10.Batch Processing

요약정리

- 시스템의 유형 3가지 구분
  - **서비스 (온라인 시스템)**
    - 클라이언트로부터 요청/지시가 올 때까지 대기
    - 성능 측정 지표 = 응답 시간 (+ 가용성)
  - **일괄 처리 시스템 (오프라인 시스템)**
    - 큰 입력 데이터를 받아 처리하는 작업을 수행하고 결과 데이터 생산 (Scheduling)
    - 성능 측정 지표 = 처리량
  - **스트림 처리 시스템 (준실시간 시스템)**
    - 준실시간 처리(near-real-time processing, nearline processing)
    - 입력 이벤트 발생 직후, 입력 데이터 소비 및 출력 데이터 생산 => 지연시간 ↓
- 맵 리듀스 (MapReduce)
  - “구글을 대규모로 확장 가능하게 만든 알고리즘”
  - ex. 하둡, 카우치DB, 몽고DB
  - 병렬 처리 시스템보다 저수준이지만 데이터 규모면에서 상당히 진보
- 표준 유닉스 도구
  - 유닉스의 철학이 대규모 이기종 분산 시스템으로 그대로 이어짐

10-1 유닉스 도구로 일괄 처리하기

단순 로그 분석

- `awk`, `sed`, `grep`, `sort`, `uniq`, `xargs` 조합으로 데이터 분석이 놀라울 정도로 잘 수행됨
- 예제

```
# /var/log/nginx/access.log
```

```
# format : $remote_addr - $remote_user [$time_local] "$request" $status $body_byte
s_sent "$http_referer" "$http_user_agent"
# (sample) 216.xx.xx.xx - - [27/Feb/2015:17:55:11 +0000] "GET /css/typrography.css
HTTP/1.1" 200 3377 "http://...com" "Mozilla/5.0 (.....)

cat /var/log/nginx/access.log | \
  awk '{print $7}' | \ # 7번째 필드 출력 ($0은 전체)
  sort  | \ # 정렬
  uniq -c | \ # 중복 제거 ('-c': 중복횟수 첫번째로 함께 출력)
  sort -r -n | \ # 정렬 ('-r': 내림차순, '-n' : 숫자로 정렬)
  head -n 5 # 맨 앞 5줄만 출력
```

- 유닉스 연쇄 명령 vs 맞춤형 프로그램
  - 취향의 문제 (간결성, 가독성 등)
  - 단, 실행 흐름은 크게 다름
- 정렬 vs 인메모리 집계
  - 유닉스 파이프라인 : 해시 테이블 x 정렬된 목록에서 같은 url 반복 노출
  - 맞춤형 프로그램 (ex. 스크립트) : 해시 테이블을 메모리에 유지
  - 뭐가 더 적합한가? => 작업 세트 (임의 접근이 필요한 메모리양) 에 따라
    - 작업세트가 작으면 인메모리 해시 테이블도 ok
    - 작업세트가 허용 메모리보다 크면 정렬 접근법 권장 (디스크를 효율적으로 사용하는 병합 정렬)
  - 리눅스에 포함된 sort 유틸리티는 메모리보다 큰 데이터셋을 자동으로 디스크로 보내고 여러 CPU 코어에서 병렬로 정렬 => 손 쉽게 큰 데이터 셋으로 확장 가능

유닉스 철학

- 유닉스 파이프
  - *“다른방법으로 데이터 처리가 필요할 때 정원 호스와 같이 여러 다른 프로그램을 연결하는 방법이 필요하다. 이것은 I/O 방식이기도 하다. (Dog McIlory, 1964)”*
- 유닉스 철학 (1978)
  - (1) 각 프로그램이 한가지 일만 하도록 작성
  - (2) 모든 프로그램의 출력은 아직 알려지지 않은 다른 프로그램의 입력으로 쓸 수 있다고 생각할 것 (출력에는 필요한 정보만. 입력 형식을 엄격하게 맞추거나 이진 형태 x, 대화형 입력 고집 x)
  - (3) S/W와 OS는 빠르게 써볼 수 있게 설계/구축

