웹 개발자를 위한 대규모 서비스를 지탱하는 기술

Created	@January 1, 2022 11:10 AM
Tags	DB 대규모 데이터
Progress	진행중
추가로 공부해야 할 부분	Column 중에 서버/인프라를 지탱하는 기술을 참고해서 Linux에서 시행해야 함

매일 발생하는 미지의 문제 - Check
대규모 데이터 처리의 어려운점
대규모 데이터의 어려움은 메모리 내에서 계산할 수 없다는 점
메모리와 디스크의 속도차
OS 레벨에서의 연구
전송속도, 버스의 속도차
규모조정의 요소
웹 애플리케이션과 부하의 관계
대규모 데이터를 다루기 위한 기초지식
대규모 데이터를 다루는 세 가지 급소
대규모 데이터를 다루기 전 3대 전제지식
OS 캐시와 분산
OS의 캐시 구조
가상 메모리 구조
Linux의 페이지 캐시 원리
페이지 캐시의 친숙한 효과
VFS
Linux는 페이지 단위로 디스크를 캐싱
LRU
(보충) 어떻게 캐싱될까?
메모리가 비어 있으면 캐싱
I/O 부하를 줄이는 방법
복수 서버로 확장시키기 (캐시로 해결될 수 없는 경우라면?)
단순히 대수만 늘려서는 확장성을 확보할 수 없다
국소성을 살리는 분산
1 - 파티셔닝
2 - 요청 패턴을 '섬'으로 분할
페이지 캐시를 고려한 운용의 기본 규칙

참고! 부하분산과 OS의 동작원리

- 데이터 구조. 메모리. OS DB 서버 인프라
- 미들웨어
- 버전 관리 시스템. 버그 추적 시스템

- 3억 5천만 레코드
  - select * from relword를 실행시 응답이 반환되지 않음

- 대규모 사이즈를 기준으로 본다면

  ex) 하테나 북마크
  - 레코드 수
    - entry 테이블 : 1,520만 엔트리
    - bookmark 테이블 : 4,500만 북마크
    - tag 테이블 : 5,000만 태그
  - 데이터 크기
    - 엔트리 : 3GB
    - 북마크 : 5.5GB
    - HTML : 200GB 이상, 태크 : 4.8G
- 대규모 데이터로의 쿼리?
  - 위의 규모의 DB에서 쿼리를 날리게 되면?

```
select url from entry use index(hoge) where eid = 9615899;
```

- use index(hoge) - 일부러 인덱스를 태우지 않고 쿼리를 던지고 있는 예
  - 1건 검색하는 데 200초가 경과해도 검색 결과가 나오지 않는다?

- 레코드 수가 수천만 건 단위, 데이터 크기는 수 GB부터 수백 GB.
  - 이 정도의 데이터가 되면, 아무 생각 없이 던진 쿼리에 응답하지 않음

    - 디버그 목적으로 데이터를 출력해보니, 엄청난 부하가 걸렸다는 말도 장난이 아닌 상황
- 데이터가 많으면 처리하는 데 시간이 걸리게 됨
    - 직감으로는 알겠으나 왜 그런가?

매일 발생하는 미지의 문제 - Check

- 크고 작은 많은 문제에 대한 노하우를 다수 보유하고 있는 것은
    - 그만큼의 트러블을 접해 왔다는 것
    - 트러블이 발생하지 않았다면 애초에 그런 문제가 시스템에 내재하고 있다는 것을 좀체 알 수 없었을 것
    - 데이터가 많아지면 왠지 수고스러울 것 같아 정도의 불안감은 있어도 실제로 어떤 노고가 있는지는 해보기 전까지는 몰랐을 것
    - 매일같이 뭔가 문제가 발생하면 이를 해결하기 위해 고민하는 것은 변함없음
        - 이렇게 되면 이렇게 해야 해라는 원리를 누군가가 알고 있는 것이 아닌 시행착오의 연속으로 노하우가 축적되어 지금에 이르렀다는 느낌
- 실제로 발생하는 문제는 미지의 문제도 많은 것이 현실임
    - 기본적인 사항은 파악해두고 '문제가 발생하면 그 자리에서 생각하자'라는 단순 명쾌한 생각도 필요하다는 것도 솔직한 생각!

