

컬럼 기반 NoSQL 개요

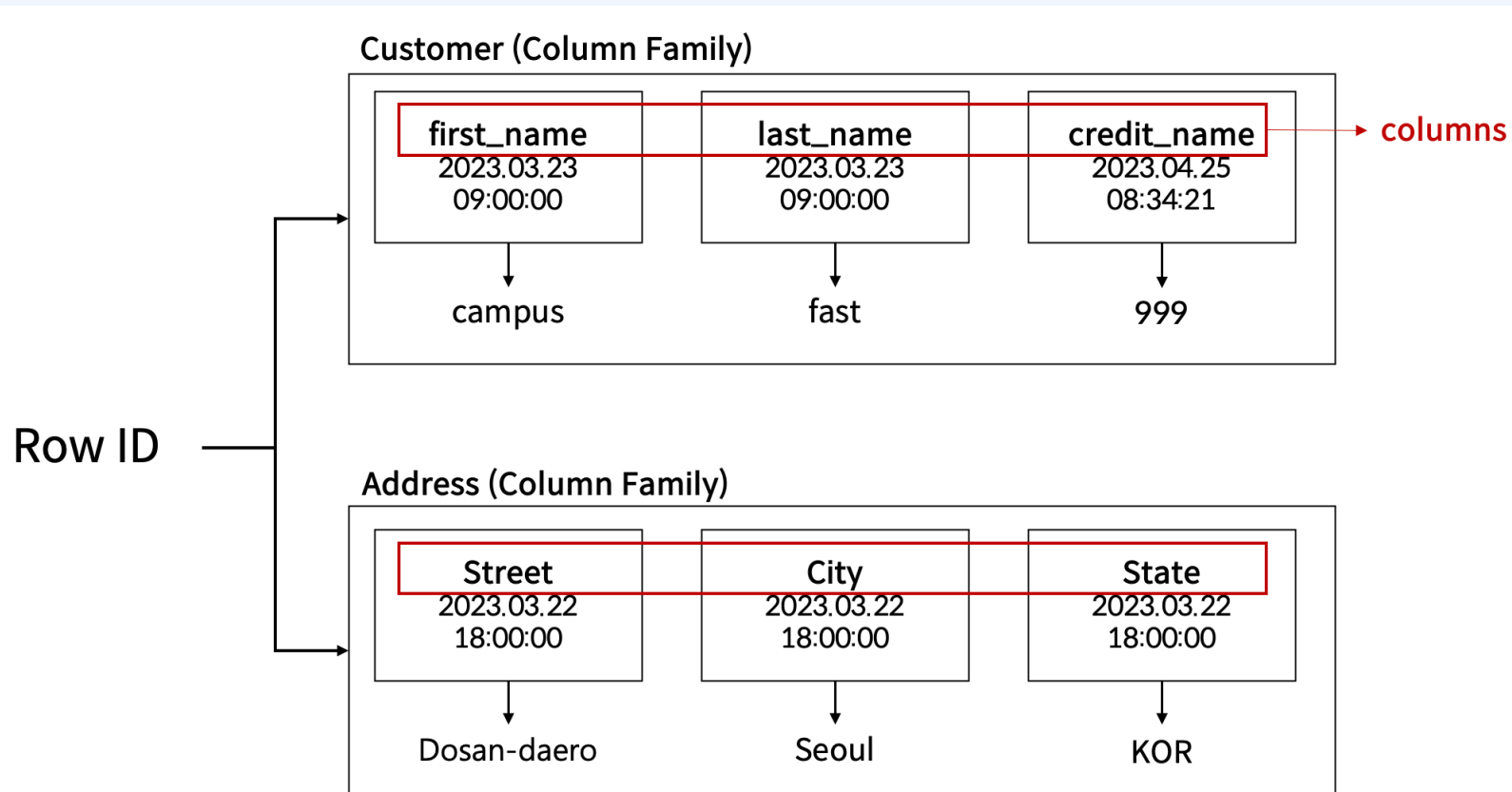
1. 컬럼 기반 NoSQL을 사용하는 이유

1. 컬럼 기반 NoSQL의 원조 - Google Bigtable

- Bigtable: A distributed Storage System for Structured Data
 - <https://static.googleusercontent.com/media/research.google.com/ko//archive/bigtable-osdi06.pdf>
- Bigtable의 핵심 특징
 1. 개발자가 동적으로 컬럼을 제어.
 2. 데이터 값이 Row 식별자, Column 이름, Timestamp로 인덱싱.
 3. 데이터 모델러와 개발자가 데이터의 저장 위치를 제어.
 4. 각 Row의 읽기, 쓰기는 원자적(Atomic)으로 처리.
 5. 각 Row는 정렬된 순서대로 유지 관리

1. 컬럼 기반 NoSQL의 원조 - Google Bigtable

- 데이터 모델 예시



1.1 개발자가 동적으로 컬럼을 제어.

- 개발자는 자유롭게 컬럼을 추가할 수 있음.
 - ex) Address Column Family에서, 도로명 주소(Street)외에 지번 주소를 추가해야 된다고 하면, LotNumber 컬럼을 추가하기만 하면 됨.
 - RDBMS에서 처럼, 테이블의 스키마를 재정의하고 업데이트 할 필요 없음.
- 단 Column Family는 자유롭게 추가가 불가능. 테이블이 생성된 시점에 고정.

1.2 데이터 값이 Row 식별자, Column 이름, Timestamp로 인덱싱.

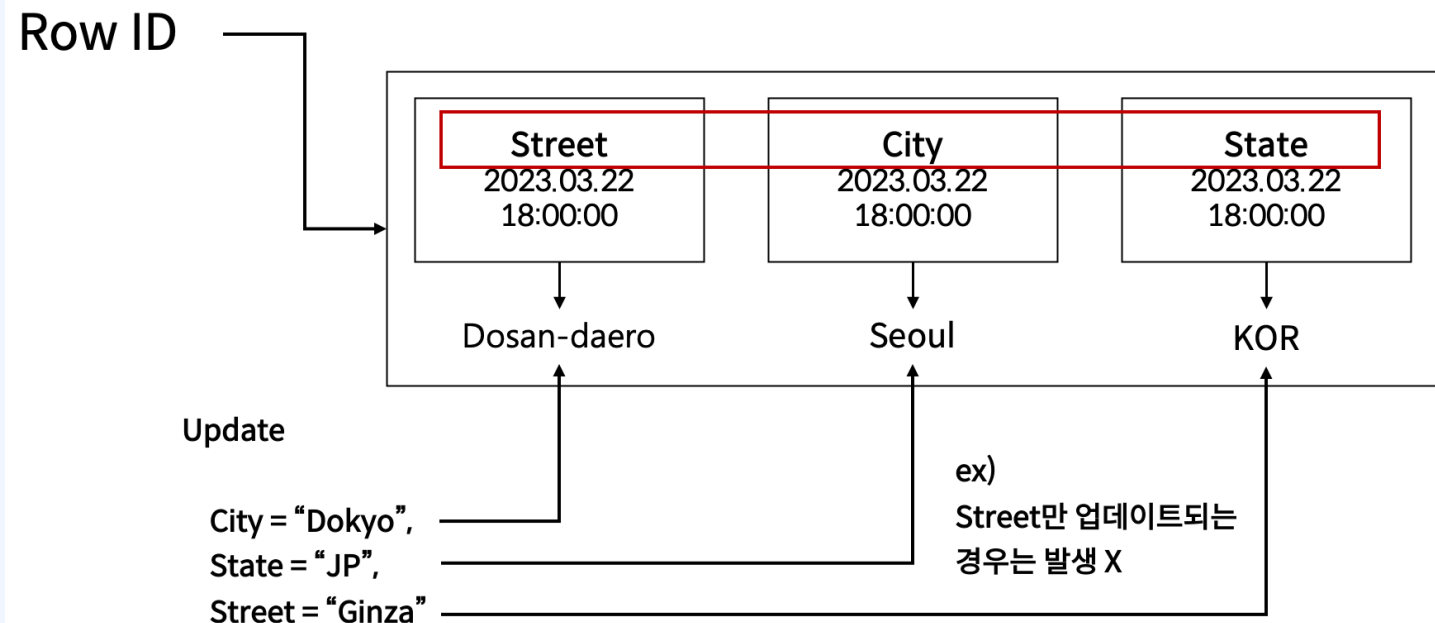
- Bigtable에서는 데이터 값을 검색할 때 Row 식별자, Column 이름, Timestamp를 인덱스로 사용.
- Row 식별자는 RDBMS의 primary key와 유사하게, Row를 고유하게 식별하는 역할을 함.
 - 단일 Row는 여러 Column Family를 가질 수 있음.
- Column 이름은 1개의 Column을 고유하게 식별하는 역할.
- 각 Column 값의 버전은 timestamp를 이용해 관리.
- Bigtable에 새로운 값을 쓸 때는 이전 값을 덮어쓰지 않고, 새로운 Timestamp와 함께 값이 추가.
 - -> Application에서는 Timestamp를 사용해 최종 버전의 Column을 찾을 수 있음.

1.3 데이터 모델러와 개발자가 데이터의 저장 위치를 제어.

- 한 Column Family 내에 존재하는 Column들은 Disk내에 연속된 위치에 저장됨. -> 단일 데이터 블록만 읽어도 질의에 대한 결과를 반환 가능

1.4 각 Row의 읽기, 쓰기는 원자적으로(Atomic) 처리.

- 읽거나 쓰는 컬럼의 수와 상관 없이, 모든 읽기, 쓰기 작업은 원자적으로 수행.
- 원자적(Atomic)?
 - 부분적인 결과를 반환하지 않는 것.
 - 읽기, 쓰기 -> 읽어야(써야) 할 모든 컬럼을 읽거나(쓰거나) 전혀 읽지 않음.(쓰지 않음)



1.5 각 Row는 정렬된 순서대로 유지 관리.

- Row id 기준으로 정렬 -> Range query (범위 질의) 가능.
- 한 가지 방식으로만 테이블을 정렬할 수 있으므로, 정렬 순서를 정할 때는 주의.

2. 컬럼 기반 NoSQL을 사용하는 이유 (RDBMS와 비교)

1. 공간 절약 가능.
2. 자유로운 스키마 구조.
3. 확장성 용이.

1. 2. -> 뒤의 Products Catalog 예시를 통해 확인

2.1. 2.2 ex) Products Catalog

- Catalog ? - 다양한 상품 목록들을 저장하고 있는 거대한 테이블.
- 상품의 카테고리 별로 다양한 속성들이 존재

- **Books**

- ISBN
- title
- author
- publisher
- year
- ...

- **T-shirts**

- brand
- color
- size
- ...
- year
- ...

- **Laptops**

- brand
- size
- cpu
- ram
- gpu
- disk_size
- battery_capacity
- ...

2.1. 2.2 ex) Products Catalog - if RDBMS

1. 각 Row 별로 빈 공간이 많이 생김.
 - 진짜 빈 공간? -> X, RDBMS에서는 공간을 차지하게 됨. -> 공간 낭비 발생.
2. 다른 카테고리의 상품이 추가되면, 컬럼이 많이 추가될 수 있음.
 - 추가될 때마다 스키마가 변경되고, 테이블을 업데이트 해야 하므로 번거로움.

[illegible]

2.1. 2.2 ex) Products Catalog - if 컬럼 기반 NoSQL

- 낭비되는 공간 X
-> 공간 최적화 가능.
- 컬럼이 추가되는 것은 단지
attribute name key가 하나 추가되는 것!
-> 기존 테이블을 업데이트할 필요 X

id	attribute name	attribute value
book1	brand	sayno
book1	author	sayno
book1	publisher	ssss
book1	year	2023
book1	ISBN	1234
t-shirt1	brand	adidas
t-shirt1	color	black
t-shirt1	size	105
laptop1	brand	lg
laptop1	cpu	8 core

2.3. 확장성 용이.

- RDBMS의 경우, 강력한 ACID transaction을 제공
- ACID transaction?
 - Atomicity (All or nothing)
 - Consistency (항상 유효한 데이터만 저장)
 - Isolation (여러 transaction이 한 table에 읽고 쓰는 작업을 할 때 서로 간섭 X)
 - Durability (transaction 수행 후 커밋된 데이터는 장애가 나더라도 잘 저장되어야 함)
- 단일 node에서 ACID 를 달성하는 것은 어렵지 않지만, 분산 환경에서는 어려움.
- 컬럼 기반 NoSQL에서는 ACID 특성을 완벽하게 지키지는 못하지만, 대신 분산 환경에서 쉽게 수평 확장이 가능함.
- Trade-off

3. 컬럼 기반 NoSQL을 사용하면 안 좋은 경우.

1. 분산 환경이 필요하지 않을 때.
2. 데이터의 일관성이 매우 중요할 때.
 - ex) 은행 계좌. 상품 주문 정보.
 - 특정 시점마다 은행 계좌의 잔액이 다르다거나, 주문 정보가 다르다 하면 매우 곤란.
 - 반면 상품의 재고 수 정보 등은 일반적으로,
일시적으로 일관성이 깨지더라도 비즈니스에 큰 영향을 끼치지 않는음.

Cassandra

1. Cassandra 개요

1. Why Cassandra?

- 가장 널리 쓰이는 오픈 소스 분산형 NoSQL 중 하나.
- NoSQL마다 세부적인 동작 방식, 용어 등에는 차이가 있지만 기본 원리는 유사하기 때문에, Cassandra의 주요 개념들을 잘 알아놓으면 다른 분산 NoSQL을 공부하기 편함.

2. Cassandra 정의

- Open source
- Distributed, Decentralized (2.1)
- Elastic scalability (2.2)
- High Availability and Fault Tolerance (2.3)
- Tuneable Consistency (2.4)
- row-oriented (2.5) (단원 제목은 column 기반 NoSQL인데 row-oriented..?)

의 속성을 가지고 있는 database

2.1 Distributed, Decentralized.

1. distributed.

- 여러 개의 node에서 실행이 되고, 클라이언트 입장에서는 마치 하나의 node인 것처럼 보임.
- 앞에서 배운 Hadoop ecosystem, Spark도 모두 분산 처리 프레임워크.
- 수평 확장이 가능.

2. decentralized.

- **Cassandra cluster 내의 모든 node들은 완전히 동등한 기능을 수행.**
(HDFS, Spark, Hbase 등 앞에서 배운 프레임워크들은 모두 Master - Worker node의 형태로, Master와 Worker node는 각기 다른 기능을 수행.)
- SPOF가 (Single Point Of Failure) 존재하지 않음.
- 특정 node가 고장이 나더라도, cluster 내의 나머지 node들을 사용해 같은 기능을 수행할 수 있음.

2.2 Elastic scalability

- Scalability? (확장성)
 - 성능 저하 없이 더 많은 요청을 지속적으로 처리할 수 있는 시스템의 아키텍처 기능.
- Vertical scalability
- Horizontal scalability
- Elastic scalability
 - Horizontal scalability의 특별한 예.
 - 새로 생성된 node는 데이터의 일부, 또는 전체 복사본을 가지고 cluster에 참여 가능.
 - 전체 클러스터의 큰 중단이나 재설정 없이 새로운 사용자의 요청에 응답할 수 있음.

2.3 High Availability and Fault Tolerance

- Availability (가용성)
 - 모든 DB의 클라이언트들이 항상 데이터를 읽고 쓸 수 있는 능력.
- Fault Tolerance.
 - 시스템의 일부에 장애가 발생하더라도 전체 시스템은 정상적으로 기능할 수 있는 특성.
- Cassandra Cluster 내의 node들이 고장 나더라도 중단 시간 없이 교체 가능.
- 여러 데이터 센터에 데이터를 복제하여, 한 데이터 센터에서 화재, 지진, 홍수 등의 재해가 발생하더라도 큰 중단 시간 없이 Availability를 제공.

2.4 Tunable Consistency

- Consistency(일관성)?
 - 모든 DB의 클라이언트들이 동일한 질의를 했을 때 동일한 값을 반환할 수 있는 능력.
 - 동시 업데이트 상황에서도 보장이 되어야 함.
- Cassandra는 이 Consistency의 정도를 조정할 수 있음.
- Consistency와 Availability는 Trade-off 관계!

2.4 Tunable Consistency - Consistency의 종류

1. Strict Consistency

- 클라이언트들이 동일한 질의를 했을 때 항상 동일한 값을 반환하도록 보장.
- 단일 node에서는 달성이 쉽지만, 분산 환경(여러 개 node)에서는, 데이터를 업데이트(추가, 갱신, 삭제)하는 연산들이 모두 동기적으로 실행되어야 함 -> block되는 시간 발생 -> 이 시간 동안은 질의에 대한 응답을 받을 수 없음. -> Availability 감소.

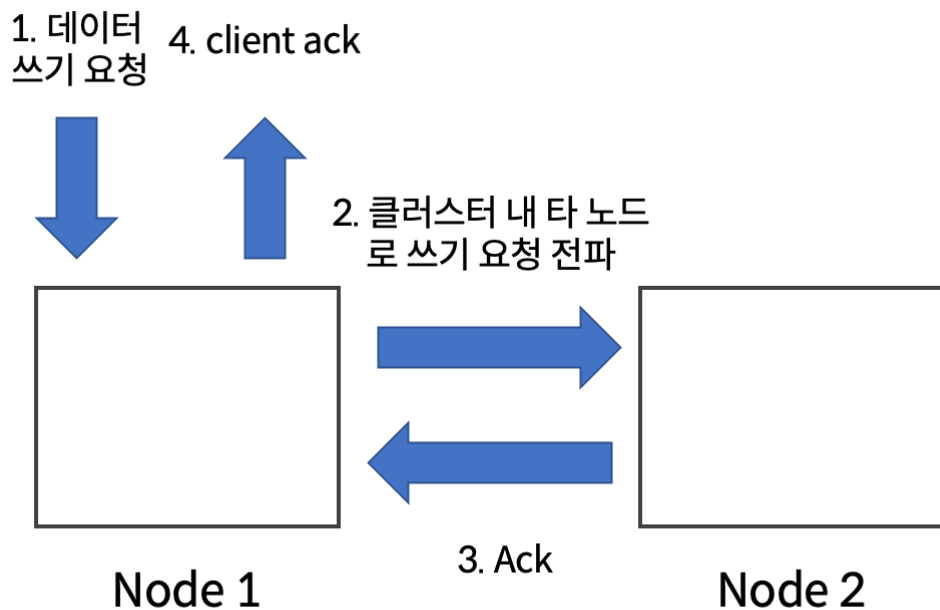
2. Weak(Eventual) Consistency

- 모든 업데이트 연산들이 Cluster의 노드들에 결국에는 전파되어 Consistency를 확보할 수 있지만, 연산들이 실행되는 중에는 Consistency를 확보하지 못함.
(= 같은 질의에 대해 다른 결과를 반환할 수 있음.)
- 대신 높은 Availability 확보 가능.

