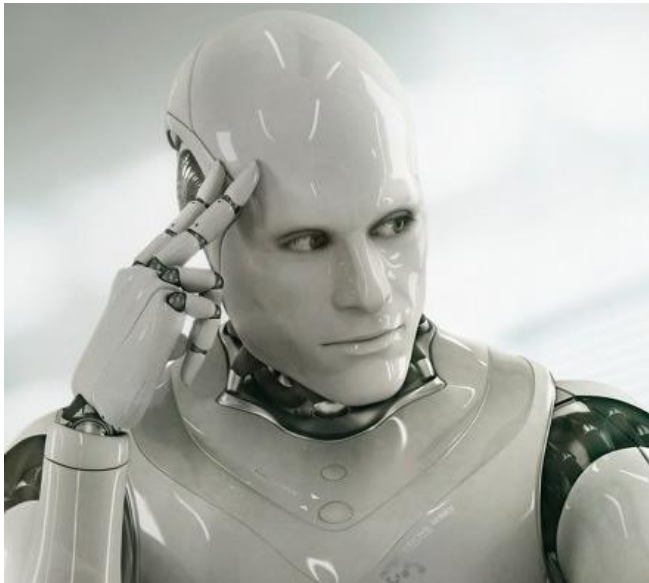


데이터 모델링

NoSQL 데이터 모델링

NoSQL은 어쩌다 등장했을까?

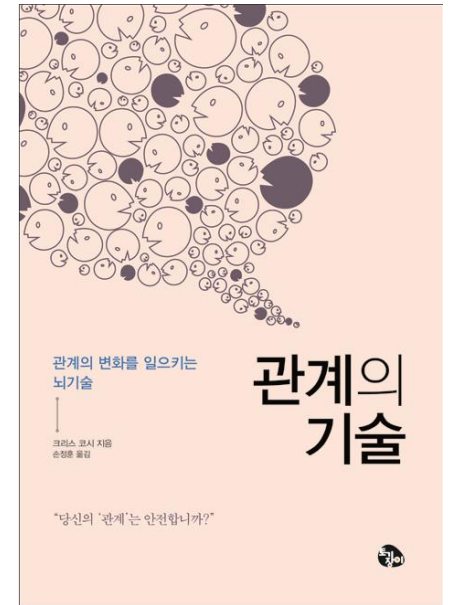
기존의 업무환경과 시스템 그리고 데이터



목적
기업업무의 자동화 / 효율화



데이터 소스
기업의 생산과 판매에서 나오는 데이터
데이터양은 한계가 있음



데이터 특징
복잡하면서 소량의 데이터
정형화된 데이터

NoSQL은 어쩌다 등장했을까?

변화된 업무환경, 시스템, 유저중심의 데이터



소셜 미디어
플랫폼 서비스



비정형 데이터



비교적 단순하면서 대량의 데이터 생성

NoSQL은 어쩌다 등장했을까?

구글과 아마존의 혁신

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vossball
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

General Terms

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform.

구글의 Bigtable 논문

아마존의 Dynamo 논문

NoSQL은 어쩌다 등장했을까?

구글과 아마존의 혁신

Bigtable: A Distributed Storage System for Structured Data

Fuji Ren, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallick



Bigtable

consistent low latency and high throughput

Bigtable is a distributed storage system for structured data. It provides a simple data model and a rich set of APIs for clients.

Bigtable is a distributed storage system for structured data. It provides a simple data model and a rich set of APIs for clients. Despite its size, Bigtable is able to handle a large number of servers, including those used by Google. This paper describes the design and implementation of Bigtable.

can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure.

구글의 Bigtable

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels



Amazon DynamoDB

AB
Reliability
face the
consistency
platform
is in
service
around
control
of the
software

This
high
core
achieving
under
versatility
that

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;

General Terms

performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform.

ting
of a
ged.
vice
this
gies
able
are
eing
for
and
able

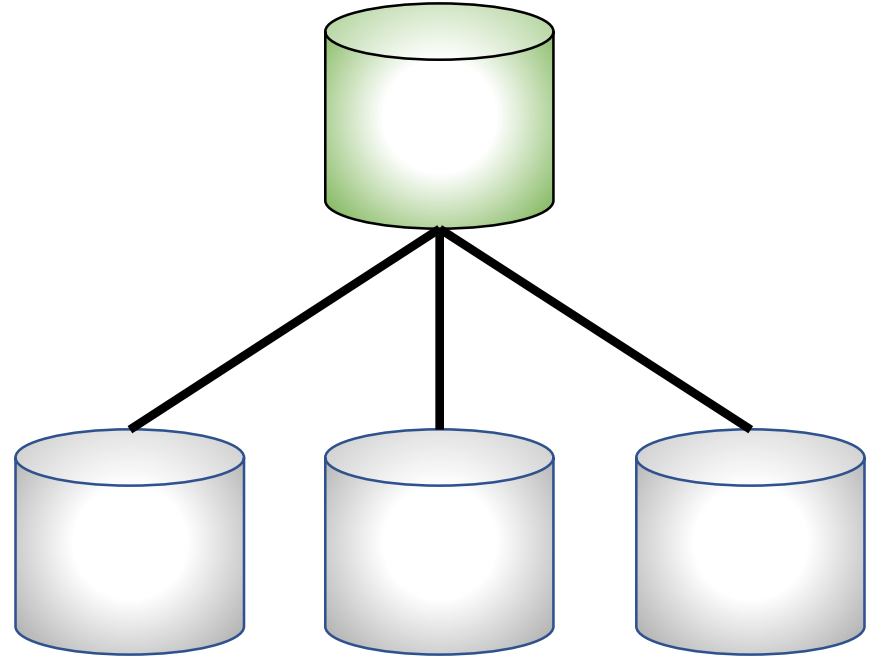
is of
ys a
ents
ware
lure
/ or

아마존의 DynamoDB

NoSQL은 어쩌다 등장했을까?