대규모 데이터 처리의 어려운점

- 메모리 내에서 계산할 수 없음
    - 왜? 메모리에 올리지 않으면 기본적으로 디스크를 계속 읽어가면서 검색하게 됨
        - 좀처럼 발견할 수 없는 상태가 되어버림
    - 데이터 건수가 많으면 그만큼 입력 데이터 건수가 늘어나므로 계산량이 많아진다는 점도 당연한 이유
        - 이 점보다도 ⇒ `디스크를 읽고 있다` 는 점이 문제가 됨
- 메모리 내에서 계산할 수 있다면, 아무리 무식한 방법으로 하더라도, 계산은 빨리 이루어져 200초나 기다리는 일은 없을 것
    - 규모가 되면 데이터가 너무 많아서 메모리 내에서 계산할 수 없음
        - 디스크에 두고 특정 데이터를 검색하게 됨

- 그리고 디스크는 메모리에 비해 상당히 느림

대규모 데이터의 어려움은 메모리 내에서 계산할 수 없다는 점

- 메모리 내에서 계산할 수 없게 되면 디스크에 있는 데이터를 검색할 필요가 있음
  - 하지만 디스크는 느리므로 I/O(Input/Output)에 시간이 걸림
  - 그러면 어떻게 대처할 것인가?

메모리와 디스크의 속도차

- 메모리 내 특정 번지에 있는 데이터를 찾는 데이터 탐색과 디스크의 특정 원반 내에 있는 데이터를 찾는 것은 얼마의 속도차가 날까?
- 10^5 ~ 10^6 배 이상 빠름
  - 메모리는 디스크보다 10만 배~ 100만 배 이상 빠르다
- 디스크는 왜 늦을까?
  - 메모리는 전기적인 부품 ⇒ 물리적 구조는 탐색속도와 그다지 관계가 없음
    - 메모리는 마이크로초(10^-6)단위로 포인터를 이동시킬 수 있음
  - 디스크는 동축 상에 원반이 쌓여있음
    - 이 원반이 회전하고 있고 여기서 데이터를 읽어낸다
    - 메모리와 달리 회전 등의 물리적인 동작을 수반함
      - 이 물리적인 구조가 탐색 속도에 영향을 줌
    - 디스크에서는 헤드의 이동과 원반의 회전이라는 두가지 물리적인 이동이 필요함
      - 오늘날의 기술로도 원반의 회전속도를 빛의 속도까지 근접시킬 수는 없음
      - 디스크에는 각각 밀리초 단위, 합해서 수 밀리초나 걸림
    - 메모리는 1회 탐색 시 마이크로초면 되지만, 디스크는 수 밀리초나 걸리는 것

- 탐색에 사용되는 것이 CPU의 캐시에 올리기 쉬운 알고리즘이나 데이터 구조라면 메모리 내용이 CPU 캐시에 올라가므로 더욱 빨라져 나노초(10^-9) 단위로 처리할 수 있음

OS 레벨에서의 연구

- 디스크는 느리지만 OS는 이것을 어느 정도 커버하는 역할을 함
  - OS는 연속된 데이터를 같은 위치에 쌓음
  - 그리고 나서 데이터를 읽을 때 1바이트씩 읽는 것이 아닌 4KB(kilobytes) 정도를 한꺼번에 읽도록 되어 있음
  - 왜?
    - 비슷한 데이터를 비슷한 곳에 두어 1번의 디스크 회전으로 읽는 데이터 수를 많게 한다
      - 디스크의 회전 횟수를 최소화하지만, 결국 회전 1회당 밀리초 단위 ⇒ 메모리와의 속도 차를 피할 수 없음

전송속도, 버스의 속도차

- 탐색속도 측면에서 메모리가 디스크에 비해 10^5~10^6배 이상 빠르다는 얘기
  - 전송속도 차이도 존재
- 메모리와 디스크 모두 CPU, 버스로 연결되어 있음
  - 탐색과 전송의 차이에 유의!
- 전송속도 → 찾은 데이터를 디스크에서 메모리로 보내거나 메모리에서 CPU로 보내는 등 컴퓨터 내부에서 전송하기 위한 속도
- Linux에서 hdparm이라는 툴을 이용해보자
- 메모리와 CPU는 상당히 빠른 버스로 연결되어 있음
  - 따라서 전송해오는 중에도 시간이 걸림
- 데이터가 많아지면 많아질수록 디스크와 메모리의 차이도 나타나게 되므로 전송속도에서도 디스크는 늦어진다
- SSD는 물리적인 회전이 아니므로 탐색은 빠르지만 버스 속도가 병목이 되거나 그 밖에 구조에 기인하는 면이 있어서 역시나 메모리만큼의 속도는 나오지 않음

