[week5] 8-9장

🌟 8장. URL 단축기 설계

1️⃣ 단계 : 문제 이해 및 설계 범위 확정

- 시스템 기본적 기능
  - URL 단축 : 주어진 긴 URL을 훨씬 짧게줄인다.
  - URL 리디렉션(redirection) : 축약된 URL로 HTTP 요청이 오면 원래 URL로 안내
  - 높은 가용성과 규모 확장성, 그리고 장애 감내가 요구

- 요구사항
  - 트래픽 규모 : 매일 1억개의 단축 URL을 만들어낼 수 있어야 함
  - 단축 URL의 길이는 짧으면 짧을수록 좋음
  - 단축 URL에는 숫자(0부터 9까지)와 영문자(a부터 z, A부터 Z까지)만 사용 가능
  - 단축된 URL을 시스템에서 삭제하거나 갱신할 수 없다고 가정

📍 개략적 추정

- 쓰기 연산 : 매일 1억 개의 단축 URL 생성
  - 초당 쓰기 연산 : 1억 / 24 / 3600 = 1160
  - 읽기 연산 : 읽기 연산과 쓰기 연산 비율을 10 : 1이라 하면 → 초당 11,600회 발생

- URL 단축 서비스를 10년간 운영한다고 가정하면 1억 * 365 * 10 = 3650억 개의 레코드를 보관해야 함
  - 축약 전 URL의 평균 길이를 100이라고 하면
  - 10년동안 필요한 저장 용량은 3650억 * 100바이트 = 36.5TB

2️⃣ 단계 : 개략적 설계안 제시 및 동의 구하기

📍 API 엔드포인트

1. **URL 단축용 엔드포인트**
   - 새 단축 URL을 생성하고자 하는 클라이언트가 단축할 URL을 인자로 실어서 POST 요청

👉 `POST /api/v1/data`

- 인자 : `{longUrl: longURLstring}`
- 반환 : 단축 URL

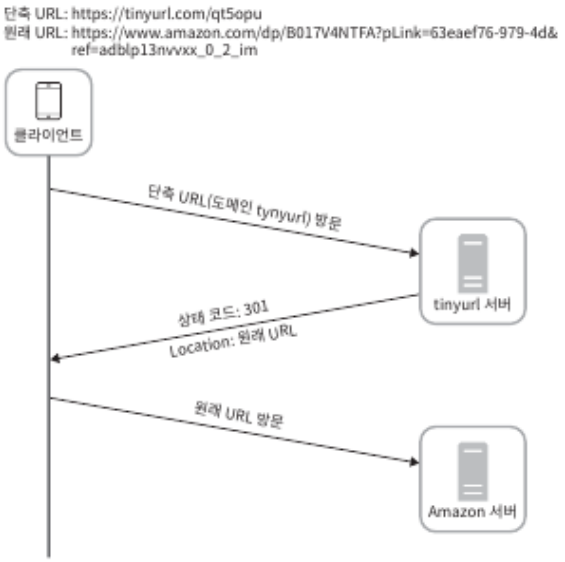
2. **URL 리디렉션용 엔드포인트**
    - 단축 URL에 대해서 HTTP 요청이 오면 원래 URL로 보내주기 위한 용도

👉 `GET /api/v1/shortUrl`

- 반환 : HTTP 리디렉션 목적지가 될 원래 URL

📍 URL 리디렉션

- 단축 URL을 받은 서버는 그 URL을 원래 URL로 바꾸어서 301 응답의 Location 헤더에 넣어 반환



- 리디렉션 응답 301과 302의 차이

- **301 Permanently Moved**
  - 해당 URL에 대한 HTTP 요청의 처리 책임이 영구적으로 Location 헤더에 반환된 URL로 이전
  - 브라우저가 응답을 캐시(cache)