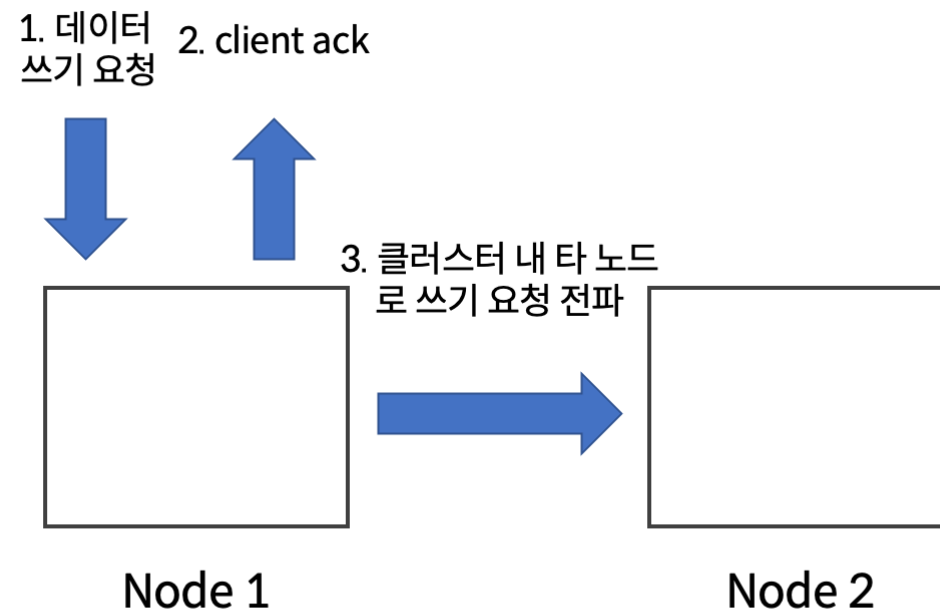
-> Cassandra는 설정에 따라 Strict, Weak Consistency 중 어떤 전략을 택할지 정할 수 있음!

관련 용어 : Brewer's CAP Theorem.

2.4 Tunable Consistency - Consistency 방법 간 비교



Strict Consistency



Eventual Consistency

2.5 Row oriented.

- Cassandra의 Data Model은 정확히는 “partitioned row store”.
- Column-based (컬럼 기반) NoSQL로 많이 알려져 있지만, 엄밀한 용어로는 맞지 않음.
- Partitioned row store?
 - Sparse, Multidimensional, sorted Hash table 형태로 데이터를 저장.
 - Data Type : Map<RowKey, SortedMap<ColKey, Col value>>
 - Partitioned?
 - 각 row 는 고유한 partition key를 가지고 있고, 이 key 값에 따라 row를 node에 분배. (spark 의 partitioning과 유사)
 - Sparse?
 - 각 row는 1개 이상의 column을 가질 수 있는데, 모든 row가 같은 column들을 갖고 있을 필요는 없음.
 - RowKey ? ColKey? partition key?

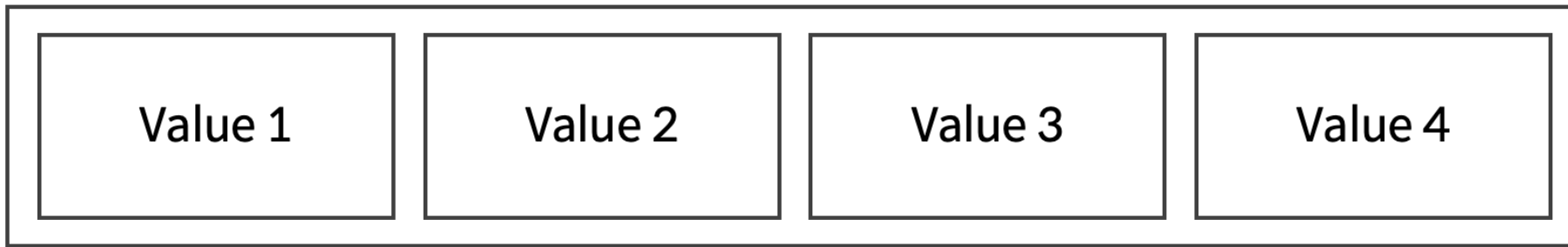
2. Cassandra Data Model

1. 개요

- Bottom up 방식으로, Cassandra의 데이터 모델을 이해
 - 간단한 구성 요소들부터 전체를 소개하는 방식.

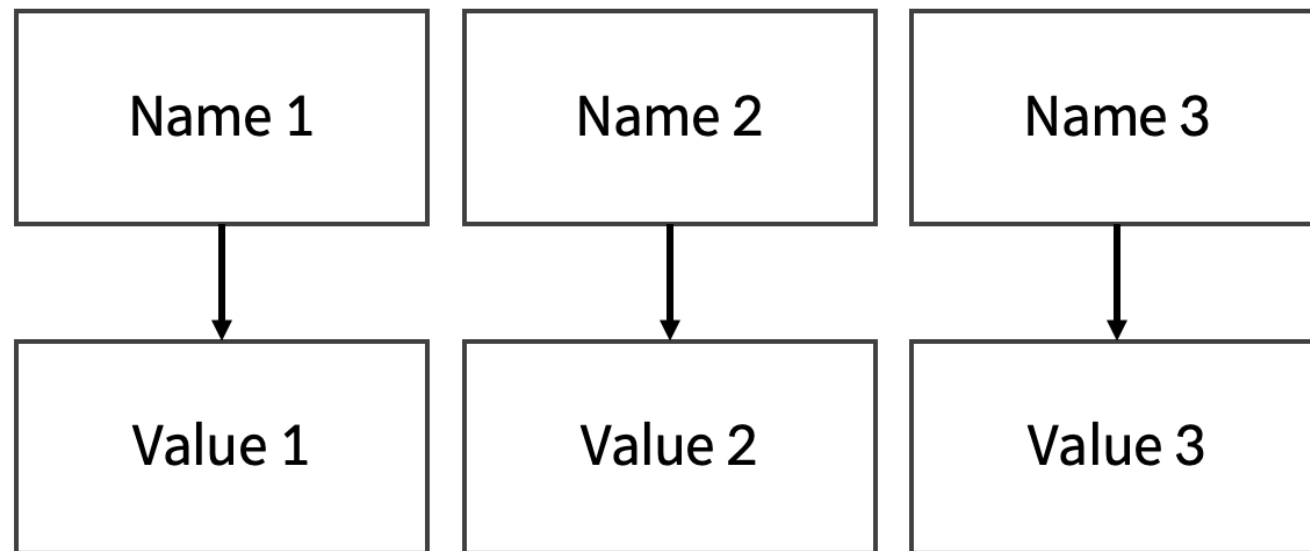
2. 가장 간단한 데이터 모델 - array or list

- 단순하지만 부족한 것이 많음.
- ex) 이 array에 어떤 값이 존재하는지 여부를 쿼리할 수 있지만, 각 값이 존재하는지 확인하기 위해 전체 array를 순회하거나, 항상 array의 동일한 위치에 각 값을 저장한 후, array의 어느 인덱스에 그 값이 있는지에 대한 정보를 따로 가지고 있어야 함.



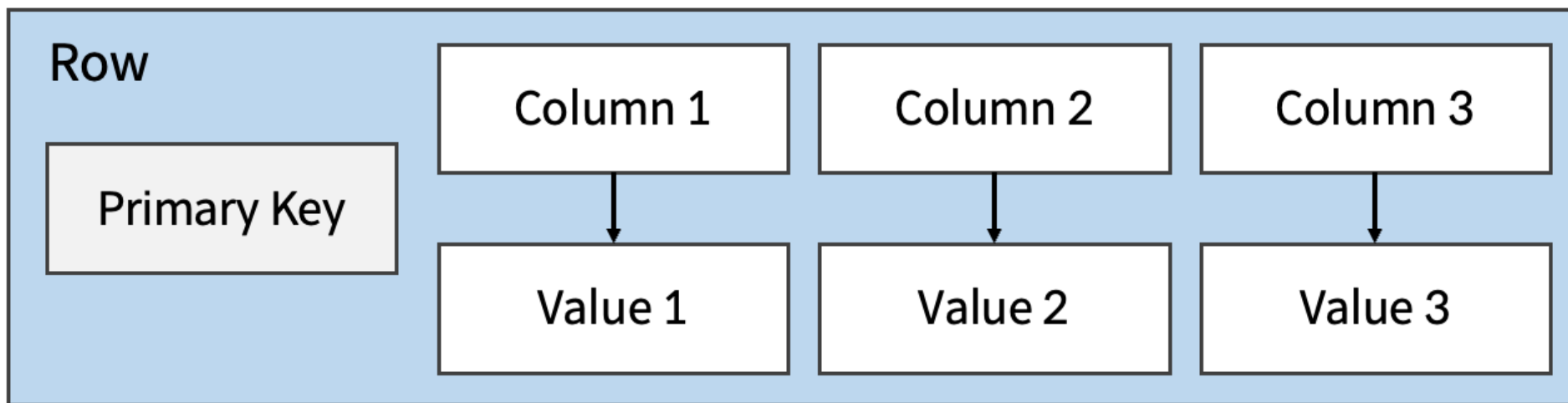
3. 조금 더 개선된 데이터 모델 - Map

- 각 Value에는 매칭이 되는 Name이 존재.
- 각 Column Name(속성) (ex: first_name, last_name, email)을 사용해 쿼리가 가능.
- 단순 Map은 여러 속성을 가지고 있는 1개의 entity에 대해서만 잘 설명이 가능함.
(ex: 사람 1명, 1개의 tweet)
- name / value 쌍의 컬렉션을 통합 불가, 동일한 Name들을 반복할 수도 없음.



4. Cassandra Row

- Row = Primary Key + (Column Name, Value) Pairs.
- Key를 이용해 한 그룹의 Column들을 1개의 set으로 묶을 수 있음.
- 고유 식별자가 되는 이 Key를 Primary Key, 혹은 Row Key라고 부름.
- 특정 Row의 정보를 가져올 때, Row 내의 모든 Column을 가져올 수도 있고, 필요한 특정 Column들만 가져올 수 있음.



5. Cassandra Table(= Column Family)

- 유사한 데이터를 모아 놓은 논리적 단위.
- Column Family와 동일한 의미

Table

Row

Primary Key

Column 1

Value 1

Column 2

Value 2

Column 3

Value 3

Row

Primary Key

Column 1

Value 1

Column 4

Value 2

6. Composite Key(Partition Key + Clustering Columns)

- Primary Key의 특별한 케이스.
- 서로 관련 있는 Row들의 그룹(= Partition)을 대표하기 위해 사용.
- Composite Key는 두 가지 요소로 구성됨.

1. Partition Key

- **Partition이 어느 노드에 저장되는지를 결정.**
- 1개 혹은 여러 개의 Column을 이용해 만들 수 있음.

2. Clustering Columns (Clustering Keys)

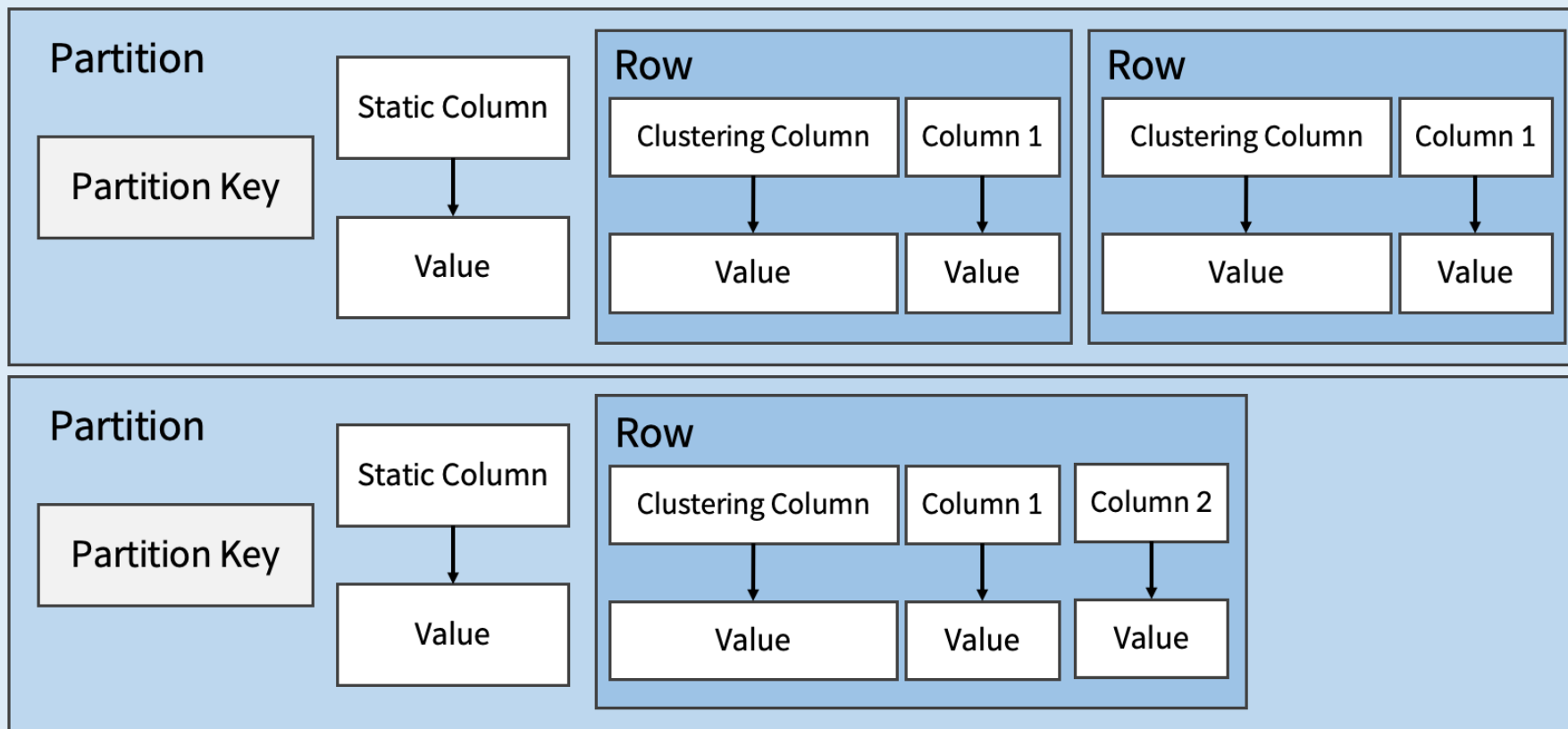
- **Partition 내의 Row들을 정렬하는 기준.**

7. Static Column

- Primary Key의 일부는 아니지만,
한 Partition 내에 모든 Row에서 공유되는 Column

8. Partition이 존재하는 경우의 Cassandra Table 구조

Table



9. 용어 정리

1. column

- 각각의 속성 이름 (ex: last_name, first_name), 이름과 매칭되는 값의 쌍.

2. row

- Primary key, 혹은 row key로 구별되는 column들을 담은 container.

3. partition

- row들의 그룹, 같은 node에 저장됨.

4. table

- partition별로 구성된 row들을 담은 container.

5. keyspace

- table들을 담은 container.

6. cluster

- 1개 이상의 node에 걸쳐 있는 keyspace들을 담은 container

3. CQL (Cassandra Query Language) 소개

1. CQL 개요

- Cassandra Query Language의 약자.
- Cassandra db와 소통하기 위한 쿼리 언어.
- RDBMS 의 SQL 커맨드와 유사한 문법 사용.
- cqlsh 라는 command line client 위에서 수행됨.

2. 실습 준비

- Docker Desktop 설치 필요.
- MAC : 실습 영상 참고
- Windows : PPT P.97 부록) Docker Desktop 설치 (Window) 참고

2. 코드 실습 링크

https://github.com/startFromBottom/fc-cassandra-practices/blob/main/part10_ch02_03.txt

4. CQL에서 Partition Key, Clustering Key 의 제약 조건

1. Recap Partition Key, Clustering Key

1. Partition Key

- Partition이 어느 노드에 저장되는지를 결정.
- 1개 혹은 여러 개의 Column을 이용해 만들 수 있음.

2. Clustering Columns (Clustering Keys)

- Partition 내의 Row들을 정렬하는 기준.
- Partition key와 마찬가지로 1개 혹은 여러 개의 Column을 이용해 만들 수 있음.

2. 코드 실습 링크

https://github.com/startFromBottom/fc-cassandra-practices/blob/main/part10_ch02_04.txt

<https://github.com/startFromBottom/fc-cassandra-practices/blob/main/data/input.csv>

3. Partition Key 제약 조건 정리

1. WHERE 절에는 Partition Key를 만드는 데 사용되는 모든 column들이 명시되어야 함.
2. WHERE 절에 Partition key 사용 시, '=', 'IN' 연산만 사용 가능. (\geq , $>$, $<$, \leq 지원 X)
3. ORDER BY 절에 Partition Key의 column은 사용 불가

4. Clustering Key(Columns) 제약 조건 정리

1. WHERE 절에 N번째 clustering column을 넣었다면, 1, ... , N-1번째 clustering columns들로 제한 되어야 함. (= WHERE 문에 반드시 넣어야 함)
2. 더 앞에 정의한 Clustering column으로 range query를 수행했다면, 뒤의 Clustering column으로 제한할 수 없음(= WHERE 조건에 넣을 수 없음)
3. Clustering Columns 으로 ORDER BY를 사용하기 위해서는 WHERE 절에서 partition key로 "IN" 조건을 사용하면 안되고, N 번째 Clustering Column으로 정렬하려면, 1, ... , N-1 번째 Clustering Column도 ORDER BY 조건에 넣어야 함.

5. CQL에서 Secondary Index의 제약 조건

1. 개요 - Secondary Index?

- CQL 사용 시,
WHERE 절에서 Primary Key의 구성 요소인 Partition Key와 Clustering Key를
만드는 데 사용되는 Column 만 사용이 가능한 것을 확인!
- 만약 Primary Key 이외의 column들을 WHERE 절에 넣어서 사용해야 한다면
어떻게 해야할까?
-> Secondary Index를 지정해야 함.

2. Secondary Index in Cassandra

- WHERE 절에 primary key 이외의 column들을 사용하고자 할 때 사용.
- Secondary Index는 별도의 column family에 저장됨.