    - (4) 프로그래밍 작업을 줄이려면 미숙한 도움보다는 도구를 사용
- => Agile, DevOps 와 매우 흡사
- 유닉스 도구들은 유연하게 조합할 수 있고 조합하여 사용했을 때 강력
    - 유닉스의 결합성 <= **동일 인터페이스**

[유닉스 철학] 동일 인터페이스

- 특정 출력을 다른 **어떤** 프로그램의 입력으로 쓰려면 (2) => **모두** 호환 가능한 인터페이스를 사용해야 함
    - 동일 인터페이스 예 : 파일, URL, HTTP
- 유닉스의 인터페이스 => 파일 (파일 디스크립터)
    - 파일은 단지 순서대로 정렬된 바이트 연속. 여러가지 것 함께 표현 가능
    - ex. 실제 파일, 프로세스 간 통신채널(유닉스 소켓, stdin, stdout), 장치 드라이버, TCP 연결 소켓 등..
- 아스키(Ascii) 텍스트
    - 관례상 많은 유닉스 프로그램들이 연속된 바이트를 아스키 텍스트 취급
    - 같은 레코드 분리자(\n) 사용하는 유닉스 유틸리티 => 상호 운용 가능
        - 0x0A, LF, Line feed(Ctrl + J)
        - 0x1E, RS, Record Separator(Ctrl + ^)
    - 큰 문제는 없지만 그다지 깔끔하지는 x
- 데이터베이스
    - 동일한 데이터 모델인 데이터베이스 간에도 데이터 이동이 쉽지않음 (통합 부족)
    - 데이터 발칸화(Balkanization) : 인터넷이 고립된 여러 섬처럼 나뉜 현상이나 프로그램 언어나 데이터 파일 포맷등이 분화 발전하는 현상

[유닉스 철학] 로직과 연결의 분리

- 표준 입력(stdin), 표준 출력(stdout) 사용
    - 유닉스 도구의 또 다른 특징
    - 한 프로세스의 stdout 을 다른 프로세스의 stdin 과 연결 => 중간 데이터 디스크에 쓰지않고 **작은 인메모리 버퍼** 사용하여 전달
- 입출력이 어떻게 이루어지는지 신경 쓸 필요 x

    - 느슨한 결합 (loose coupling)
    - 지연 바인딩 (late binding)
    - 제어 반전 (inverse of control)
- 직접 작성한 프로그램을 끼워넣어 os 지원 도구처럼 사용 가능
- 제약 사항
    - 여러 개의 입력을 받거나 여러 개의 출력이 필요한 경우 사용이 까다로움
        - 출력을 파이프를 이용해 네트워크와 연결 불가 (without `netcat`, `curl`)
    - 프로그램의 I/O가 프로그램 자체와 서로 묶이는 경우
        - 프로그램이 파일을 직접 열어 읽고 쓰거나, 서브 프로세스로 다른 프로그램 구동하거나, 네트워크 연결하거나
        - 입출력 연결하는 유연성 감소

[유닉스 철학] 투명성과 실험

- 유닉스 도구는 불친절하고 단순하지만, 진행 사항 파악이 쉬움
    - 일반적으로 입력파일은 불변 처리
    - 어느 시점이든 파이프라인 중단하고 출력 확인 가능 (`less`)
    - 특정 파이프라인 단계의 출력을 파일에 쓰고, 다음 단계에 입력으로 사용 가능 (재시작 good)
- 가장 큰 제약은 **단일 장비** 에만 실행된다는 점
    - => 하둡같은 도구가 필요한 이유

10-2 맵리듀스와 분산 파일 시스템

- 유닉스 도구와 비슷하지만 **수천 대의 장비로 분산 실행** 가능
    - 입력 수정 x => 부수 효과 x
    - 출력 파일은 순차적으로 한번씩만 작성 (수정 x)
- 입출력
    - 유닉스 도구 : stdin / stdout
    - 맵리듀스 : 분산 파일 시스템 상의 파일 (하둡 => HDFS)
- 분산 파일 시스템