    → 같은 단축 URL에 요청을 보낼 때 캐시된 브라우저가 원래 URL로 요청 보냄
- **302 Found**
  - 주어진 URL로의 요청이 '일시적으로' Location 헤더가 지정하는 URL에 의해 처리되어야 한다.
  - 클라이언트 요청이 언제나 단축 URL 서버에 먼저 보내진 후에 원래 URL로 리디렉션

> 첫 번째 요청만 단축 URL 서버로 전송되기 때문에 서버 부하를 줄이는 것이 중요하면 301이 좋다
>
> 클릭 발생률이나 발생 위치를 추적하는 트래픽 분석이 중요할 때는 302가 좋다

- URL 리디렉션 구현 방법 : 해시 테이블 사용
  - 해시 테이블에 <단축 URL, 원래 URL> 쌍을 저장한다고 가정하면
    - 원래 URL = hasTable.get(단축 URL)
    - 301 또는 302 응답 Location 헤더에 원래 URL을 넣은 후 전송

📍 URL 단축

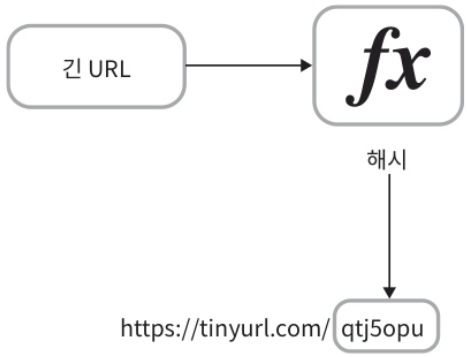


그림 8-3

- 긴 URL을 해시 값으로 대응시킬 해시 함수 fx를 찾는 것이 중요
- 해시 함수 요구사항
  - 입력으로 주어지는 긴 URL이 다른 값이면 해시 값도 달라야 한다.
  - 계산된 해시 값은 원래 입력으로 주어졌던 긴 URL로 복원될 수 있어야 한다.

3️⃣ 단계 : 상세 설계

📍 데이터 모델

- 모두 해시 테이블에 저장 → 메모리 유한, 비용 이슈
- `<단축 URL, 원래 URL>`의 순서쌍을 **관계형 데이터베이스**에 저장

url	
PK	id
	shortURL longURL

그림 8-4

📍 해시 함수

> 원래 URL을 단축 URL로 변환하는 데 쓰이는 해시 함수가 계산하는 단축 URL 값을 hashValue라 지칭

- **해시 값 길이**

- 문자 개수 : 0-9 + a-z + A-Z = 62개
- 62^n **≥ 3650억** 인 n의 최솟값을 찾아야 한다 → **n = 7**일 때부터

n	URL 개수
1	62^1 = 62
2	62^2 = 3,844
3	62^3 = 238,328
4	62^4 = 14,776,336
5	62^5 = 916,132,832
6	62^6 = 56,800,235,584
7	62^7 = 3,521,614,606,208 = ~3.5조(trillion)
8	62^8 = 218,340,105,584,896

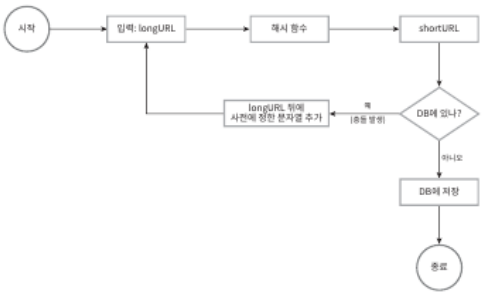
표 8-1

해시 함수 구현 기술 두 가지

- **해시 후 충돌 해소**
  - CRC32, MD5, SHA-1같이 잘 알려진 해시 함수 사용 → 길이가 7보다 길어진다.
  - 계산된 해시 값에서 처음 7개 글자만 사용 → 해시 결과가 충돌할 확률이 높아진다.
  - 충돌이 해소될 때까지 사전에 정한 문자열을 해시값에 덧붙여서 해결