3. Secondary Index 사용시 주의할 점

Cassandra은 여러 Node에 데이터를 분할해 관리하기 때문에,
각각의 Node는 자신이 소유하고 있는 Partition에 저장된 데이터를 기반으로
secondary index의 복사본을 소유, 유지하고 있어야 함.

-> secondary index를 이용해 쿼리할 때, 많은 경우에서 여러 Node를 조회해야 하기 때문에, 고비용 발생!

주의할 점)

1. Cardinality가 너무 큰(= 고유한 값들이 많이 존재.) Column을
Secondary Index로 사용하지 않기.
 - index를 생성하는 비용이 너무 큼.
2. Cardinality가 너무 작은(ex: Boolean Field) Column을
Secondary Index로 사용하지 않기.
 - index의 크기만 커지고, index를 사용하는 이점이 크지 않음.
3. 자주 업데이트되거나 삭제되는 Column을
Secondary Index로 사용하지 않기.

4. 실습 코드 링크

https://github.com/startFromBottom/fc-cassandra-practices/blob/main/part10_ch02_05.txt

6. Consistency

1. 개요 - Recap Consistency

- Consistency(일관성)
 - 모든 DB의 클라이언트들이 동일한 질의를 했을 때, 동일한 값을 반환할 수 있는 능력.
 - 동시 업데이트 상황에서도 보장이 되어야 함.
- Consistency와 Availability(가용성)은 Trade-off 관계.
- Cassandra는 이 Consistency의 정도를 조정할 수 있음. -> ?
 - READ, WRITE 연산에 대해 Consistency의 Level을 정할 수 있음!

2. Write, Read Consistency in Cassandra

가정 : Cassandra 클러스터에 $N(\geq 2)$ 개의 node가 존재.

1. Write

- **N 개 중 몇 개의 Replica Node에 쓰기가 성공적으로 수행됐을 때, 클라이언트에게 성공 응답을 보낼 것인가?**
 - 언젠가는 모든 Replica Node에 쓰기 연산이 성공.
 - 모든 Replica Node에서 쓰기 연산이 성공하는 것을 기다린 후에 성공 응답을 보낼 것인가?
 - 1개 Replica Node에서만 쓰기 연산이 성공해도 성공 응답을 보낼 것인가?

2. Read

- **쿼리의 결과를 클라이언트에게 보낼 때, 몇 개의 Replica Node를 확인할 것인가?**
 - 모든 Replica Node 내의 데이터를 확인할 것인가?
 - 1개의 Replica Node 내의 데이터만 확인할 것인가?

Replica Node?

3. 용어 정리 - Replication Factor(RF), Quorum(Q)

1. Replication Factor (RF)

- 특정 Row를 한 Datacenter의 클러스터 내에 몇 개의 Node에 복사해놓을지 결정.
- ex) 클러스터 - 10개 Node, RF = 3
 - 10개 Node 중 3개의 Node에 특정 Row를 복사해놓음.

2. Quorum (Q)

- Write/Read 연산의 성공 여부를 확인할 때, Replica Node 들 중 몇 개의 Node를 확인할 것인지를 결정.
- ex) $Q = 2$, $RF = 3 \rightarrow$ 3개의 Replica Node 중에 2개의 Node를 확인.
- **Cassandra의 Consistency level은 클러스터 내의 전체 Node 수와는 관계가 없고, RF에 의해 결정!**
- Replication Factor와 Quorum의 관계
 - $Q = \text{floor}(RF/2 + 1)$
 - ex) $RF = 3 \rightarrow Q = 2$

4. Write Consistency in Cassandra

Q) 어느 Node에 데이터를 쓸지
어떻게 결정?

- Partition Key
- Partitioner

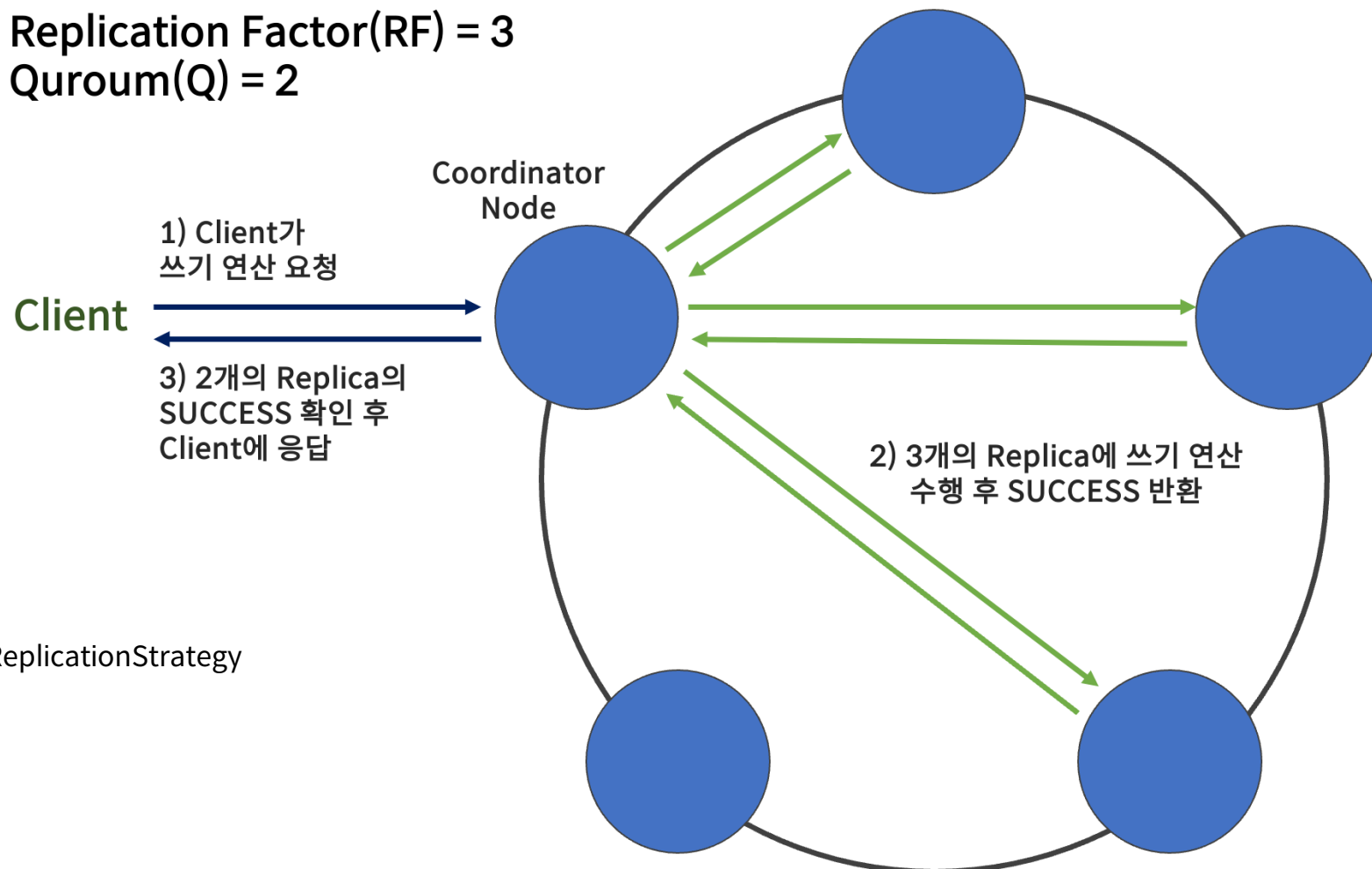
(default: Murmur3Partitioner)
`org.apache.cassandra.dht.IPartitioner`
구현

Q) 데이터를 어느 Node들에
복사할지 어떻게 결정?

- SimpleStrategy
- NetworkTopologyStrategy

`org.apache.cassandra.locator.AbstractReplicationStrategy`
상속해 구현

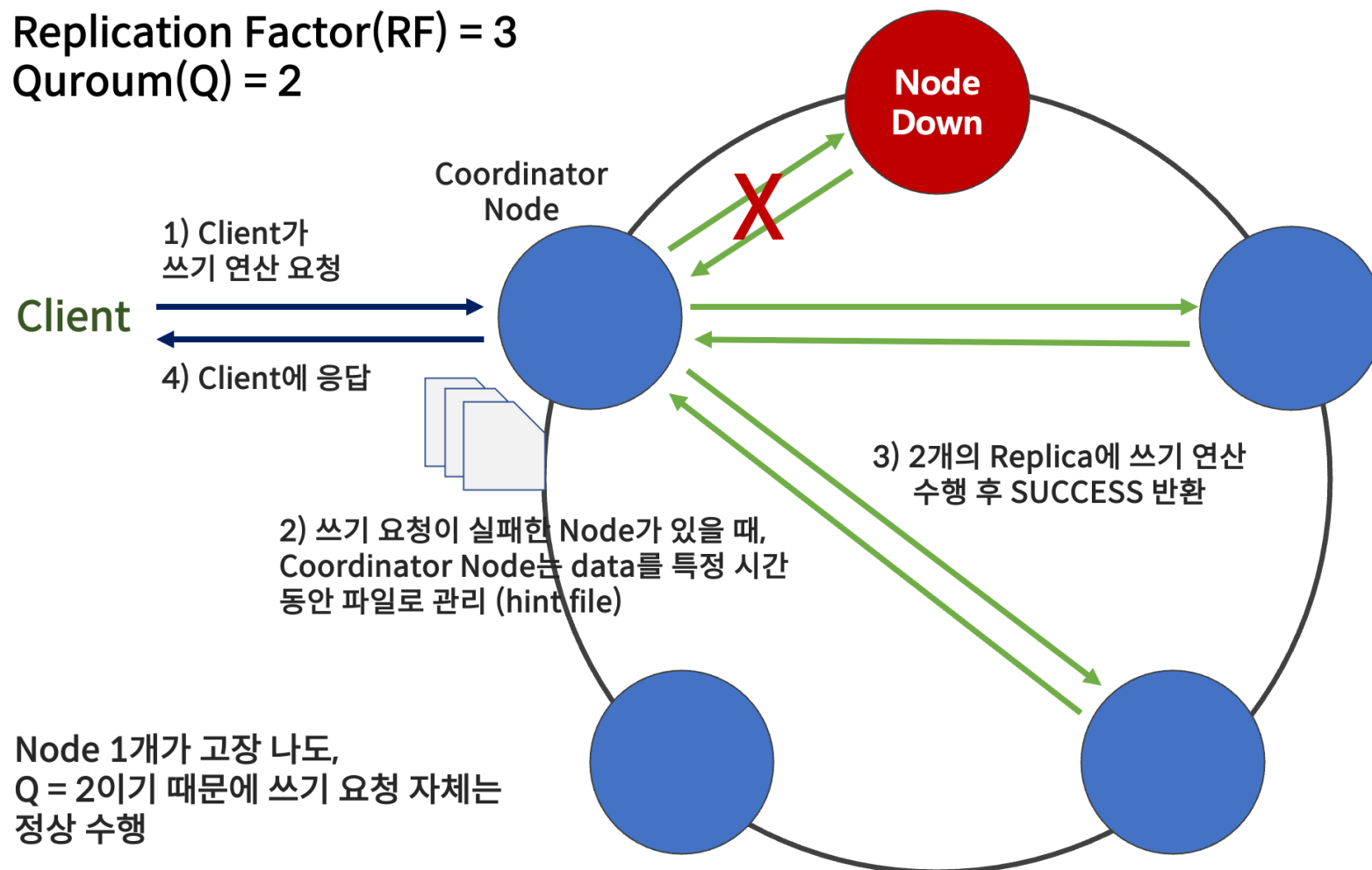
Replication Factor(RF) = 3
Quorum(Q) = 2



4.1 Hinted Handoff

- 쓰기 연산 중 특정 Node에 장애 발생 시, Consistency가 깨지는 것을 막는 방법
- 고장난 Node가 있을 때 hint file을 생성하고, Node가 복구된 후 hint file을 바탕으로 해당 Node에 쓰기 수행

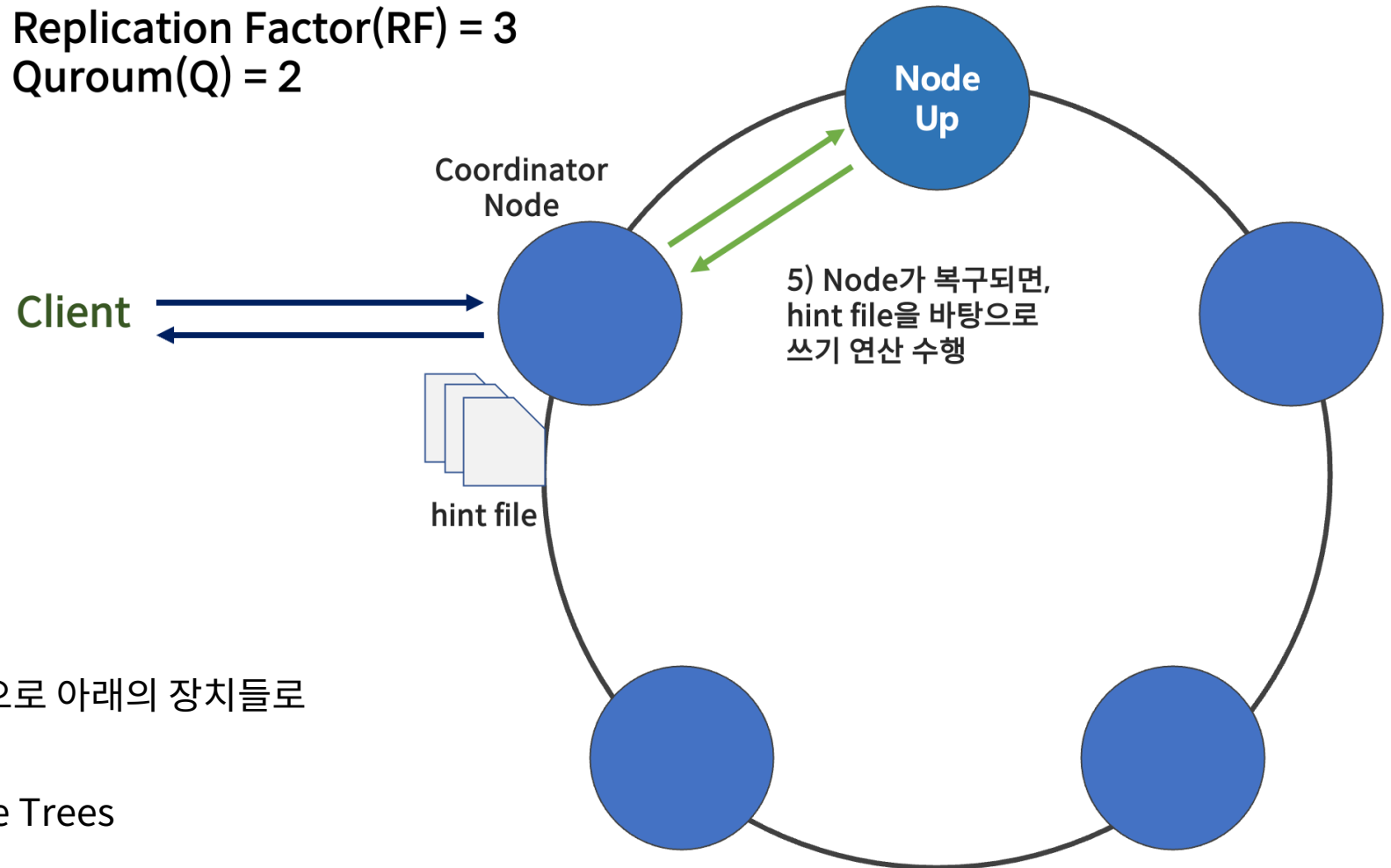
Replication Factor(RF) = 3
Quorum(Q) = 2



Node 1개가 고장 나도,
Q = 2이기 때문에 쓰기 요청 자체는
정상 수행

4.1 Hinted Handoff

Replication Factor(RF) = 3
Quorum(Q) = 2



- Hinted handoff 외에 추가적으로 아래의 장치들로 Consistency를 보장
- Anti-entropy, Repair, Merkle Trees

4.2 Write Consistency Quorum

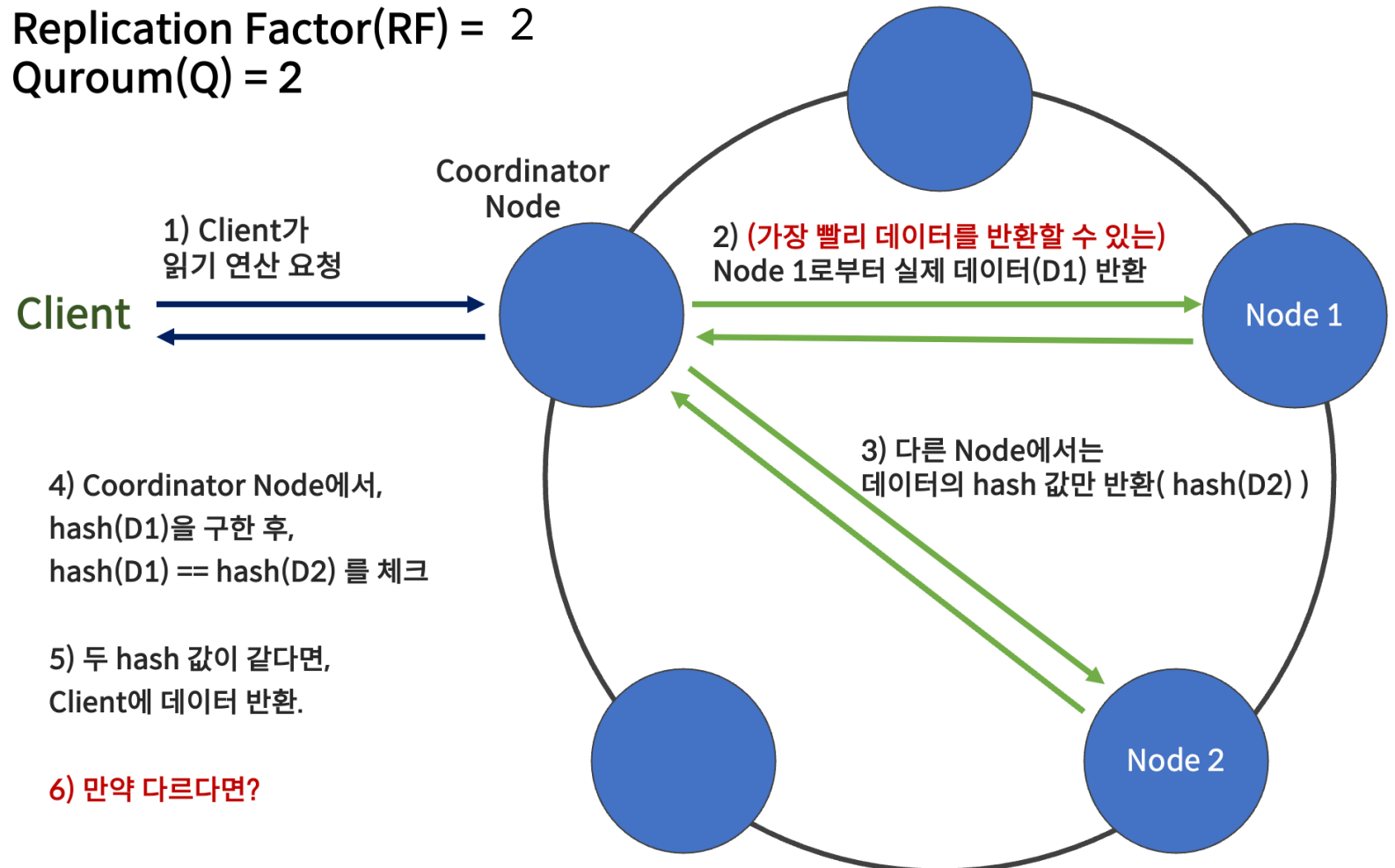
- Quorum 값이 클수록, Consistency 는 잘 보장되지만, Availability는 감소.
- ex) $Q = 1$
 - 1개의 Replica Node에만 쓰기 연산이 잘 수행됐는지 확인.
 - 빠른 응답 가능, 그러나 다른 Replica Node에는 값이 잘 써진 것을 보장하기 어려움.
- ex) $Q = ALL$
 - Cluster 내의 모든 Replica Node에서 쓰기 연산이 잘 수행됐는지를 확인.
 - 모든 Replica Node에 값이 잘 써진 것을 보장 가능. 느린 응답 시간.