이제는 새로운 데이터 저장 기술이 필요하다

Not
Only
NoSQL
Relational



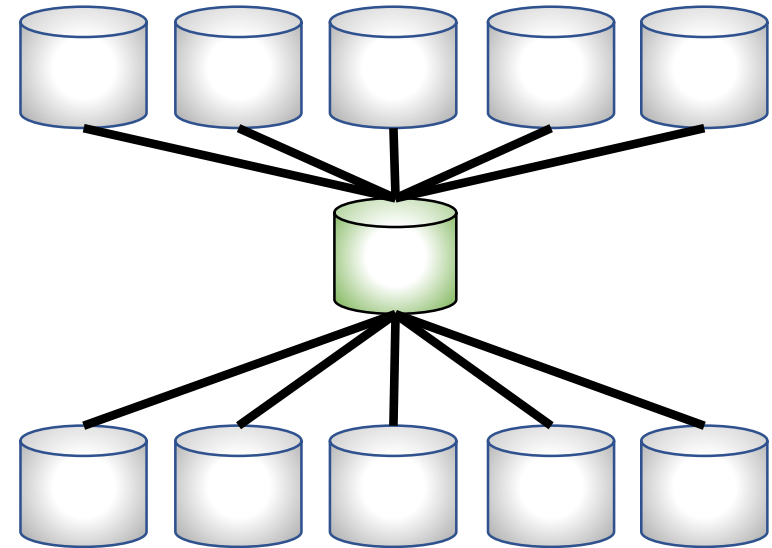
NoSQL 특징



NoSQL 특징



페타바이트 급
(테라바이트 천배)
대용량 처리 가능



수십대 분산
확장성
무종단

NoSQL 특징

컬럼은 어떤 이름이든 어떤 타입이든 허용
각기 다른 컬럼이름과 타입 사용 가능

키는
동일한
타입

"유저1" : "유저1은 이름이 Jane이며 2019년 10월에 가입했음"

"유저2" : {"name" : "john", friends : [Jane, jeff]}

"유저3" : ["jeff", 2018-12-01, "jeff@mail.com"]

스키마가 유연하다

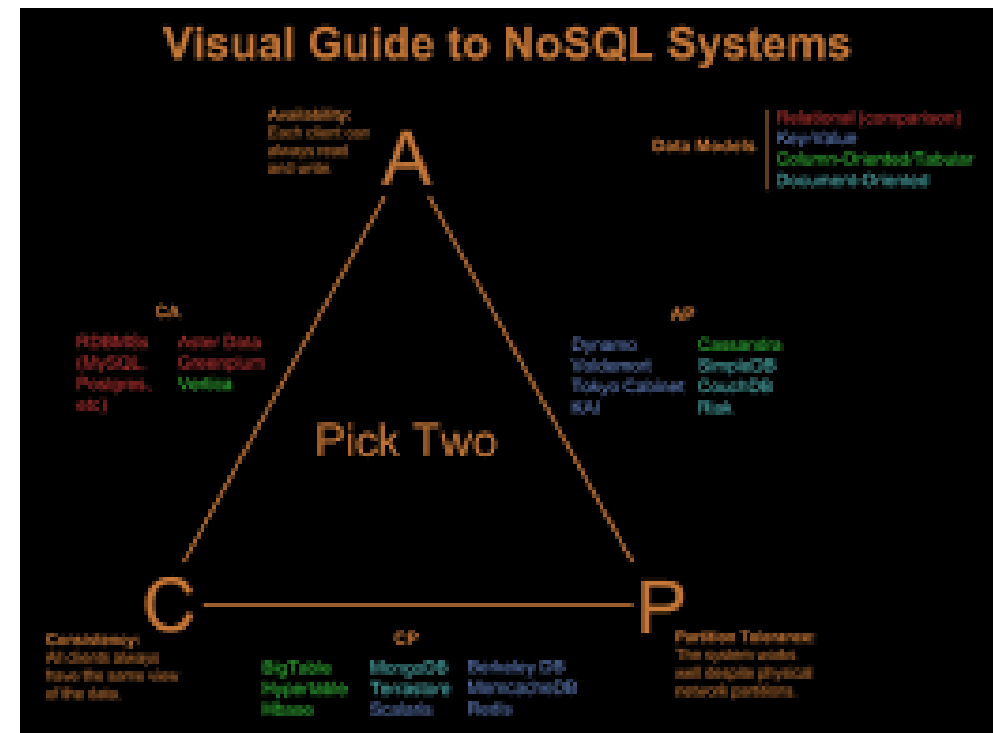
NoSQL 특징

- 분산형 구조이기 때문에 분산시스템의 특징인 CAP 이론을 따름

CAP 이론

분산 컴퓨팅 환경은 Consistency(일관성), Availability(가용 성), Partitioning(부분결함용인) 3가지 특징 을 가지고 있으며, 이중 2가지만 만족할 수 있다는 이론임

- Consistency : 분산된 노드 중 어느 노드로 접근하더라도 데이터 값이 같아야 함.
- Availability : 클러스터링된 노드 중 하나 이상의 노드가 FAIL이 되더라도, 정상적으로 요청을 처리할 수 있는 기능을 제공
- Partition Tolerance : 클러스터링 노드 간 에 통신하는 네트워크가 장애를 겪더라도 정상 적으로 서비스를 수행할 수 있는 기능