- 현대의 컴퓨터에서는 메모리와 디스크 속도차를 생각하고 애플리케이션을 만들어야 한다
  - 확장성을 생각할 때 매우 본질적이면서도 어려운 부분

- Linux 단일 호스트의 부하 - ‘서버/인프라를 지탱하는 기술’ 요약 : 찾아서 읽어보자

규모조정의 요소

- 웹 서비스에서는
  - 스케일업 - 고가의 빠른 하드웨어를 사서 성능을 높이는 전략
  - 스케일아웃 - 저가이면서 일반적인 성능의 하드웨어를 많이 나열해서 시스템 전체 성능을 올리는 전략
  - 스케일 아웃 전략이 더 나은 이유는?
    - 웹 서비스에 적합한 형태이고 비용이 저렴하다는 점과 시스템 구성에 유연성이 있다는 점이 포인트
    - 하드웨어 가격이 성능과 비례하지 않는다는 것
      - 동일한 성능을 확보할 때 저가의 하드웨어를 나열해서 확보하는 편이 더 낫다는 것
- 스케일아웃은 하드웨어를 나열해서 성능을 높이는 ⇒ 하드웨어를 횡으로 전개해서 확장성을 확보해가게 된다
  - 이 때 CPU 부하의 확장성을 확보하기는 쉽다
- 웹 애플리케이션에서 계산을 수행하고 있을 때,
  - HTTP 요청을 받아 DB에 질의하고 DB로부터 응답받은 데이터를 가공해서 HTML로 클라이언트에 반환할 때는 기본적으로 CPU 부하만 소요되는 부분
    - 서버 구성 중에 프록시나 AP(Application Server)가 담당할 일
- DB 서버 측면에서는 I/O 부하가 걸림

웹 애플리케이션과 부하의 관계

- 프록시, AP, DB
- 프록시에서 AP 서버로 요청이오고 DB에 도달해서 I/O가 발생함
  - 이 I/O가 발생해서 되돌아온 콘텐츠를 변경한 후 클라이언트로 응답함
  - 기본적으로 AP 서버에는 I/O 부하가 걸리지 않고 DB 측에 I/O 부하가 걸림
- AP 서버는 CPU 부하만 걸리므로 분산이 간단함
  - 왜? 기본적으로 데이터를 분산해서 갖고 있는 것이 아님
    - 동일한 호스트가 동일하게 작업을 처리하기만 하면 분산할 수 있음

        - 대수를 늘리기만 하면 간단히 확장해 갈 수 있음
    - 결국 새로운 서버를 추가하고자 한다면 원래 있던 서버와 완전히 동일한 구성을 갖는 서버
        - 심하게 말하면 복사본을 마련해서 추가하면 됨
- 요청을 균등하게 분산하는 것은 로드밸런서(Load Balancer)라는 장치가 해줌

- I/O 부하에는 문제가 있음
    - DB를 함께 놓는다고 하면, AP에서 DB로의 쓰기가 발생했을 때, 이전에 있던 DB와 새로운 DB의 데이터를 어떻게 동기화할 것인가?
    - 이전 DB 데이터를 새로운 DB에 복사하면 좋겠지만 이전 DB에 쓰인 내용을 어떻게 새로운 DB에 쓸 것인가라는 상황에 놓임
    - 쓰기는 간단히 분산할 수가 없음

- DB 확장성을 확보하는 것은 상당히 어려움
    - 디스크가 느리다는 문제도 여기에 영향을 미침
    - 디스크 I/O를 많이 발생시키는 구성으로 되어 있으면 속도차 문제가 생김
    - 데이터가 커지면 커질수록 메모리에서 처리 못하고 디스크 상에서 처리할 수 밖에 없는 요건이 늘어남
    - 즉, 대규모 환경에서는 I/O 부하를 부담하고 있는 서버는 애초에 분산시키기 어려움
    - 디스크 I/O가 많이 발생하면 서버가 금세 느려지는 본질적인 문제가 존재

- CPU 부하의 규모조정은 간단하다
    - 같은 구성의 서버를 늘리고 로드밸런서로 분산
    - 웹, AP 서버, 크롤러
- I/O 부하의 규모저정은 어렵다
    - DB
    - 대규모 데이터

