검색의 규율

사람들이 찾을 수 있게 만드는 기술

ThakiCloud (Hyojung Han)

검색의 규율

사람들이 찾을 수 있게 만드는 기술

ThakiCloud (Hyojung Han)

Contents

1 1장. 완전 일치가 죽는 자리

1.1 존재하는 기록을 못 찾는 사람

사용자한테 쪽지가 온다. "지난주에 등록한 프로젝트를 못 찾아요." 데이터베이스 클라이언트를 연다. 기록은 있다. created_at은 지난 화요일이고, 제목도 기억하는 그대로다. 아무 문제 없어 보인다. 그런데 같은 이름을 검색창에 치면 빈 결과다.

이 상황은 묘한 성질. 데이터는 있고 기능은 있고 그런데 사용자는 못 찾는 것이다. 둘 중 어느 하나만 탓하면 설명이 안 된다. 원인은 그 사이의 짧은 구간에 있다. 사용자가 친 문장이 데이터베이스의 조건으로 번역되는 구간. 이 구간은 대부분의 개발자가 가장 먼저 코드로 쓰고, 가장 적게 고민하는 부분이다.

대부분의 첫 검색은 이렇게 생겼다.

```
SELECT * FROM projects
WHERE name = $1;
```

깔끔하고 빠르다. 인덱스도 탄다. 그리고 테스트에서 항상 통과한다. 등록한 제목을 그대로 쳤기 때문이다. 하지만 테스트는 네 문장으로 쓰고 사용자는 사용자의 문장으로 친다. 이 두 어휘집 사이의 간극이 검색 기능을 죽이는 자리다.

1.2 두 개의 입구

위 쪽지를 조사한 개발자는 보통 두 갈래로 나뉜다. 한 갈래는 데이터베이스 클라이언트를 열고, SELECT를 직접 쓴다. LIKE '%프로젝트%' 같은,

사용자가 친 것보다 훨씬 관대한 조건으로. 그리고 "있잖아, 내가 찾았는데?"라고 답한다. 다른 갈래는 제품의 검색창을 열어서, 사용자가 친 그대로를 다시 친다. 그리고 "버그 맞다"라고 답한다.

두 갈래가 다른 결과를 보는 이유는, 두 갈래가 다른 문장을 쳤기 때문이다. 개발자가 직접 SQL을 쓸 때는, 답을 이미 알고 있다. 답을 알고 있는 사람은 조건을 풀지 않는다. "이것은 분명히 title에 들어있다"는 확신을 갖고 LIKE를 쓴다. 사용자는 답을 모른다. 답을 모르는 사람은 기억나는 문장을 친다.

여기에 첫 번째 교훈이 숨어 있다. 검색 기능을 시험할 때, 개발자가 답을 아는 상태에서 조건을 짜는 것은 시험이 아니다. 답을 모른다는 전제에서, 기억나는 문장을 치는 것만 시험이다. 이 책의 20개 쿼리 연습은 정확히 이 전제를 강제하는 장치다.

지원 티켓도 같은 눈으로 보자. 검색에 대한 티켓은 대부분 번역에 대한 티켓이다. 사용자가 "못 찾아요"라고 하면, 개발자는 "버그"를 듣는다. 하지만 사용자가 말하고 있는 것은, "내가 친 문장이 결과를 안 낳았어요"다. 그 문장은 시스템이 받은 유일한 입력이자 유일한 증거다. 티켓이 들어오면, 데이터베이스를 열기 전에 먼저 물어보자. 사용자가 어떤 문장을 쳤는가. 그걸 모르면, 수정은 추측이 된다.

1.3 완전 일치의 약속

WHERE name = 'x' 구절은 사용자에게 아주 엄격한 약속을 한다. 제목을 통째로 그대로 치고 글자 하나 다르지 않고 앞뒤 공백이 없어야 한다.

이 약속은 데이터를 등록한 사람이 검색하는 사람인 내부 도구에서는 괜찮다. 같은 사람이 같은 문장으로 다시 치니까. 하지만 누구든 검색하는 제품에서는 깨진다. 세 달 전에 "3분기 성장 프로젝트"라고 등록한 사람과, 오늘 그 프로젝트를 찾는 사람은 다른 사람일 수 있다. 같은 사람이라도

세 달 뒤의 기억은 저장소가 아니다. 기억은 원본 형식을 유지하지 않는다. 띄어쓰기도, 반각 전각도, 마지막 글자도.

더 까다로운 경우는, 등록한 사람의 문장 자체가 데이터베이스 안의 다른 문장과 다를 때 있다. 등록 화면의 제목 필드에 "ACME 계약"이라고 쓴 사람이, 검색 화면에서 "ACME"라고 치면, 완전 일치는 아무것도 모른다. 등록할 때와 찾을 때의 문장이 같은 시스템 안에서도 바뀌기 때문이다.

완전 일치는 키로 조회하는 기술이지, 찾는 기술이 아니다. 사용자가 검색창을 열면 그 사람은 키를 조회하려는 게 아니라, 흐릿한 기억을 문장으로 번역하려고 한다. 그 번역을 받아들이는 쪽이 데이터베이스인데, 완전 일치는 그중에서 가장 덜 관대한 엔진이다.

1.4 사용자가 실제로 치는 다섯 가지

검색창을 제품에 넣고 일주일만 지켜봐도, 입력은 대략 다섯 가지 형태로 수렴한다.

첫째, 등록된 이름 그대로. "3분기 성장 프로젝트". 가장 단순하고 완전 일치만으로도 처리되는 유일한 유형이다. 실제 제품에서 이 비율은 생각보다 낮다. [추정]

둘째, 이름의 일부. "성장"이라고만 치는 경우. 가장 빈번한 형태이고 부분일치를 전제로 한다. 이 유형은 다시 세 갈래로 갈린다. 앞부분만, "3분기". 중간만, "성장". 뒷부분만, "프로젝트". B-tree 인덱스는 앞부분만 찾아준다. 나머지는 2장에서 다룬다.

셋째, 오타. 자리가 바뀐 글자, 빠진 글자. "크리오투"를 "크리튜"라고 친다. 관용성을 전제로 한다.

넷째, 데이터에 없는 단어. 데이터에는 "청구서"라고 되어 있는데, 사용자는 "인보이스"를 친다. 저장한 쪽의 어휘와 찾는 쪽의 어휘가 다른 격차 문제다.

이 유형은 매칭 알고리즘으로는 안 풀린다. 알고리즘이 아무리 관대해도, 저장된 글자에 없는 문장은 찾지 못한다. 어휘 지도가 필요한 이유다.

다섯째, 조건 섞임. "진행중인 거, 지난 달의". 사용자는 검색과 필터링을 동시에 하려는 중이다. 그런데 검색창은 단어만 받는다. 이 유형은 3장의 필터로 풀지만 사용자가 왜 문장 속에 조건을 집어넣는지부터 이해해야 한다. 조건을 별도 입력으로 분리할 장치를 볼 수 있을 때까지, 사용자는 검색 문장 하나로 조건까지 말하려는 게 자연스럽다.