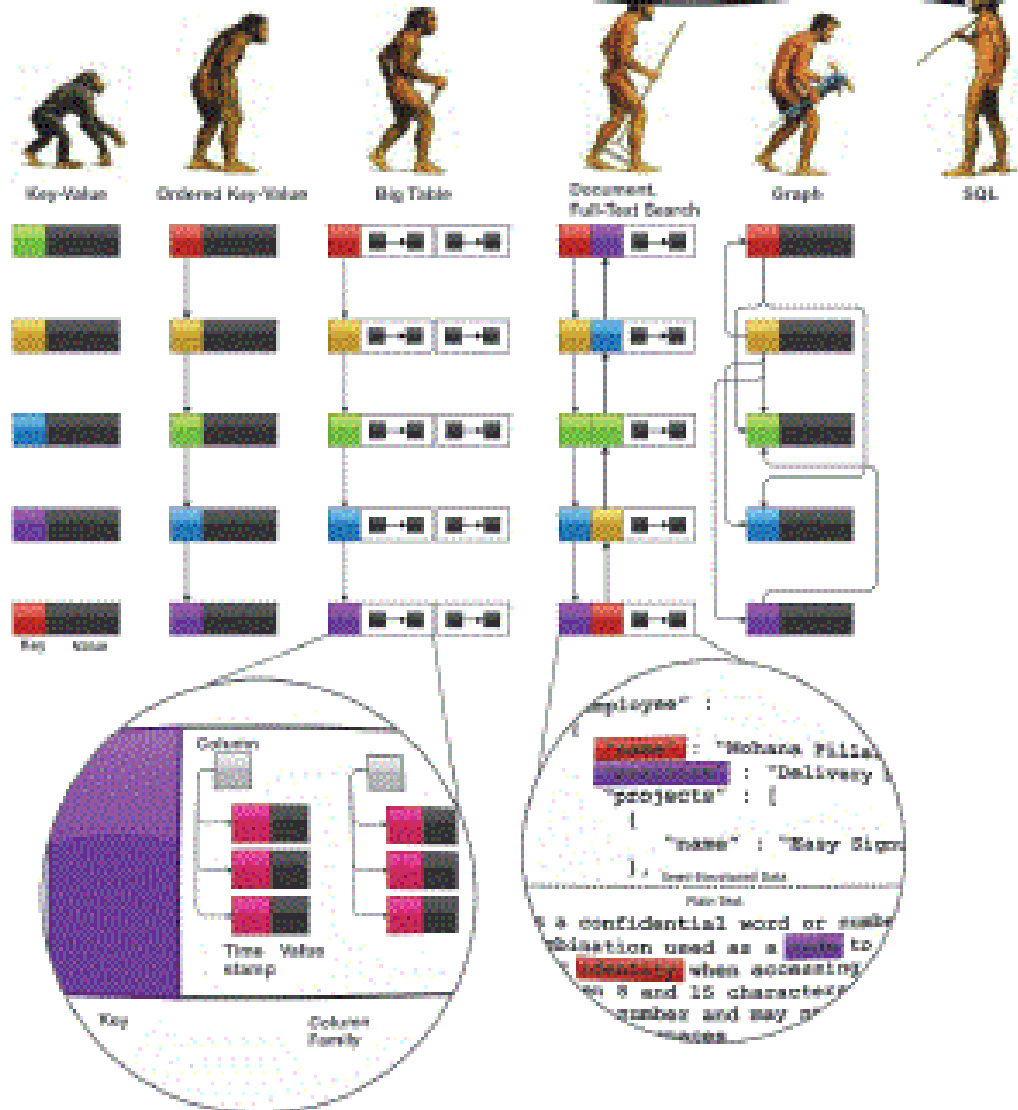


NoSQL data models

- NoSQL model의 진화

- Key-Value stores : Coherence, redis, Kyoto cabinet
- BigTable-style databases : Hbase, Cassandra
- Document databases : mongoDB, CouchDB
- Full Text Search Engine : elastic search
- Graph databases : neo4j, FlockDB

그만 따라해!



기존 SQL과 relational model은

- 데이터 사용자는 aggregated(하나로 엮여진) 정보에 관심, 그 부분을 제공하는데 SQL이 집중을 하고 있다.
- 트랜잭션 보증, 스키마, 참조 무결성 등에 집중

NoSQL : new evolution of data model

Key-value store
매우 간단하지만 강력한 모델 다양한 기법 적용 가능

range key 처리가 어려움

Ordered Key-value store
range key 효율적 처리, 데이터 조합 성능 상향

data modeling 어려움

data modeling 어려움

BigTable-style
value modeling 가능
map-of-maps-of-maps, namely, column family
columns, timestamped version

GraphDB
비즈니스 엔티티 투명하게 제공
하나의 모델에 하나의 map 또는 document
계층적인 모델링 기법

map of maps위주의 구조

key를 이용한 검색만 가능

DocumentDB
임의의 복잡도를 가진 스키마,
인덱스 가질 수 있음

full text search engine
value로 인덱스
유연한 스키마

NoSQL data models – key/value store

- 대부분 NoSQL 시스템은 key/value model을 지원
- 모든 key는 unique하며 하나의 value를 가지고 있음
- put(key,value), value := get(key) 형태 API 사용
- 마치 programming language에서 제공하는 map형태

key1	value1
key1	value2



NoSQL data models – column family

- BigTable이 지원하는 data model
- 하나의 key와 (column:value)를 페어하여 여러 개의 필드를 가짐
- RDB 테이블과 비슷한 형태

key1	column1	column2	column3
	value1	value2	value3
key1	column1	column2	column3
	value1	value2	value3
⋮			

