NoSQL 스터디

5장 ~ 7장

YBIGTA 11기 ENGINEERING TEAM 김연진

5장 일관성

5.1 업데이트 일관성

- **쓰기 충돌 (write-write conflict)**
  - 두(여러) 사람이 동시에 같은 데이터 항목을 업데이트하는 문제

- **업데이트 손실 (lost update)**
  - 업데이트하였지만 그 값이 손실된 경우
  - 예: A와 B가 같은 항목에 업데이트를 시도해서 A의 값으로 업데이트한 직후 B의 값으로 업데이트하는 경우

- **비관적 방법**
  - 충돌이 발생하는 것을 **방지** (싹을 자름)
  - 쓰기 잠금 (write lock)
- **낙관적 방법**
  - 충돌이 발생하도록 **놔두고** 충돌이 **발생하면** 이를 탐지해 적절한 **조치**를 취함 (나중에 처리)
  - 조건적 업데이트
    - 클라이언트가 업데이트를 하기 전에 자신이 마지막으로 읽은 시점 이후에 값이 변경되었는지 확인

5.1 업데이트 일관성

- **단일서버**
  - 업데이트 직렬화 과정 명확 (하나 고르고 다음 고름)
- **서버가 한 대가 넘는 경우 (ex 피어-투-피어 복제)**
  - 문제: 두 노드가 업데이트를 다른 순서로 적용 가능 → 노드마다 다른 값 생길 수 있음
  - 분산 시스템에서 동시성: 모든 노드가 같은 순서로 연산 적용하는 것이 보장되는 순차적 일관성

- **쓰기 충돌**을 처리하기 위한 **낙관적 접근법**
  - 두 업데이트 모두 저장 후 충돌 발생했다고 표시하는 방법(도 있다)

5.2 읽기 일관성

- 비일관적 읽기 (읽기-쓰기 충돌)
  - 데이터베이스의 일관성이 깨진 경우
  - ex) 누군가 읽고 나서 쓰기 전에 다른 사람이 같은 항목을 읽는 경우
  - 비일관적인 읽기-쓰기 충돌 피하기 위해 RDBMS는 **트랜잭션** 개념 지원

- 논리적 일관성
  - 다른 데이터 항목이 함께 의미를 가지도록 보장하는 것

5.2 읽기 일관성

Q: NoSQL은 트랜잭션 지원 안하니까 일관성도 지원 안하나요? 똥DB네

책: (일반적으로) 아니다.

1. 트랜잭션이 없다는 것은 보통 일부 NoSQL에만 해당하는 것이다. 그래프 데이터베이스는 지원함 (누가 써)
2. 원시적 업데이트를 지원하지만 한 집합에 대해서만 지원 (한 집합 내에서는 논리적 일관성 제공)

5.2 읽기 일관성

- 하지만 모든 데이터가 같은 집합에 들어갈 수 있는 것은 아님
- ➔ 따라서 여러 집합에 영향을 미치는 업데이트가 있는 경우 클라이언트가 비일관적 읽기를 수행할 가능성은 미해결 상태로 남는다..... (비일관적 읽기 발생할 가능성이 있다는 뜻)

- **비일관성 윈도**
  - 비일관성이 존재하는 시간의 길이
  - 아마존 문서에 따르면 심플DB 서비스의 비일관성 윈도는 보통 1초 미만

- **복제 일관성**
  - 같은 데이터 항목을 다른 복제본에서 읽어도 같은 값임을 보장

- **결과적 일관성**
  - 특정 시점에는 일관성 불일치가 있을 수 있지만, 더이상 업데이트가 없다면 결국에는 모든 노드가 같은 값으로 업데이트됨

- **낡은(stale) 데이터**
  - 유효기간이 지난 데이터

5.2 읽기 일관성

- **스티키 세션**
  - 세션이 유지되는 동안은 한 노드만 사용
  - 가장 쉬운 방법

- 다른 방법은
  - 버전 스탬프 사용
  - 데이터 저장소와의 모든 상호작용에 세션이 사용한 가장 최근 버전 스탬프를 사용하도록 함

- 사용자가 상호작용하는 동안 트랜잭션을 열어놓는 것은 일반적으로 좋은 생각이 아님
- 사용자가 데이터 업데이트하려고 할 때 정말 충돌이 발생해 오프라인 잠금을 초래할 위험이 있기 때문

