스 화면에서만 읽히는

값은 비정규화를 받을 자격이 없다. 매 페이지 로드에서 읽히는 값은 자격이 있다. 빈도가 이익의 분자.

두 번째 질문. 신선한 계산은 얼마나 비싸나. join 한 번이면 계산은 싸다. 읽기 쪽에서 치르는 것도 괜찮다. 이력 로그의 일 년치를 훑는 계산은 비싸다. 읽기 쪽은 결국 부서진다. 계산 비용이 분모.

세 번째 질문. 부패가 얼마나 치명적인가. 멤버 수가 하나 어긋나는 것은 표피 문제. 다음 날 치유된다. 청구 금액이 어긋나는 것은 표피가 아니다. 분쟁이 되고 환불이 되는 문제. 돈을 건드리는 사실의 부패는 레드라인.

네 번째 질문. 치유가 싼가. 소스에서 밤마다 도는 잡으로 다시 계산할 수 있는 값은 안전하다. 부패에 상한선이 있다. 다시 계산하려면 사람이 역사를 뒤져야 하는 값은 안전하지 못하다. 영원히 썩는다.

네 답이 네 결정의 모양으로 교차한다.

상황	판단	예시
읽기 많고 계산 비싸	과감한 비정규화	사용량 rollup
읽기 많고 계산 비싸고 치명	스냅샷	청구 금액
읽기가 드물다	정규화 유지	관리자 화면
계산이 싸다	읽기 때 계산	join 한 번

두 번째 행이 가장 흥미롭다. 자주 읽고 계산이 비싸고 부패가 치명적인 값. 청구 금액이 정확히 이 모양이다. 스냅샷이 답이다. 청구서는 발행 시점의 금액을 저장하고, 이후로는 교정하지 않는다. usage_events의 행도 바꾸지 않는다. 청구서는 문서이지 뷰가 아니다. 문서는 자기 진실을 보존할 권리가 있다.

3.4 워크스루: member_count

앞 섹션에 나온 organizations의 member_count를 네 질문에 걸어서 한 바퀴 돌려 보자.

얼마나 자주 읽히나. 조직 목록 화면은 각 조직 옆에 멤버 수를 보여 준다. 이 화면은 사용량 화면 다음으로 많이 도는 화면이다.

계산은 얼마나 비싸나. 신선하게 계산하려면, 조직 목록의 각 행마다 membership의 카운트를 해야 한다. 조직이 백 개면 membership의 group by 전체다. 멤버십이 만 개면 빠르다. 백만 개면 스캔이 되고 읽기 쪽은 제품이 크는 속도로 더 느려진다. 계산 비용은 시간이 갈수록 오른다.

부패가 얼마나 치명적인가. 멤버 수가 하나 어긋나도 청구 금액에는 영향이 없다. 접근 권한에도 영향이 없다. 표피 문제.

치유가 싼가. membership의 진실은 membership 테이블에 있다. 밤마다 각 조직의 카운트를 다시 계산해 대조하는 잡이 모든 어긋남을 잡는다. 재계산이 가능하다.

결론: 비정규화. member_count를 organizations 행에 둔다. membership의 INSERT와 DELETE에서 같은 트랜잭션으로 갱신한다. 밤에 치유 잡을 돌린다. 그리고 한 가지 더. 대조 쿼리를 관리자 화면에 붙인다. 어긋남이 가장 큰 조직 열 개를 보여주는 쿼리. 대조 쿼리가 비어 있으면, 밤에 잘 수 있다.

이것이 비정규화를 결정한다는 말의 전부다. 컬럼 하나를 더하는 것이 아니다. 컬럼, 트랜잭션, 잡, 대조 쿼리 네 가지가 한 묶음이다. 하나를 빠뜨리면 지킬 자 없는 약속이 된다.

3.5 비정규화의 두 얼굴

비정규화는 한 가지인 것처럼 다뤄 왔는데, 사실은 구분해야 할 두 얼굴이 있다.

첫 번째는 속도를 위한 비정규화. 부패해도 되는 유도 값, 치유가 가능한 값. member_count, last_login_at, rollup. 이 얼굴의 진실은 결국 같다는 뜻. 치르는 가격은 부패 창이다.

두 번째는 진실을 위한 비정규화. 교정되면 안 되는 값. 스냅샷. 이 얼굴의 진실은 그 당시 같음을 의미한다. 치르는 가격은 소스가 바뀌면 따라가지 못한다는 점이다.