이 다섯 밖에도 빈 입력이 있다. 플레이스홀더를 보고 커서를 깜빡이게 하고 사용자가 그냥 떠나는 것. 검색창에서 가장 빈번한 이벤트다. 이 유형은 풀이가 없다. 다만 빈입력이 잦다면, 사용자가 뭘 쳐야 할지 몰랐거나, 쳐도 되는 줄 몰랐다는 뜻이므로, 플레이스홀더와 첫 페이지의 결과 모양을 다시 보자.

이 다섯 유형은 풀이가 완전히 다르다. 둘째에는 부분일치가, 셋째에는 유사도 계산이, 넷째에는 어휘 지도가, 다섯째에는 필터가 필요하다. 하나의 완전 일치 구절로 전부 처리하면, 전부 실패한다.

검색창의 플레이스홀더 문장도 이 분류와 관련 있다. "이름을 검색하세요"라고 적으면, 사용자는 이름 전체를 쳐야 한다고 믿는다. "프로젝트 검색"이라고 적으면, 이름도 설명도 치면 된다고 믿는다. 플레이스홀더는 기능 설명이다. 사용자가 어떤 사전으로 말해야 답이 나오는지, 한 줄로 미리 말해주는 장치다.

1.5 검색은 번역 문제

검색 기능은 두 사전 사이를 통역하는 것이라는 전제가 좋다. 첫 번째 사전은 사용자의 것이다. 그 사물이 사용자 머릿속에서 어떤 문장으로 불리나. 두 번째 사전은 데이터의 것이다. 데이터베이스에는 어떤 문장이 저장되어 있나. 데이터베이스는 저장이 충실할 뿐, 사용자 사전에 대해서는 아무것도

모른다. 저장된 형태 그대로의 문장으로만 대답할 수 있다.

검색을 만드는 개발자의 일은, 쿼리를 쓰기 전에 이 두 사전을 가까워지게 만드는 것이다. 손 쓸 수 있는 지렛대는 세 개다.

첫째, 저장을 관대하게 만든다. 등록할 때 동의어가 있으면 함께 저장한다. "인보이스"라고 치면 "청구서" 필드도 함께 쓴다던가. 이건 데이터 모델의 문제고 4장의 search_text에서 다시 나온다.

둘째, 검색을 관대하게 만든다. 부분일치, 유사도, 트라이그램. 이 지렛대가 2장의 전부다.

셋째, 사전 사이의 격차를 직접 잇는다. 작은 동의어 지도, "이거 찾으시는 거?"라는 제안. 3장의 주제다.

세 지렛대를 전부 당기지 않아도 된다. 하지만 최소한 둘째는 당겨야 한다. 완전 일치는 출발점이지 답이 아니다.

1.6 20개 쿼리 연습

코드에 손을 대기 전에, 값싸고 효용 큰 연습을 하나 하자. 네가 실제로 친다는 20개 검색어를 적는 것이다. 좋은 20개가 아니라, 바쁠 때 정말로 치겠더란 20개.

청구서를 관리하는 제품이라면 이렇게 나올 수 있다. "3월", "ACME", "ACME 미지급", "큰 청구서", "지난주 실패한 결제", "ACME 청구서 4월". 적다 보면 바로 드러난다. 절반은 완전 일치로는 못 찾고 일부는 부분일치로도 못 찾는다. "큰 청구서" 같은 것은 단어 검색의 영역이 아니라 조건 검색의 영역이다.

적은 20개를 위에서 분류한 다섯 유형으로 나누고, 지금의 검색창이 몇 개를 답할 수 있는지 세어보자. 절반도 안되면 부끄러워할 필요 없다. 그게 정상 상태다. 이 책은 정확히 그 숫자를 올리기 위한 것이다.

제품에 이미 사용자가 있다면, 이 20개는 더 정확해질 수 있다. 지원 메일과 채팅에서 "이걸 못 찾겠어요"라는 문장을 모은다. 그리고 그 문장을, 사용자가 치는 그대로 20개에 넣는다. 실제 사용자의 어휘는, 네가 적어둔 것보다 훨씬 낯설다. "큰 청구서"를 "1억 넘는 거"라고 말하는 사용자가 있다.

이 연습에는 또 한 용도가 있다. 검수 기준이 된다. 검색을 개선했을 때 아무 문장이나로 테스트하지 않는다. 같은 20개로 다시 돌린다. 잡힌 수가 늘어나는 것이 개선의 진짜 측정이다. 이 책의 뒤 장들에서도 이 20개를 계속 테스트 세트로 쓰자.

1.7 사용자의 잘못이 아니다

검색이 실패했을 때의 첫 반응으로 사용자를 탓하는 경우가 많다. "이름을 통째로 쳤으면 찾았을 거예요." 이 답은 기술적으로 맞는 경우가 많다. 하지만 답이 설계는 아니다.

사용자를 교육하는 데 드는 비용은 작지 않다. 사용자가 열 명이면 대신 쳐줄 수 있다. 천 명이면 아무도 대신 쳐줄 수 없다. "이렇게 치면 찾을 수 있어요"라는 한마디는 매번 지원 티켓이다. 티켓에는 내 시간이 들어가고 그 시간은 다음 기능에 못 쓰는 시간이다.

그 반대쪽이 규율이다. 사용자가 정확히 쳐야만 찾게 되는 검색은, 사용자 의존형 검색이다. 사용자 의존형 시스템은 사용자가 바뀔 때 고인다. 사용자는 바뀐다. 데이터를 넣는 사람도, 찾는 사람도, 쓰는 말도 바뀐다. 변하지 말아야 하는 쪽은 검색이다.

"사용자가 뭘 배워야 했나"로 잴 질문을 기억하자. 검색을 쓰기 전에 배워야 한다면, 그 검색은 아직 완성되지 않았다. 이 질문이 유일한 기준은 아니다. 하지만 검색을 고칠 때마다 이 질문을 건드리면, 설계가 옳은 방향으로만 흘러가게 된다.

1.8 이 장의 규율

검색은 사용자가 친 문장과 어떤 컬럼을 비교하는 행위가 아니다. 사용자의 의도를 데이터베이스가 답할 수 있는 조건으로 번역하는 행위다. “기록은 있는데 못 찾는다”는 고장은 저장의 실패가 아니라, 거의 항상 번역의 실패다.

첫걸인은 검색 엔진을 넣거나 확장자를 설치하는 것이 아니다. 사람들이 실제로 어떤 문장을 치는지 기록하는 것이다. 다음 장에서는 데이터베이스가 문장을 어떻게 자르고, 자른 조각으로 어떻게 빠르게 찾는지를 다룬다.