5.3 일관성 완화

- **일관성은 좋은 것이지만… 슬프게도 희생할 수밖에 없는 경우도…있다..**
  - 모든 것은 트레이드 오프다 - 성배 초

- ㄹ

5.3 일관성 완화

5.3.1 CAP 정리

- **CAP**
  - 일관성 Consistency
  - 가용성 Availability
  - 분단 허용성 Partition tolerance

- **일관성은 이미 설명했다.**

- **가용성**
  - 클러스터의 한 노드와 통신할 수 있으면 그 노드에서 읽기와 쓰기가 가능해야 한다

- **분단 허용성**
  - 클러스터 내 통신 두절로 클러스터가 여러 조각으로 분단돼 서로 통신할 수 없게 되더라도(분단뇌), 클러스터가 잘 동작해야 한다

5.4 지속성 완화

- 일관성의 핵심은 원자적, 독립적 작업 단위로 요청을 직렬화하는 것이다
- 그러나 지속성 완화에 대해서는 사람들이 대부분 비웃는다
- 데이터 저장소가 업데이트된 내용을 잃어버리면 무슨 의미가 있겠는가
- 그러나 높은 성능을 위해 지속성을 얼마간 희생하고 싶을 때가 있다.. ← 마지막 저장 이후 변경사항 모두 날아간다는 대가가 있음

5.5 정족수

- 강력한 일관성을 얻으려면 얼마나 많은 노드가 관여해야 하는가?

- 쓰기 정족수
  - 복제할 때 강력한 일관성 보장 위해 승인해야하는 충분한 노드의 수
  - 쓰기에 참여하는 노드 수(W)는 복제에 관여하는 수(N)의 절반을 넘어야 한다
  - $W > N/2$

- 복제 인수
  - 복제본의 수
  - 대다수 전문가들은 좋은 복원력을 갖는 데 복제 인수가 3이면 충분하다고 말한다

5.6 더 읽을거리

- 타텐바움과 반 스틴(Tanenbaum and Van Steen)의 글 - 분산 시스템의 기본을 잘 정리했음
- IEEE 컴퓨터 2012 2월호 - CAP 정리의 영향력 확대에 대한 특별 기획

5.7 요점

- **쓰기 충돌**은 두(여러) 클라이언트가 동시에 같은 데이터를 쓰려 할 때 발생.
- **읽기-쓰기 충돌**은 한 클라이언트가 쓰고 있는 도중에 다른 클라이언트가 비일관적 데이터 읽을 때 발생

- **비관적 방법**은 충돌 방지를 위해 데이터 레코드에 잠금 사용
- **낙관적 방법**은 충돌 탐지해서 해결

- 분산 시스템에 읽기-쓰기 충돌이 발생하는 것은 일부 노드는 업데이트O지만 X인 노드 있는 시점 있기 때문
- **결과적 일관성**: 쓰기가 다른 모드로 전파되는 어떤 시점에 시스템이 일관성 있는 상태가 된다는 뜻

- 클라이언트는 보통 자신이 쓴 것에 대한 일관성 필요
- 즉 클라이언트가 어떤 값을 **쓴 즉시 그 값을 읽을 수 있어야** 함
- 읽기와 쓰기가 다른 노드에서 수행되면 제공 어려울 수 있음

5.7 요점

- 좋은 **일관성** 얻으려면: 데이터 연산에 많은 노드가 관여해야 함.
- 하지만 이렇게 하면 **지연** 길어짐
- 따라서 보통은 **일관성과 지연 사이에서 타협**해야

- CAP 정리는 네트워크 분산 발생한 경우 데이터 **가용성과 일관성 사이에서 절충**해야 함을 알려줌

- 지속성 또한 지연과 절충 가능
- 특히 데이터 복제에 실패하는 경우에도 서비스가 중단되지 않기를 바랄 경우

- 강력한 일관성 유지하려고 모든 복제본에 접근할 필요 X
- 충분한 **정족수**만 채우면 된다

6장 버전 스탬프

6.1 업무 트랜잭션과 시스템 트랜잭션

- 사용자는 트랜잭션을 생각할 때 **업무 트랜잭션**을 생각함
  - 제품 목록을 보고 적당한 가격의 위스키 한 병을 고른 다음, 신용카드 정보 입력하고 주문 승인하는 것과 같음
  - 그러나 이 모든 과정에 DB에서 제공하는 시스템 트랜잭션에서 이루어지지는 않음
  - 신용카드 정보 입력 전에 밥 먹으러 나가면 그동안 DB에 잠금 설정 해야 하기 때문