    - **HDFS** (Haddop Distributed File System)
    - GlusterFS, OFS(Quantcast File)
    - AWS S3, Azure Blob Storage, Openstack Swift (객체저장소)
- HDFS는 비공유 원칙 기반
    - 비공유 아키텍처 (shared-nothing, scale out)
        - 일반적 데이터센터 네트워크에 연결된 컴퓨터면 충분
    - vs 공유 디스크 방식
        - NAS(network Attached Storage), SAN(Storage Area Network)
        - 중앙 집중 저장 장치를 위한 맞춤 하드웨어나 특별한 네트워크 인프라 필요
- HDFS는 개념적으로 큰 하나의 파일 시스템
    - 각 장비에서 실행되는 데몬 프로세스로 구성 => 다른 노드가 해당 장비의 파일에 접근 가능하게끔 네트워크 서비스 제공
    - 데몬이 실행중인 모든 장비의 디스크 사용 가능
    - 중앙서버 (네임노드, NameNode)가 파일 블록이 저장된 장비 위치 추적
- HDFS는 뛰어난 확장성
    - 대규모 확장 가능 (범용 하드웨어 + 오픈소스 소프트웨어로 훨씬 저렴한 비용)
- 파일 블록 복제
    - 여러 장비에 동일 데이터 복사 [5장]
    - **삭제 코딩(erasure coding)** 방식으로 적은 부담으로 손실된 데이터 복구 (like RAID)

맵리듀스 작업 실행하기

- 맵 리듀스 = 분산 파일 시스템 위에서 대용량 데이터셋을 처리하는 코드 작성 프로그래밍 프레임워크
- 유닉스 도구로의 단순 분석과 유사한 데이터 처리 패턴
    - (1) 입력파일을 레코드로 쪼갠다 (separator = `\n`)
    - (2) 각 레코드마다 매퍼 함수를 호출해 키와 값 추출
    - (3) 키 기준으로 키-값 쌍 모두 정렬

        - (4) 정렬된 키-값 쌍 전체 댇상으로리듀스 함수 호출 (같은 키값은 서로 인접 -> 쉽게 결합)
    - 맵리듀스 작업 4단계
        - 1단계) 입력 형식 파서 : 파일 → 레코드
        - 2단계) 맵 (Map) : 레코드 → 키, 값 추출
        - 3단계) 정렬 단계 : 매퍼 출력이 내부적으로 이미 정렬
        - 4단계) 리듀스 (Reduce) : 키, 값 → 출력 레코드
    - 2 가지 콜백 함수 구현 필요
        - **매퍼 (Mapper)** : 정렬에 적합한 형태로 데이터 준비
            - 모든 입력 레코드마다 독립적으로 한 번씩만 호출
        - **리듀서 (Reduccer)** : 정렬된 데이터 가공
            - 같은 키를 모으고 해당 값의 집합을 반복해 리듀서 함수 호출
    - 맵리듀스 분산 실행
        - 병렬 수행 코드를 직접 작성안하고도 동시 처리 가능 (신경x)
        - 매퍼, 리듀서 : 하둡 (Java Class), MongoDB, CouchDB (Javascript Function)
        - 파티셔닝 기반
            - 매퍼 - 입력 파일 블록수 기반
            - 리듀서 - 사용자 지정 (키 해시값 사용)
        - 데이터 가까이에서 연산하기 (입력 파일있는 장비에서 맵리듀스 작업 수행) => 지역성 ↑ 네트워크 부하 ↓
        - 셔플 (shuffle)
            - 리듀서를 기준으로 파티셔닝하고 정렬한 뒤 매퍼로부터 데이터 파티션을 복사하는 과정
            - 정렬된 순서를 유지하며 병합 (임의 x)
    - 맵리듀스 워크플로 (workflow)
        - 하나의 맵리듀스 출력을 다른 맵리듀스의 입력으로 연결 (파일 경로를 통한 암묵적 방식)
        - 일괄 처리 작업의 출력은 성공했을 때만 유효 (실패시 남은 출력 제거)

- 하둡 도구 예
  - 맵리듀스 작업간 수행 의존성을 위한 스케줄러 예 : Oozie, Azkaban, Luigi, **Airflow**, Pinball
  - 다중 맵리듀스 연결을 위한 하둡용 고수준 도구 예 : Pig, Hive, Cascading, Crunch, FlumeJava