2 2장. 단어를 자르는 법: 인덱스와 토큰

2.1 문장을 조각으로: 토큰화

검색 엔진이 문장을 다루기 전에는 먼저 자른다. “hello world invoice”라는 입력을 [“hello”, “world”, “invoice”]라는 조각, 즉 토큰으로 나누는 것. 영어에서는 이 작업이 쉽다. 공백이 단어 경계이고 소문자로 내리면 된다. “Hello”와 “hello”는 같은 토큰이다.

한국어는 다르다. 공백은 단어 경계가 아니다. “검색기능”은 띄어쓰기 없이 쓰이기도 하고 “검색 기능”은 띄어쓰고 쓰인다. 둘 다 같은 뜻인데, 공백 기준으로 자르면 한쪽은 두 토큰, 다른 쪽은 한 토큰이 된다. 저장된 데이터가 “검색기능”이면, 사용자가 “검색 기능”을 쳐도 토큰 일치로는 못 만난다.

더 근본적인 문제는 형태다. “검색하다”, “검색을”, “검색이”는 같은 뿌리에서 나오지만 문자열로는 서로 다르다. 영어에서는 “search”와 “searches”도 다르지만 접미사가 짧고 규칙적이라 Stemmer라는 간단한 도구로 다룰 수 있다. 한국어의 어미는 길고 규칙이 많아서, 이 정도로는 안 통한다. 형태소 단위로 자르려면 전용 분석기가 필요하다.

여기에 실용적인 선을 하나 그어 두자. 1인 규모에서 한국어 형태소 분석을 처음부터 제대로 세팅하는 것은, 검색 기능을 만드는 일보다 더 큰 프로젝트가 된다. [추정] 분석기 라이브러리를 고르고 등록할 때 적용하고 검색할 때 적용하고 둘이 같은 도구를 쓰는지 검증하고 데이터가 쌓인 뒤에는 다시 분석하는 재처리까지 한다. 이 일은 검색 기능의 두세 배를 먹는다.

대신에, 문장을 자르지 않고 “안에 들어 있다”는 조건으로 찾는 방법, 즉 부분일치를 우선으로 다루자. 전문 인덱스의 한계는 알고 있는 정도로

충분하다. 검색의 언어 문제에서 1인 개발자가 가장 싼 대안은, 정교한 분석기보다 관대한 매칭이다.

구체적으로 보면 이렇다. 토큰이 "검색기능" 한 조각으로 저장되어 있는데, 사용자가 "검색 기능"을 친다. 공백으로 자르면 ["검색", "기능"] 두 조각이 된다. "기능"은 저장된 토큰에 없고 "검색"은 "검색기능"의 앞부분일 뿐이라, 토큰 일치로는 어떤 조합도 안 맞는다. 반대로 부분일치에서는, 저장된 "검색기능" 안에 "기능"이 들어 있으므로, "기능"만으로 찾는다. 한국어 이름 검색에서 부분일치가 더 맞는 답인 이유가 여기 있다.

이 장의 나머지 대부분이, 이 관대한 매칭을 어떻게 빠르게 만드는가에 대해다.

2.2 LIKE의 함정: 앞모시 와일드카드

부분일치의 가장 직관적인 작성법은 이것이다.

```
SELECT * FROM projects
WHERE name LIKE '%성장%';
```

문제는 인덱스가 안 탄다는 것이다. B-tree 인덱스는 정렬된 순서로 저장이 된다. 앞에 '성장'이 오는 행을 찾으면, 정렬 순서 덕분에 앞에서부터 읽어 원하는 자리를 빠르게 뛸 수 있다. 하지만 어디에나 '성장'이 들어가는 행을 찾으면, 정렬 순서는 도움이 안 된다. 성장이 행의 중간에 있는지도, 끝에 있는지도 몰라서, 한 행 한 행을 전부 읽어야 한다.

이걸 순차 스캔이라고 부르고 EXPLAIN 결과에 Seq Scan이라고 적힌다. 10만 행 표라면, 검색 한 건마다 10만 행을 읽는다.

데이터가 적을 때는 아무 일도 안 일어난다. 1천 행이면 전부 읽어도 1밀리세컨트 [추정] 이내다. 그래서 함정은 늘 나중에 온다. 사용자가 "왜 검색이 느려졌냐"고 묻는 그 날, 표는 50만 행이 되어 있다. 그때 "예전에는

빨랐는데?"라는 질문을 받으면, 답은 항상 같다. 예전에는 읽어야 할 행이 적었기 때문이다.

전부 읽는 스캔은 느린 것만으로도 끝나지 않는다. 데이터베이스는 하나인데, 쿼리는 여러 개다. 검색 쿼리가 CPU를 잡으면, 결제 확인이든 메일 발송이든 같은 서버의 다른 쿼리가 기다린다. 사용자는 검색만 느려진 게 아니라, 제품 전체가 무거워진다고 느낀다. 1인 개발자의 서버에는 우선순위 조절할 인력이 없으므로, 검색이 전부를 읽지 않는 것은 성능 문제이자 공존 문제다.

2.3 세 개의 탈출로

부분일치를 인덱스로 만들 수 있는 방법은 세 갈래 있다.

첫째, trigram 인덱스. Postgres의 pg_trgm 확장이다. 원리가 단순하다. 문장을 3글자씩 겹쳐가며 썰어서 그 조각들을 인덱스에 올린다. "성장추진"에는 "성장추", "장추진" 같은 3글자 조각이 들어 있고 이 조각 목록이 인덱스에 저장된다. 사용자가 "성장추"를 찾으면, 그 조각을 인덱스에서 찾아내고 해당 조각을 가진 후보 행만 검사한다. 전부 읽는 대신, 후보만 읽는 것이다.

```
CREATE EXTENSION pg_trgm;
CREATE INDEX idx_projects_name_trgm
 ON projects USING gin (name gin_trgm_ops);
```

이후의 쿼리는 ILIKE로 쓰면 된다. 대소문자를 구분하지 않으니까.

```
SELECT id, name FROM projects
WHERE name ILIKE '%성장%'
ORDER BY name
LIMIT 20;
```

장점 두 가지. 하나는 언어를 가리지 않는다는 것. 한국어, 영어, 무엇이든 글자로 된 텍스트에 작동한다. 1~2글자 검색만 주의하면 된다. 짧은 검색어는 만드는 조각이 흔해서, 인덱스로 골라도 후보가 표의 대부분이 된다. 결국 전부 읽는 것과 비슷하므로, 검색창에서 최소 길이를 제한하는 게 보통이다. 두 번째 장점은, 같은 인덱스로 "성장으로 시작"도 "성장이 들어간"도 다룬다는 것. 한 인덱스에 두 종류의 질문이 탄다.