    → 충돌 해소할 수 있지만 단축 URL 생성할 때 한 번 이상 데이터베이스 질의 필요

    → 데이터베이스 대신 블룸 필터 사용



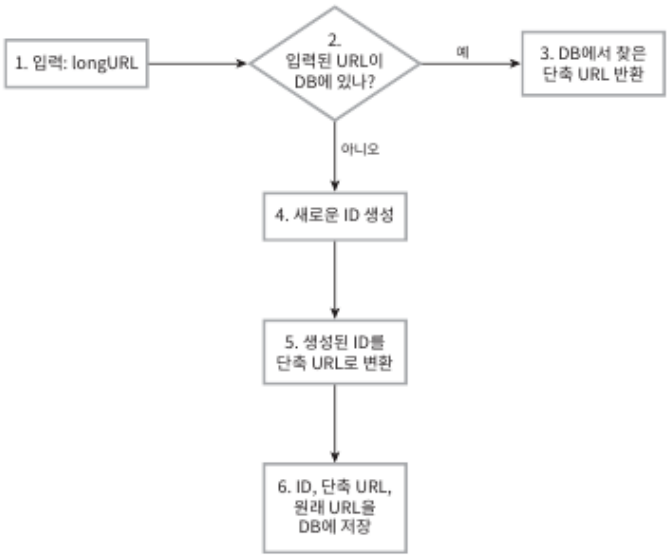
- **base-62 변환**
  - 62진법 : 0은 0으로, 9는 9로, 10은 a로, 11은 b로, ... 35는 z로, 36은 A로, ... 61은 Z로 대응
  - 10진수 11157 → 2 x62^2 + 55 x 62^1 + 59 x 62^0 = [2, 55, 59] = [2, T, X]

- 두 접근법 비교

해시 후 충돌 해소 전략	base-62 변환
단축 URL 길이 **고정**	단축 URL 길이 **가변적** (ID값 커지면 같이 길어짐)

해시 후 충돌 해소 전략	base-62 변환
유일성이 보장되는 ID 생성기 필요 없음	전역적 유일성 보장 ID 생성기 필요
충돌 가능해서 해소 전략 필요	ID의 유일성이 보장된 후에 적용 가능한 전략이므로 충돌 불가능
ID로부터 단축 URL을 계산하는 방식이 아니라서 **다음에 쓸 수 있는 URL을 알아내는 것이 불가능**	ID가 1씩 증가하는 값이라고 가정하면 다음에 쓸 수 있는 단축 URL이 무엇인지 쉽게 알아낼 수 있어 보안상 문제

📍 URL 단축기 상세 설계



1. 입력으로 긴 URL을 받는다.
2. 데이터베이스에 해당 URL이 있는지 검사한다.
    a. 데이터베이스에 있는 경우

        → 해당 URL에 대한 단축 URL을 만든 적이 있는 것이므로 데이터베이스에서 해당 단축 URI을 가져와서 클라이언트에 반환

    b. 데이터베이스에 없는 경우

        → 해당 URL은 새로 접수된 것이므로 유일한 ID를 생성하고 데이터베이스의 기본 키로 사용
3. 62진법 변환을 적용해서 ID를 단축 URL로 만든다.
4. ID, 단축 URL, 원래 URL로 새 데이터베이스 레코드를 만든 후 단축 URL을 클라이언트에 전달