리듀스 사이드 조인과 그룹화

- 사용자 활동 이벤트 분석 예제
  - 활동 이벤트 (activity event) or 클릭스트림 데이터 (clickstream data)
  - 원격 데이터베이스에 질의한다는 건 일괄 처리가 비결정적이라는 뜻
  - 데이터베이스의 사본 (ex. ETL) 를 추출해 분산 파일 시스템에 넣는 방법
- **리듀스 사이드 조인 (Reduce-Side Join)**
  - 실제 조인 로직을 리듀서에서 수행
  - 매퍼는 입력데이터 준비 역할 (입력 데이터에 가정 x)
- **정렬 병합 조인** (SMB, Sort-Merge Join)
  - 매퍼 출력이 키로 정렬된 후 (sort) 리듀서가 조인의 양측에 정렬된 레코드 목록 병합 (merge)
  - 특정 id의 모든 레코드를 한번에 처리. 한번에 한 id의 레코드만 메모리에 유지. 네트워크 x
  - 보조 정렬 (secondary sort) : 리듀서가 작업 레코드 재배열
- 같은 곳으로 연관된 데이터 가져오기
  - 같은 키 (주소) 를 가진 키-값 쌍은 모두 같은 리듀서 호출
  - 맵리듀스는 데이터 모으는 연산 (물리적 네트워크 통신) 과 처리하는 로직 (애플리케이션 로직) 분리
  - 실패가 발생해도 애플리케이션 코드에서는 고민 no (재시도)
- 그룹화
  - SQL `GROUP BY`
  - 맵리듀스로 그룹화 구현 => 키-값 생성 시 **그룹화할 대상을 키**로 설정
  - 그룹화 사용 예시) 세션화 (sessionization)
- 쏠림(skew) 다루기

        - 불균형한 활성 데이터베이스 레코드 = 린치핀 객체 (linchpin object) = 핫 키 (hot key)
        - 한 리듀서에 많은 레코드가 쏠리는 현상 = 핫스팟
        - 맵 리듀서는 모든 매퍼, 리듀서가 끝나야하므로 지연시간 ↑
        - 핫스팟 완화 알고리즘
            - Pig의 쏠린 조인(skewed join)
            - Crunch의 공유 조인(shared join)
            - Hive의 맵 사이드 조인 (map-side-join)
        - 핫 키로 그룹화/집계하는 2 단계
            - (1) 레코드를 임의의 리듀서로 보내 처리해서 핫 키 레코드의 일부를 그룹화 하고 간소화 값 출력
            - (2) 첫 단계의 출력으로 나온 값을 키별로 모두 결합해 하나의 값으로

맵 사이드 조인

- **맵 사이드 조인 (Map-Side Join)**
    - 입력 데이터에 대한 특정 가정
    - 축소된 맵리듀스 (리듀서 x 정렬 x)
    - 매퍼는 단지 입력 파일 블럭 하나를 읽어 다시 분산 파일 시스템에 출력
- **브로드캐스트 해시 조인** (Broadcast Hash Join)
    - 메모리에 올릴 정도로 작은 데이터 셋과 큰 데이터 셋을 조인
    - 큰 입력 파티션 하나를 처리하는 각 매퍼는 작은 입력 전체를 읽고 (broadcast), 작은 데이터셋은 각 파티션의 해시 테이블에 적재 (hash)
    - 인메모리 해시 테이블 적재 대신 로컬 디스크 읽기 전용 색인으로 저장도 가능
    - Pig의 복제 조인, Hive의 맵 조인, 캐스케이딩, 크런치, Impala (질의 엔진)
- **파티션 해시 조인** (Partitioned Hash Join)
    - 두 입력 모두를 같은 키, 같은 해시 함수, 같은 수로 파티셔닝하여 조인
    - 각 맵퍼 해시 테이블에 적재해야 할 데이터의 양 ↓
    - Hive의 버킷 맵 조인(bucketed map join)
- 맵 사이드 병합 조인 (map-side merge join)