- 문제: 계산과 결정이 변경되기 전 데이터에 대해 이루어졌을 경우
- 해결: 오프라인 동시성 (일반적으로 사용되는 기법)
  - 특히 낙관적 오프라인 잠금이 유용
  - : 트랜잭션 커밋 전에 업데이트하려는 데이터를 읽어서 처음 읽었을 때와 변경됐는 지 확인
  - 방법(중 하나): DB 레코드에 어떤 형태든 버전 스탬프 포함시킴

6.1 업무 트랜잭션과 시스템 트랜잭션

- 버전스탬프 만드는 방법
  1. 자원이 업데이트될 때마다 항상 증가하는 **카운터** 이용
     - 장: 어떤 버전이 나중 버전인지 알기 쉬움
     - 단: 중복X도록 생성하려면 단일 마스터 필요
  2. **GUID**(유일성이 보장되는 큰 난수) 생성
     - 장: 누구나 생성할 수 있고(마스터 아니라도) 중복 X
     - 단: 크기가 큼, 어느 것이 최신인지 모름
  3. **자원 내용**으로 **해시** 만듦
     - 장: 해시 키 크기 충분히 크면 유일 가능, 누구나 생성 가능
     - 단: 어떤게 최신인지 직접 비교 불가, 길이 길어짐
  4. 마지막 업데이트의 **타임스탬프** 사용
     - 장: 적당히 짧고, 최신인거 직접비교 가능, 단일마스터 필요 X, 여러 장비가 생성가능
     - 단: 한 노드에서 시간 잘못되면 온갖 종류의 데이터 오류 발생 가능, 간격 크면 중복 위험

- 두 가지 이상의 방법 조합해 버전 스탬프 생성 가능
  - ex) 카우치DB: 카운터 + 콘텐츠 해시 ➔ 대부분의 경우 최신 비교 가능

6.2 다중 노드에서의 버전 스탬프 (p.78)

- 단일노드 / 마스터-슬레이브 복제처럼 기준 데이터가 **단일 노드에서 관리**되는 경우 → 기본적인 버전 스탬프도 잘 동작
- 파어-투-피어 모델 → 단일 지점 없으므로 다른 방법 필요

- 카운터가 가장 단순
- 마스터가 여럿인 경우에는 더 복잡한 방법 필요
  - 모든 노드가 버전 스탬프의 히스토리 갖게 하기 ← 클라이언트가 버전 스탬프 히스토리 들고 있거나 서버 노드가 버전 스탬프 히스토리 유지하고 데이터 요청에 응답할 때 히스토리 포함시켜야 함 (버전 관리 시스템에서는 이런 방식 사용, NoSQL은 X)
  - 타임스탬프 문제 많음. 시간 어긋나면 문제 多, 쓰기 충돌 탐지 X ➔ 단일 마스터에서만 OK, 보통은 카운터가 낫다

- 피어-투-피어 NoSQL 시스템에서 가장 흔하게 사용되는 방법은?
- 벡터 스탬프
  - 각 노드의 카운터로 구성된 카운터의 집합
  - 자기 노드 업데이트 & 통신 시 다른 노드 동기화
  - 비일관성 발견에 요긴. 해결은 X (해결 방법은 작업 분야에 따라 다를 것이다 …)

7장 맵-리듀스

7.1 맵-리듀스 기본 개념

- **맵**: 입력이 단일 집합이고 출력은 여러 개의 키-값 쌍인 함수
- 입력: 주문
- 출력: 상품에 포함된 상품에 대한 키-쌍 값
- 레코드 하나에 대해 동작

- **리듀스**: 똑같은 **키**에 대한 맵의 출력 여러 개를 취해 그 값을 결합
- 맵 함수가 특정 주문에 대해 상품을 100개 리턴하면, 리듀스 함수는 이를 수량과 매출의 총합 하나로 결합