NoSQL data models – ordered key/value store

- 데이터 쓰기 작업에서 내부적으로 key를 sorting하여 저장
- 쓰기작업은 좀 더 느리지만 range read작업은 효율적
- Hbase, Cassandra

key1	value1
key1	value2
⋮	


NoSQL data models

– document key/value store

- key-value store의 확장된 형태
- value는 구조화된 document type(ex. xml, json, yaml)
- 복잡한 계층구조 표현가능
- DBMS에 따라 추가기능 지원

key1	xml,json,yaml document
key1	xml,json,yaml document

•
•
•



```
"user" : {  
  "name" : "Albert",  
  "address" : { "state" : "California",  
                "nationality" : "USA"  
              },  
  "friendship" : ["John", "Geoff"]  
}
```

Relational modeling VS NoSQL modeling

모델링 원칙

- 전형적으로 가용한 데이터 구조에 기반
- Application의 특징적인 데이터 접근 패턴에 기반
- What answers do I have?
- What question do I have?
- 데이터 모델링 -> 쿼리설계
- 쿼리/성능 정의 -> 데이터 모델링

Relational

NoSQL

Relational modeling VS NoSQL modeling

모델링 기법

- 1) 도메인 분석
- 2) entity와 relationship 식별
- 3) table 추출
- 4) 쿼리 설계
- 5) 쿼리 구현

Relational

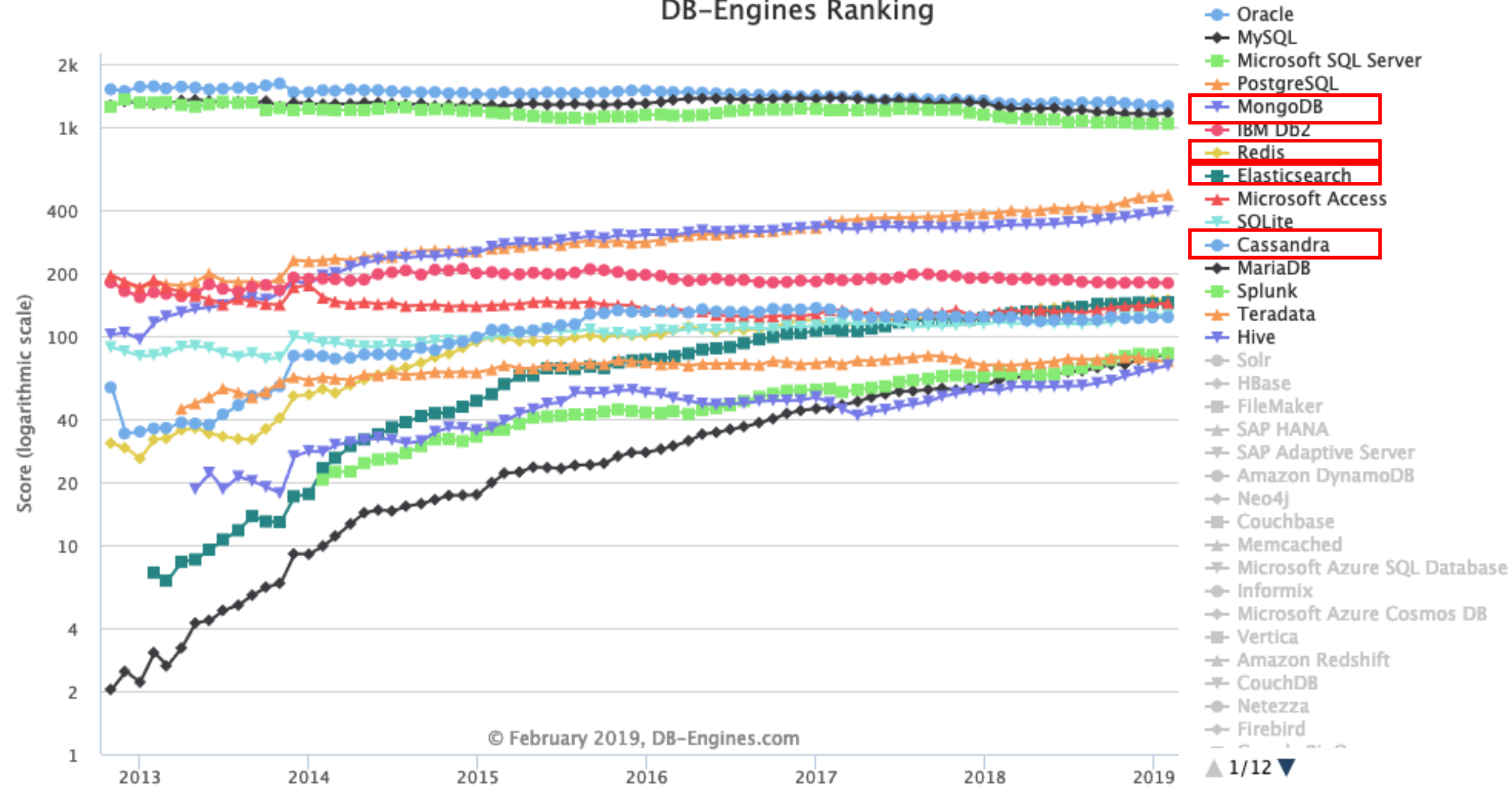
- 1) 도메인 분석
- 2) 쿼리결과 설계
- 3) 쿼리설계
- 4) 데이터 저장 모델 설계
- 5) 쿼리구현