📍 URL 리디렉션 상세 설계

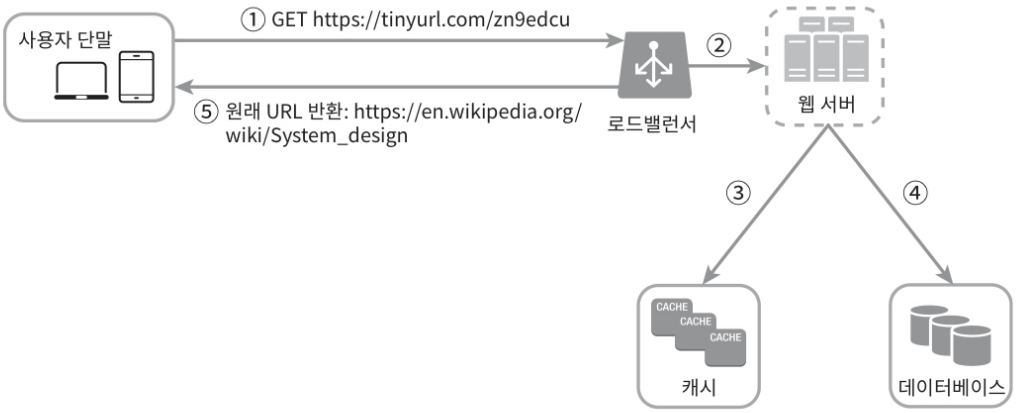


그림 8-8

- 쓰기보다 읽기를 자주 하는 시스템, <단축URL, 원래URL> 쌍을 캐시에 저장하여 성능 높임
- 로드밸런서의 동작 흐름

1. 사용자가 단축 URL을 클릭한다.
2. 로드밸런서가 해당 클릭으로 발생한 요청을 웹 서버에 전달한다.
3. 단축 URI이 이미 캐시에 있는 경우에는 원래 URL을 바로 꺼내서 클라이언트에게 전달한다.
4. 캐시에 해당 단축 URL이 없는 경우에는 데이터베이스에서 꺼낸다. 데이터베이스에 없다면 아마 사용자가 잘못된 단축 URL을 입력한 경우일 것이다.
5. 데이터베이스에서 꺼낸 URL을 캐시에 넣은 후 사용자에게 반환한다.

4️⃣ 단계 : 마무리

더 고려해 볼 사항

- **처리율 제한 장치** : 엄청난 양의 URL 단축 요청이 밀려들 경우
- **웹 서버의 규모 확장** : 웹 계층이 무상태 계층이므로 웹 서버 증설, 삭제 자유롭게 가능
- **데이터베이스의 규모 확장** : 데이터베이스의 다중화 혹은 샤딩
- **데이터 분석 솔루션** : 어떤 링크를 얼마나 많은 사용자가 클릭했는제 ,언제 주로 클릭했는지

- **가용성, 데이터 일관성, 안정성**

🌟 9장. 웹 크롤러 설계

1️⃣ 단계 : 문제 이해 및 설계 범위 확정

- 기본 알고리즘
  1. URL 집합이 입력으로 주어지면, 해당 URL들이 가리키는 모든 웹 페이지를 다운로드한다.
  2. 다운받은 웹 페이지에서 URL들을 추출한다.
  3. 추출된 URL들을 다운로드할 URL 목록에 추가하고 위의 과정을 처음부터 반복한다.

- 질문을 통해 파악한 기능 요구사항
  - 웹 크롤로의 주된 용도 : 검색 엔진 인덱싱
  - 매달 10억 개의 웹 페이지 수집
  - 새로 만들어진 or 수정된 웹 페이지도 고려
  - 수집한 웹 페이지는 5년간 저장
  - 중복된 콘텐츠를 갖는 페이지는 무시

- 기능 요구사항 외 고려할 속성
  - **규모 확장성** : 수십억 개의 페이지 → 병행성(parallelism) 활용
  - **안정성** : 잘못 작성된 HTML, 무반응 서버, 장애, 악성 코드 링크
  - **예절(politeness) :** 짧은 시간 동안 너무 많은 요청을 보내서는 안 된다
  - **확장성** : 새로운 형태 콘텐츠(예: 이미지)를 지원하기 쉬워야 한다