7.2 분할과 결합

- 리듀스 함수만 가지는 경우
- 여러 노드에서 각각 수행한 맵 작업의 결과를 모아서 하나의 리듀스로 보냄
- 동작은 하지만, 병렬성 늘리고 데이터 전송향 줄이려면 할 일이 더 있다
  1. 맵 작업 결과를 파티션으로 나누기
     - 여러 리듀스 작업을 병렬적으로 실행 가능
     - 각 노드에서 맵의 결과를 키별로 나눈다 (보통 여러 개 키가 파티션으로 묶임)
     - 모든 노드로부터 하나의 파티션에 대한 테이터를 → 해당 파티션에 대한 하나의 그룹으로 결합한 다음 → 리듀스로 보냄
  2. 맵과 리듀스 단계 사이에서 노드 간 이동하는 데이터 양 줄이기
     - 데이터 상당 부분 반복적, 같은 키에 대한 키-값 쌍 여럿 포함
     - 같은 키에 대한 데이터를 값 하나로 결합해 데이터 크기 줄임
     - 결합 가능한 리듀스 함수
       - 결합 함수는 사실 리듀스 함수다
       - 많은 경우 최종 리듀스 결과 만드는 데 같은 함수가 사용될 수 있다
       - 이를 위해 리듀스 함수를 특별한 형태로 만들어야 함
       - 리듀스 함수의 출력이 그 입력과 연결되어야 함
     - 결합 가능하지 않은 경우: 특정 차를 주문한 고유(중복X) 고객 수

7.3 맵-리듀스 계산 조합

- 맵 - 한 집합에 대해서만 적용 가능
- 리듀스 - 한 키에 대해서만 적용 가능
- 이런 제약 조건에서도 잘 동작하게 하려면 프로그램의 구조를 다르게 생각해야 함

- 예: 평균 계산하는 경우
  - 평균들을 단순 평균내서 전체 평균을 구할 수 없음
  - 전체 개수를 구해야 한다

7.3 맵-리듀스 계산 조합

7.3.1 2단계 맵-리듀스 예제

- 맵-리듀스 계산이 복잡해지면, 파이프와 필터를 이용해 작업을 여러 단계로 나누는 것이 좋다
- (유닉스의 파이프라인처럼) 각 단계의 출력이 다음 단계의 입력이 되도록 함

- 예: 2011년도와 그 전년도의 월별 제품 판매 실적 비교
  1. 한 제품에 대한 특정 년/월의 판매 실적 집계
  2. 첫 단계의 결과(input) ➔ 전년도 같은 월의 수치와 비교해 결과를 만듦