대규모 데이터를 다루기 위한 기초지식

- 프로그래머를 위한 대규모 데이터 기초
  - 대규모 데이터는 메모리에서 처리하기 어렵고 디스크는 느리다
    - 또한, 분산하기도 곤란하다는 어려움이 있다는 것
  1. 프로그램을 작성할 때의 요령
  2. 프로그램 개발의 근간이 되는 기초라는 점에서 전제로 알아두었으면 하는 것

대규모 데이터를 다루는 세 가지 급소

- 대규모 시스템을 고민하게 만드는 대규모 데이터
  - 어떻게 하면 메모리에서 처리를 마칠 수 있을까?
  - 메모리에서 처리를 마쳐야 하는 이유
    - 디스크 탐색 횟수가 확장성, 성능에 크게 영향을 주기 때문
  1. 디스크 탐색 횟수 최소화, 국소성을 활용한 분산 실현
  2. 데이터량 증가에 강한 알고리즘, 데이터 구조
     a. 선형 검색 ⇒ 이분 검색 (O(n) → O(log n))
     b. log n으로 수십 번만에 마칠 수 있음(레코드 10만 건이 있을 때)
  3. 데이터 압축, 정보검색기술
     a. 압축해서 데이터량을 줄일 수 있다면, 읽어내는 탐색 횟수도 적어지게 됨
        i. 디스크 읽는 횟수를 최소화할 수 있다는 것
        ii. 메모리에 캐싱하기 쉬워짐
        iii. 데이터가 크면 메모리에서 넘치거나 디스크에 저장해도 읽어내기에 시간이 걸림
     b. 확장성 면에서 DB에만 맡겨서 해결할 수 없을 때
        i. 특정 용도에 특화된 검색엔진 등을 만들어서 해당 검색 시스템을 웹 애플리케이션에서 이용하는 형태로 전환한다면 속도를 제대로 확보할 수 있기 때문

대규모 데이터를 다루기 전 3대 전제지식

1. OS 캐시

2. 분산을 고려해서 RDBMs를 운용할 때는 어떻게 해야만 하는 가
3. 대규모 환경에서 알고리즘과 데이터 구조를 사용한다는 것

- 대수가 많다고 해서 빠른 것은 아니라는 것
  - DB 서버가 I/O 부하의 규모조정이 어렵기 때문에 DB 서버쪽이 반드시 대수가 많아지는 것은 아님

- AP 서버는 늘리면 늘릴수록 점점 빨라짐 ⇒ 부족해지면 늘리면 그만
  - 반면 DB 서버는 늘리더라도 의미가 없는 경우가 자주 있음

OS 캐시와 분산

- 대규모 데이터를 다룰 때의 포인트
  - I/O 대책에 대한 기반은 OS에 있음
  - OS 캐시로 제대로 처리할 수 없게 되었을 때 분산에 대해 고려해보는 것

OS의 캐시 구조

- OS의 캐시 구조를 알고 애플리케이션 작성하기 ⇒ 페이지 캐시
  - 디스크와 메모리 간 속도차가 10^5~10^6배 이상 난다고 했는데, 원래 OS에는 디스크 내의 데이터에 빠르게 액세스할 수 있도록 하는 구조가 갖춰져 있음
  - OS는 메모리를 이용해서 디스크 액세스를 줄인다
    - 원리를 알고 이를 전제로 애플리케이션을 작성한다면?
      - OS에 상당부분을 맡길 수 있음
  - 그것이 바로 OS 캐시
- 그 원리가 바로 OS 캐시
  - Linux의 경우는 페이지 캐시나 파일 캐시, 버퍼 캐시라고 하는 캐시 구조를 가짐
    - 파일 캐시라고 하는 것은 적절치 않다??
    - Linux 페이지 구조
- 가상 메모리 = 스왑 이라는 설명은 잘못되었다
  - OS는 가상 메모리 구조를 갖추고 있음

- 가상 메모리 구조?
  - 논리적인 선형 어드레스를 물리적인 물리 어드레스로 변환하는 것
  - 선형 어드레스(0xbffff4444) ===⇒ 페이징 구조 ===⇒ 물리 어드레스 (0x00002123)
- 스왑?
  - 가상 메모리를 응용한 기능 중 하나로 물리 메모리가 부족할 때 2차 기억장치 (주로 디스크)를 메모리로 간주해서 외형상의 메모리 부족을 해소하는 원리