5. Read Consistency in Cassandra

(가장 빨리 데이터를 반환할 수 있는)

-> Coordinator Node에 데이터가 존재한다면, 다른 Node에서 실제 데이터 반환할 필요 X

Replication Factor(RF) = 2
Quorum(Q) = 2



5.1 Snitch - 가장 빨리 데이터를 반환할 수 있는 Node 정보를 제공

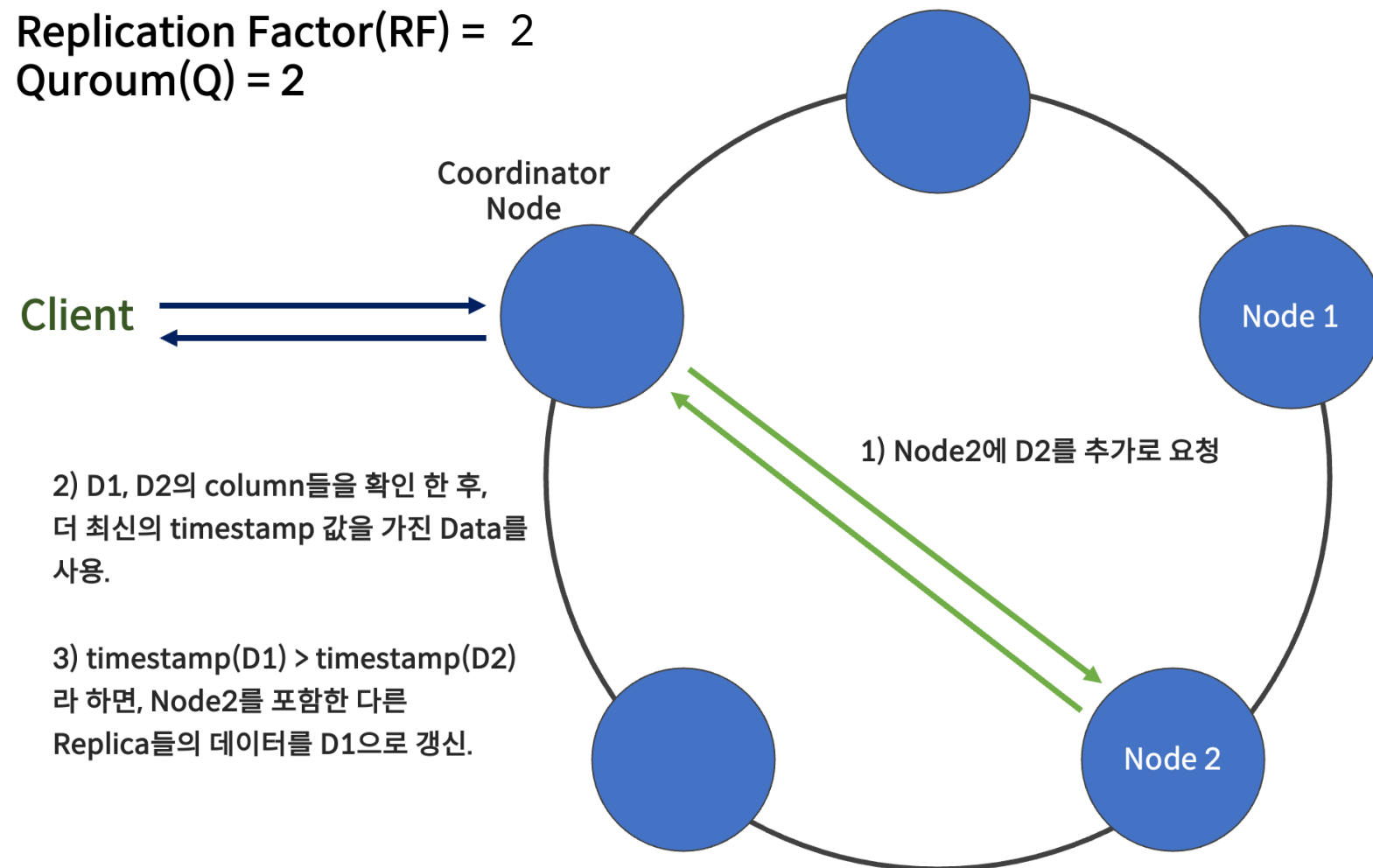
- Cassandra Cluster의 network topology(네트워크의 구성 방식, 배치 형태), 각 Node간의 관계 정보를 가지고 있는 Job.
- 이러한 정보를 바탕으로, Snitch는 읽기 연산 수행 시 어느 Node에서 가장 빠르게 데이터를 반환할 수 있는지를 확인 가능.

5.2 Read Repair

앞의 예시에서,
 $\text{hash}(D1) \neq \text{hash}(D2)$ 인 경우.

- Read Repair를 얼마나 자주 할지 설정 가능.

Replication Factor(RF) = 2
Quorum(Q) = 2

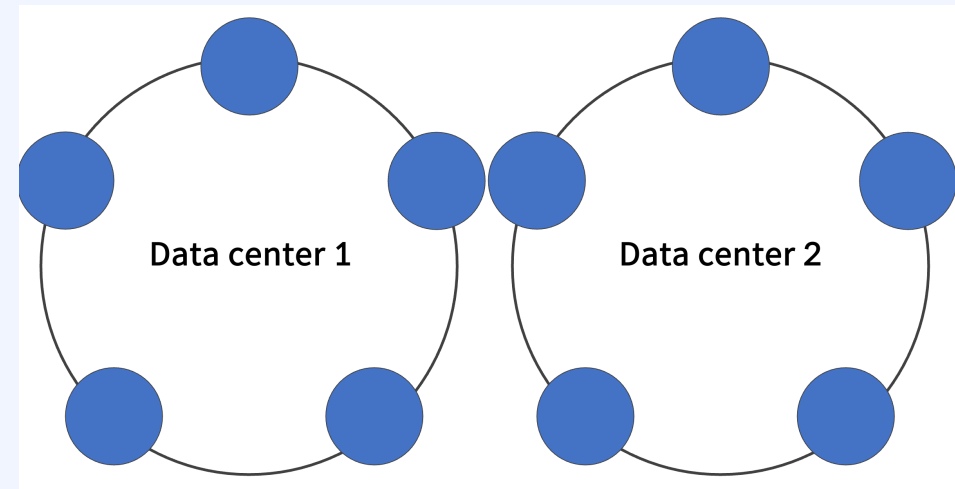


5.3 Read Consistency Quorum

- Quorum 값이 클수록, Consistency 는 잘 보장되지만, Availability는 감소.
- ex) $Q = 1$
 - 1개의 Replica Node에서만 읽기 연산.
 - 빠른 응답 시간, 다른 Node로부터는 동일한 값을 읽을 수 있을지 보장 X
- ex) $Q = ALL$
 - Cluster 내의 모든 Node에 읽기 연산. (정정 : Node -> Replica Node)
 - 모든 Node 값으로부터 동일한 값을 읽는 것이 보장. 느린 응답 시간.

6. Consistency - LOCAL_QUORUM

- 앞의 예시들에서는 Data Center 1개인 경우를 가정.
- Q) 만약 여러 개의 Data Center가 있는 상황이라면..?
- Write
 - LOCAL_QUORUM = 2?
 - 각 datacenter에서, 2개의 Replica에서 성공하면 성공으로 간주.
- Read
 - 응답 시간 문제로 지원 X



7. Storage Component의 종류

1. 개요

- Cassandra에서는 다양한 Storage Component를 사용해 데이터를 저장.
(Hbase 등 다른 분산 NoSQL도 비슷함)
- Storage Component는 크게 두 가지로 나눌 수 있음.

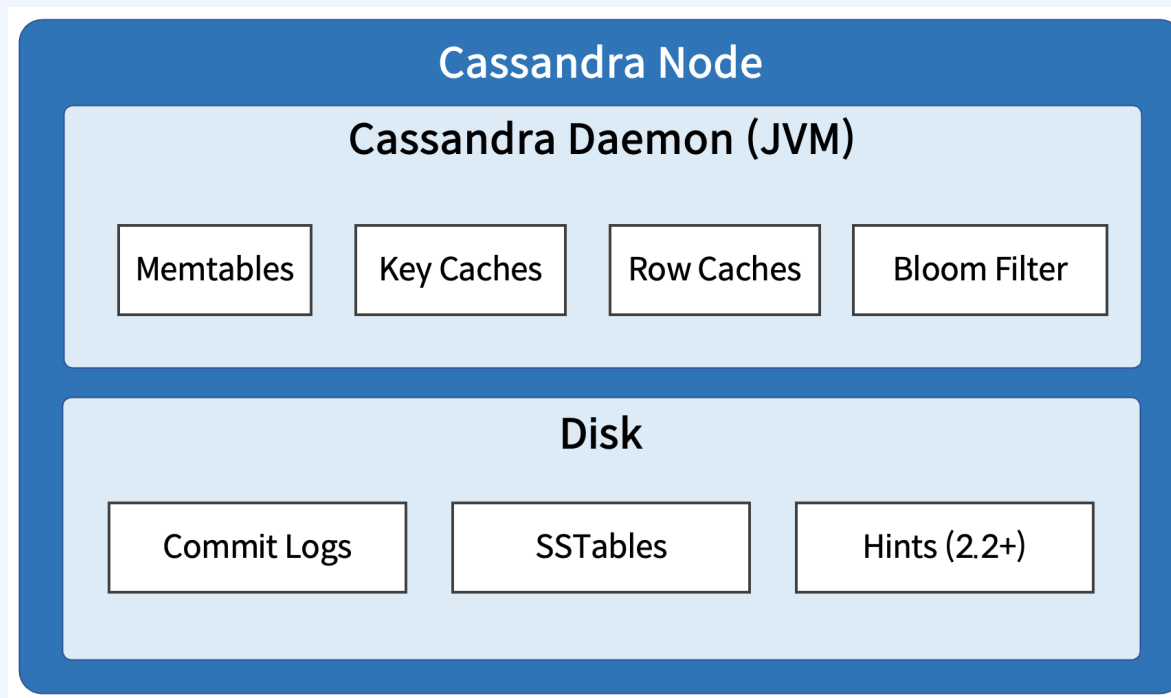
1. In-Memory (JVM)

- Memtable
- Caches(Key, Row)
- Bloom Filter

2. Disk

- Commit log
- SSTable
- Hints (2.2+)

- 이러한 구조의 기본 디자인은
LSM-Tree에서 기원.
(Log-Structured Merge-Tree)



2. Memtable (In-Memory)

- In-memory hash table.
- 1개의 table (Column Family)는 여러 개의 Memtable을 가질 수 있음.
- 데이터 구조)
 - {
 k1: [{c1, v1}, {c2, v2}, {c3, v3}],
 k2: [{c1, v4}, {c4, v5}]
}
 - k : Row Key(Primary Key) c : column name v : column value
- Cassandra 초기 버전에서는 JVM heap 에 저장되었는데, 2.1 ver 부터는 일부를 native memory에 저장할 수 있도록 설정 가능. (장점 : GC 의 영향을 덜 받음)
- Memtable의 크기가 특정 threshold를 넘으면, Disk의 SSTable이라는 자료 구조로 기록, 그 후 새로운 Memtable이 생성. (non-blocking 연산)

3. Cache (In-Memory)

- Cache의 목적 : 읽기 성능을 높이기 위함.
- Cassandra에서 Cache의 종류
 1. Key Cache (기본 값 : 사용)
 - Map (key : partition key, value : row index entries)
 - disk에 있는 SSTable에 빠르게 접근하기 위한 목적.
 - JVM heap에 저장.
 2. Row Cache(기본 값 : 사용 X)
 - index가 아닌 row 전체를 cache
 - 자주 접근하는 row에 대해서는 아주 빠르게 접근 가능, 대신 메모리 용량을 많이 차지.
 - Off-heap에 저장.
 3. Counter Cache (> v2.1, 기본 값 : 사용)
 - 자주 읽는 counter에 대한 lock 경합을 줄여, counter 성능을 개선.

node 재시작 시 빠르게 warm-up하기 위해 , 주기적으로 Cache 를 Disk에 저장

4. Bloom Filter (In-Memory)

- Cache와 비슷하게 읽기 성능을 높이기 위해 사용.
 - 특정 원소가 집합에 속하는지 여부를 검사하는데 사용되는 **확률적 자료 구조**.
 - bit-vector, hash function 사용
 - Bloom Filter에 의해 어떤 원소가 집합에 속한다고 판단된 경우, 실제로는 원소가 집합에 속하지 않는 긍정 오류(False-positive)가 발생하는 것이 가능함.
 - 반대로 원소가 집합에 속하지 않는 것으로 판단되었는데 실제로는 원소가 집합에 속하는 부정 오류(False-negative)는 절대로 발생하지 않음.
- > 특정 Data가 Disk 상에 존재하는지 여부에 대해 Bloom Filter를 이용해 매우 빠르게 확인 가능.
- Key cache의 특수한 예
 - False Positive가 발생할 확률은 조정할 수 있음.
(bloom filter size 커질 수록 False Positive가 발생할 확률은 감소.)
 - 각각의 Bloom Filter는 1개의 SSTable에 대응됨.

5. Commit logs (Disk)

- node에서 쓰기 요청이 수행되면, 가장 먼저 transaction을 Commit Log에 기록.
- Cassandra의 durability (내구성)을 지원하는 장애 복구 메커니즘에 사용됨.
- 쓰기 요청이 Memtable까지 도달하지 못하는 경우에, Commit log에 기록되기 전까지는 node에서 쓰기가 성공한 것으로 간주하지 않음.
- Node가 어떠한 이유로 고장 나고 후에 복구되었을 때, Commit Log에 저장하였던 transaction을 재실행하여 데이터를 복구함.
 - 이 경우에만 Commit Log를 읽고, client가 Commit Log를 읽을 일은 없음.
- Commit Log에 다 기록이 되면, 그 후에 Memtable에 데이터 쓰기 연산 수행.
(전체 쓰기 과정의 순서는 다음 강의에서 다룰 예정)

6. SSTable (Disk + In-Memory)

- Sorted String Table의 약자, Bigtable 논문에서 소개된 개념.
- Immutable 자료 구조, application 혹은 client에 의해 변경되지 않음.
- SSTable 사용 시, 모든 쓰기 연산은 append-only.
 - RDBMS의 경우, 데이터를 쓰기 전에 어느 위치에 써야할지 탐색해야 함.
 - 반면 Cassandra는 탐색 없이 새로 쓰기만 함. -> Cassandra가 쓰기 성능이 좋은 이유!

SSTable은 크게 네 가지로 구성.

1. Index file (Disk) (6.1)
 2. Summary file (in-memory) (6.2)
 3. Data File (Disk) (6.3)
 4. Bloom Filter (in-memory)
- 주기적으로 Compaction을 수행하여 SSTable을 병합, Compaction ?

6.1 SSTable - Index file (Disk)

- partition key와, partition key의 대응되는 DataFile의 위치에 대한 정보를 저장하는 파일.
- 1개의 SSTable 당 1개의 Index File 존재.
- Entry는 partition key의 token 값에 따라 정렬
- token ? : 각 partition을 식별하기 위한 64-bit 정수 ID ($-2^{63} \sim 2^{63} - 1$)
 - hash function을 이용하여 partition key에 대한 token을 계산하고, 그 token 값에 따라 데이터는 특정 node에 할당됨.

partition key	location of DataFile
partition key 1	location of DataFile
partition key 13	location of DataFile
partition key 97	location of DataFile

6.2 SSTable - Summary file (In-memory)

- index file에 있는 partition key의 범위와, 그 범위에 해당하는 Index File의 위치 정보를 담고 있는 파일

Index File

partition key	location of DataFile
partition key 1	location of DataFile
partition key 13	location of DataFile
partition key 97	location of DataFile
...	...