두 얼굴을 구분하는 것은 하나다. 소스가 바뀌면 갱신하는가. 갱신하면 속도 얼굴이고 치유를 증명해야 한다. 갱신하지 않으면 진실 얼굴이고 생성 시점의 카피가 맞았음을 증명해야 한다. 네 질문 가운데 치명을 묻는 것은 두 번째 얼굴을, 치유를 묻는 것은 첫 번째 얼굴을 가리킨다. 둘을 섞는 모델, 밤마다 갱신되는 스냅샷과 교정되지 않는 캐시, 는 둘 다 잃는다.

3.6 LLM 앱의 만료되는 캐시

LLM 앱에는 비정규화의 특별한 종족이 있다. 모델의 가격이다.

모델 가격은 프로바이더가 가격표를 바꾸면 바뀐다. 그런데 이미 일어난 사용량의 비용은 소급 바뀌면 안 된다. 발생 시점의 비용은 스냅샷이고 cost_minor로 usage_events에 이미 박혀 있다. 현재 가격은 속도 얼굴의 값이다. 두 값이 동시에 산다.

현재 가격은 TTL이 있는 캐시. 프로바이더의 pricing API를 한 시간마다 뽑아, 캐시와 대조해, 바뀌었다면 갱신한다. 갱신 시각도 함께 남긴다. TTL은 장식이 아니다. TTL이 바로 부패 창. 캐시를 하루에 한 번 갱신하면, 오늘 바뀐 가격의 경우 최대 하루 동안 틀린 가격을 붙인다. 그 창을 제품이

선언해야 한다. 가격이 한 시간 단위로 갱신된다, 는 것은 명세. 명세는 모델 문서에 쓴다.

중요한 것은 두 값을 분리하는 일. pricing 캐시가 바뀌어도 usage_events의 cost_minor는 건드리지 않는다. 이미 일어난 행의 cost_minor는 그날의 진실. pricing 캐시가 하루를 틀렸다면, 그날의 청구가 틀렸을 뿐, 지난 청구는 틀리지 않는다. 스냅샷은 문서고 캐시는 뷰라는 성질이 여기 있다.

그리고 대조 쿼리. 청구서를 발행하는 순간, usage_events의 cost_minor 합계와 청구서 금액이 같아야 발행된다. 다르면 발행하지 않는다. 게이트다.

3.7 반복으로 남은 네 패턴

판단은 몇 가지 패턴에 가라앉는다. 당신은 그 패턴을 재사용하게 된다.

캐시 필드 패턴. 유도 값을 부모 행에 두고 같은 트랜잭션에서 갱신하고 주기적 잡으로 치유한다. member_count, last_login_at 같은 것들. 규칙은 하나. 잡은 트랜잭션을 대신하는 것이 아니다. 트랜잭션을 놓친 날을 위한 보험. 트랜잭션이 본업이고 잡은 안전망이다.

스냅샷 패턴. 문서가 생성 시점에 필요한 사실을 저장한다. 청구서의 금액, 통화, 세율, 환율. 규칙은 하나. 스냅샷은 갱신되지 않는다. 소스가 바뀌면, 가격 변경이든 세율 변경이든, 발행된 문서에는 도달하지 않는다. 과거 청구서를 소급 정정해야겠다, 는 기분이 드는 날, 제품의 약속이 틀렸다는 신호다.

이력 더 rollup 패턴. append만 하는 진실의 원천과, 읽기를 위한 유도 테이블. usage_events가 진실이고, 일일 usage_rollups (organization_id, day, total_tokens, total_cost)가 읽기다. 규칙은 하나. rollup은 재계산 가능해야 한다. rollup을 지우고 로그에서 다시 계산하면 같은 값이 나와야 한다. 이 재계산 가능성이 rollup을 캐시와 구분한다. 캐시는 썩을 수 있다.

rollup은 증명할 수 있다.

JSON 백 패턴. 흔한 경로는 컬럼으로, 나머지는 백으로. LLM 앱은 전형이다. 생성 기록에는 모델 이름, 토큰 수, 비용이 있다. 이들은 조회되므로 컬럼. 프로바이더 응답 헤더, 재시도 횟수, 프롬프트 버전. 이들은 디버깅 용이므로 백. 규칙은 하나. 백 안의 key에 불변식, 상태 기계나 제약, 을 쓰는 날, 백은 더 이상 백이 아니다. 태어나지 못한 테이블. 꺼내라.