가상 메모리 구조

- 가상 메모리 구조가 존재하는 가장 큰 이유는?
  - 물리적인 하드웨어를 OS에서 추상화하기 위해
  - 메모리, OS, 애플리케이션 프로세스
  - 메모리에는 주소가 붙어있다
    - 0x00002123이라는 32비트 주소가 붙어 있음
    - 그러나 1의 어드레스를 직접 프로그램에서 사용하게 되면 여러 곤란한 일이 일어난다
- 프로세스에서 메모리를 필요로 하게 되면, 느닷없이 메모리 내 주소를 가져오는 것이 아니라
  - OS가 메모리에서 비어 있는 곳을 찾는다
  - 메모리는 OS가 관리하고 있으며, 비어 있는 곳을 반환할 때 0x00002123과는 다른 어드레스를 반환한다
  - 왜?
    - 개별 프로세스에서는 메모리의 어느 부분을 사용하는지 관여하지 않고, `반드시 특정 번지부터 시작` 하는 것으로 정해져 있으면 다루기 쉽기 때문

    ex) 유닉스의 공유 라이브러리는 프로세스 내의 지정된 주소로 할당되도록 되어 있음

      - 프로세스 내에서 이 특정 어드레스는 예약되어 있다
        - 이 때 만일 시작주소가 각기 다르다면?
          - 메모리를 확보해야 할 주소위치를 찾기가 매우 어려울 것
- OS 관련 참고서를 더 자세히 읽어보자

- Point!
  - OS라는 것은 메모리를 직접 프로세스로 넘기는 것이 아니라 일단 커널 내에서 메모리를 추상화하고 있다는 점 ⇒ 가상 메모리 구조
  - 어드레스 변환에 있는 다양한 장점이란?

- 디스크의 경우에도 OS가 모아서 읽어낸다는 것 ⇒ 메모리를 확보할 때 그와 마찬가지 방식으로
  - 메모리 1바이트씩 액세스 하는 것이 아니라 적당히 4KB 정도를 블록으로 확보해서 프로세스에 넘긴다
  - 여기서 1개의 블록을 페이지라고 함
  - OS는 프로세스에서 메모리를 요청받으면 페이지를 1개 이상, 필요한 만큼 페이지를 확보해서 프로세스에 넘기는 작업을 수행한다

Linux의 페이지 캐시 원리

- OS : 확보한 페이지를 메모리상에 계속 확보해두는 기능을 가짐
  - 프로세스는 디스크로부터 데이터를 어떻게 읽어내는가?
    1. 우선 디스크로부터 4KB 크기의 블록을 읽어낸다
       a. 읽어낸 것은 한 번은 메모리 상에 위치시켜야 함
          i. 왜?
             1. 프로세스는 디스크에 직접 액세스할 수 없기 때문
       b. 어디까지나 프로세스가 액세스할 수 있는 것은 (가상) 메모리
          i. 따라서 OS는 읽어낸 블록을 메모리에 쓴다
    2. 그 다음, OS는 그 메모리 주소를 프로세스에 (가상 어드레스로서) 알려준다
       a. 그러면 프로세스가 해당 메모리에 액세스하게 된다
- 데이터 읽기를 마친 프로세스가 `이번 디스크 읽기는 끝나고 데이터는 전부 처리했으므로 더 이상 불필요` 하게 됐어도
  - 메모리를 해제하지 않고 남겨둔다
  - 왜?
    - 다음에 다른 프로세스가 같은 디스크에 액세스할 때에는 남겨두었던 페이지를 사용할 수 있음

        - 남겨두었던 페이지를 사용할 수 있음 ⇒ 디스크를 읽으러 갈 필요가 없게 된다
        - `페이지 캐시`
- 커널이 한 번 할당한 메모리를 해제하지 않고 계속 남겨두는 것