앱 쪽에도 흔히 지킴이가 있다. 입력이 3글자 미만이면, 요청을 보내지 않거나 결과를 안 보여주는 것. 이것은 UX의 다듬기가 아니라, 쿼리의 보호다. 1~2글자 쿼리는 후보가 표의 대부분이 되므로, 인덱스가 있어도 플랜러가 이를 수 있다. 검색창에서 짧은 힌트를 보여주는 것이, 데이터베이스를 버텨지게 하는 것보다 값싸다.

둘째, 전문 인덱스. Postgres의 tsvector, to_tsquery 체계다. 문장을 사전에 토큰 벡터로 변환해서 저장하고, 그 벡터로 검색한다. 랭킹 함수 ts_rank도 함께 준다. 잘 되는 자리는 명확하다. 긴 본문, 영어 중심, "이 단어가 몇 번, 어디에 등장하나"가 중요할 때. 막히는 자리는 짧은 한국어 이름이다. Postgres의 한국어 토큰화 지원은 약하고, 기본 설정으로는 "검색기능"과 "검색 기능"을 같은 토큰으로 못 자른다. MySQL 쪽은 ngram 파서가 있어 조금 낫다. 하지만 역시 형태소 수준은 아니다.

셋째, 전용 검색 엔진. Meilisearch, Typesense 같은 것. 오타 관용, 즉석 제안, 다국어, 빠른 UX. 성능과 편리야말로 그 존재 이유다. 하지만 1인 규모에서 이 선택은 데이터베이스 옆에 또 하나의 서비스를 돌린다는 의미다. 동기화, 백업, 배포, 모니터링. 4장에서 이 선을 더 자세히 다룬다.

세 갈래를 고르는 기준은, 사실 두 질문으로 줄어든다. 첫 번째. 저장된 데이터가 긴 문서인가, 짧은 이름인가. 긴 문서라면 전문 인덱스가, 짧은 이름이라면 trigram이 맞다. 두 번째. 사용자는 오타를 치는가. "crewt"를 쳐서도 답을 기대하는 사용자가 있다면, trigram은 거기까지고 전용 엔진의 영역이다. 이 두 질문에 대한 답이 데이터베이스 쪽으로 기울면,

그쪽을 고른다. 운영할 서비스가 하나 줄어드는 것은, 1인 규모에서 가장 큰 이점이다.

2.4 무엇을 인덱싱할까: 컬럼과 가중치

name만 검색하는 것은 검색이 아니다. 사용자가 "ACME"를 치면, 이름이 아니라 설명에 ACME가 들어간 행도 나와야 한다. 여러 컬럼을 OR로 이어가면 되지만 그 순간 "어떤 행이 더 관련 있는가"라는 문제가 생긴다.

"청구서"가 제목에 들어간 행은, "청구서"가 설명 맨 뒤에 한 번 들어간 행보다 관련이 높다. 이 차이를 만드는 것이 가중치다. 전용 엔진은 내장 랭커가 해주지만 데이터베이스 쪽에서는 손으로 점수를 만든다.

```
SELECT id, name, description,
 (CASE WHEN name ILIKE $1 THEN 3 ELSE 0 END
 + CASE WHEN description ILIKE $1 THEN 1 ELSE 0 END) AS score
FROM projects
WHERE name ILIKE $1 OR description ILIKE $1
ORDER BY score DESC, updated_at DESC
LIMIT 20;
```

제목 3점, 설명 1점, 동점이면 최근 업데이트 순. 이 단순함이 의도적이다. ts_rank도 결국 "어디에, 몇 번"에 점수를 주는 식이다. 처음부터 복잡한 수식을 만들지 마라. 세 컬럼, 세 점수, 하나의 타이브레이커. 1장의 20개 쿼리로 검증되면, 그 다음에 손을 대도 된다.

trigram의 경계도 알고 가자. trigram은 부분 문자열을 찾지, 비슷한 문자열을 찾지 못한다. "crew"가 저장되어 있으면 "crewt"는 못 만난다. 3글자 조각이 완전히 같아야 인덱스가 움직이므로, 한 글자가 바뀐 오타는 후보에조차 못 든다. 오타 관용은 3장의 유사어 제안, 또는 전용 엔진의 몫이다. trigram이 주는 것은 "빠른 부분일치"일 뿐, "유사도"는 아니다.

인덱스는 점수를 주는 컬럼마다 필요하다. name과 description 둘 다 ILIKE로 쓰면, trigram 인덱스도 두 개다. 인덱스는 읽기를 빠르게 한다. 대신 쓰기를 느리게 만든다. 쓸 일이 적은 읽기 중심 표라면 감수할 가치가 있다. 하지만 매 행이 자주 바뀌는 표라면, 인덱스 두 개가 매 쓰기마다 두 번 갱신된다는 걸 잊지 마라.

2.5 1인 규모를 위한 실용선

방법	잘 되는 자리	막히는 자리
완전 일치	키 조회	부분, 오타
trigram	짧은 한국어 포함	2글자 이하
전문 인덱스	긴 본문, 영어	짧은 한국어
전용 엔진	오타, 다국어	운영 부담

수치를 붙이면 이렇다. 몇 만 행에서 몇 십만 행 사이의 SaaS 표라면, trigram GIN 인덱스 두 개(제목과 설명)로 수년간 버틴다. 쿼리는 대개 100~300밀리세컨트 안쪽에 머물고, [추정] 그 이상 걸리는 건 대부분 인덱스가 아닌 다른 부분이다. 4장에서 그 다른 부분을 본다.

행이 천만 단위로, 또는 오타 관용이 제품의 얼굴이 되는 순간이 오면, 전용 엔진으로의 이주가 논의를 시작할 때가 된다.

하지만 반대 방향의 과잉 설계를 경계한다. 5천 행에 Meilisearch를 앞세우는 건, 성능 문제를 운영 문제로 바꾸는 일이다. 지금의 느림이 LIKE 때문인지, 행 수 때문인지, 페이지 구조 때문인지 모른 채로 새 서비스를 더하는 것은, 증세 없이 약을 더하는 것과 같다. EXPLAIN으로 원인을 확인한 다음에 고를 것.

측정도 일관되게 하자. 제품에서 실제로 잘 치는 5개 검색어를 골라, psql에서 직접 쿼리를 돌리고 시간을 재면 된다. 500밀리세컨트 안쪽이면,

그 규모에서는 문제가 아니다. [추정] 500을 넘는 쿼리가 있다면, EXPLAIN으로 Seq Scan을 찾고 인덱스를 확인한다. 시간을 재고 Seq Scan을 찾는, 이 두 동작이 이 장의 실용선 전부다.

다음 장에서, 검색이 행을 찾아낸 다음에 어떤 방식으로 보여야 사용자가 "이거다"라고 멈추는지 다룬다.

3 3장. 결과를 보여주는 디자인: 필터, 정렬, 페이지

3.1 검색창은 시작일 뿐

사용자의 목표는 쿼리가 아니다. 답이다. “ACME 미지급”을 치고 20줄의 목록이 뜨는 순간까지가 절반의 일이다. 그 목록에서 “아, 이거다”라고 멈추게 만드는 나머지 절반이 이 장의 주제다.