LLM 앱에 한 가지 더. 생성의 출력은 문서. 입력 프롬프트, 모델, 그 당시의 비용, 출력을 같은 행에 두라. 사용자가 보는 챗 메시지와 시스템이 기록하는 기술 이벤트는 별개의 것. 메시지는 편집과 삭제가 있다. 이벤트에는 없다. 둘을 모델에서 혼동하면, 제품이 재생성 기능을 넣는 날이 마이그레이션의 날이다.

3.8 잘못된 쪽을 고르는 냄새

모델은 완벽하지 않다. 판단은 그날의 정보로 한 것이다. 냄새가 다시 판단할 때가 왔음을 알려 준다.

사건 하나에 테이블 다섯 개를 갱신하는 쓰기 경로. 너무 간절하다. 비정규화할 것이 아닌 것을 비정규화했다. 몇 개를 읽기 쪽으로 되돌려 보라.

join 네 개를 가진 뜨거운 화면. 너무 게으르다. 복사가 필요했던 것을 정규화했다. 읽기를 rollup이나 캐시 필드로 옮기자.

아무도 조회하지 않는 컬럼을 모두가 지키는 것. 좀비. 쓰기를 공짜로 먹는다. 읽는 사람이 없으면 drop하라.

키 서른 개로 자란 JSON 백. 위장한 모델. 제품이 테이블이 예상하지 못한 형을 키웠다. 흔한 key를 컬럼과 테이블로 꺼내라.

잡이 치유하지 않는 유도 값. 지킬 자 없는 약속. 가장 나쁜 냄새. 재계산 잡을 만들거나, 값을 없애라.

냄새는 죄가 아니다. 냄새는 방향. 모델은 살아 있다. 제품의 읽기 쓰기 비율이 바뀌면도그 바뀐다. 규율은 처음에 완벽한 모델을 만드는 것이 아니다. 규율은 언제나 지금, 모델이 예산의 어느 쪽을 치르고 있는지 아는 일.

다음 장에서 설계 도면을 떠나 프로덕션 데이터베이스로 간다. 모델은 바뀌어야 한다. 데이터는 이미 거기 있다. 서비스를 멈추지 않고 바꾸는 법이 마지막 규율이다.

4 마이그레이션과 인덱스

지금까지의 장은 설계 도면의 테이블을 다뤘다. 이번 장은 프로덕션 테이블을 다룬다. 프로덕션 테이블에는 데이터가 있다. 고객은 지금 그 데이터를 보고 있다. 제품은 데이터를 바꾸길 원한다. 빈 테이블의 형을 바꾸는 것은 무료다. 데이터가 있는 테이블의 형을 바꾸는 것은 프로젝트다. 이 장은 그 프로젝트의 규율을 설명한다.

4.1 첫 원리: 이중 버전 기간

스키마 변경은 순간이 아니다. 기간이다. 그 기간 동안 구 코드와 새 코드가 같은 테이블을 읽고 쓴다. 구 코드는 새 형을 모른다. 새 코드는 구 형을 참아야 한다. 이 기간을 이중 버전 기간이라고 부른다. 마이그레이션의 모든 규칙은 여기서 나온다.

첫 번째 규칙. 구 코드와 새 코드가 동시에 볼 수 없는 변경을 하지 않는다. 컬럼 삭제는 구 코드가 보지 못한다. 구 코드가 지워진 컬럼을 조회하는 날은 에러가 난다. 컬럼 이름 변경도 같다. 구 코드는 옛 이름을 읽고 새 코드는 새 이름을 읽고 테이블에는 이름이 하나뿐이다. 그래서 안전한 마이그레이션은 언제나 두 단계다. 먼저 넣고 구 코드가 완전히 사라진 후에 지운다.

이것에서 굳어진 방법이 있다. 확장, 이동, 축소, 의 세 단계다.

확장. Expand. 둘 다 살 수 있게 형을 넓힌다. nullable 컬럼을 더하고 새 테이블을 더한다. 새 코드는 구 형과 새 형에 모두 쓴다. 구 코드는 영향받지 않는다. 새 컬럼을 보지 못한다.