페이지 캐시의 친숙한 효과

- 예외인 경우를 제외하고 모든 I/O에 투과적으로 작용
- Linux에서는 디스크에 데이터를 읽으러 가면 꼭 한 번은 메모리로 가서 데이터가 반드시 캐싱됨
    - 따라서 두 번째 이후의 액세스가 빨라짐
- 현대의 OS는 페이지 캐시와 비슷한 구조를 가짐
    - OS를 가동시켜 두면 메모리가 허락하는 한 디스크상의 데이터를 계속 캐싱하게 됨
        - OS를 계속 가동시켜 두면 빨라짐
- Windows 머신 같은 경우에 우리가 재부팅하는 경우 존재
    - 재부팅하지 않는 편이 디스크를 읽어낼 때 캐시가 작용하기 쉬워 속도는 빨라짐
    - 부팅 직후에는 캐시가 없음 ⇒ 디스크 I/O가 발생하기 쉬워 다소 버벅거리는 것처럼 느낄 수 있음

VFS

- 디스크의 캐시는 페이지 캐시에 의해 제공되지만, 실제 이 디스크를 조작하는 디바이스 드라이버와 OS 사이에는 `파일 시스템` 이 있음
- Linux : ext3, ext2, ext4, xfs 등 몇몇 파일시스템이 있음
    - 그 하위에 디바이스 드라이버가 있고 실제로 이 드라이버가 하드디스크를 조작
- 파일시스템 위에는 VFS(Virtual File System - 가상 파일시스템)이라는 추상화 레이어가 존재
    - 파일시스템은 다양한 함수를 갖추고 있음
        - 그 인터페이스를 통일하는 것이 VFS의 역할
- VFS는 또한 페이지 캐시의 구조를 가지고 있음
    - 어떠한 파일시스템을 이용, 어떤 디스크를 읽어내더라도 반드시 동일한 구조로 캐싱

    - 매우 바람직한 구조 ⇒ 사람들이 여러 종류의 PC, 다양한 하드웨어, 여러 파일시스템을 사용하고 있음
        - 모두 같은 구조로 동일하게 캐싱되므로 전부 마찬가지로 생각해도 된다는 의미
- VFS의 역할 : 파일시스템 구현의 추상화와 성능에 관련된 페이지 캐시 부분

Linux는 페이지 단위로 디스크를 캐싱

- ex) 디스크 상에 4GB 정도의 매우 큰 파일이 있고, 메모리 2GB 밖에 없다고 하면?
    - 2GB 중에 500MB 정도를 OS가 프로세스에 할당했다고 하면?
        - 그러면 1.5GB 정도 여유가 있게 되는데 4GB 파일을 캐싱할 수 있는가?
- OS는 읽어낸 블록 단위만으로 캐싱할 수 있는 범위가 정해짐
    - 디스크 상에 배치되어 있는 4KB 블록만을 캐싱 ⇒ 특정 파일의 일부분만, 읽어낸 부분만을 캐싱할 수 있음
        - 이렇게 디스크를 캐싱하는 단위가 페이지
- `페이지 = 가상 메모리의 최소단위`

LRU

- 메모리 여유분이 1.5GB 있고 파일을 4GB 전부 읽게 되면?
    - 구조상으로는 LRU - 가장 오래된 것을 파기하고 가장 새로운 것을 남겨놓는 형태로 되어 있음
        - 최근에 읽은 부분이 캐시에 남고 과거에 읽은 부분이 파기되어 간다
    - DB도 계속 구동시키면 캐시가 점점 최적화되어 가므로 기동시킨 직후보다 점점 뒤로 갈수록 부하, I/O가 내려가는 특성을 보임

(보충) 어떻게 캐싱될까?

- i노드와 오프셋
- 어떻게 일부분만 캐싱될 수 있는가?
    - Linux는 파일을 i노드 번호라고 하는 번호로 식별

        - 해당 파일의 i노드 번호와 해당 파일의 어느 위치부터 시작할지를 나타내는 오프셋
            - 이 두 가지 값을 키로 캐싱
                - 이 두 가지를 키로하면 `어떤 파일의 어느 위치를` 이라는 쌍으로 캐시의 키를 관리할 수 있음
                    - 결과적으로 파일 전체가 아닌 파일의 일부를 캐싱해갈 수 있음
- 파일이 아무리 크더라도 이 키로부터 해당 페이지를 찾을 때의 데이터 구조는 최적화되어 있음
    - Radix Tree : OS(=커널) 내부에서 사용되고 있는 데이터 구조
    - 파일이 아무리 커지더라도 캐시 탐색속도가 떨어지지 않도록 개발된 데이터 구조
        - 따라서 커다란 파일의 일부분을 캐싱하거나 작은 파일의 일부분을 캐싱하더라도 동일한 속도로 캐시를 찾을 수 있도록 함