검색창은 입구일 뿐이고 결과 페이지가 제품이다. 사용자가 제품을 “찾기 좋은”이라고 느끼는 것은 검색창에 자판을 누르는 순간이 아니라 결과가 보이는 화면에서다. 그 화면은 세 질문에 답해야 한다.

이건 뭔가. 행의 미리보기가, 클릭 없이 “이게 맞다”를 판단하게 해야 한다.

어느 순서가 맞는가. 사용자가 기대하는 순서대로 나와야, 쓸 만한 결과를 첫 화면에서 본다.

아직 안 나온 건 어떤 모양인가. 필터와 페이지가, 사용자가 “더 찾아야 하나, 다른 표현으로 시도해야 하나”를 결정하게 한다.

검색창의 반응 속도도 이 장의 주제다. 사용자가 자판을 누를 때마다 쿼리를 보내면, “A”, “AC”, “ACME”로 세 번의 요청이 생긴다. 대부분 300밀리세컨트 [추정] 정도 타이머를 두고, 멈출 때만 보내는 방식으로 풀 수 있다. 쿼리가 500밀리세컨트를 넘는다면, 타이머를 늘리는 것보다, 이전 결과를 지우지 않는 쪽이 낫다. 새로운 결과가 올 때까지 이전 목록을 유지하면, 화면은 흔들리지 않는다.

검색창의 위치도 같다. 상단의 검색창은 전체 검색이고 목록 위의 검색창은 이 목록을 좁히는 것이다. 사용자는 둘을 다르게 다룬다. 목록 위에 있는

검색창에서는, 목록을 줄이려고 친다. 상단에서는, 무언가를 찾으려고 친다. 한 제품에는 둘 다 있을 수 있다. 중요한 것은, 그 역할이 섞이지 않는 것이다.

세 질문 전부 UI의 문제처럼 보인다. 하지만 전부가 쿼리의 문제다. 미리보기가 보여줄 필드를 정하면, 쿼리가 가져올 컬럼이 정해진다. 순서를 정하면 ORDER BY가 정해진다. "아직 안 나온 건"을 말하려면, 결과 수와 다음 페이지의 조건이 정해진다. 화면은 쿼리의 그림자다. 그림자를 고치려 하면, 늘 원형부터 고치게 된다.

3.2 필터와 검색어의 조합

"진행중인 걸"이라는 말은 검색어가 아니라 조건이다. 제품이 둘을 합쳐줘야 한다. 검색창 옆에 상태 필터를 두고 두 입력이 AND로 만난다.

```
WHERE status = $2
 AND (name ILIKE $1 OR description ILIKE $1)
```

필터의 모양은 단순할수록 좋다. 상태를 고르는 세 가지, 기간을 고르는 세 가지. 드롭다운에 열두 가지 옵션을 나열하면, 사용자는 매 검색마다 옵션을 읽는다. 읽는 횟수가 늘면, 멈추는 횟수는 줄어든다. 1장에서 분류한 다섯째 유형, 조건 섞임은, 조건을 눈에 보이는 독립 입력으로 분리하는 순간부터 줄어든다. 사용자가 문장 속에 숨겼던 조건을, 화면 위로 꺼내준 것이다.

순서를 고민할 가치가 있는 순간이 하나 있다. 필터가 매우 선택적이고 검색어가 느릴 때. status가 5개 값 중에 하나고 그 값이 표의 5퍼센트[추정]라면, 먼저 걸렀다가 남은 5퍼센트에서 ILIKE를 돌리는 쪽이 훨씬 싸다. Postgres의 플랜러는 보통 알아서 잘 한다. 하지만 EXPLAIN으로 확인한 적 없다면, 확인해 두자. 4장에서 다시 나온다.

필터 상태는 URL에 남아야 한다. 사용자가 상태 필터로 좁혀서 유망한 결과를 보다가, 그 페이지를 남한테 보내고 싶을 때가 온다. 필터가

브라우저의 기억에만 있으면, 보낸 링크는 필터 없는 페이지가 된다. 쿼리 스트링 한 줄의 문제다. 페이지의 상태가 URL로 표현되면, 페이지가 공유 가능해진다. 브라우저의 뒤로 버튼도 같은 상태에 데려가 준다.

또 하나, 개수 표시 문제. "1,042개 중 37개" 같은 문장은 친근하지만 그 1,042를 세려면 전체 행을 읽어야 한다. 몇 십만 행 표에서 매 검색마다 전체 카운트를 세면, 결과가 20줄인데 쿼리는 두 번이고, 그중 하나가 전부 읽을 수 있다. 실용선은 단순하다. 카운트 쿼리에도 LIMIT 21을 걸어, 21행이 나오면 "20개 이상"이라고 표현하거나, 카운트를 아예 생략한다. 사용자는 정확한 숫자보다 첫 페이지를 먼저 본다.

3.3 빈 결과 페이지는 설계 대상

검색 결과가 0개인 것은 버그가 아니다. 그것은 정보다. 그런데 대부분의 제품은 그 자리에 아무것도 안 보인다. 빈 화면은 "없다"를 말하기는 해도, "다음에 뭘 치나"를 말해 주지 못한다.

빈 결과 페이지에는 네 가지를 놓자.

첫째, 무엇을 쳤는지 복기. "ACME 미지급"에 대한 결과라고 적는다. 사용자가 다른 탭에서 다른 쿼리를 돌린 뒤로 돌아왔을 때를 위해서다.

둘째, 필터를 풀라는 제안. "조건을 지우고 다시 찾기" 버튼 하나. 다섯 유형 중 다섯째, 조건 섞임이 여기에서 풀린다.

셋째, 더 짧은 단어를 권한다. "ACME 미지급"이 0건이면, "ACME"만으로도 시도해 보라고 말한다. 두 단어가 합쳐지면 조건이 너무 좁아지는 경우가 많다.

넷째, 유사한 단어 제안. "이거 찾으시는 거?" 아래에 2~3개. 구현은 생각보다 쉽다. trigram 유사도로, 저장된 제목 중에서 쿼리와 가장 비슷한 상위 3개를 뽑아, 임계값 이상이면 보여준다.

```
SELECT name FROM projects
ORDER BY name % $1 DESC
LIMIT 3;
```

그리고 0결과 자체를 기록한다. 매 검색에서 쿼리, 결과 수, 시각을 한 행씩 넣는다. 이 로그가 쌓이면, 0결과가 자주 모이는 단어군을 볼 수 있고, 그게 곧 어휘 격차의 보고서다. 사용자가 "인보이스"를 40번 쳤는데 데이터에는 "청구서"만 있다면, 동의어 지도에 한 줄 추가할 이유가 생긴 것이다. 이 로그의 효용은 4장에서 더 크게 나온다.