이동. Migrate. 데이터를 옮긴다. 구 행을 새 형으로 백필한다. 세트로, 속도를 줄여, 서비스가 살아 있는 동안. 그리고 읽기 쪽을 새 형으로 돌린다.

축소. Contract. 구 형을 지운다. 더해진 컬럼, 임시 테이블, 둘 다 쓰던 코드 경로. 축소 전에 며칠을 본다. 구 코드도, 새 코드도, 구 형을 건드리지 않는 것을 확인한 날, 지울 수 있다.

예를 하나 보자. 단통화 정수 amount를 다중 화폐 형으로 바꾼다. 확장: amount_minor와 currency 컬럼을 nullable로 더하고 둘 다 쓰는 새 코드를 배포한다. 이동: currency IS NULL인 행을 백필한다. currency는 KRW, amount_minor는 amount. 세트로 돌린다. 읽기 쪽을 새 컬럼으로 돌린다. 축소: 몇 버전 배포 후에 amount 컬럼을 drop한다. 각 단계는 작고 각 단계는 관찰 가능하고, 각 단계는 멈출 수 있다. 멈출 수 있는 것이 마이그레이션의 가장 중요한 성질이다. 어느 순간에 멈춰도, 시스템은 동작하는 형에 있다.

4.2 백필의 순서, 락이 무서운 이유

마이그레이션의 일은 두 종류다. DML과 DDL. 위험도는 완전히 다르다.

DML은 행의 일이다. 자기 데이터의 INSERT, UPDATE, DELETE. 이것은 무료다. 언제든 할 수 있다. 걱정할 것은 시간뿐이다. 세트 백필은 루프다. id 구간을 잡고 업데이트하고 잠깐 멈추고 다음 구간. 이 루프에는 세 가지 규율이 있다.

첫째, 멱등이다. 백필은 몇 번이고 돌려도 되아야 한다. currency IS NULL인 행만 UPDATE 하는 백필은 두 번 돌려도 안전하다. 멱등하지 않은 백필은 다시 돌릴 수 없는 백필이고 다시 돌릴 수 없는 백필은 첫 실패에서 멈추는 백필이다.

둘째, 진행이 보인다. 어디까지 왔는가에 대답할 수 있어야 한다. 코드 속의 변수가 아닌 행으로 확인한다. migration_progress 테이블에, 대상 테이블, 마지막 id, 갱신 시각. 잡이 죽어서 다시 살아난 날, 행에서부터 이어서 돌린다.

셋째, 속도를 줄인다. 세트 크기와, 세트 사이 잠깐 멈춤을 정한다. I/O를 전부 먹는 백필은 제품을 늦게 만드는 백필이다. 백필의 속도는 미덕이 아니다. 미덕은 고객이 모르게 끝나는 일이다. 백만 행을 만 행 세트로 돌리면 몇 시간이 걸리는 것은 normal 이다 [추정]. 그보다 훨씬 더 걸린다면, 쿼리의 형이 틀렸다는 신호다.

DDL은 형의 일이다. 컬럼 추가, 인덱스 추가, 타입 변경. 위험은 락이다. DDL이 테이블 전체의 락을 잡는 날, 그 테이블로의 쓰기는 멈춘다. 쓰기가 멈추지 않는 프로덕션에서, 이것은 정지다. 현대의 데이터베이스는 많은 DDL을 non-locking으로 만들었다. nullable 컬럼 추가는 최근 버전에서는 거의 즉시고, 인덱스 생성에는 쓰기를 막지 않는 CONCURRENT 옵션이 있다. 그러나 거의, 는 언제나, 가 아니다. 버전과 컬럼의 형과 기본값의 유무가 결과를 바꾼다.

일	위험	규율
nullable 컬럼 추가	낮음	버전 확인
NOT NULL 추가	높음	두 단계, 추가 후 백필
인덱스 생성	높음	CONCURRENT
타입 변경	매우 높음	새 컬럼, 이동, 삭제
컬럼 삭제	매우 높음	축소에서만

DDL이 피할 수 없는 날, 순서는 하나다. 프로덕션에서 하지 않는다. 프로덕션 데이터의 사본에서 먼저 한다. 락을 잡아도 괜찮은 사본에서. 사본에서 재는 시간이 프로덕션의 시간이고 락의 위험은 프로덕션의 것. 사본에서 십 분이면, 프로덕션에서는 한 시간일 수 있다. 규모와 부하가 다르다 [추정].