        - 입력 데이터셋이 같은 파티셔닝, 같은 키 기준 **정렬** 된 경우 사용 가능 (sort-merge)
    - 맵 사이드 조인을 사용하는 맵리듀스 워크플로
        - 맵 사이드 조인 vs 리듀스 사이드 조인 => 출력구조 다름
            - 리듀스 사이드 조인 : 조인 키로 파티셔닝, 정렬
            - 맵 사이드 조인 : 큰 입력과 동일한 방법으로 파티셔닝, 정렬
        - 맵 사이드 조인은 크기, 정렬, 입력 데이터의 파티셔닝 같은 제약 사항 => 물리적 레이아웃 파악 필수
            - HCatalog, Hive metastore

일괄 처리 워크플로의 출력

> 🤔 애초에 그래서 일괄처리를 왜 쓰는데?

- 데이터 베이스 질의 구분 => OLTP를 분석 목적과 구별
    - OLTP 질의 : 색인 사용하여 사용자에게 보여줄 소량의 레코드만 특정키로 조회
    - OLAP 분석 질의 : 대량의 레코드를 스캔해 그룹화/집계 연산하여 보고서 형태로 출력
- **그렇다면 일괄 처리는?**
    - 트랜잭션 처리도 분석도 X
    - 분석에 가깝지만 SQL 질의도 아니고 출력은 보고서가 아닌 다른 형태 구조
- 검색 색인 구축
    - 일괄 처리로 색인 구축 효율적 (병렬화, 읽기 전용, 불변)
    - 색인 갱신 방법 : 전체 색인 워크플로 재수행하여 색인 대치, 증분 색인
- 일괄 처리의 출력으로 키-값을 저장
    - 일괄 처리 워크플로 출력 예 : 검색 색인, 머신러닝 시스템 (분류기), 추천 시스템 ..
    - **일괄 처리의 출력** => 일종의 가 됨

      데이터 베이스

        - 질의 방법 => 일괄 처리 작업 내부에 새로운 데이터베이스를 구축해 분산 파일 시스템의 작업 출력 디렉터리에 데이터베이스 파일 저장 (직접 하나씩 데이터베이스에 요청보내는 것은 Bad)
        - 데이터 파일은 읽기전용, 불변, 서버에 bulk
    - 일괄 처리 출력에 관한 철학
        - 인적 내결함성 (human fault tolerance) : 버그 코드로 부터 복원 가능 여부
        - 비가역성 최소화 (minimizing irreversibility)
        - 입력 불변, 실패 출력 폐기 => 실패 시 재실행 반복 가능
        - 동일 입력 파일 집합 사용
        - 연결작업과 로직의 분리
    - 유닉스와의 차이점
        - 구조화된 파일형식 (avro, parquet) 사용으로 저수준 구문 변환 작업 최소화 가능 + 스키마 발전 가능

하둡과 분산 데이터베이스의 비교

- **대규모 병렬 처리 (MPP, Massively Parallel Processing)** 데이터베이스
    - MPP 데이터베이스 : 하나의 쿼리를 여러개의 프로세스로 병렬처리하는 데이터베이스
        - 'MPP 데이터베이스란' 참고
    - 맵리듀스의 개념은 이미 수십년전에 MPP DB에서 구현된 것
    - MPP vs 맵리듀스
        - 💣MPP 데이터베이스 : 장비 클러스에서 분석 SQL 질의를 병렬 수행하는 것에 초점
        - 🗂️맵리듀스 + 분산 파일 시스템 : 아무 프로그램이나 실행할 수 있는 OS 같은 속성 제공
- 저장소의 다양성
    - 💣데이터베이스는 특정 모델 (관계형 or 문서형) 에 따라 데이터 구조화 필요
    - 🗂️하둡은 어떤 형태라도 상관없이 HDFS 덤프가능
    - 현실적으로 하둡처럼 데이터를 **빨리** 사용가능하게, **한 곳에 모으는** 작업만으로도 가치 (like 데이터 웨어하우스)