메모리가 비어 있으면 캐싱

- 이 부분은 직접 linux에서 실행해보자

I/O 부하를 줄이는 방법

- 캐시에 의한 I/O 경감효과는 매우 큼
    - 캐시를 전제로 I/O를 줄이기 위한 대책을 세워가는 것이 유효하다는 것

1. 1) 데이터 규모에 비해 물리 메모리가 크면 전부 캐싱할 수 있음
    a. 다루고자 하는 데이터 크기에 주목
2. 대규모 데이터 처리에는 데이터 압축이 중요
    a. 압축해서 저장해두면 디스크 내용을 전부 그대로 캐싱해둘 수 있는 경우가 많음
    b. ex) LZ법 등 일반적인 압축 알고리즘의 경우
        i. 압축률은 보통이더라도 텍스트 파일을 대략 절반 정도로 압축할 수 있다
            1. 4GB의 텍스트 파일이라면 메모리 2GB인 머신으로 뒷부분 절반 정도는 거의 캐싱할 수 없었던 것이
                a. 압축해서 저장해두면 2GB로 캐싱할 수 있는 비율이 상당히 늘어나게 됨

3. 2) 경제적인 비용과의 밸런스를 고려하고자 한다는 점

복수 서버로 확장시키기 (캐시로 해결될 수 없는 경우라면?)

- 메모리를 늘려서 전부 캐싱할 수 있다면 좋겠지만, 당연히 데이터를 전부 캐싱할 수 없는 규모가 될 수 있음
  - 복수 서버로 확장시키는 방안을 생각해보자
- AP를 늘려야 하는 이유
  - 기본적으로 CPU 부하를 낮추고 분산시키기 위해서
- DB 서버를 늘려야 할 때는 반드시 부하 때문만은 아니고 오히려 캐시 용량을 늘리고자 할 때 or 효율을 높이고자 할 때인 경우가 많음
- AP 서버를 늘리는 것과 DB 서버를 늘리는 것은 둘 다 서버를 늘리는 것이지만 필요한 리소스, 요구되는 리소스가 전혀 다름
  - DB 서버는 늘리면 좋다는 논리가 들어맞지 않음
- `I/O 분산에는 국소성을 고려한다`?

단순히 대수만 늘려서는 확장성을 확보할 수 없다

- 캐시 용량을 늘려야 한다고 했지만, 사실 단순히 대수를 늘리는 것만으론 X
  - 단순히 데이터를 복사해서 대수를 늘리게 되면 애초에 캐시 용량이 부족해서 늘렸는데 그 부족한 부분도 그대로 동일하게 늘려가게 되는 것
  - 변함없이 일부분이 캐싱되지 않는 상황이 됨
    - 곧 다시 병목이 되는 상황
- 서버를 늘림으로써 시스템 전체로서는 아주 조금은 빨리질지도 모르지만, 증설비용에 대비해서 성능향상은 극히 부족한 것
  - 확장성을 확보하려고 할 때— 서버를 늘려서 10배 ~ 100배 정도는 빨라져야 함

국소성을 살리는 분산

- 캐시 용량을 늘리기 위해 어떻게 하면 여러 대의 서버로 확장시킬 수 있는지?
  - 국소성 = locality
- 데이터에 대한 액세스 패턴을 고려해서 분산시키는 것 = 국소성을 고려한 분산

- 데이터로 액세스하는 경향에 대한 처리방식에 따라 특정한 방향으로 치우치는 경우가 많음
- ex) 서버 1,2 양측에 별다른 액세스 패턴을 고려하지 않고 분배한 경우
  - 2로의 액세스는 여전히 계속됨 ⇒ 서버 1이 2에 해당하는 데이터 영역도 캐싱해야 할 필요가 있음
  - 액세스 패턴을 고려해서 분배한 경우, 2에 해당하는 데이터 영역은 더 이상 액세스 되지 않아
    - 그만큼 캐시영역을 다른 곳으로 돌릴 수 있음
  - 서버 2에서도 동일한 입장으로 생각 가능함
- 결국 시스템 전체로서는 메모리에 올라간 데이터량이 늘어나게 됨
- 캐싱할 수 없는 부분이 없어짐
  - 메모리는 디스크보다 10^6정도 빠름 ⇒ 그만큼 덕을 보게 됨