4.3 인덱스: 따로 정렬한 복사본

인덱스는 컬럼의 복사본이다. 정렬된 복사본. 테이블은 삽입 순서이고 인덱스는 조회 순서다. 데이터베이스는 인덱스로 행을 찾아, 테이블로 간다. 그래서 인덱스는 자기 순서를 따르는 읽기에 빠르고 복사본을 유지해야 하는 쓰기에 느리다. INSERT마다 그 테이블의 모든 인덱스에 세금을 낸다.

첫 번째 원리. 인덱스는 쿼리를 위해 존재한다. 이 테이블에 인덱스가 필요하겠다, 는 느낌이 오면, 질문이 틀린 것이다. 질문은 어떤 쿼리가 느리다, 는 것이다. 그 느린 쿼리를 들고 WHERE와 JOIN과 ORDER BY의 형을 본다.

복합 인덱스에는 순서가 있다. 등식 조건이 먼저, 범위 조건이 나중. 이 조직의 이번 달 사용량은 WHERE organization_id = ? AND occurred_at >= ? 로 생긴다. 인덱스는 (organization_id, occurred_at). 반대로, (occurred_at, organization_id)는 이 쿼리에 쓸모없다. 앞 컬럼의 범위 조건이 뒷 컬럼의 선택성을 삼킨다.

unique 인덱스는 제약을 가진다. email의 unique, (user_id, organization_id)의 unique. 제약은 인덱스로 강제된다. unique 제약을 추가하는 날은 인덱스를 추가하는 날이고, 지우는 날은 인덱스를 지우는 날이다.

검증은 EXPLAIN으로 한다. 쿼리를 EXPLAIN으로 돌리고 플랜의 모양을 본다. seq scan이 늘 나쁜 것은 아니다. 작은 테이블이나 테이블의 대부분을 가져가는 조회에는 seq scan이 가장 빠르다. 문제는 큰 테이블에서 인덱스를 써야 할 조회가 seq scan인 경우. 그리고 역도. 대부분을 가져가는 인덱스 scan도 문제다. 인덱스는 돌아가는 길이다.

가장 큰 함정은 소량 데이터에서의 검증이다. 천 행에서 빠른 테스트는 억 행에서 거짓말을 한다. 플랜너의 선택은 행의 수에 따라 바뀐다. 테스트 데이터베이스에서 쓸모없는 인덱스가 프로덕션에서는 유일한 길이고, 역도 그렇다. 그래서 플랜은 프로덕션 데이터의 사본에서, 적어도 같은 규모의

데이터에서, 검증한다.

마지막 규율은 정리다. 아무도 쓰지 않는 인덱스는 매 쓰기에 공짜로 세금을 내는 것. 대부분의 데이터베이스는 인덱스의 사용 횟수를 기록한다. Postgres에는 pg_stat_user_indexes가 있고 scan 횟수를 볼 수 있다. 한 달에 한 번, scan이 0인 목록을 본다. 전부 지우지는 않는다. 쿼리의 형이 바뀌지 않았는지, 드물지만 중요한 조회, 월간 보고서, 복구 경로, 의 인덱스가 아닌지 확인하고, 그때 drop한다. 인덱스는 자산이 아니다. 인덱스는 읽기의 배당을 치르는 부채다.

4.4 1인 운영자를 위한 마이그레이션 전 체크리스트

1인 개발자에게 온콜 팀이 없다. 규율 자체가 온콜이다. 프로덕션 마이그레이션 전에 다음 항목을 순서대로 거친다. 한 시간 안쪽의 일이다 [추정]. 그런데 그것이 따분한 오후와 새벽 두 시의 경계다.

하나. 복구 지점이 있는가. 마이그레이션 전의 백업, 또는 PITR 위치. 지금 확인한 백업이어야 한다.

둘. 프로덕션 데이터의 사본에서 돌렸는가. DDL의 시간, 백필의 시간, 플랜의 변화, 전부 사본에서.

셋. 멱등한가. 백필을 두 번 돌려도 되는가. 마이그레이션을 중간에서 이어서 돌릴 수 있는가.

넷. 세트 크기와 예상 시간을 정했는가. 그리고 속도 제한. 백필이 가질 수 있는 최대 I/O는 얼마인가.