7.3 맵-리듀스 계산 조합

- 예: 2011년도와 그 전년도의 월별 제품 판매 실적 비교
  1. 한 제품에 대한 특정 년/월의 판매 실적 집계

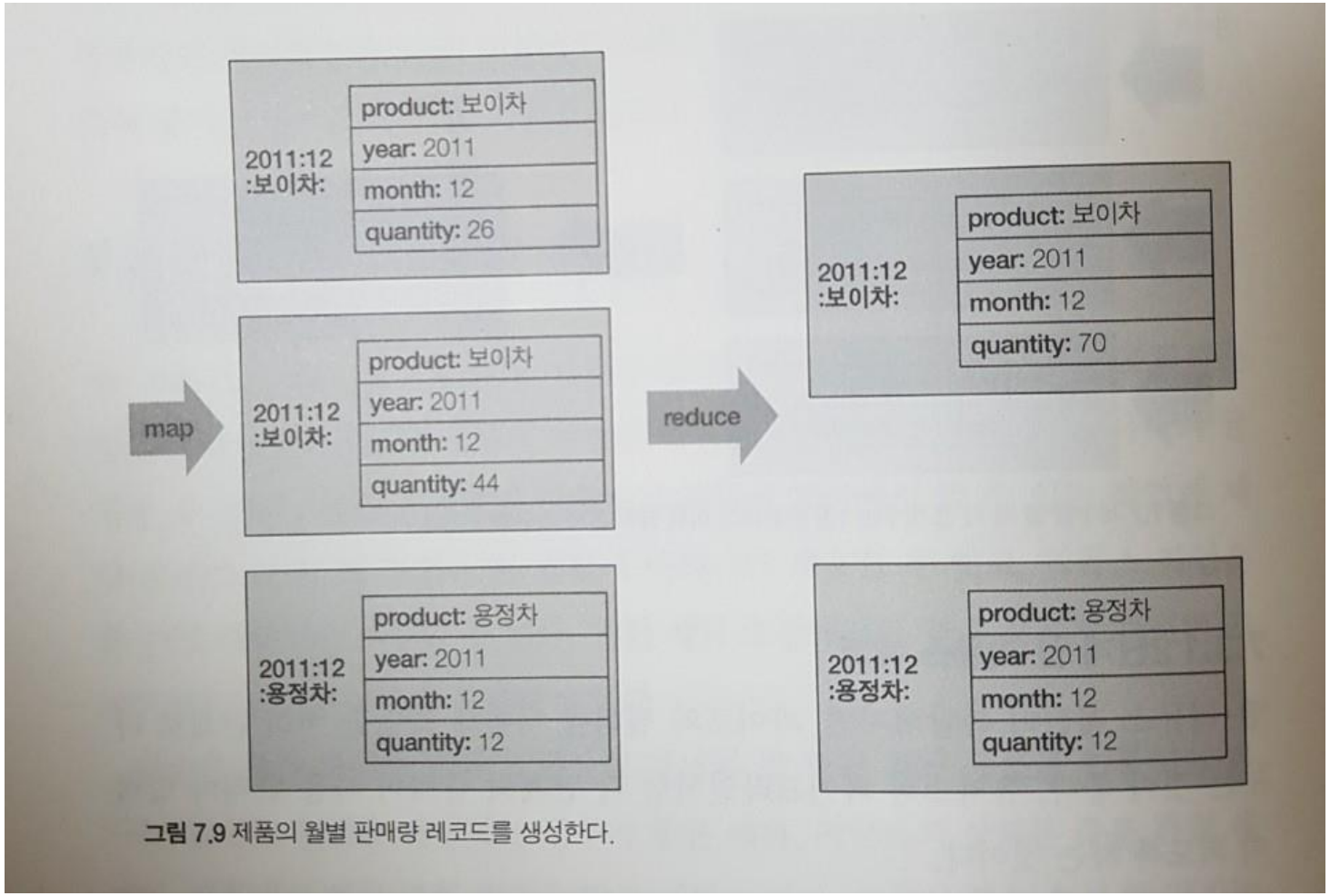

그림 7.9 제품의 월별 판매량 레코드를 생성한다.

7.3 맵-리듀스

7.3.1 2단계 맵-리듀스 예제

- 맵-리듀스 계산이 복잡해지면
- (유닉스의 파이프라인처럼)

- 예: 2011년도와 그 전년도의
  1. 한 제품에 대한 특정 년
  2. 첫 단계의 결과(input)

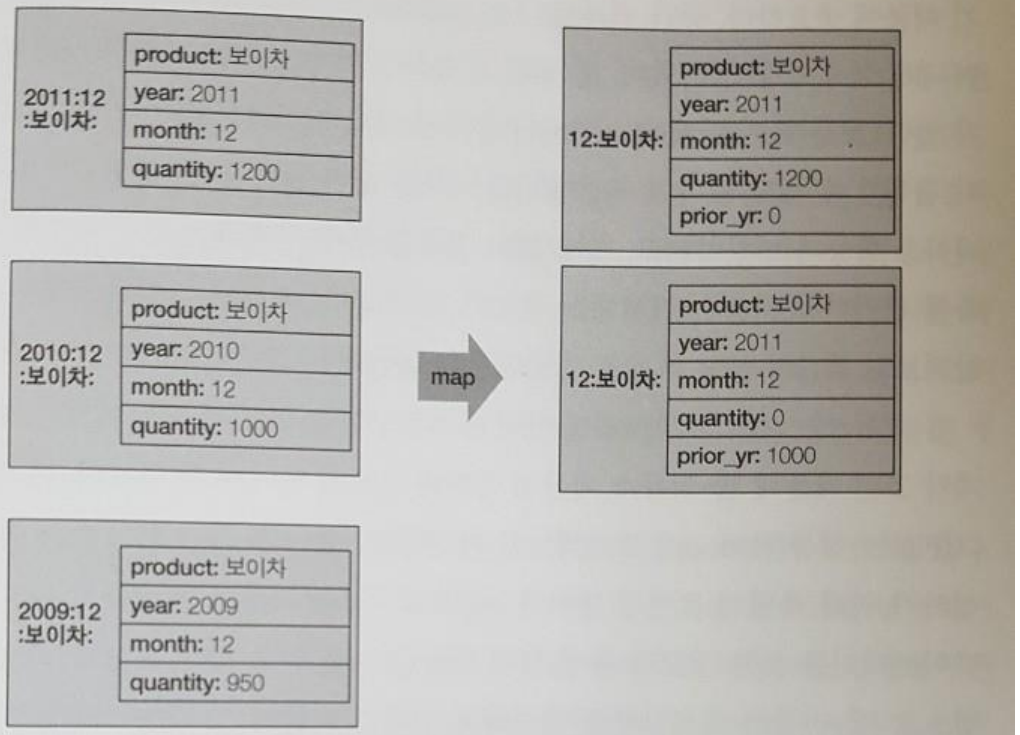

그림 7.10 둘째 단계에서 맵 함수는 연도별 비교를 위한 기본 레코드를 생성한다.