📍 개략적 규모 추정

- 매달 10억 개의 웹 페이지 다운로드

  → **QPS** = 10억 / 30일 / 24시간 / 3600초 = 대략 400페이지/초

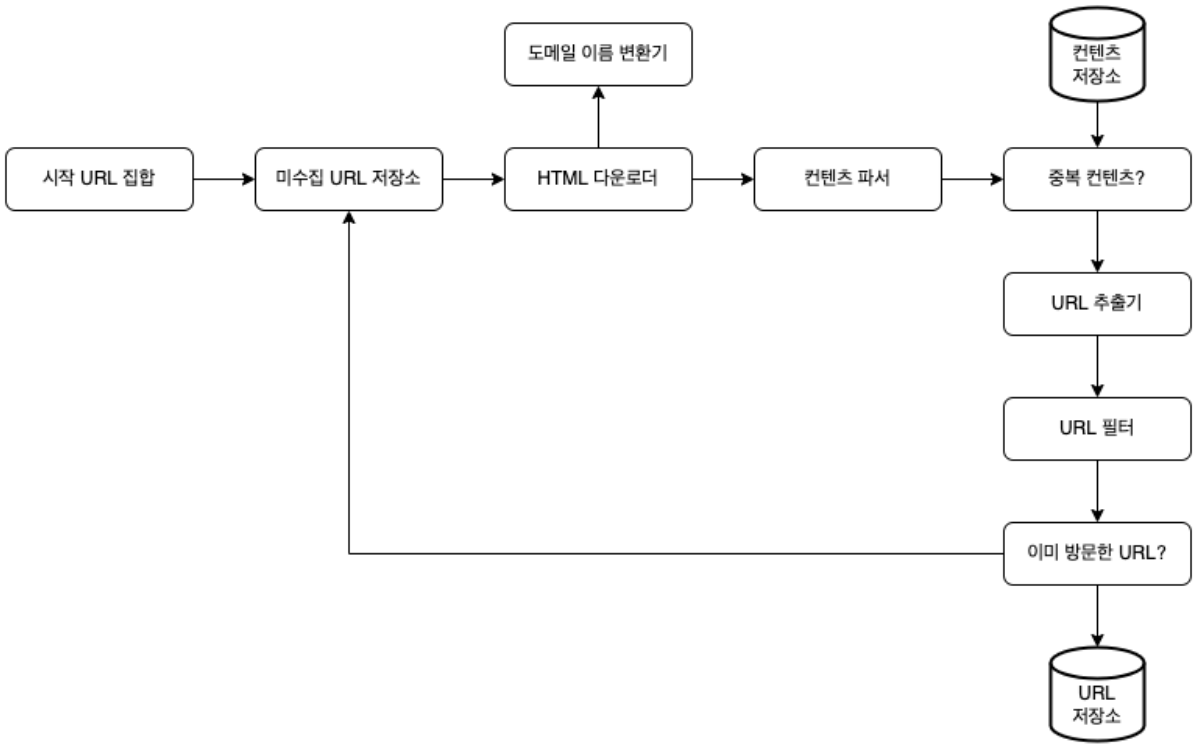
  → 최대(peak) QPS = 2 x QPS = 800

- 웹 페이지의 크기 평균이 500k라 가정

  → 10억 페이지 x 500k = 500TB/월

  → 5년간 보관하면 500TB x 12개월 x 5년 = 30PB 용량 필요

2️⃣ 단계 : 계략적 설계안 제시 및 동의 구하기



📍 시작 URL 집합

- 웹 크롤러가 크롤링을 시작하는 출발점
- 전체 웹 크롤링하는 경우 가능한 한 많은 링크를 탐색할 수 있도록 하는 URL

  → 전체 URL 공간을 작은 부분집합으로 나누는 전략 (예: 나라별, 주제별, ..)

📍 미수집 URL 저장소

- 크롤링 상태를 1️⃣ **다운로드 할 URL**, 2️⃣ **다운로드된 URL**의 두 가지로 나눠서 관리
- 1️⃣ **다운로드 할 URL**를 저장 관리하는 컴포넌트를 **미수집 URL 저장소(URL frontier)** → FIFO 큐

📍 HTML 다운로더

- 인터넷에서 **웹 페이지를 다운로드**하는 컴포넌트
- 다운로드할 페이지의 URL은 미수집 URL 저장소가 제공

📍 도메인 이름 변환기

- 웹페이지를 다운 받기 위해 필요한 절차
- HTML 다운로더가 도메인 이름 변환기를 사용하여 **URL에 대응되는 IP주소를 알아낸다**.