        - data lake, enterprise data hub
        - 데이터 해석은 소비자에게 (schema-on-read)
        - “원시 데이터가 오히려 좋아” => 초밥 원리 (sushi principle)
    - ETL 구현에 사용하기도 (데이터 모델링을 하더라도 수집과는 분리된 단계)
- 처리 모델의 다양성
    - 💣MPP 데이터베이스는 monolithic 구조. 설계된 질의 유형에 좋은 성능 (but 한정)
    - 🗂️맵리듀스 이용 시 자신이 작성한 코드를 대용량 데이터 셋에서 쉽게 실행 가능
    - HDFS + 맵리듀스 + SQL 질의 엔진 (Hive) => 어려운 다양한 일괄 처리 가능
    - 하둡의 개방성
        - SQL, 맵리듀스보다 더 다양한 모델 등장
        - 데이터 이동 필요 없이 유연한 지원
        - 임의 접근 가능한 OLTP 데이터베이스 (Hbase), MPP 스타일의 분석 데이터베이스 (Impala) => HDFS
- 빈번하게 발생하는 결함을 줄이는 설계
    - MPP 데이터베이스 vs 맵리듀스 2가지 차이
        - 결함을 다루는 방식
        - 메모리 및 디스크 사용 방식
    - 💣MPP 데이터베이스는 한 장비만 죽어도 전체 질의 중단. 가능하면 메모리에 많은 데이터 유지
    - 🗂️맵 리듀스는 개별 태스크 실패에 큰 영향 x. 되도록 디스크에 데이터 기록 (내결함성, 데이터량)
    - 맵 리듀스는 대용량 작업과 예상치 못한 태스크 종료가 빈번한 경우 적합

10-3 맵리듀스를 넘어

- 맵리듀스는 분산 시스템에서 가능한 여러 프로그래밍 모델 중 단지 하나
    - 데이터 양, 자료 구조, 데이터 처리 방식에 따라 다른 도구가 더 적합할 수도

- 맵리듀스를 편하게 쓰기위해 추상화된 다양한 고수준 프로그래밍 모델 (단, 모델 자체 문제 주의)

중간 상태 구체화

- 맵리듀스는 다른작업과 모두 독립적 (로직과 연결의 분리)
- **중간 상태 (Intermediate state)** 를 파일로 기록하는 과정 => **구체화 (materialization)**
  - 장점
    - 내구성 (내결함성 확보)
  - 단점
    - 모든 선행 작업 태스크가 종료될때까지 대기 (수행시간 slow)
    - 매퍼 중복
    - 임시 데이터 (중간 상태) 도 복제되는 과잉 조치
  - vs 스트리밍 (ex. 유닉스 파이프의 인메모리 버퍼를 사용한 입출력 전달)
- **데이터플로 엔진 (dataflow engine)**
  - 분산 일괄 처리 연산 엔진. 전체 워크플로를 독립된 하위작업이 아닌,

    작업 하나로서 다루는 엔진
  - **Spark**, Tez, Flink
  - vs 맵리듀스
    - 더 유연한 방법으로 조합 가능 => **연산자 (operator)**

      함수
    - 연산자의 출력과 다른 연산자의 입력을 연결하는 여러가지 선택지 (키로 재파티셔닝 및 정렬, 정렬 스킵, 브로드캐스트 ..)
    - 수행속도 훨씬 빠름
  - 장점
    - 값비싼 작업(ex. 정렬)은 실제 필요할때만 수행
    - 필요없는 맵 태스크는 없다
    - 모든 조인과 데이터 의존 관계를 명시적 선언 => 지역성 최적화
    - 연산자 간 중간 상태는 로컬 디스크나 메모리에 기록