Summary File

start : partition key 1 end : partition key 97	index location
start : partition key 100 end : partition key 256	index location
start : partition key 257 end : partition key 512	index location

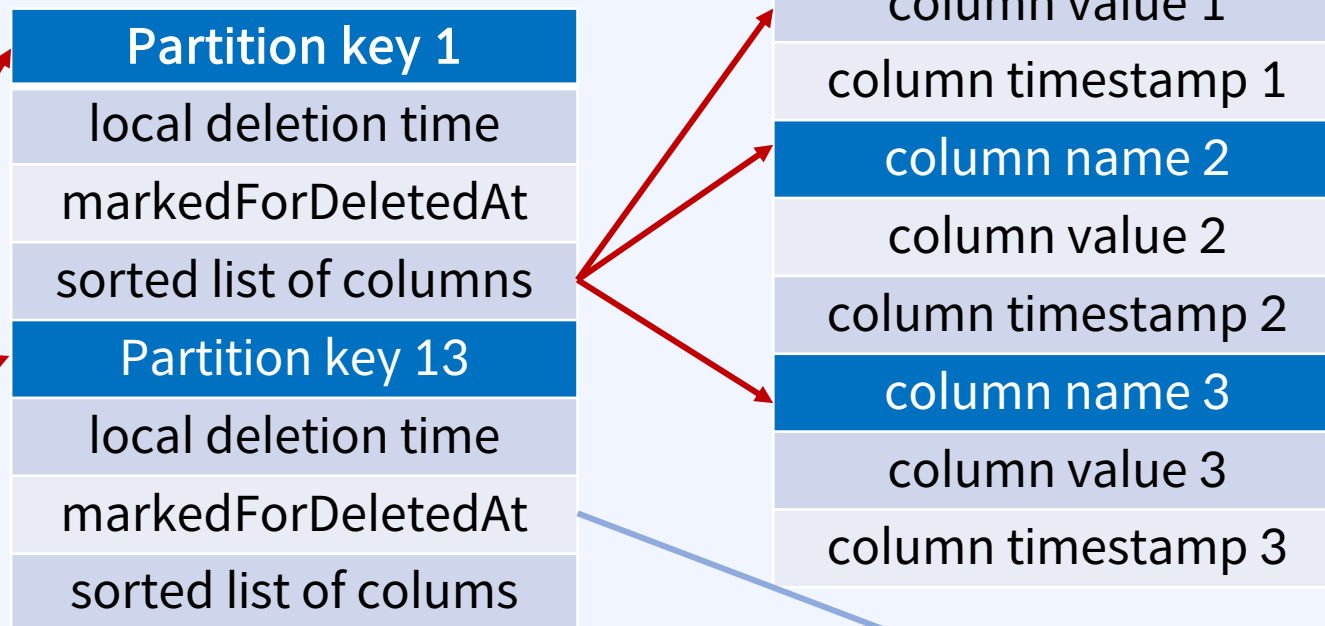
6.3 SSTable - Data file (Disk)

- 실제 Data가 저장되는 파일
- DataFile로부터 실제 데이터를 읽는 거는 크게 두 과정
 1. 해당 row를 찾기 위한 Disk 탐색.
 2. 그 row 내의 column들에 접근하기 위한 순차 탐색

Index File

partition key	location of DataFile
partition key 1	location of DataFile
partition key 13	location of DataFile
partition key 97	location of DataFile
...	...

Data File

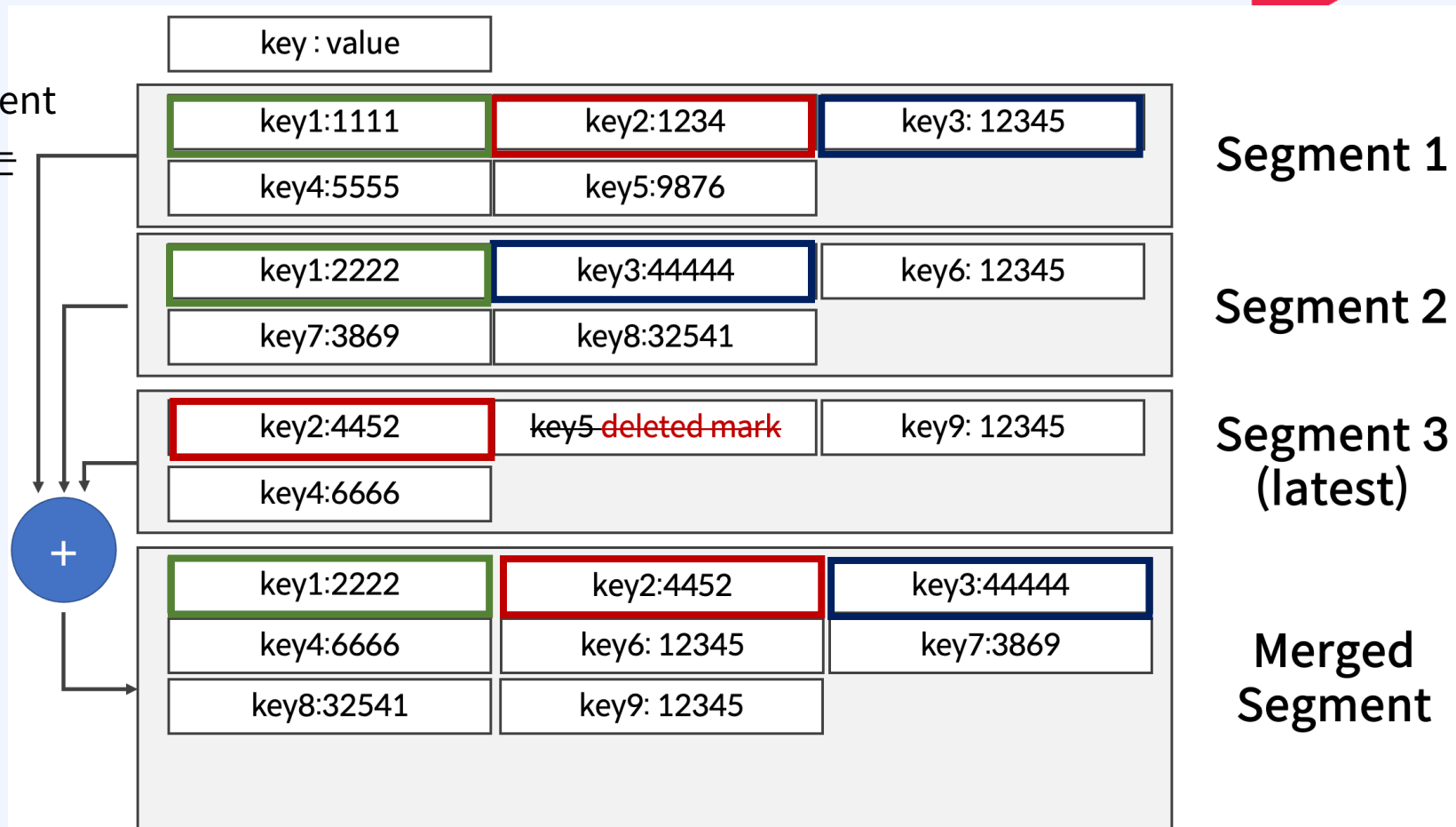


7. Compaction

- SSTable은 immutable하고, 쓰기 연산은 append-only이기 때문에 Cassandra는 높은 쓰기 속도를 확보 가능.
- 그러나 주기적으로, 만들어진 SSTable들을 압축하는 것은, 빠른 읽기 성능을 지원하고 오래된 데이터들을 정리하는 데 있어 중요.
- compaction 동안, 여러 개의 SSTable은 1개의 SSTable로 병합됨.
 - key들은 병합되고, column들은 병합되고, 사용되지 않는 값은 삭제되고 새 index가 생성됨.
- Compaction 과정에서 delete mark(=tombstone)가 있는 데이터를 삭제.
- Compaction 도중에는 일시적으로 Disk I/O가 많이 발생 가능.
 - Old SSTable들을 읽고, New SSTable을 쓰는 과정.
- non-blocking. (compaction 중 client는 Read, Write 연산이 가능)
- 여러 Compaction 전략을 선택 가능
 - Sized-tiered, Leveled, Time window

7. Compaction - ex)

- SSTable 병합 시, 여러 Segment에 같은 key가 존재하는 경우, 더 최신의 것을 기준으로 함.
- SSTable 병합 중에 실제로 데이터를 삭제.



8. Tombstone

- Cassandra에 delete query를 날리면, 실제로 데이터가 바로 삭제되지 않고, 대신 데이터가 삭제되었다는 것을 표시하는 Tombstone이라는 marker를 표시.
- Tombstone이 마크된 데이터는 query시에 결과로 나타나지 않음.
- gc_grace_seconds 설정을 조정해, Tombstone 마크된 데이터를 어느 기간 후에 삭제할지를 조정 가능
- 분산 환경에서 Tombstone을 사용하는 이유?
만약 Tombstone을 사용하지 않고, 바로 데이터를 삭제한다면?

8.1 Tombstone 사용 이유 예시

가정) Cassandra 클러스터의 3개의 Node에 존재하는 D1 데이터를 바로 삭제.

[D1], [D1], [D1]

-> 이 시점에 1개 Node에 문제가 생기면, 삭제 연산이 수행되지 않아 D1이 남을 수 있음.

[], [], [D1]

-> 이 Node가 복구되면, 아래처럼 Node간의 데이터가 싱크될 수 있음.

[D1], [D1], [D1] ➔ 분명 D1을 삭제했지만 재생성됨.

Tombstone 사용 시

[D1], [D1], [D1]

-> 위와 동일하게 1개 Node에 문제가 생겼을 때, 그 Node에서는 tombstone이 마킹 X

[D1, tombstone(D1)], [D1, tombstone(D1)], [D1]

이 Node가 복구 된 후에, D1이 삭제 대상이라는 것은 모든 Node에서 알고 있음.

[D1, tombstone(D1)], [D1, tombstone(D1)], [D1, tombstone(D1)]

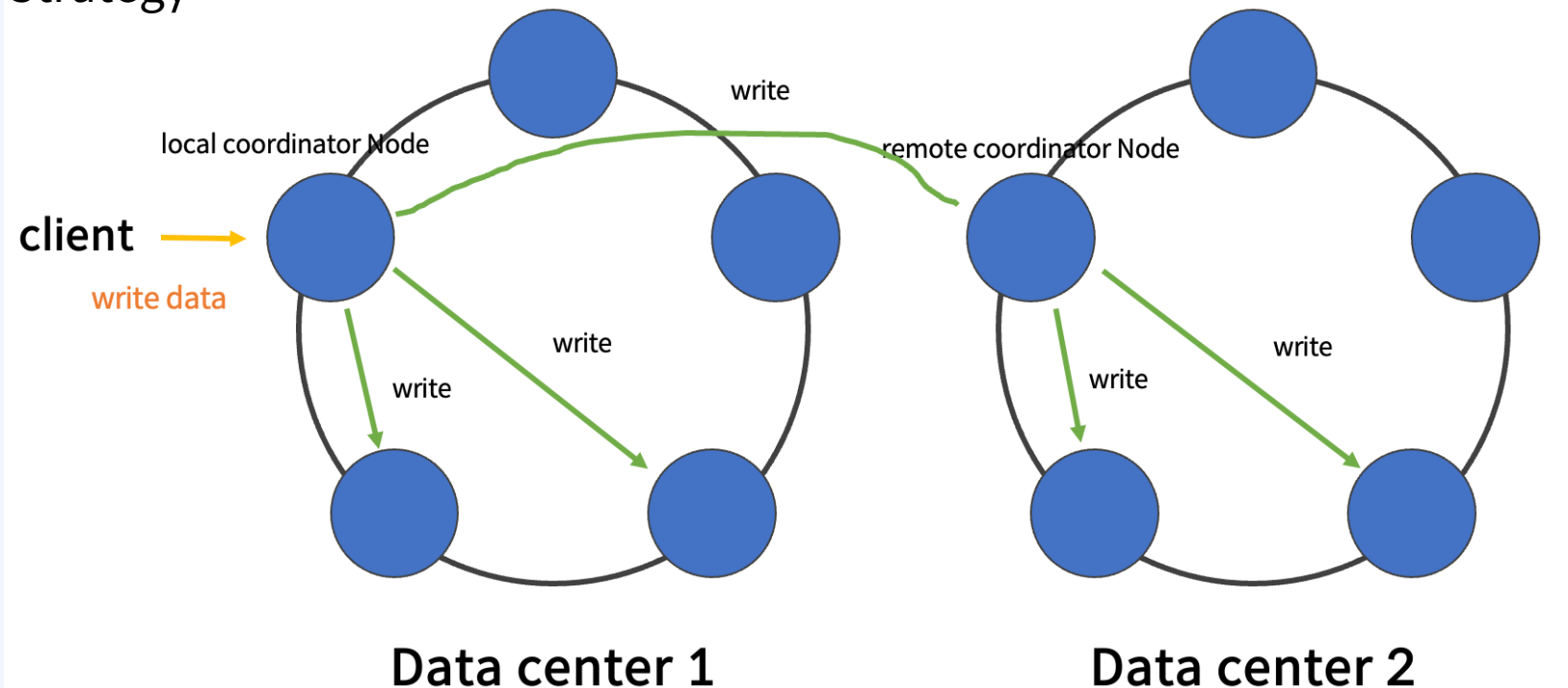
8. 데이터 쓰기 과정

1. 개요

- Cassandra cluster에서 쓰기 연산은 크게 두 과정.
 1. Cluster의 Node 간 쓰기 과정
 2. 한 Node 내에서의 쓰기 과정

2. Cluster의 Node 간 쓰기 과정

- data를 어느 Replica Node들에 써야할지 어떻게 결정?
 - Partitioner
 - Replication Strategy



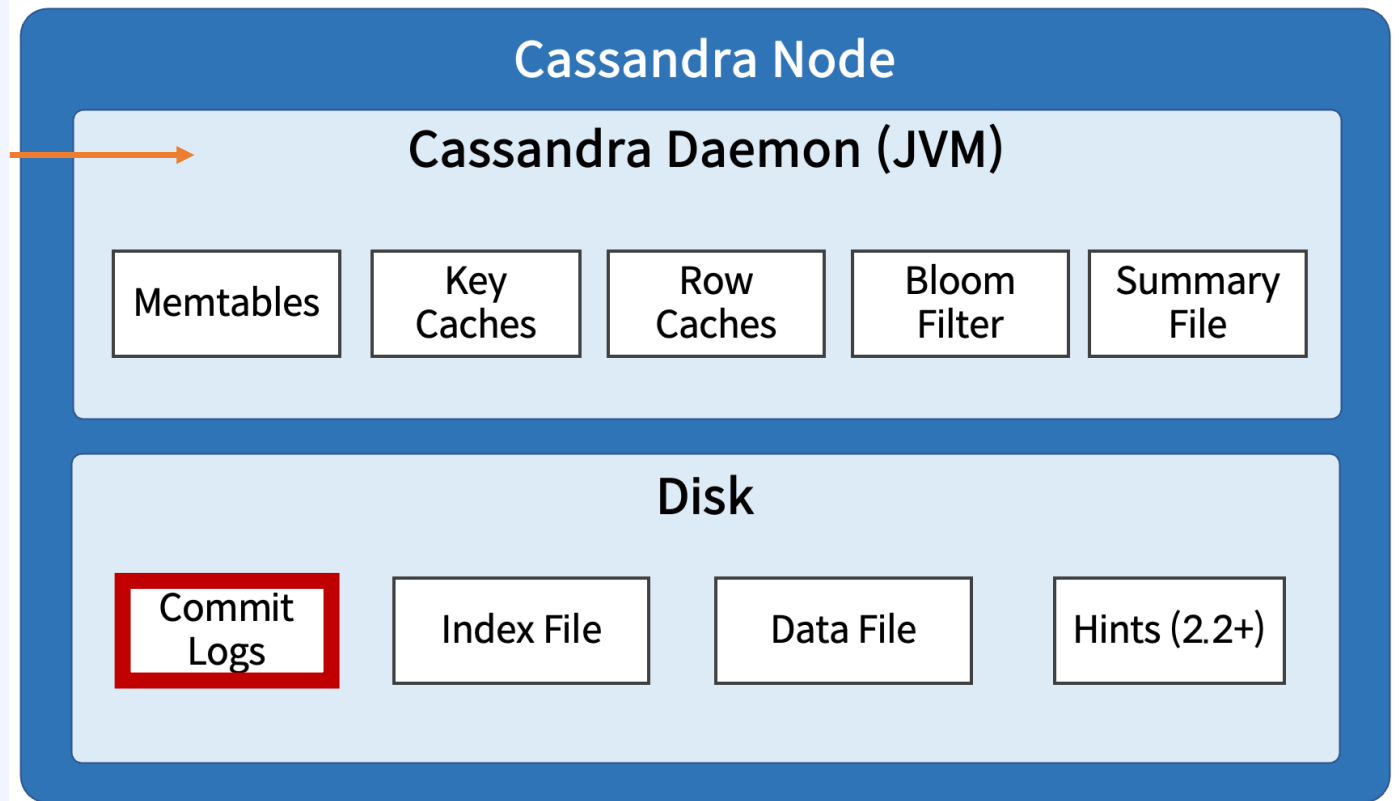
3. 한 Node 내에서의 쓰기 과정

1. commit log에 트랜잭션이 기록됨.