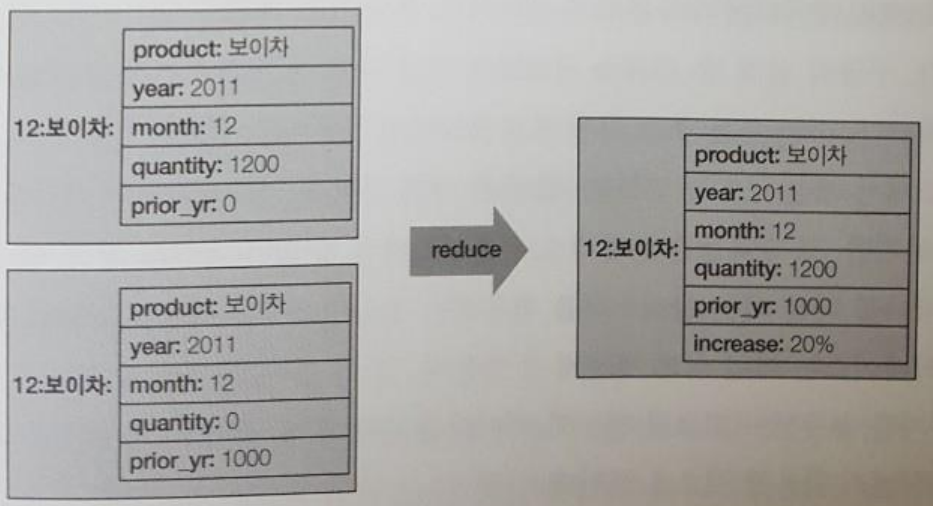

그림 7.11 리듀스는 불완전한 레코드를 병합하는 단계다.

7.3 맵-리듀스 계산 조합

7.3.2 점증적 맵-리듀스

- 맵-리듀스 계산은 시간 오래 걸리는 경우 많음
- **점증적 업데이트** 할 수 있도록 구조 만들어서 계산을 최소화

- **맵**: 처리가 쉬움
- 변경된 입력 데이터에 대해서만 맵 함수 재실행
- 서로 격리되어 있어 점증적 업데이트도 쉬움

- **리듀스**: 좀더 복잡 ← 이유: (맵의 결과를 모아서 계산하므로) 맵의 결과에 변경이 있는 경우 다시 계산해야함
- 파티션 있다면 ➔ 변경 x 파티션은 다시 계산 x도 됨
- 결합 있다면 ➔ 입력 단계 변경 X 경우 결합 재실행 X
- 리듀스 함수가 결합가능
  - 변경 형태가 '추가'(+만) ➔ 기존 결과 + 새로 추가한 데이터로 리듀스 실행
  - 파괴적 변경 있음(업데이트/삭제 있음) ➔ 리듀스 연산 쪼개고 입력이 변경된 단계만 재계산
- 본질적으로: 의존성 네트워크(Dependency Network) 사용

7.4 더 읽을거리

- 자신이 사용하는 DB의 문서를 가장 먼저 보세요
- 맵-리듀스 작업 구조를 어떻게 만들지에 대한 일반적인 정보를 위해 <하둡 책>

7.5 요점

- 맵-리듀스는 클러스터에서 계산의 병렬화를 쉽게 해주는 패턴

- **맵**: 집합으로부터 데이터 읽어서 → 관련된 **키-값 쌍**으로 압축
- 맵은 한번에 레코드 하나만 읽으므로 병렬화될 수 있고 해당 레코드 저장한 노드에서 실행됨

- **리듀스**: **한 키**에 대해 맵 작업의 결과로 많은 값을 취해 출력 하나로 요약
- 각 리듀스 함수는 한 키에 대해 작용 → 키로 병렬화 가능

- 입력-출력 같은 형태인 리듀스 함수는 **파이프라인**으로 연결 가능
- 이렇게하면 병렬성 향상, 전송할 데이터량도 감소

- **맵-리듀스 연산**은 **파이프라인**으로 연결 가능 (리듀스의 결과를 다른 연산의 맵에 입력으로 제공가능)

- 맵-리듀스 계산 결과가 많이 사용되면 이를 구체화 뷰로 저장 가능
- 구체화 뷰는 (모든 것을 처음부터 다시 계산하는 대신,) 변경된 부분만을 계산하는 **점증적 맵-리듀스 연산**으로 업데이트 가능