📍 콘텐츠 파서

- 웹 페이지 다운로드하면 **파싱(parsing)**과 **검증(validation)** 절차를 거쳐야 한다

  → 문제 발생/저장공간 낭비 방지
- 크롤링 서버 안에 콘텐츠 파서를 구현하면 크롤링 과정 느려짐 → 독립된 컴포넌트로

📍 중복 콘텐츠인가?

- 29% 웹 페이지 콘텐츠 중복 → 같은 콘텐츠를 여러 번 저장할 수 있음
- **자료 구조를 도입**하여 데이터 중복을 줄이고 데이터 처리에 소요되는 시간 줄임

- 두 HTML 문서를 비교할 때 **웹 페이지의 해시 값을 비교**하면 효율적

📍 콘텐츠 저장소

- HTML 문서 보관 시스템
- 데이터의 유형, 크기, 유효 기간이나 저장소 접근 빈도 등을 종합적으로 고려
- 예시에서는 디스크와 메모리를 동시에 사용
  - 데이터 양이 너무 많으므로 대부분의 콘텐츠는 디스크에 저장
  - 인기 있는 콘텐츠는 메모리에 두어 접근 지연시간 감소

📍 URL 추출기

- HTML 페이지를 파싱하여 링크들을 골라내는 역할
- 상대경로는 전부 절대 경로로 변환

📍 URL 필터

- 다음 같은 URL을 크롤링 대상에서 배제
  - 특정한 콘텐츠 타입이나 파일 확장자를 갖는 URL
  - 접속 시 오류가 발생하는 URL
  - 접근 제외 목록(deny list)에 포함된 URL

📍 이미 방문한 URL?

- 이미 방문한 URL, 미수집 URL 저장소에 보관된 URL 추적 가능한 자료구조 → 블룸 필터, 해시 테이블

  → 같은 URL 여러 번 처리하는 일 방지 → 서버 부하 줄이고 무한 루프 방지

📍 URL 저장소

- 이미 방문한 URL 보관하는 저장소

📍 웹크롤러 작업 흐름

1. 시작 URL들을 미수집 URL 저장소에 저장한다.
2. HTML 다운로더는 미수집 URL 저장소에서 URL 목록을 가져온다.
3. HTML 다운로더는 도메인 이름 변환기를 사용하여 URL의 IP 주소를 알아내고, 해당 IP 주소로 접속하여 웹 페이지를 다운받는다.
4. 콘텐츠 파서는 다운된 HTML 페이지를 파싱하여 올바른 형식을 갖춘 페이지인지 검증한다.
5. 콘텐츠 파싱과 검증이 끝나면 중복 콘텐츠인지 확인하는 절차를 개시한다.
6. 중복 콘텐츠인지 확인하기 위해서, 해당 페이지가 이미 저장소에 있는지 본다.
    a. 이미 저장소에 있는 콘텐츠인 경우에는 처리하지 않고 버린다.
    b. 저장소에 없는 콘텐츠인 경우에는 저장소에 저장한 뒤, URL 추출기로 전달한다.
7. URL 추출기는 해당 HTML 페이지에서 링크를 골라낸다.
8. 골라낸 링크를 URL 필터로 전달한다.
9. 필터링이 끝나고 남은 URL만 중복 URL 판별 단계로 전달한다.
10. 이미 처리한 URL인지 확인하기 위하여, URL 저장소에 보관된 URL인지 살핀다. 이미 저장소에 있는 URL은 버린다.
11. 저장소에 없는 URL은 URL 저장소에 저장할 뿐 아니라 미수집 URL 저장소에도 전달한다.