- 쓰기 요청에 timestamp가 만약 포함되어 있지 않으면,
cassandra에서
현재 시간으로 세팅

SSTable : Summary, Index, Data File, Bloom Filter

Write data 1

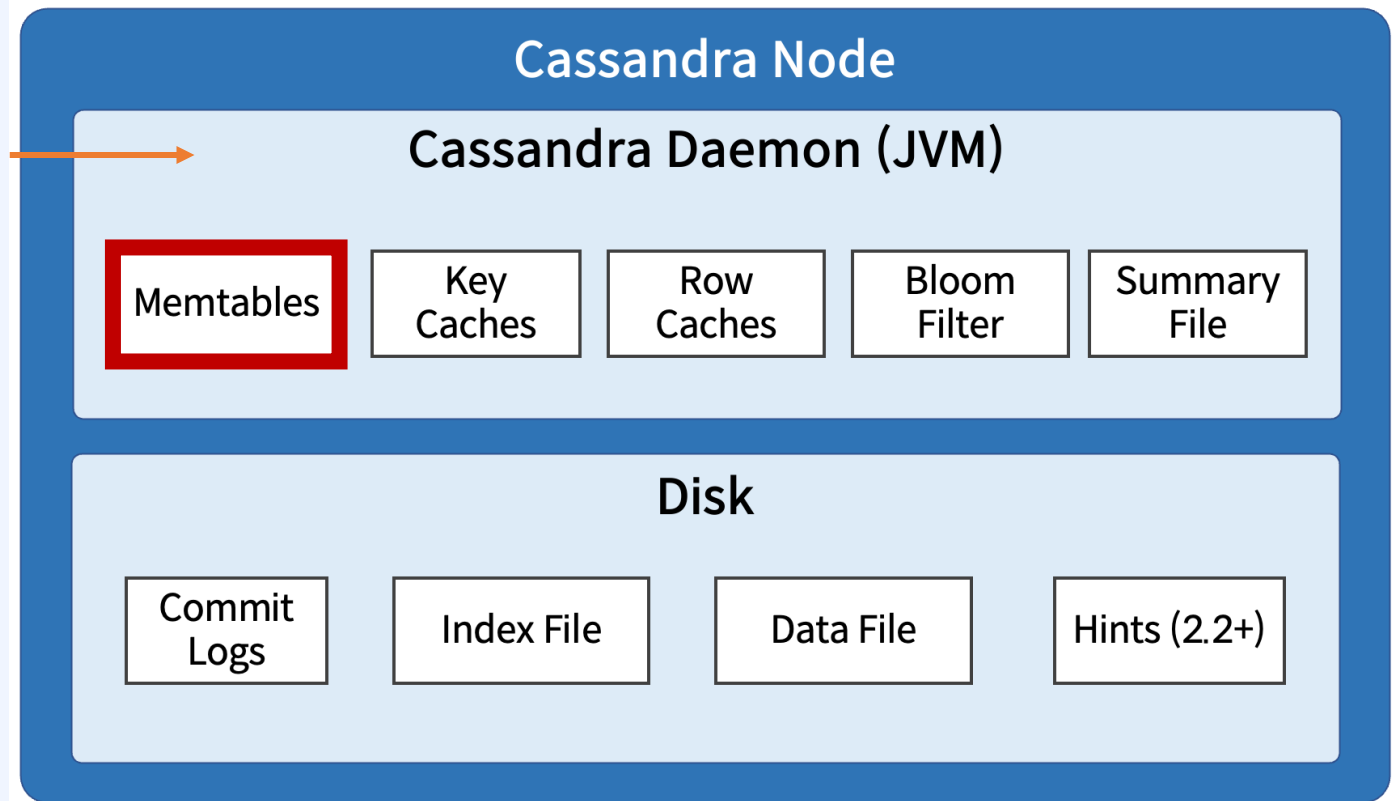


3. 한 Node 내에서의 쓰기 과정

2. Memtable에 Data 쓰기.

SSTable : Summary, Index, Data File, Bloom Filter

Write data 1

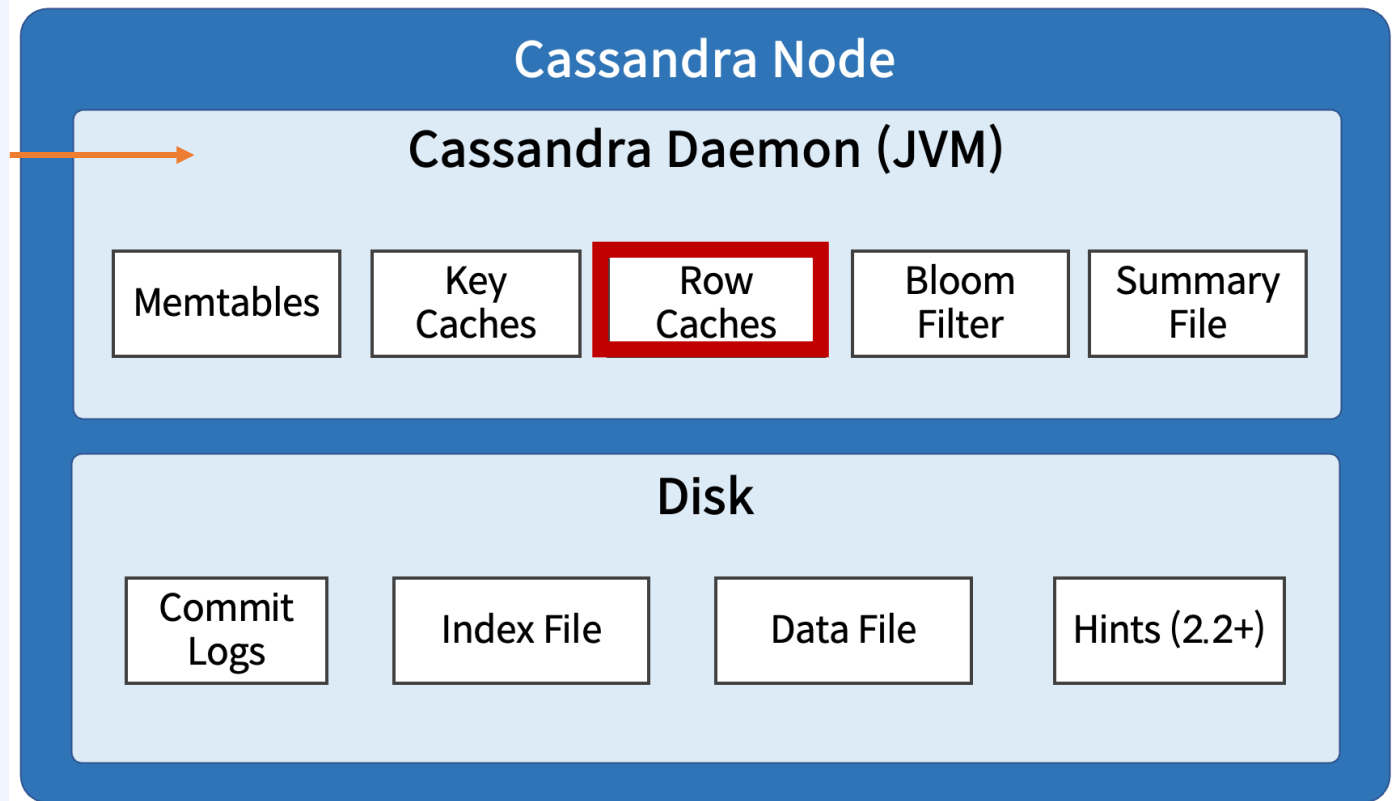


3. 한 Node 내에서의 쓰기 과정

3. 만약 Row Cache를 사용하도록 설정했다면, 그 data 에 해당하는 Row는 Row cache에서 제거.

SSTable : Summary, Index, Data File, Bloom Filter

Write data 1

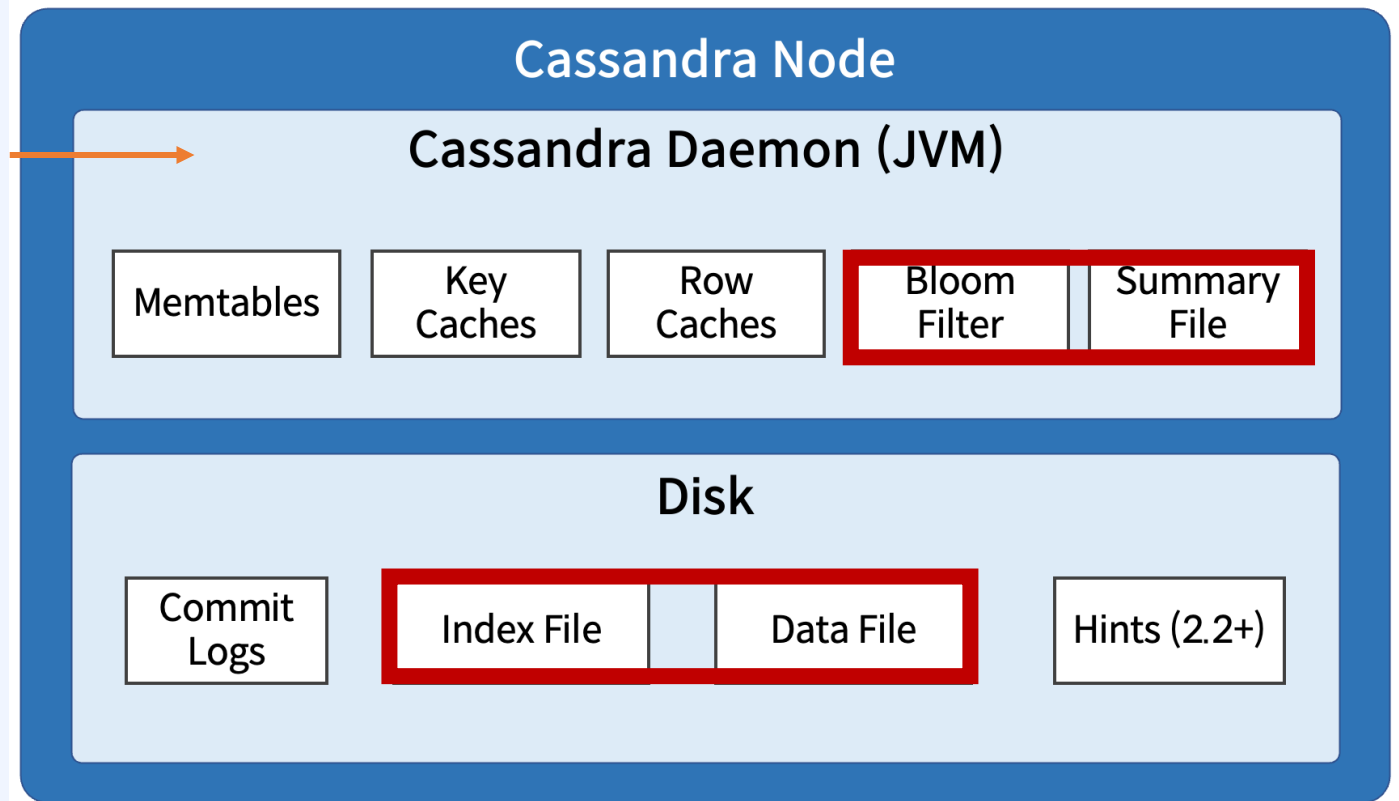


3. 한 Node 내에서의 쓰기 과정

4. Memtable의 크기가 특정 threshold보다 커지면 Disk의 SSTable에
Memtable 방출(flush) - Compaction

SSTable : Summary, Index, Data File, Bloom Filter

Write data 1



9. 데이터 읽기 과정

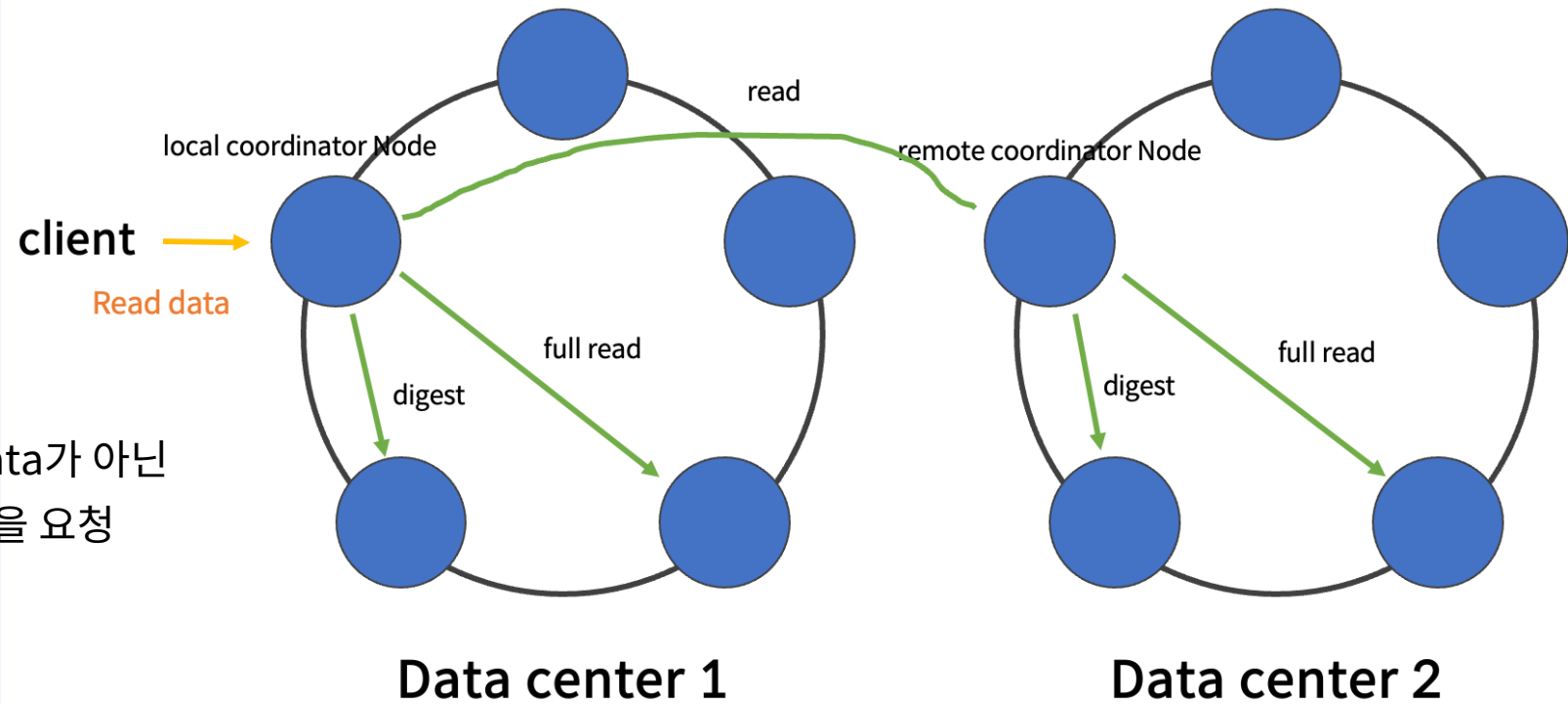
1. 개요

- Cassandra cluster에서 읽기 연산은 크게 두 과정.
 1. Cluster의 Node 간 읽기 과정
 2. 한 Node 내에서의 읽기 과정

2. Node 간 읽기 과정

- 1) read 요청 : coordinator Node에 도달
- 2) coordinator Node는 data를 포함하고 있는 Node들에 요청을 보내 data를 가져옴.

digest? -> 실제 data가 아닌
data의 hash 값만을 요청



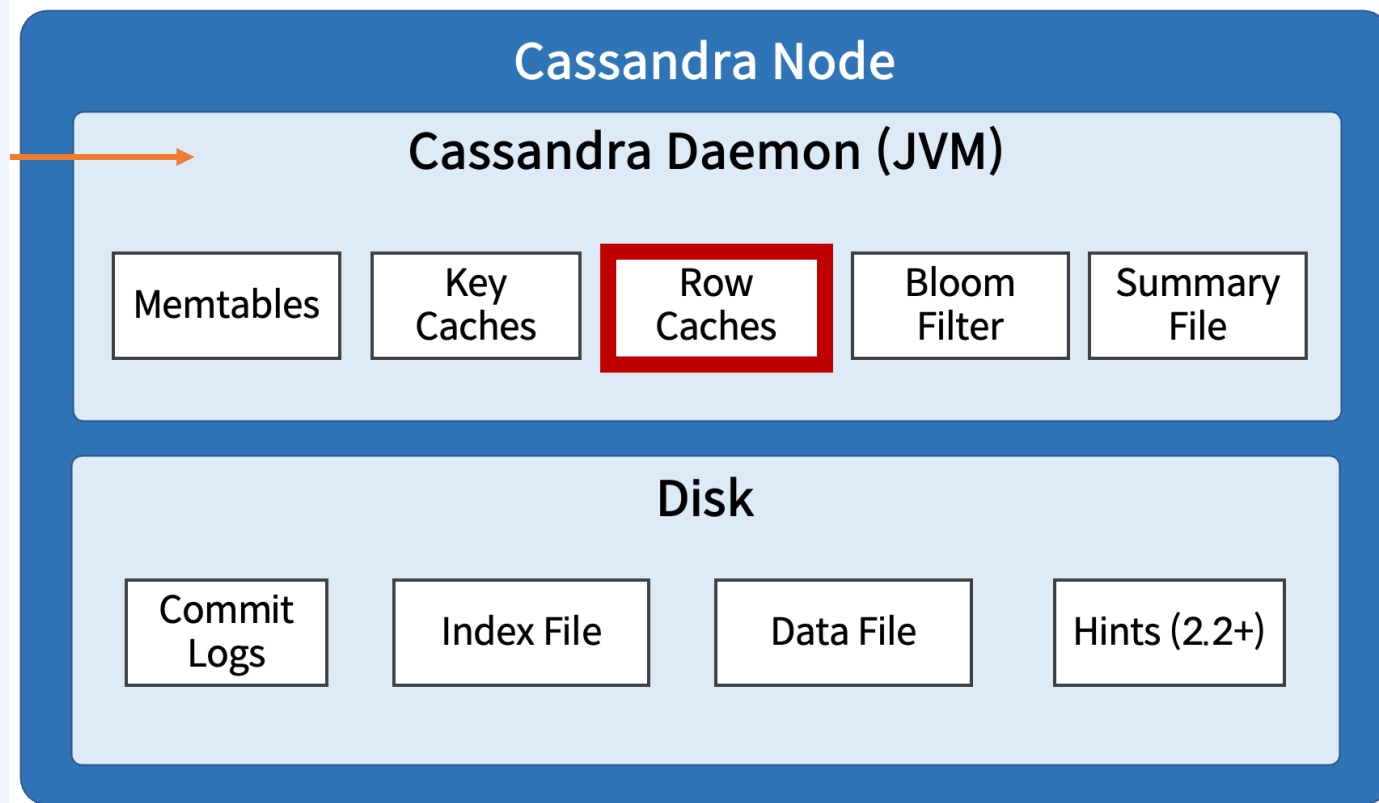
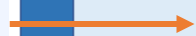
3. Node 내에서 데이터 읽기 과정