마지막 안전망도 하나. 0결과이고 유사어 후보도 없는 자리에, 인기 검색어를 보여준다. 이번 달에 가장 많이 검색된 다섯 문장. 이건 사용자의 문제를 풀지는 못한다. 하지만 동작하는 쿼리를 하나 쥐여주는 것은 된다. 아무것도 없는 화면보다, 시험해 볼 것이 하나 있는 화면이 낫다.

빈 결과 페이지에 절대 놓지 말 것도 있다. 내부 오류, SQL 문장의 조각, "no rows returned" 같은 기계 문장. 기계 문장은 다음 행동을 말해주지 못한다.

3.4 정렬의 긴장: 관련도 대 시간

정렬은 두 개의 정직한 답 사이에서 고르는 문제다. 관련도 순은 "당신이 친 것에 가장 가까운 것부터"이고 시간 순은 "최근에 변한 것부터"다. 둘 다 옳고, 둘 다 때로는 틀리다.

실용선은 조건부다. 검색어가 없으면 시간 순. 검색어가 있으면 관련도 순. 사용자가 아무것도 안 친 채로 목록을 여는 순간에는, "최근에 무엇이 변했는지"가 기본 기대다. 반대로 "ACME"를 친 순간에는, 관련성이 우선이 되어야 한다. 이 규칙은 한 줄로 정의할 수 있다. 쿼리가 비어 있으면 updated_at DESC, 그렇지 않으면 score DESC.

정렬 옵션을 여러 개 주지 마라. 두 개면 충분하고 셋 이상이면 사용자는 매 검색마다 정렬을 다시 고른다. 고르는 횟수가 늘면, 멈추는 횟수는 줄어든다.

또 하나, 동점 처리. 관련도 점수가 같은 행이 여러 개면, 그 순서는 정의가 안 된다. 페이지를 넘길 때마다 같은 행의 순서가 바뀌면, 사용자는 목록이 움직인다고 느낀다. id를 마지막 타이브레이커로 늘 붙인다.

```
ORDER BY score DESC, updated_at DESC, id;
```

이 한 컬럼이, 결과 목록의 흔들림을 끝낸다.

정렬에는 일관성이라는 또 한 면이 있다. 사용자가 같은 쿼리를 두 번 치면, 두 목록은 같은 순서여야 한다. 순서가 바뀌면, 사용자는 데이터가 움직였다고 느낀다. 흔한 원인은 id 없이 동점을 처리하는 것, 그리고 시간이 지나면 바뀌는 값으로 정렬하는 것이다. 정렬 키에 시간이 지남으로써 바뀌는 값이 들어가면, 타이브레이커로 불변의 키를 붙여라.

3.5 페이지네이션: OFFSET의 한계

20개씩 페이지를 넘기면, SQL은 이 모양이 된다.

```
LIMIT 20 OFFSET 40;
```

OFFSET 40은, 첫 40개를 찾아서 버리라는 뜻이다. 데이터베이스는 실제로 그렇게 한다. OFFSET이 커질수록, 버려야 할 행이 커진다. 100페이지라면 2,000개를 버린다. 이 비용은 관련도 정렬에서는 더 아프다. 정렬 키가 복합 점수라면, 버리는 행도 전부 점수를 계산한 후다.

전문적인 답은 키셋 페이지네이션이다. 마지막에 본 행의 키를 기억하고 그보다 뒤만 가져온다.

```
WHERE (updated_at, id) < ($last_updated_at, $last_id)
ORDER BY updated_at DESC, id DESC
LIMIT 20;
```

이건 인덱스를 탄다. 하지만 관련도 점수로는 쓸 수가 없다. 점수는 쿼리마다, 행마다 바뀌기 때문에, "그 점수보다 작은 것"이라 쓸 수 없다. 키셋은 정렬 키가 데이터 자체의 값일 때만 성립한다.

1인 제품을 위한 실용선은, 검색은 3페이지를 넘지 않는다는 관찰이다. [추정] 사용자가 10페이지까지 넘기는 것은, 거의 항상 "다른 표현으로 치는 게 낫다"는 신호다. 그래서 페이지 번호를 끝없이 늘리기보다, "더 보기"를 몇 번이고 그 이상은 필터 또는 빈 결과 제안을 권하는 쪽이 경험상 낫다. 무한 스크롤은 검색에 맞지 않는다. 사용자가 관련 행의 총량을 몰라야, 다음 단어를 칠 이유가 생긴다.

3.6 미리보기의 모양

결과 행이 클릭 없이 "이거다"를 말하게 하려면, 매 행에 세 가지를 놓자.

첫째, 제목. 결과 행에서 제목이 작으면, 사용자가 매 행을 확대해서 읽는다.

둘째, 하나의 식별 맥락. 소유자, 날짜, 상태, 숫자. "ACME 4월 청구서, 미지급, 8,200달러" 같은 형태. 제목만으로는 구분하지 못하는 행들이, 이 한 칸으로 구분된다.

셋째, 매칭 하이라이트. 검색한 단어가 행 안에서 어디에 들어갔는지, mark 태그로 표시한다. ILIKE의 경우, 앱 코드에서 위치를 찾아 감싸면 된다. 사용자가 20줄을 훑을 때, 매칭 위치를 보면, "이건 관련 있다"를 한눈에 판별한다.

식별 맥락의 그 한 칸은, 엔티티에 따라 다른 것으로 채워진다. 주문이면 금액과 날짜다. 고객이면 회사명과 최근 활동이다. 문서면 작성자와

수정일이다. 고르는 기준은 하나다. 사용자가 비슷한 제목의 두 행을 보았을 때, 이 칸 하나로 열지 않고 구분할 수 있느냐. 그게 그 엔티티의 식별 맥락이다.

그냥 목록이다.

3.7 이 장의 규율

결과 페이지가 검색을 완성한다. 빈 결과도, 정렬도, 미리보기도, 사용자의 다음 행동을 설계하는 것들이다. 다음 장에서는, 이 검색이 규모가 커지는 과정에서 실제로 넘어지는 자리를, 순서대로 정리한다.