3️⃣ 단계 : 상세 설계

📍 DFS를 쓸 것인가, BFS를 쓸 것인가

- 웹 = 유향 그래프 → **페이지 = 노드**, **하이퍼링크(URL) = 에지(edge)**
- 깊이 가늠이 어렵기 때문에 DFS보다는 BFS 사용

  → FIFO 큐 : 한 쪽으로는 탐색할 URL을 집어넣고, 다른 한 쪽으로는 꺼내기만

→ 두 가지 문제 존재

- 한 페이지에서 나오는 링크의 상당수는 같은 서버로 되돌아간다.

  → 링크를 병렬로 처리하는 서버가 수많은 요청으로 과부하 위험

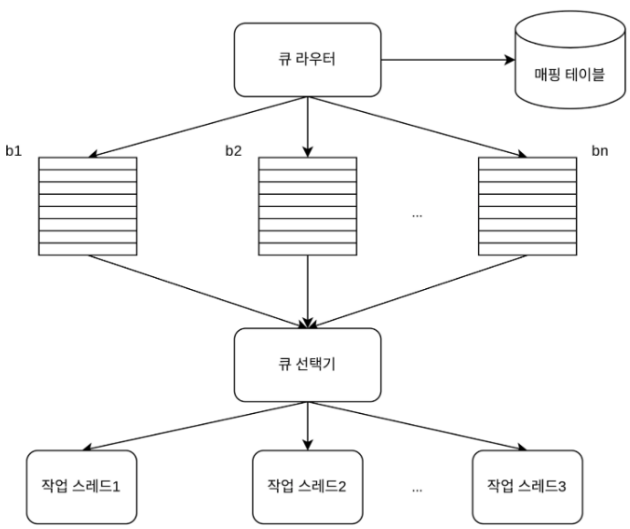
- 표준적 BFS 알고리즘은 URL 간에 우선순위를 두지 않는다.

  → 페이지 순위, 사용자 트래픽 양, 업데이트 빈도로 처리 우선순위 구별 필요

📍 미수집 URL 저장소

- '예의'를 갖춘 크롤러 = URL 사이의 우선순위와 신선도를 구별하는 크롤러
- 예의
  - 동일 웹사이트에 대해서는 한 번에 한 페이지만 요청
  - 호스트명(hostname)과 작업 스레드(worker thread) 사이의 관계 유지

    → 각 다운로드 스레드가 별도 FIFO 큐를 갖고 있어서, 해당 큐에서 꺼낸 URL만 다운로드하도록



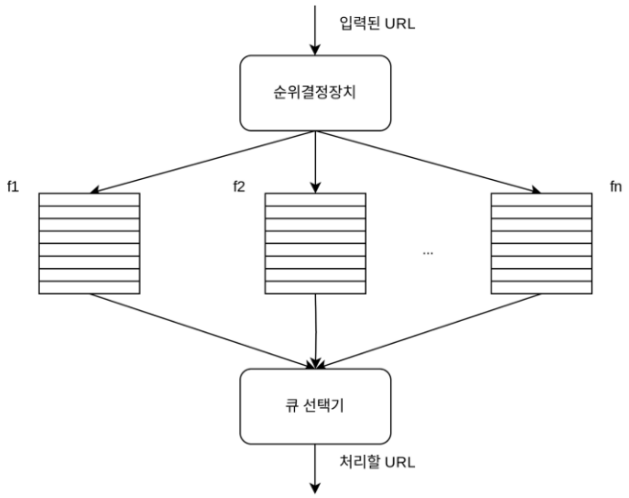
호스트	큐
wikipedia.com	b1
apple.com	b2
...	...
nike.com	bn

^ 매핑 테이블 예시

  - **큐 라우터** : 같은 호스트에 속한 URL은 언제나 같은 큐로 가도록 보장하는 역할
  - **매핑 테이블** : 호스트 이름과 큐 사이의 관계를 보관하는 테이블
  - **FIFO 큐** : 같은 호스트에 속한 URL은 언제나 같은 큐에 보관