1) Row Cache에 data 1이 존재하는지 확인

-> 만약 존재한다면 Row Cache에 저장된 Row를 그대로 반환

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1



3. Node 내에서 데이터 읽기 과정

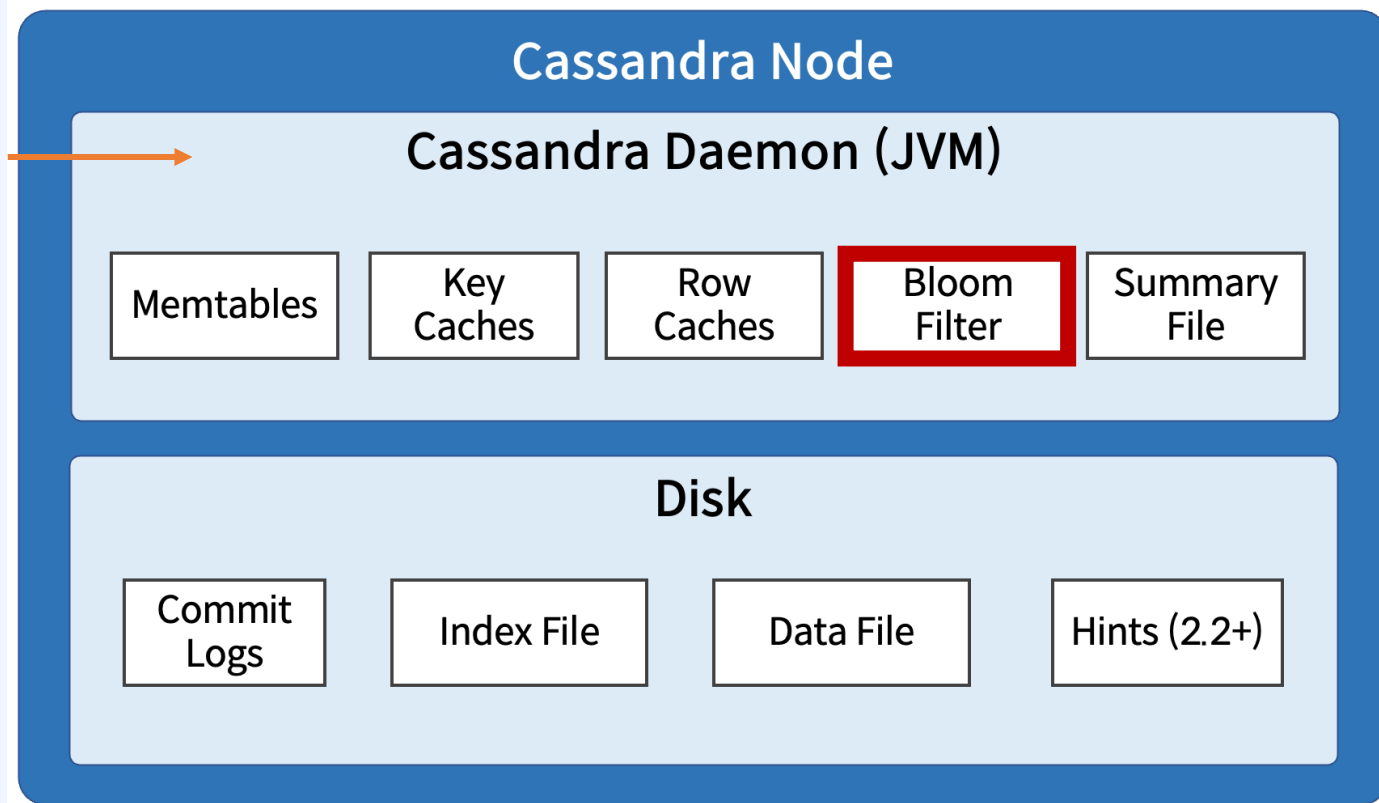
2) 만약 Row Cache에 data1이 없거나 Row Cache를 사용하지 않도록 설정

-> 각 SSTable의 모든 Bloom Filter들을 조회하여

data1이 SSTable에 존재하는지
여부만 확인

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1

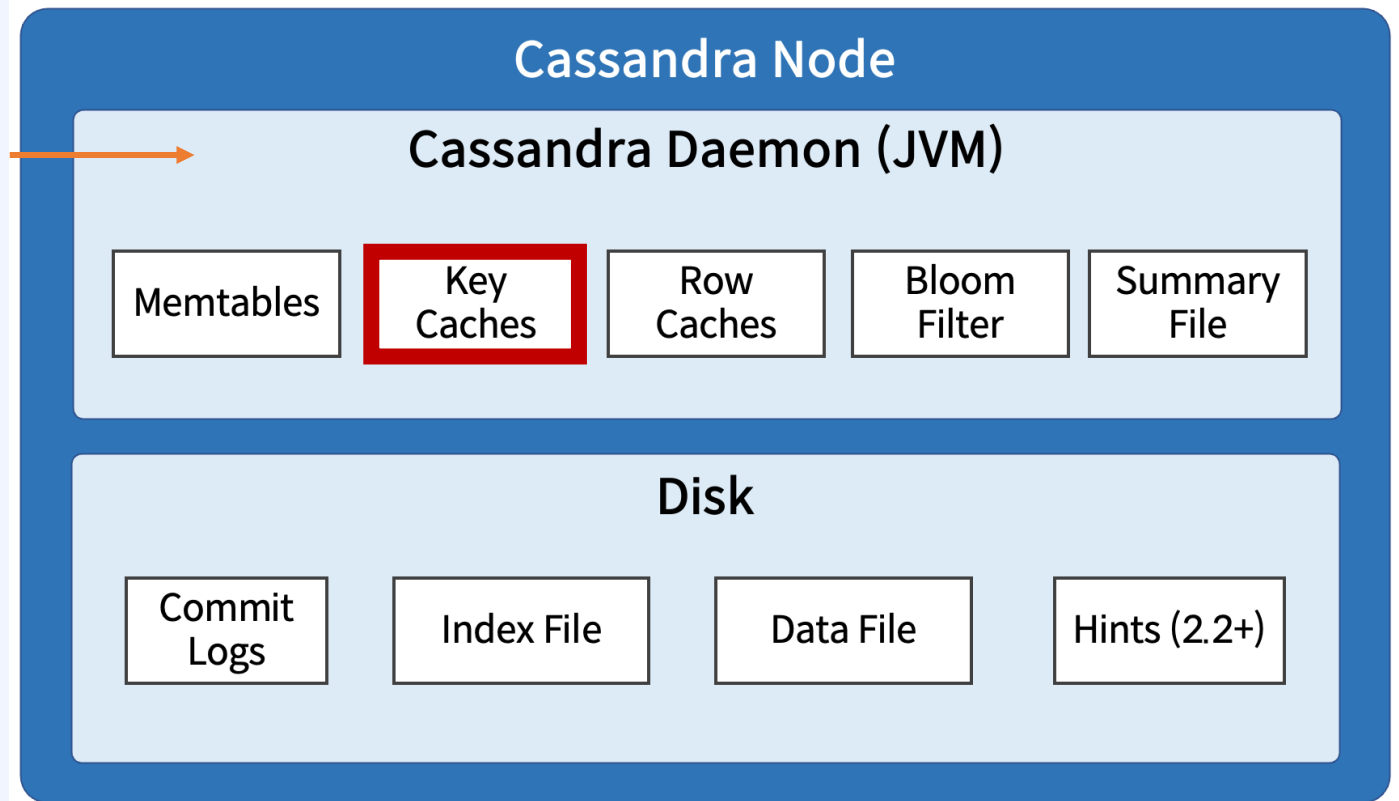


3. Node 내에서 데이터 읽기 과정

3) Bloom Filter 조회 결과가 positive (data1이 존재)이고, Key cache를 사용하도록 설정
-> Key Cache에서 Key가 존재하는지를 확인.

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1



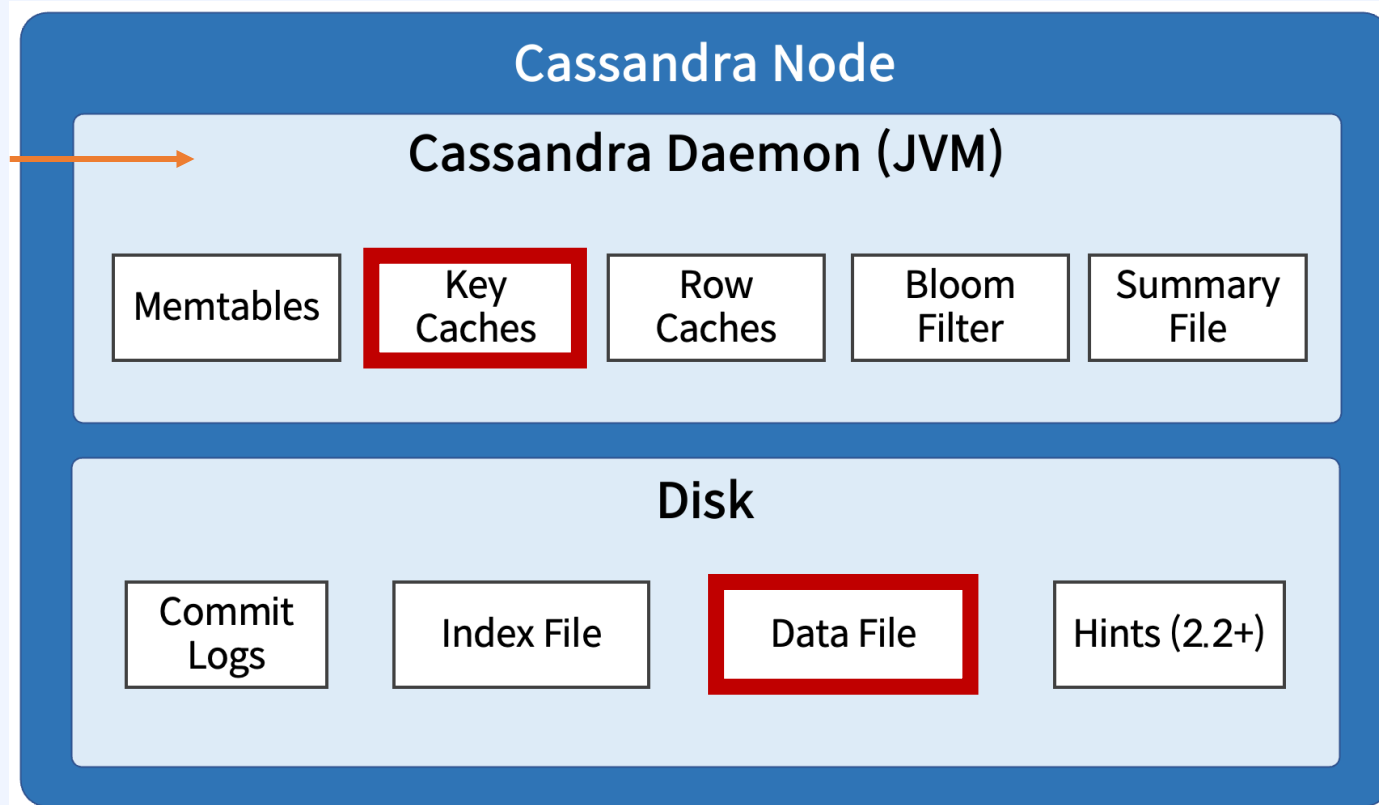
3. Node 내에서 데이터 읽기 과정

4) 만약 Key Cache에 Key entry가 존재.(1개의 Key가 여러 SSTable에 존재할 수 있음)

-> 매핑되는 Data File들로부터 data1을 가져옴.

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1



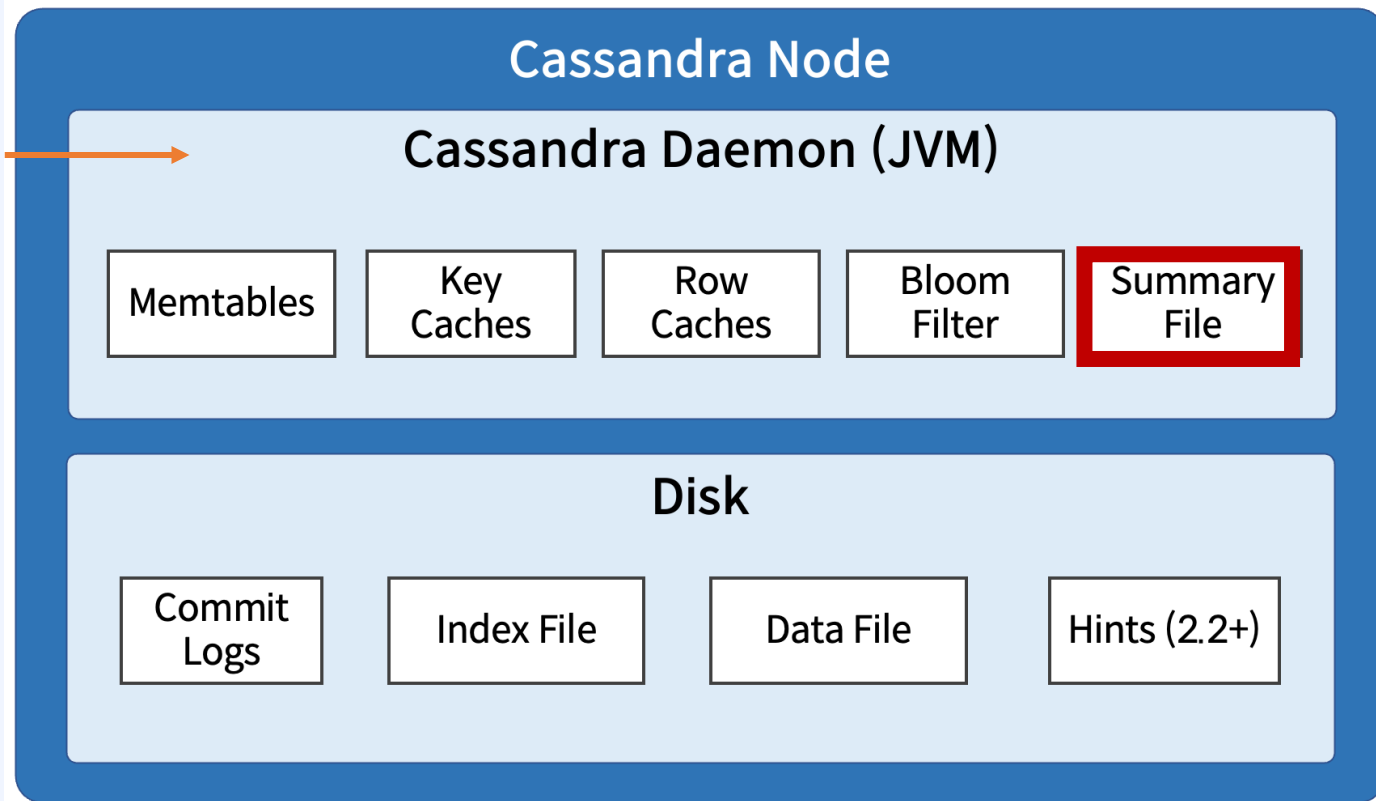
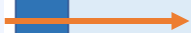
3. Node 내에서 데이터 읽기 과정

5) 만약 Key cache가 Key entry를 갖고 있지 않거나, Key cache를 사용하지 않도록 설정

-> Summary File로부터 Index File의
partition key의 offset 정보를
얻음.

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1



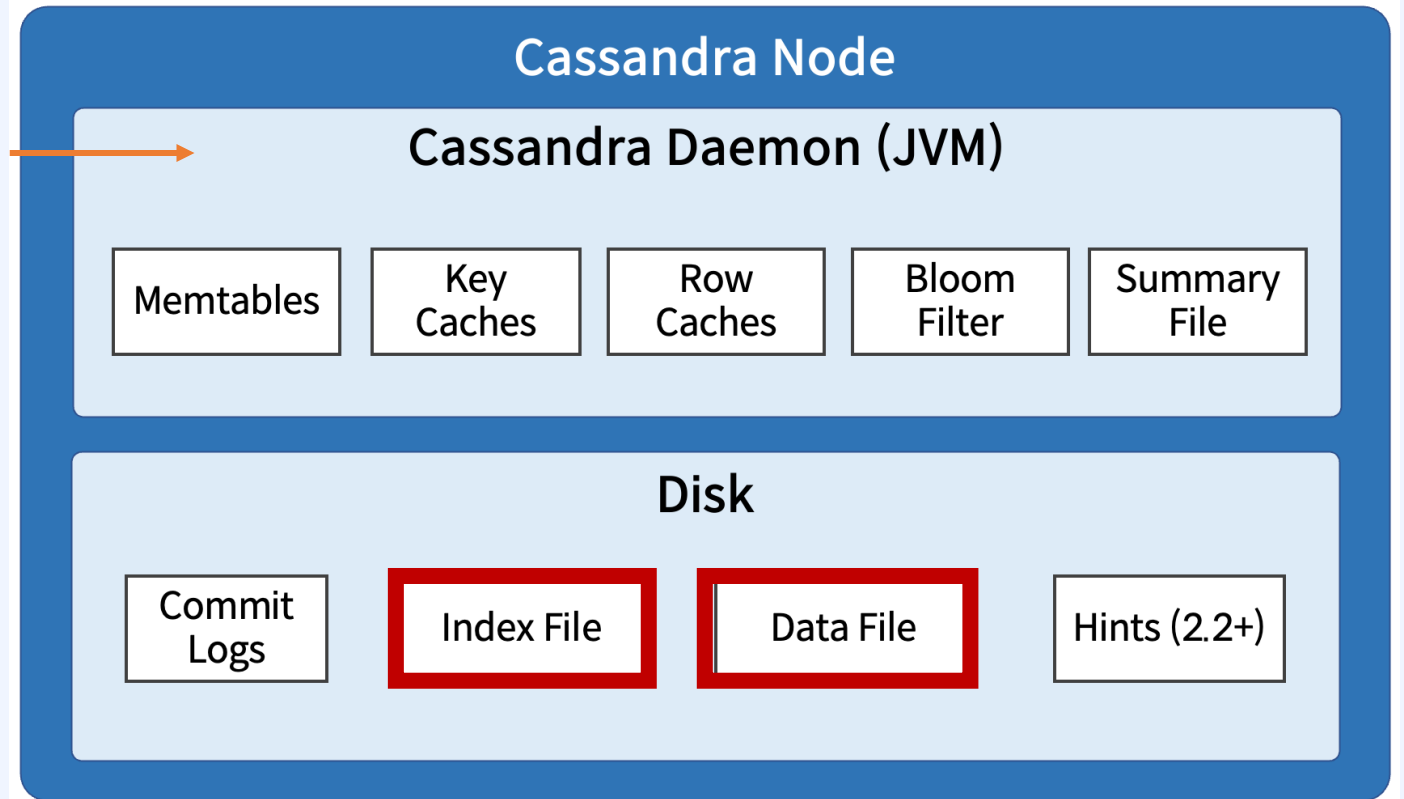
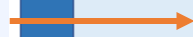
3. Node 내에서 데이터 읽기 과정

6) index File에서 offset 지점부터 순차 탐색하여, Data File내의 data의 위치 정보를 얻음.