NoSQL

NoSQL 설계시 고려사항

- 관계형 보다 더 깊은 데이터 구조와 접근 알고리즘에 대한 이해
- 쿼리가 실제 몇 개의 노드에 걸쳐서 수행되는지에 대한 이해
- DB와 application과 함께 인프라(네트워크,스토리지), 운영에 대한 디자인을 같이 해야함 만능잡부
- 대부분 별도의 인증절차가 없으므로 보안에 대한 고려를 해야 함 (방화벽이나 Reverse Proxy 등)

DB-Engines Ranking

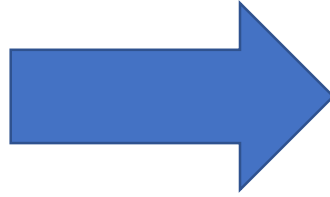


NoSQL 데이터 모델링 패턴

- Denormalization
- Aggregates
- Application side joins

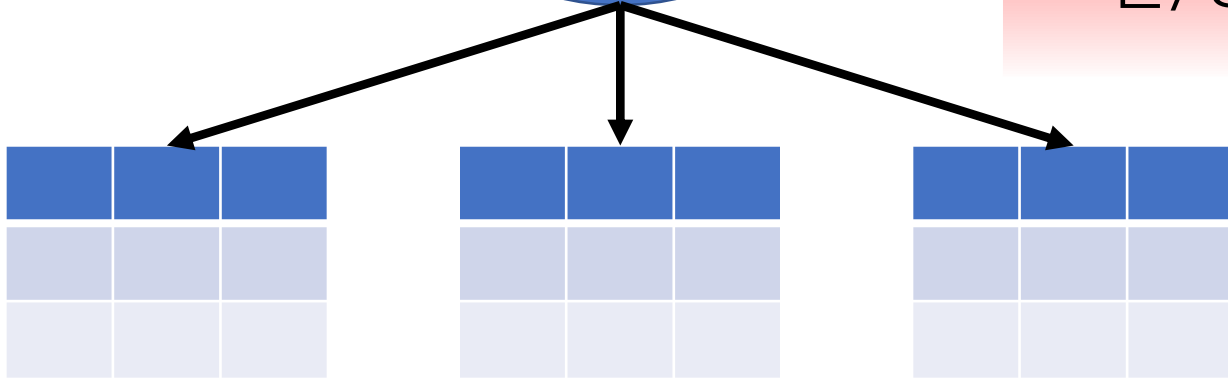
Denormalization

비정규화



데이터
중복 허용

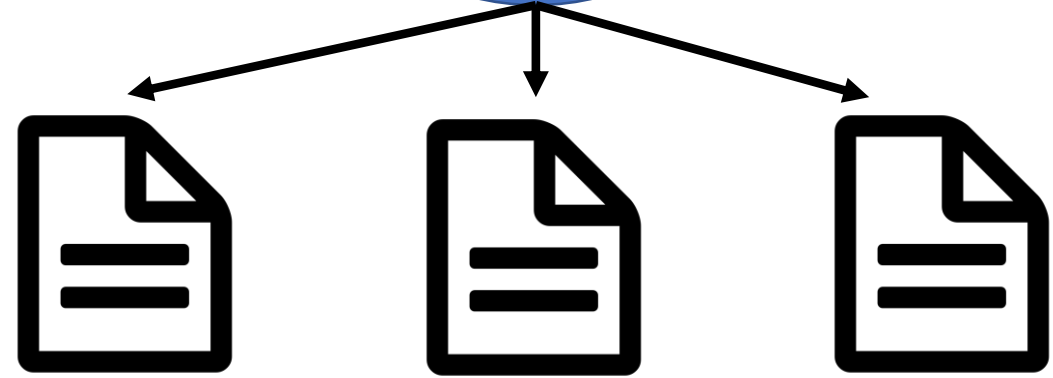
동일
데이터



다중 테이블에 중복

쿼리
효율/성능

동일
데이터



다중 Document에 중복

Denormalization의 trade-off

Query data volume
I/O per query

모든 데이터 한곳에 모여서
쿼리 당 I/O숫자를 줄임

VS

total data
volume

여러 테이블이나 도큐먼트에 중복저장
사이즈가 필연적으로 증가

Processing
complexity

복잡도를 증가시키는 JOIN 없음
쿼리 친화적 구조
전체적인 쿼리 프로세싱 단순화
수행시간 단축

Denormalization - Example

요구사항

- 사용자별로 이름, 나이, 성별, 우편번호를 출력
- 주어진 데이터는 유저 : 이름, 나이, 성별,
도시 : 도시명, 주, zipcode

Denormalization - Example

Relational의 경우

User				
User_ID(PK)	Name	Age	Sex	City(FK)

City			
City_ID(PK)	City	State	ZipCode

```
select u.name,u.age,u.sex,c.zipcode from user u, city c where u.city = c.city_id
```

Denormalization - Example

NoSQL의 경우

User				
User_ID(PK)	Name	Age	Sex	City(FK)

City			
City_ID(PK)	City	State	ZipCode

Relational과 같은 테이블로 쿼리 시 2번을 날려야 함

```
select $city,name,age,sex from user where user_id="사용자ID"  
select zipcode from city where city=$city
```

Denormalization - Example

NoSQL의 경우

애초에 쿼리부터 설계함

```
select city,name,age,sex,zipcode from user where userid="사용자ID"
```