4 4장. 규모가 커지기 전에 넘어지는 자리

4.1 인덱스는 약속이지 보증이 아니다

2장에서 인덱스를 만들었다. 하지만 인덱스가 있는 쿼리는, 인덱스를 쓰지 않을 수 있다. 플랜러는 비용을 계산해서, 전부 읽는 쪽이 싸다고 판단하면 순차 스캔을 고른다. 검색이 느리다고 불평이 온 첫날에 할 일은, EXPLAIN이다.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, name FROM projects
WHERE name ILIKE '%성장%'
LIMIT 20;
```

결과에 Seq Scan이 보이면, 인덱스가 탄 게 아니라 표를 읽은 것이다. Index Scan 또는 Bitmap Index Scan이어야 한다. trigram 인덱스가 왜 안 타나를 묻자. 가장 흔한 답은 검색어가 2글자 이하라는 것. 짧은 검색어는 만드는 조각이 흔해서, 인덱스를 써도 후보가 표 전체에 가까워지므로, 플랜러가 이를 수 있다. 두 번째로 흔한 답은, 인덱스를 만든 컬럼이 쿼리의 컬럼과 다르다는 것. name에 인덱스인데 description으로 검색했다거나.

EXPLAIN은 검색 코드를 건드릴 때마다 실행할 도구다. 일주일에 한 번씩만 실행해서는 안 된다. 성능 문제가 "느려졌다"에서 시작하면, 증명이 없기 때문에 추측 싸움이 된다. EXPLAIN의 한 줄은 그 싸움을 끝낸다.

3글자 이상인데도 Seq Scan이 보인다면, 두 번째 원인을 보자. trigram 인덱스도, 매칭 후보가 너무 많으면 무시된다. "프로" 같은 흔한 3글자 검색어를 치면, 표의 절반이 후보가 될 수 있다. 후보가 절반이면, 인덱스로 좁혔다 다시 읽는 것이 전부 읽는 것보다 비싸다고 플랜러는 계산한다.

그래서 인덱스가 있는 것과, 인덱스가 쓰인 것은 별개다. EXPLAIN 없이는 알 수 없다.

여기에 한 가지를 덧붙이자. LIMIT 20이 붙어도, 점수 순으로 정렬해야 한다면, 데이터베이스는 WHERE를 통과한 모든 행에 먼저 점수를 매겨야 한다. LIMIT은 마지막에 행을 자를 뿐, 계산 자체를 줄여주지는 않는다. 그래서 2장의 점수 쿼리는, WHERE로 먼저 좁힌 뒤에 점수를 매기는 것이 원칙이다. 점수를 먼저 매기고 WHERE로 거르는 순서는, 읽을 행이 많은 쿼리가 된다.

4.2 결과 페이지의 N+1

검색 쿼리는 20개의 id를 가져온다. 그 다음에, 결과 페이지가 매 행의 상세, 소유자 이름, 최근 활동을 추가로 조회한다. 행별로 하나씩 조회하면, 쿼리는 1 + 20 + 20 + 20, 61개다. 한 번의 페이지 뷰에 61번 왕복.

목록 화면의 고전적 문제다. 이것은 검색만의 문제가 아니다. 하지만 검색 결과 페이지는 특히 자주 당한다. 그 이유는, 검색 결과가 "여러 테이블의 정보를 섞어 보여주는" 미리보기를 요구하기 때문이다.

수정은 둘 중 하나다. JOIN으로 한 쿼리에 넣거나, WHERE id = ANY(배열)로 두 번째 쿼리에 20행을 한 번에 가져온다.

```
SELECT owner.name, p.id, p.name
FROM projects p
JOIN owners ON owners.id = p.owner_id
WHERE p.id = ANY($ids)
```

미리보기에 실제로 필요한 컬럼만 가져오자. 결과 행에 4개 필드가 보이는데, 30개 컬럼의 전체 행을 20번 가져오는 것은, 전송과 메모리 둘 다 낭비다.

서버가 먼저 안다. N+1이 드러나는 자리는, 보통 사용자가 아니다. 데이터베이스의 연결 사용량이, 검색 트래픽과 함께 오르내린다면, 그 모양은 N+1이다. 쿼리 로그를 보면, 같은 SQL이 20번 반복되어 있다. 한 번의 페이지 뷰에 나는 쿼리가 몇 개인지 세어보자. 10개를 넘으면, 구조의 문제다. [추정]

4.3 search_text 컬럼: 하나의 대가

여러 컬럼을 OR로 검색하는 방식에는, 유지보수의 벽이 있다. 검색에 넣을 컬럼을 하나 추가할 때마다, 검색 쿼리도, 인덱스도, 가중치도 손을 대야 한다. 다섯 컬럼의 OR ILIKE는 플랜러도, 읽는 사람도 어렵다.

대안은, 쓰는 쪽에서 미리 합쳐둔 컬럼이다. search_text.

```
ALTER TABLE projects ADD COLUMN search_text text;
```

쓰는 쪽에서, search_text에 name과 description과 tags를 합쳐서 저장한다. 읽는 쪽은 한 컬럼만 다룬다. 인덱스도 한 개. 가중치는, search_text에 제목을 맨 앞에 놓는 것으로 근사할 수 있다. 더 정직한 건, 제목 매칭을 따로 한 컬럼으로 남겨, 제목이면 3점, search_text면 1점으로 매기는 것이다.

대가도 명확하다. search_text는 쓰는 모든 경로에서 갱신되어야 한다. 모델의 저장, 직접 SQL, 배치 스크립트, 관리자 화면. 한 곳이라도 놓이면, 그 행은 검색에서 영원히 사라진다. 데이터는 있는데 못 찾는다는 1장의 고장이, 이번에는 정말로 저장의 고장이 된다.

방어는 둘이다. 첫 번째, 갱신을 트리거 또는 단일 메서드에 모아, "직접 컬럼에 쓰는" 코드를 만들지 않는다. 두 번째, 정기적으로 search_text를 재구성하는 재생성 잡을 둔다. 읽기 중심 데이터라면, 하루 한 번의 재생성은 싼 보험이다. 재생성 잡은, search_text와 원 컬럼이 다른 행이 0개여야

한다는 것을 검사할 수도 있다. 검사에서 0이 아니면, 갱신 경로가 어딘가에 빠져 있다는 증거다.

트리거로 쓰는 모양은 이렇다.

```
CREATE OR REPLACE FUNCTION update_search_text()
RETURNS trigger AS $$
BEGIN
 NEW.search_text := NEW.name || ' ' || NEW.description || ' ' ||
  ↪  NEW.tags;
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

트리거는 앱 코드 밖에 있는 안전망이다. 관리자가 SQL로 직접 행을 고쳐도, search_text는 따라 갱신된다. 앱 코드만의 규칙은, 앱 코드를 우회하는 순간 무너진다.

4.4 검색 로그는 제품 데이터

3장에서 0결과를 기록하자고 했다. 제품의 요구 기록이다. 이 로그는, 검색 기능의 성능 기록이 아니다.

한 달치를 세 가지만 봐도 된다.

0결과가 반복되는 단어. "인보이스"가 40번이면, 어휘 격차의 결정적 증거다. 동의어 지도 한 줄, 또는 등록 시 저장 형식 하나로 해결.

많이 잡히지 않는 큰 쿼리. 결과 수는 많지만 사용자가 아무 행도 클릭하지 않는 쿼리. 미리보기가 "이거다"를 말해 주지 못했다는 증거다. 3장의 미리보기 세 가지로 돌아가서 고친다.

상위 몇 개의 쿼리. 검색은 입력이 다양하지만 인기 쿼리는 반복된다.