1 - 파티셔닝

- 국소성을 고려한 분산을 실현하기 위해선 파티셔닝이라는 방법을 주로 사용
  - 파티셔닝 - 한 대였던 DB 서버를 여러 대의 서버로 분할하는 것
    - 분할 방법은 여러가지가 존재 ⇒ 테이블 단위 분할이 제일 간단
- 같이 액세스하는 경우가 많으므로 같은 서버에 위치시킴
  - 테이블 단위로 분할했으면 각각 서버로 보내 처리될 수 있도록 애플리케이션을 변경할 필요가 있음
- 다른 분할 방법 : 테이블 데이터 분할
  - 특정 테이블 하나를 여러 개의 작은 테이블로 분할한다
  - ex) ID의 첫 문자로 파티셔닝을 하고 있음
    - 예를 들어, ID의 첫 문자가 a~c인 사람의 데이터는 서버1, d~f인 사람의 데이터는 서버2와 같이 나눈다
      - 특정 데이터에 대한 요청이 올 때는 그것에만 액세스하면 되므로, 메모리 측면에서는 국소성이 작용하고,
      - 그 서버에 있는 사용자는 전부 액세스가 가능해 특정 서버 안에 있는 데이터만 캐싱하면 됨
- 이러한 분할의 문제

    - 분할의 입도를 크거나 작게 조절할 때 데이터를 한 번 병합해야 함

2 - 요청 패턴을 '섬'으로 분할

- 용도별로 시스템을 섬으로 나누는 방법
    - 이전에는 DB 테이블이나 DB 내 데이터를 기준으로 액세스를 분배했다면
    - HTTP 요청의 User-Agent나 URL 등을 보고, 통상의 사용자이면, 1번 섬, 일부 API 요청이면 3번 섬, bot이라면 2번 섬으로 나누는 방법을 사용
- 검색 봇은 그 특성상 아주 오래된 웹 페이지에도 액세스하러 온다
    - 사람의 경우라면 좀처럼 액세스하지 않을 페이지에도 액세스하고, 또한 광범위하게 액세스한다
        - 이렇게 되면 캐시가 작용하기 어려움
            - 동일한 페이지에 잇따라 방문하는 경우에는 캐시로 성능을 끌어올리기 쉽지만
                - 이처럼 광범위한 액세스에는 그럴 수가 없음
- 캐싱하기 쉬운 요청, 캐싱하기 어려운 요청을 처리하는 섬이라는 부분으로 나누어 두게 되면
    - 전자는 국소성으로 인해 안정되고 높은 캐시 적중률을 낼 수 있음
    - 후자의 요청이 전자의 캐시를 어지럽히므로 섬으로 나누는 경우에 비해 전체적으로는 캐시 효율이 떨어짐
- 링크를 따라가는 봇은 좀처럼 순회를 멈추지 못하는 문제가 있음 ⇒ 그래서 섬이라는 부분으로 구분을 해두어야 함
- 정해진 테이블만 액세스한다면, 그 부분만 캐싱이 잘 적용되도록 별도로 가려서 섬으로 나누는 것도 유효함

페이지 캐시를 고려한 운용의 기본 규칙

Point 1. OS 기동 직후에 서버를 투입하지 않는다는 것

- 갑자기 배치하면 캐시가 없으므로 오직 디스크 액세스만 발생하게 됨
    - 실제로 규모가 큰 곳에서는 서버가 내려감 ⇒ 시스템이 다운됨
    - 안이한 운용은 금물

    - 그렇다면?
        - OS를 시작해서 기동하면 자주 사용하는 DB의 파일을 한 번 cat해줌
            - 그러면 전부 메모리에 올라가게 됨
            - 그렇게 한 후 로드밸런서에 편입시킨다

Point 2. 성능 평가 or 부하시험

- 초기값을 버려야 함 ⇒ 최초의 캐시가 최적화되어 있지 않은 단계와 올라가 있을 때 낼 수 있는 속도는 완전히 다름
    - 성능평가 및 부하시험도 캐시가 최적화된 후에 실시할 필요가 있음

참고! 부하분산과 OS의 동작원리

- OS 캐시, 멀티스레드나 멀티프로세스, 가상 메모리 구조, 파일시스템
    - 다양한 장치가 하드웨어를 효율적으로 사용하기 위해 어떤 원리를 갖추고 있는지를 비롯해 장점과 단점을 확실히 알 수 있음
- OS의 장단점 + 시스템 전체를 최적화할 수 있음