다섯. 배포 순서가 맞나. DB 변경이 먼저, 코드가 나중. 역은 절대 아니다. 확장 전에 새 코드가 오면, 에러다.

여섯. 이중 버전 기간의 형을 검증했는가. 구 코드와 새 코드가 동시에 도는 동안, 둘 다 동작하는가. 의도적으로 구 버전을 돌리는 스테이징이 검증의

자리다.

일곱. 인덱스가 CONCURRENT인가. 마이그레이션에 인덱스 생성이 있으면, 막는 생성은 정지다.

여덟. 멈출 자리를 정했는가. 어느 단계에서, 무엇이 잘못되면, 멈추고 그때 시스템은 어떤 형인가.

아홉. 롤백 경로가 문서화되었는가. 확장의 롤백은 쉽다. 더한 것을 지우면 된다. 축소의 롤백은 어렵다. 지운 컬럼은 돌아오지 않는다. 그래서 축소는 관찰 기간이 가장 긴 단계다.

열. 검증 쿼리가 있는가. 마이그레이션 후에, 구 형의 행 수와 새 형의 행 수가 같고 돈의 합계가 같고 샘플 행이 같다. 이것 없이 끝나지 않는다.

체크리스트는 의식이 아니다. 각 항목은 실제 실패의 무덤이다. 어젯밤의 백업, 다시 돌릴 수 없었던 백필, 확장보다 먼저 온 새 코드, 쓰기를 막았던 인덱스. 1인 운영자는 안전을 사람으로 사지 않는다. 순서로 산다.

체크리스트를 실행한 흔적을 남기자. 체크리스트는 한 줄의 메모로 끝난다. 언제, 어떤 사본, 어떤 버전, 몇 행, 몇 초. 다음 마이그레이션의 시작에서 그 한 줄을 읽으면, 이번의 예상 시간이 지나간다. 기록이 쌓이면, 예상은 숫자가 된다.

4.5 작은 팀의 리듬

마이그레이션의 빈도는 한 번의 크기보다 중요할 수 있다. 한 달에 한 번 큰 마이그레이션을 하는 팀은 매일 그 불안과 산다. 주에 다섯 번 작은 마이그레이션을 하는 팀은 그것을 생각하지 않는다.

이유는 마이그레이션의 위험이 이중 버전 기간에 있기 때문이다. 기간이 짧고 기간의 형이 단순할수록 안전하다. 큰 마이그레이션이 피할 수 없다면, 쪼갠다. 백필을 두 번의 마이그레이션으로, 테이블의 앞부분을

먼저, 뒷부분을 나중으로. 확장과 축소를 릴리스에 나누어. 각 조각이 따분해지고 따분한 조각은 밤을 새우지 못한다.

섞지 않는 것도 같다. 스키마 변경과 동작 변경을 하나의 릴리스에 섞지 않는다. 형이 바뀌고 동작이 바뀌고 무언가가 깨지면 어느 쪽이 깼는지 모른다. 스키마 변경은 혼자 릴리스된다, 가능하면 다른 변경 없는 릴리스에서. 그 규율이 순간의 롤백을 가능하게 한다.

캘린더에 넣어라. 주 1회, 스키마 일만 하는 30분 slot. 컬럼을 더하고 인덱스를 더하고 백필을 돌리는 것, 전부 그 slot 안에. 그 slot을 캘린더에 넣지 않으면, 마이그레이션은 쌓이고 쌓인 마이그레이션은 언젠가 그 큰 마이그레이션이 된다.

4.6 미리 치른 비용

책을 한 줄에 올려 보자.

제품의 형은 테이블에서 정해진다. 테이블은 사실을 어디에, 몇 번 저장할 것인가의 규율로 정해진다. 그리고 규율이 바뀔 날에는, 낮에, 작은 조각으로, 바꾼다.

확장, 이동, 축소. 스냅샷과 rollup과 백. 멱등한 백필과 세트 루프와 EXPLAIN으로 검증한 인덱스. 어휘는 많지 않다. 습관은 하나. 형의 비용을 필요할 때에서 쌀 때로 옮기는 일.

새벽 두 시의 마이그레이션은 운이 아니다. 설계의 날에 정하지 않은 형이 쌓은 청구서다. 이 책의 규율은 그 청구서를 분할 납부하는 일이다. 조용한 화요일 오후에. 형은 조용하다. 형을 조용히 두는 날, 제품은 조용하다.