User					
User_ID(PK)	Name	Age	Sex	City(FK)	Zipcode

중복이고 뭐고 간에 쿼리에 충실하게 User table만듬

Aggregates

대부분의 NoSQL 솔루션은
기본적으로 유연한 스키마 제공

- Nested entities를 구성하여 복잡하고 다양한 구조가 가능
- 1:N 관계 최소화 -> JOIN 연산 하지 않음
- 구조가 유연하여 다양한 도메인 활용가능

Aggregates

key-value store & Graph DB

- 일반적으로 저장되는 value와 데이터에 제약 없음
- key 디자인 시, composite key 활용하여 다수의 레코드를 하나의 entity로 구성 가능
- 다수의 entity를 포함한 하나의 document 또는 table 구성이 가능하며 하나의 조합키로 표현이 가능

Aggregates

BigTable model

- value 영역에 다양한 column family와 한개 이상의 column set

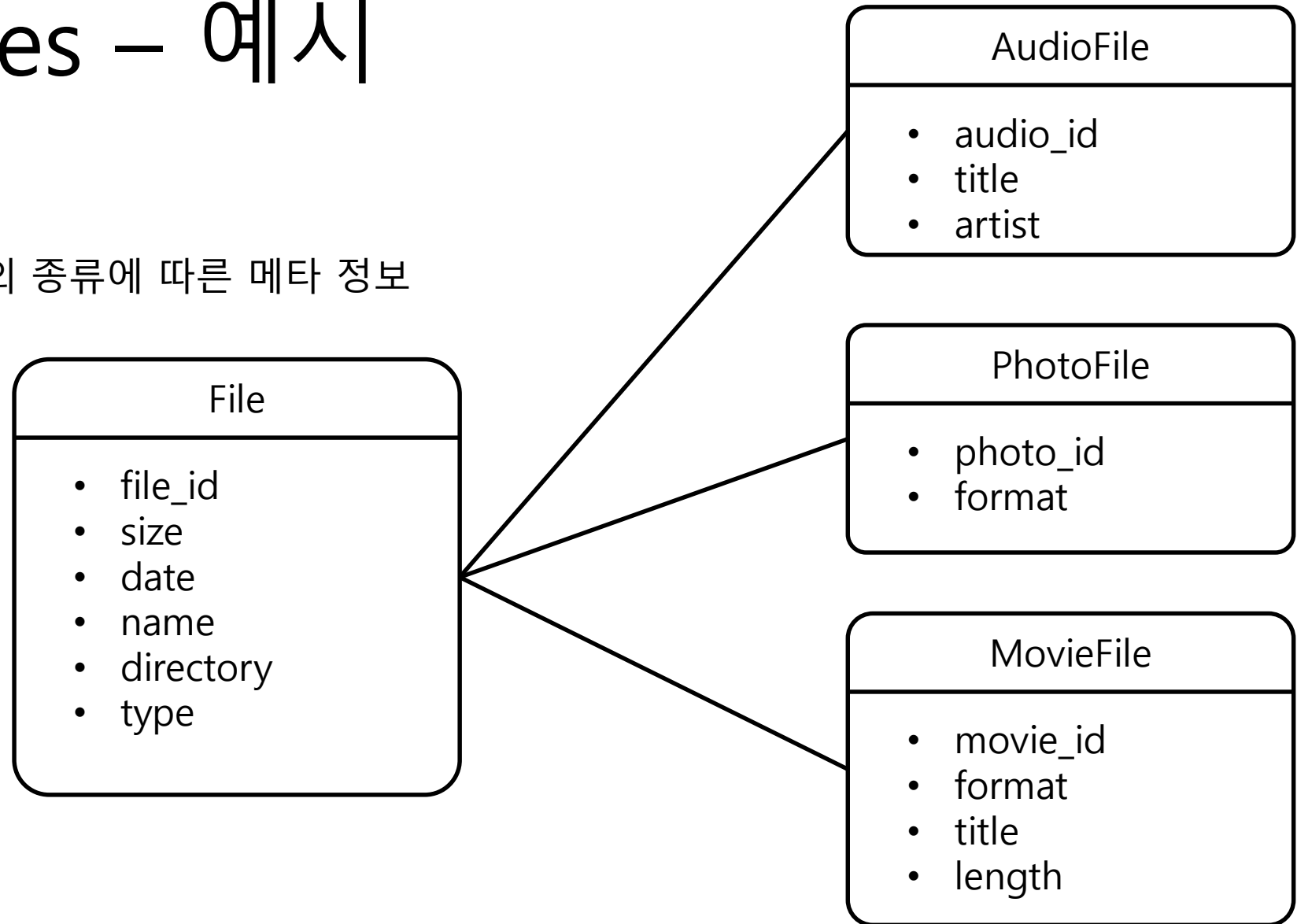
Aggregates

Document DB

- 구조상 태생적으로 Schema-less

Aggregates – 예시

- 파일 공유 서비스
- 파일 정보와 파일의 종류에 따른 메타 정보



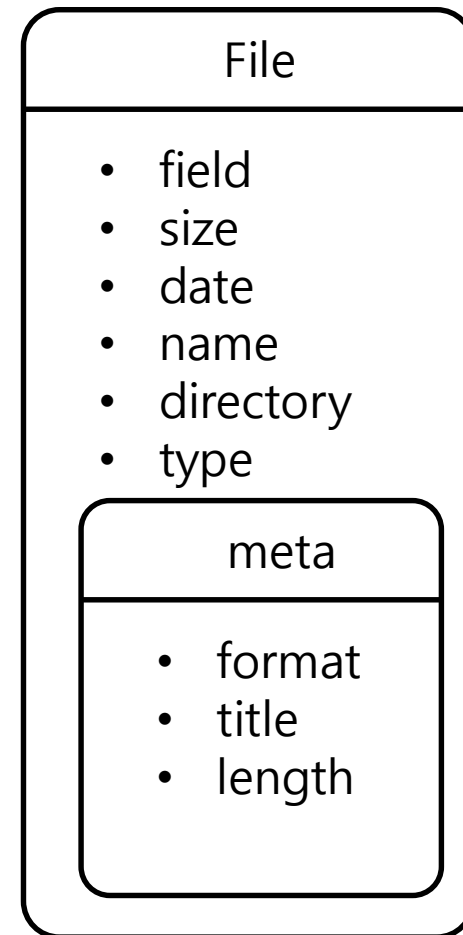
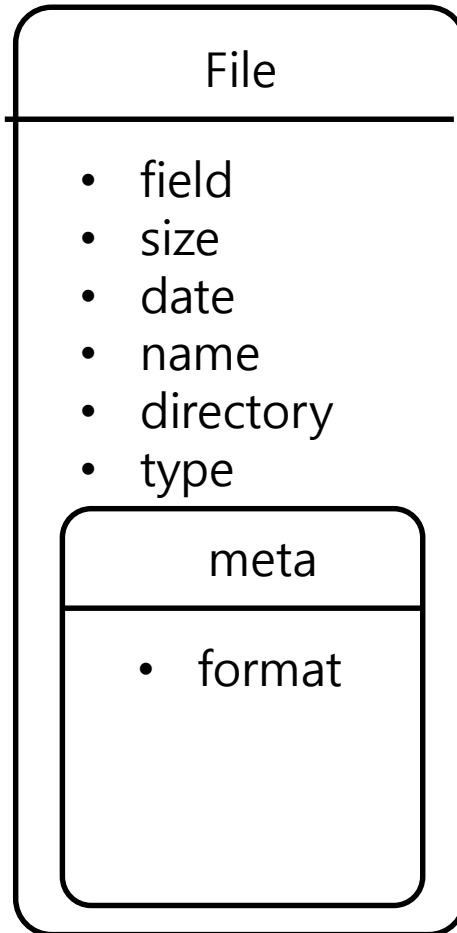
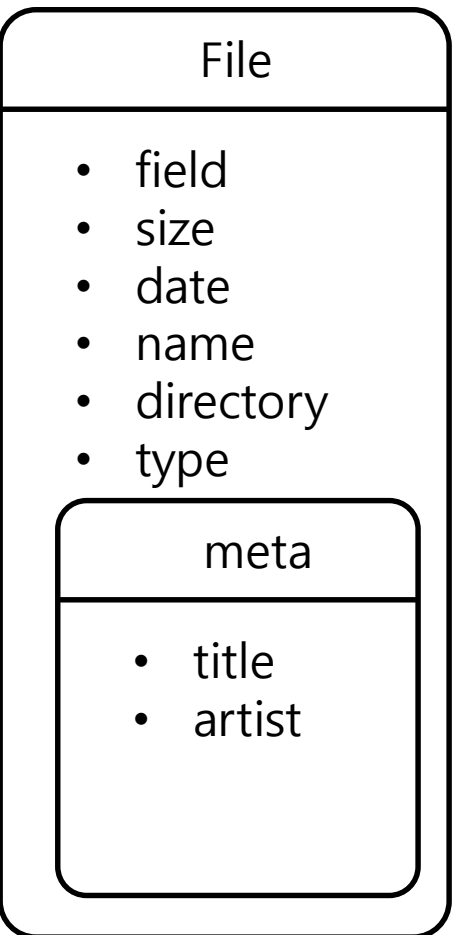
Aggregates

RDBMS VS NoSQL

- 데이터 입력 시, 반드시 테이블 구조에 맞춰야 함
- column수, 이름, 데이터 타입 준수
- 모든 row는 같은 column 가져야 함
- Scheme-less / Soft scheme
- key만 같다면 각 row의 데이터가 제멋대로라도 상관없음
- 같은 column을 가질 필요 없고, 타입이 달라도 가능
- value에 저장되는 row 구조는 일치할 필요가 없음
- 구조는 가지되 각각 다른 것을 허용함