        - 입력 준비되는 즉시 실행 가능 (선행 단계 전체 완료 대기 x)
        - 새로운 연산자 실행 시 이미 존재하는 JVM 사용
    - 데이터플로 엔진의 내결함성 (Fault tolerance)
        - 중간 상태를 사용하지않는 데이터플로 엔진의 내결함성 확보 접근법 => 재계산
        - Spark (RDD 추상화 - 데이터 조상 추적), Flink (연산자 상태 체크포인트)
        - 데이터 재연산의 포인트는 “해당 연산이 **결정적인지** 파악” 하는 것 (= 비결정적 원인 제거)
    - 데이터플로 엔진의 구체화
        - 데이터플로 엔진은 일부(agg)를 제외하고 파이프라인 방식 실행 가능
        - 작업 완료시 출력을 지속성 있는 어떤 곳 (=분산 파일 시스템) 에 다시 기록
        - 모든 중간 상태를 기록하는 수고 x

그래프와 반복 처리

- 그래프 처리의 필요성
    - 그래프 처리 != 비순환 방향 그래프 (directed acyclic graph, DAG)
        - DAG는 데이터플로 엔진의 작업 연산자. 데이터 흐름이 그래프로 구성
        - 그래프 처리는 데이터 자체가 그래프 형식
    - 이행적 폐쇄 (transitive closure) : 특정 조건에 도달할때까지 인접 정점 조인
        - 맵리듀스로 반복적 스타일로 구현시 비효율적
- 프리글 (Pregel) 처리 모델
    - = 벌크 동기식 병렬 (BSP, bulk synchronous parallel) : 일괄 처리 그래프 최적화 방법
    - 한 정점이 다른 정점으로 ‘메세지를 보낼’ 수 있다.
        - 맵리듀스가 매퍼가 특정 리듀서를 호출해 ‘메세지를 전달’ 과 비슷
        - 차이점은 반복에서 사용한 메모리 상태 기억
    - 정점 상태 제외하고, 정점 사이 메세지는 내결함성/지속성. 메세지 처리는 고정 횟수 내 처리
        - 액터 모델 (분산 액터 프레임워크) 랑 비슷

        - 차이점은 타이밍 보장 (각 반복에서 이전 반복에서 보내진 모든 메세지 전달)
    - 내결함성
        - 반복이 끝나는 시점에 모든 정점 상태를 주기적 저장 (체크포인트)
        - 프로그래밍 모델 단순화를 위해 프레임워크 차원에서 완벽히 결함 복구
    - 병렬 실행
        - 어떤 물리장비에서 정점이 실행되는지 알필요 x. “정점과 같이 생각하기”
        - 그래프 알고리즘은 장비간 통신 오버헤드 ↑ (최적화 파티셔닝 x)
        - 그래프가 단일 장비에 넣기 너무 크다면 프리글 같은 분산 접근법 필수

고수준 API와 언어

- 고수준 API와 언어
    - 직접 맵리듀스 작업 작성은 매우 어려움 => 고수준 API 인기
    - 이런 데이터플로 API는 일반적으로 관계형 스타일의 빌딩 블록을 사용해 연산 표현
    - 장점
        - 적은 코드 작성
        - 대화식 사용 지원 (여러 접근법 실험 가능)
        - 시스템의 생산성 높은 사용 및 장비의 효율적 사용
- 선언형 질의 언어로 전환
    - 선언적인 방법으로 조인 지정시 => (맵리듀스, 데이터플로 계승자들의 내장된) 질의 최적화기가 최적 방법 결정
    - 장점
        - **코드 임의 실행 및 임의 형식의 데이터 읽기 가능**
        - 칼럼 기반 저장 레이아웃으로 최적화 가능
        - Hive, Spark DataFrame, Imapala Vectorized 수행 => 캐시 히트율 ↑ or 함수 호출 회피
- 다양한 분야를 지원하기 위한 전문화
    - 표준화된 처리 패턴 => 재사용 가능한 공통 빌딩 블록 구현

        - 재사용 구현의 예 : Mahout, MADlib, 공간 알고리즘 (ex. K-nearest neighbor)
        - 일괄 처리 엔진은 점차 광범위한 영역에서 필요 알고리즘을 분산 수행하는데 사용
    - 일괄 처리 시스템 vs MPP 데이터베이스
        - 점차 비슷해지고 있다 (결국엔 둘다 데이터 저장하고 처리하는 시스템)
        - 일괄 처리 엔진은 내장 기능 + 고수준 선언적 연산자
        - MPP 는 프로그래밍이 가능한 유연성