SaaS라면 “상위 고객명”, “상위 제품명”이 며칠 내내 상위 10개를 차지한다. [추정] 이 쿼리의 결과를 짧은 시간 캐싱하면, 가장 자주 열리는 페이지가 제일 빨라진다. 캐싱의 유효기간은 짧게, 쓰기가 일어나면 무효화. 또는, 이 규모에서는 캐시 없이도 인덱스로 충분하므로, 아예 안 하는 것. 둘 다 정답이고 고르는 기준은 2장의 실용선과 같다.

한 단계 위를 가려면 클릭을 기록한다. 결과 행마다 클릭이 몇 번 있었는지를 남기면, 미리보기가 실제로 어디서 멈추게 하는지 안다. 클릭이 한 번도 없는 쿼리는, 결과 수가 몇이든 실패한 검색이다. 이 세 가지는 같은 테이블에 들어간다. 쿼리, 결과 수, 클릭 수. 한 줄의 로그가 검색 기능의 전체 건강을 담는다.

로그의 작성은 어렵지 않다. 검색 요청을 거치는 함수 한 곳에, 쿼리, 결과 수, 시각을 한 행씩 넣는 것. 주 10분, psql에서 세 쿼리. 분석은 BI 도구가 아니다.

캐시를 넣을 때도 이 로그가 기준이 된다. 상위 쿼리만 캐싱하고 유효기간은 짧게, 그리고 한 가지 규칙을 잊지 않는다. 쓰기가 일어나면 캐시를 모두 버린다. 속도를 조금 잃지만 틀린 데이터를 보여주는 일은 없다. 검색에서 캐시의 위험은 빠름이 아니다. “방금 등록한 게 왜 안 나와”라는 질문으로 돌아온다는 점이 위험하다.

4.5 이주의 선: 언제 전용 엔진으로 갈까

결론부터. 지금 없는 성능을 위해, 이주하지 않는다. 이주는, 데이터베이스가 못 하는 기능을 원할 때 시작한다.

데이터베이스가 버티는 선은, trigram 인덱스가 살고 EXPLAIN이 Index Scan을 보이고, 쿼리가 300밀리세컨트 안쪽인 한이다. [추정] 몇 십만 행 SaaS라면, 이 선은 수년. 이 선 안에 있으면, Meilisearch를 추가하는 순간부터, 동기화 잡, 두 소스의 불일치, 백업, 배포가 네 개의 새로운 일로

추가된다.

이주가 논의를 받을 신호는 네 가지다. 오타 관용이 제품의 얼굴이 되었다. “crewt”를 치는 사용자가 정정을 안 치고도 답을 받아야 한다면, trigram은 거기까지다. 다국어가 되었다. 한 제품에 한국어와 영어 데이터가 섞이고 각각의 언어에 맞는 매칭이 필요해졌다. 행이 천만 단위로 넘어섰다. 그리고 팀이 서비스를 하나 더 돌릴 사람이 생겼다. 이 신호가 하나도 없다면, 데이터베이스에 남아 있는 것이 싼 쪽이다.

이주를 실제로 한다면, 형식은 보통 같다. 쓰는 쪽에서 변경을 전용 엔진으로 동기화하는 잡을 두고 읽는 쪽만 엔진으로 돌린다. 데이터베이스는 여전히 진실의 원천이고, 엔진은 인덱스의 또 다른 형태다. 이 선을 잃으면, “어느 쪽이 맞나”를 묻는 아침이 온다.

4.6 늘 따라오는 필터: 테넌트

SaaS라면, 모든 검색에 한 가지 필터가 처음부터 붙어 있다. WHERE tenant_id = $1. 이걸 빼면, A 사용자는 B 사용자의 데이터를 찾게 된다. 이 필터는 전제다. 기능이 아니다. 그리고 인덱스 설계에도 관여한다.

이론에서는 플래너가 tenant_id 인덱스와 name의 trigram 인덱스를 bitmap AND로 합칠 수 있다. 하지만 결합의 비용은 오르고 실제로 쓰이는지는 EXPLAIN으로 확인할 수밖에 없다. 흔히 보는 패턴은 이렇다. 테넌트의 행이 적으면, tenant_id로 그 테넌트의 행을 읽고, 거기에 ILIKE를 거는 식이다. 빠르다. 테넌트의 행이 몇 십만이면, 그 테넌트의 검색은 느려진다.

대안은, 검색 컬럼 자체를 테넌트 인지하게 만드는 것이다. search_text에 tenant_id와 name을 함께 넣거나, 가장 큰 테넌트에게는 별도 인덱스를 준다. 이 결정은 규모에 따라 바뀌지 않는다. 열 명짜리 제품인 지금부터 이 구조를 기억해 두면 된다. 첫 큰 테넌트가 들어올 때가 아니라.

4.7 출시 전 10개 체크

마지막으로, 검색을 출시하기 전에, 또는 크게 건드린 후에, 줄줄이 확인하는 목록.

1. 20개 쿼리 세트를 돌렸고 잡힌 수가 이전에 늘었는가.
2. 1~2글자 검색은 제한하거나, 전부 읽는다는 것을 알고 있는가.
3. EXPLAIN으로, 실제 검색 쿼리가 인덱스를 쓰는지 확인했는가.
4. 빈 결과 페이지에, 풀기, 짧은 단어, 유사어 제안이 있는가.
5. 정렬에 id 타이브레이커가 있어서 페이지 넘김이 흔들리지 않는가.
6. LIMIT 상한이 있어서 아무도 100만 행을 못 가져가는가.
7. 검색 로그가, 쿼리와 결과 수를 기록하는가.
8. 한 페이지 뷰에, 나는 쿼리가 10개 미만인가.
9. search_text 갱신 경로가, 모든 쓰기 경로에서 동작하는가.
10. 검색창의 플레이스홀더가, 무엇을 검색하는지 말하는가.

4.8 끝

검색은, 사용자가 친 문장을 데이터베이스가 아는 문장으로 번역하는 기능이다. 이 책에서 다룬 규율은 결국 하나다. 번역의 각 단계를, 증명으로 다뤄라. 추측으로는 안 된다. 20개 쿼리가 번역의 테스트 세트를, EXPLAIN이 인덱스의 증명을, 검색 로그가 다음 번역 개선의 자료가 된다.

"찾는" 기능은 단순해 보인다. 데이터베이스에 기록이 있고 사용자가 문장을 치면, 맞는 행이 나오는 것. 그런데 그 "맞는"을 정의하는 순간, 어휘, 형태, 순서, 페이지, 규모가 전부 문제를 낸다. 1인 개발자에게 이 기능은, 제품의 얼굴 중에서도 특히 얼굴이 된다. 사용자가 제품을 "찾기 좋은"이라고 느끼는 것은, 거의 항상 검색에서 느껴진다고 나는 생각한다.

그래서 검색은, "나중에"를 정해두지 않는 것이 낫다. 다음 주에 20개 쿼리를 적어 보자. 이 책의 나머지는, 그 20개에 대한 답이다.