-> Data File로부터 Data 가져옴

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1

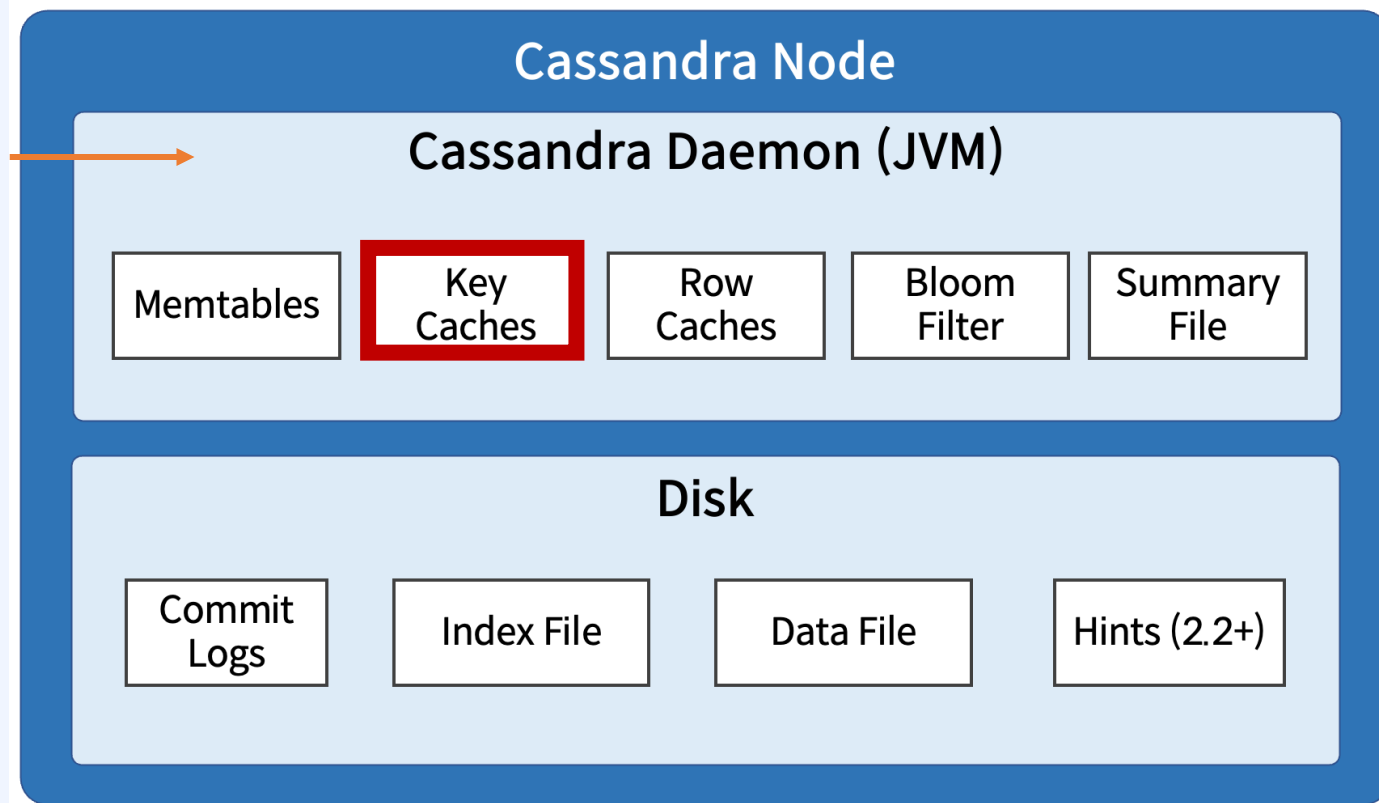
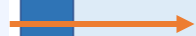


3. Node 내에서 데이터 읽기 과정

7) 4), 6) 에서 Data를 잘 읽었고, Key Cache를 사용하도록 설정이 되었다면,
Key Cache의 값을 갱신.

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1

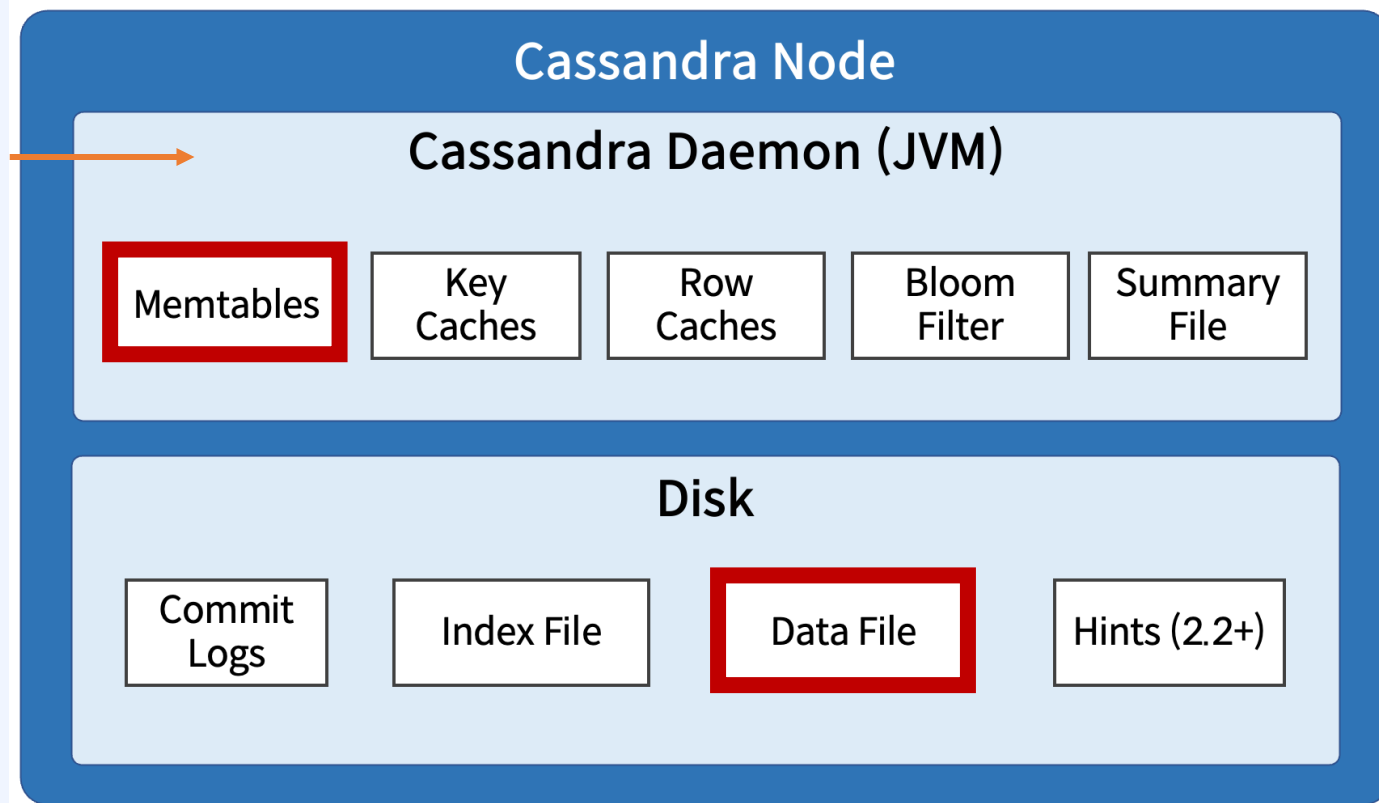
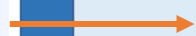


3. Node 내에서 데이터 읽기 과정

9) SSTable의 data와 Memtable의 data를 병합함. 병합 시 각 column에 대해 더 최신인 timestamp인 것을 기준으로 함.

SSTable : Summary, Index, Data File, Bloom Filter

Read data 1

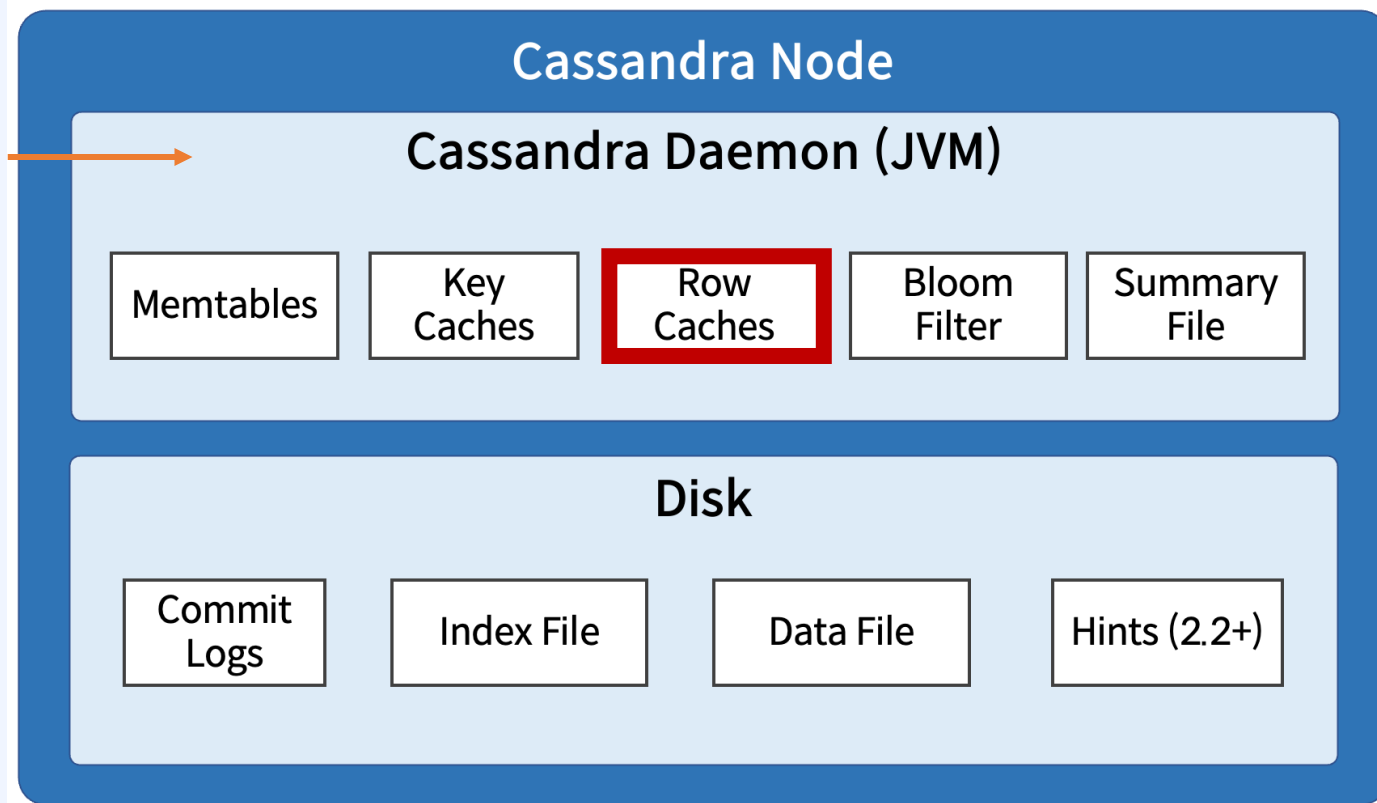
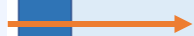


3. Node 내에서 데이터 읽기 과정

10) 만약 Row Cache를 사용하도록 설정되었다면, Row Cache의 data 갱신

SSTable : Summary, Index, Data File, Bloom Filter

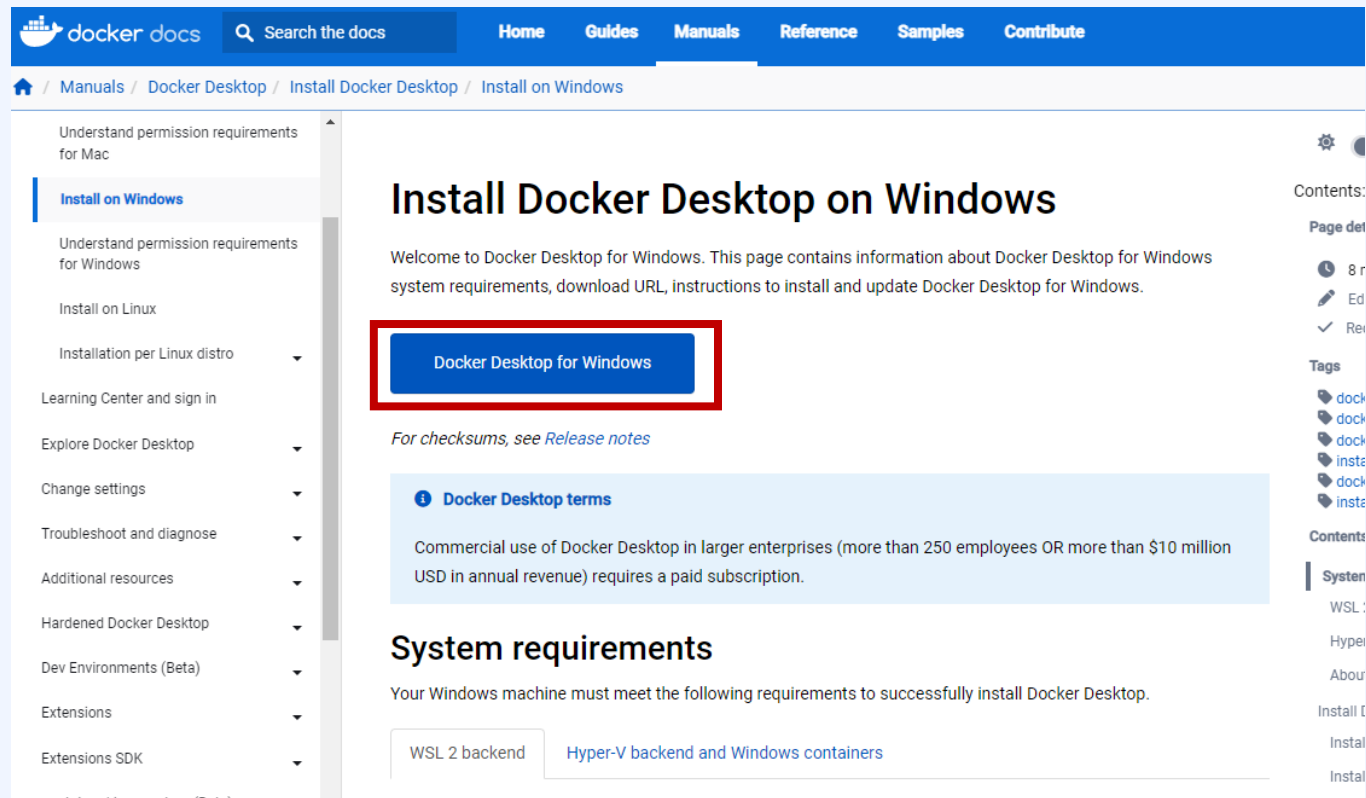
Read data 1



부록) Docker Desktop 설치 (Window)

1. Docker Desktop 설치

- <https://docs.docker.com/desktop/install/windows-install/> 에서 다운로드



1. Docker Desktop 설치

- Docker Desktop 다운로드 후 실행하면 WSL 관련 업데이트 문구 뜨는 경우 존재. -> 커널 업데이트 실행
- <https://learn.microsoft.com/ko-kr/windows/wsl/install-manual#step-4---download-the-linux-kernel-update-package>

4단계 - Linux 커널 업데이트 패키지 다운로드

1. 최신 패키지를 다운로드합니다.

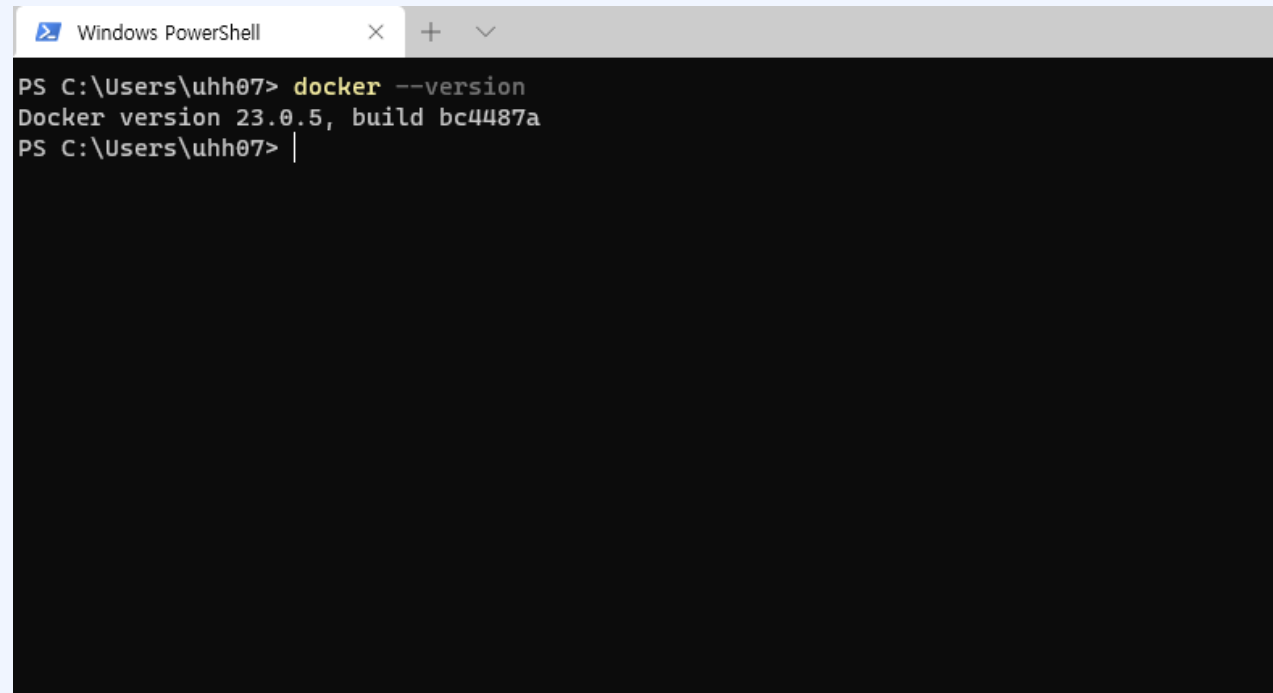
- x64 머신용 최신 WSL2 Linux 커널 업데이트 패키지

① 참고

ARM64 머신을 사용하는 경우 ARM64 패키지를 대신 다운로드하세요. 사용하고 있는 머신의 종류를 잘 모르는 경우 명령 프롬프트 또는 PowerShell을 열고 `systeminfo | find "System Type"` 을 입력합니다. 주의: 비 영어 Windows 버전에서는 "시스템 유형" 문자열을 변환하여 검색 텍스트를 수정해야 할 수 있습니다. find 명령에 대한 따옴표는 이스케이프해야 할 수도 있습니다. 예를 들어 독일어 `systeminfo | find '"Systemtyp"'` 입니다.

1. Docker Desktop 설치

- 터미널에서 \$ docker --version 입력해 잘 설치된 것을 확인.

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Windows PowerShell' with standard window controls. The terminal text shows the command 'docker --version' being executed, resulting in the output 'Docker version 23.0.5, build bc4487a'. The prompt 'PS C:\Users\uhh07>' is visible at the start and end of the command line.

```
PS C:\Users\uhh07> docker --version
Docker version 23.0.5, build bc4487a
PS C:\Users\uhh07> |
```