- **큐 선택기** : 큐들을 순회하면서 큐에서 URL을 꺼내서 해당 큐에서 나온 URL을 다운로드하도록 지정된 작업 스레드에 전달
- **작업 스레드** : 전달된 URL을 다운로드하는 작업 수행

- 우선순위



  - **순위결정장치(prioritizer)** : URL을 입력으로 받아 우선순위를 계산한다.
  - **큐** : 우선순위별로 큐가 하나씩 할당. 우선순위 높으면 선택될 확률도 올라간다.
  - **큐 선택기** : 임의 큐에서 처리할 URL을 꺼내는 역할. 순위가 높은 큐에서 더 자주 꺼내도록

- 신선도
  - 웹 페이지는 수시로 추가, 삭제, 변경 → 주기적 재수집 필요 → 최적화하여 수행
    - 웹 페이지의 변경 이력(update history) 활용
    - 우선순위를 활용하여, 중요한 페이지는 좀 더 자주 수집

- 미수집 URL 저장소를 위한 지속성 저장장치
  - 대부분의 URL은 디스크에 두지만 IO비용을 줄이기 위해 메모리 버퍼에 큐를 두고 버퍼에 있는 데이터를 주기적으로 디스크에 기록

📍 HTML 다운로더

- `로봇 제외 프로토콜(Robot Exclusion Protocol) : Robots.txt`
  - 웹사이트가 크롤러와 소통하는 표준적 방법, 규칙
  - 파일에 크롤러가 수집해도 되는 페이지 목록이 들어있다 → 주기적으로 다시 다운받아 캐시에 보관
- `성능 최적화`
  1. **분산 크롤링** : 크롤링 작업을 여러 서버에 분산, 각 서버는 여러 스레드를 돌려 다운로드
  2. **도메인 이름 변환 결과 캐시**
     - DNS 요청을 보내고 결과 받는 작업의 동기적 특성 → 성능 병목 발생
     - DNS 조회 결과로 얻어진 도메인 이름과 IP 주소 사이의 관계를 캐시에 보관, 주기적으로 갱신
  3. **지역성** : 크롤링 작업을 수행하는 서버를 지역별로 분산
  4. **짧은 타임아웃** : 최대 얼마나 기다릴지 미리 정해둔다
- `안정성`
  - **안정 해시(consistent hashing)** : 다운로더 서버에 부하 분산할 때
  - **크롤링 상태 및 수집 데이터 저장** : 장애 발생에도 쉽게 복구하도록 지속적 저장장치에 기록
  - **예외 처리** : 예외가 발생해도 전체 시스템이 중단되지 않도록
  - **데이터 검증** : 시스템 오류 방지
- `확장성`
  - 새로운 형태의 콘텐츠를 쉽게 지원하도록 → 새로운 모듈 (예: PNG 다운로서, 웹 모니터)
- `문제 있는 콘텐츠 감지 및 회피`
  1. **중복 콘텐츠** → 해시나 체크섬(check-sum) 사용하여 탐지
  2. **거미 덫(spider trap)**
     a. 크롤러를 무한 루프에 빠뜨리도록 설계한 웹 페이지
     b. URL의 최대 길이 제한 → 만능 해결책은 없음 (이상할 정도로 많은 웹 페이지 가진 경우 일반적으로 그러함)
  3. **데이터 노이즈** : 광고, 스크립트 코드, 스팸 URL 제외

4️⃣ 단계 : 마무리

- 추가로 논의해 볼 사항
  - **서버 측 렌더링(server-side rendering)** : 링크를 즉석에서 만들어내는 경우 → 페이지를 파싱하기 전에 동적 렌더링 적용
  - **원치 않는 페이지 필터링** : 자원 유한 → 스팸 방지 컴포넌트
  - **데이터베이스 다중화 및 샤딩** : 데이터 계층의 가용성, 규모 확장성, 안정성 향상
  - **수평적 규모 확장성** : 무상태 서버로 만드는 것이 중요
  - **가용성, 일관성, 안정성**
  - **데이터 분석 솔루션**