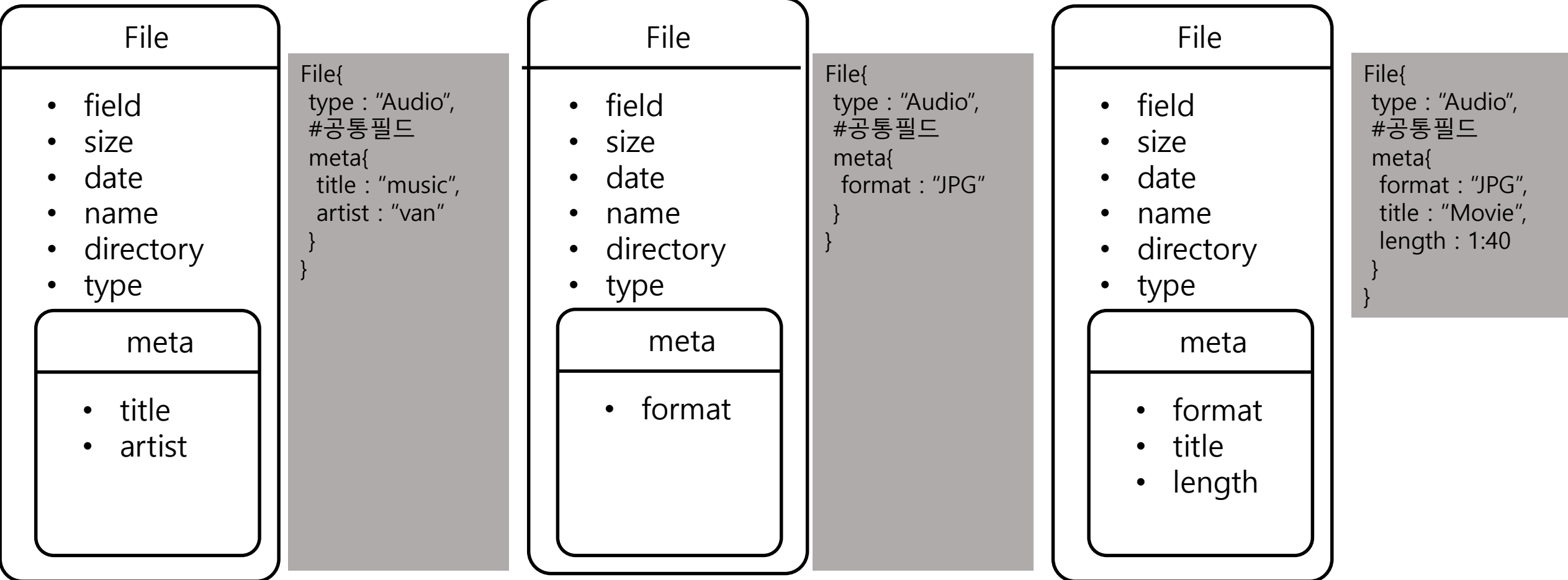
Aggregates – 예시

- Scheme-less 특성을 이용해서 하나의 테이블로 합침



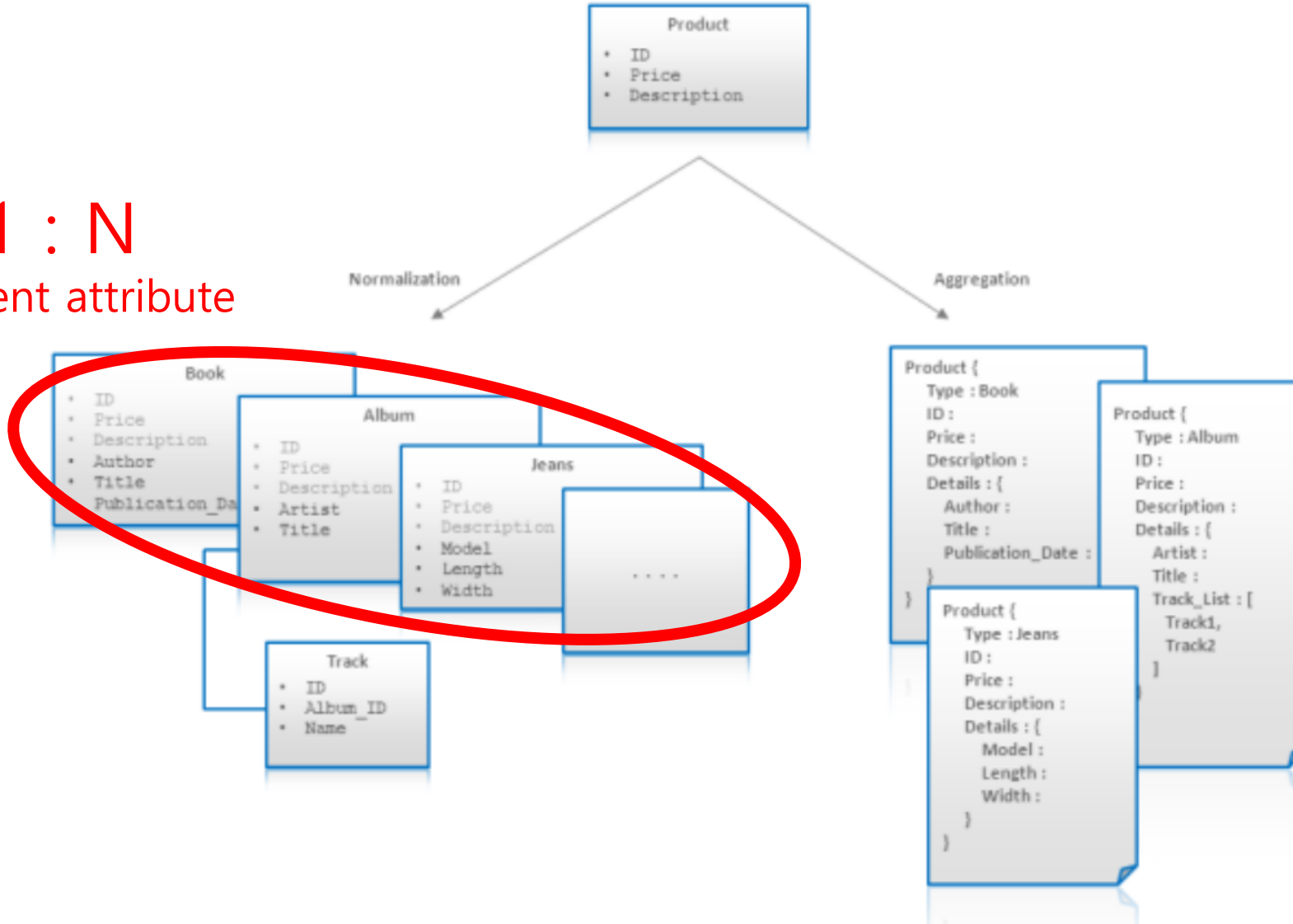
Aggregates – 예시

- Scheme-less 특성을 이용해서 하나의 테이블로 합침

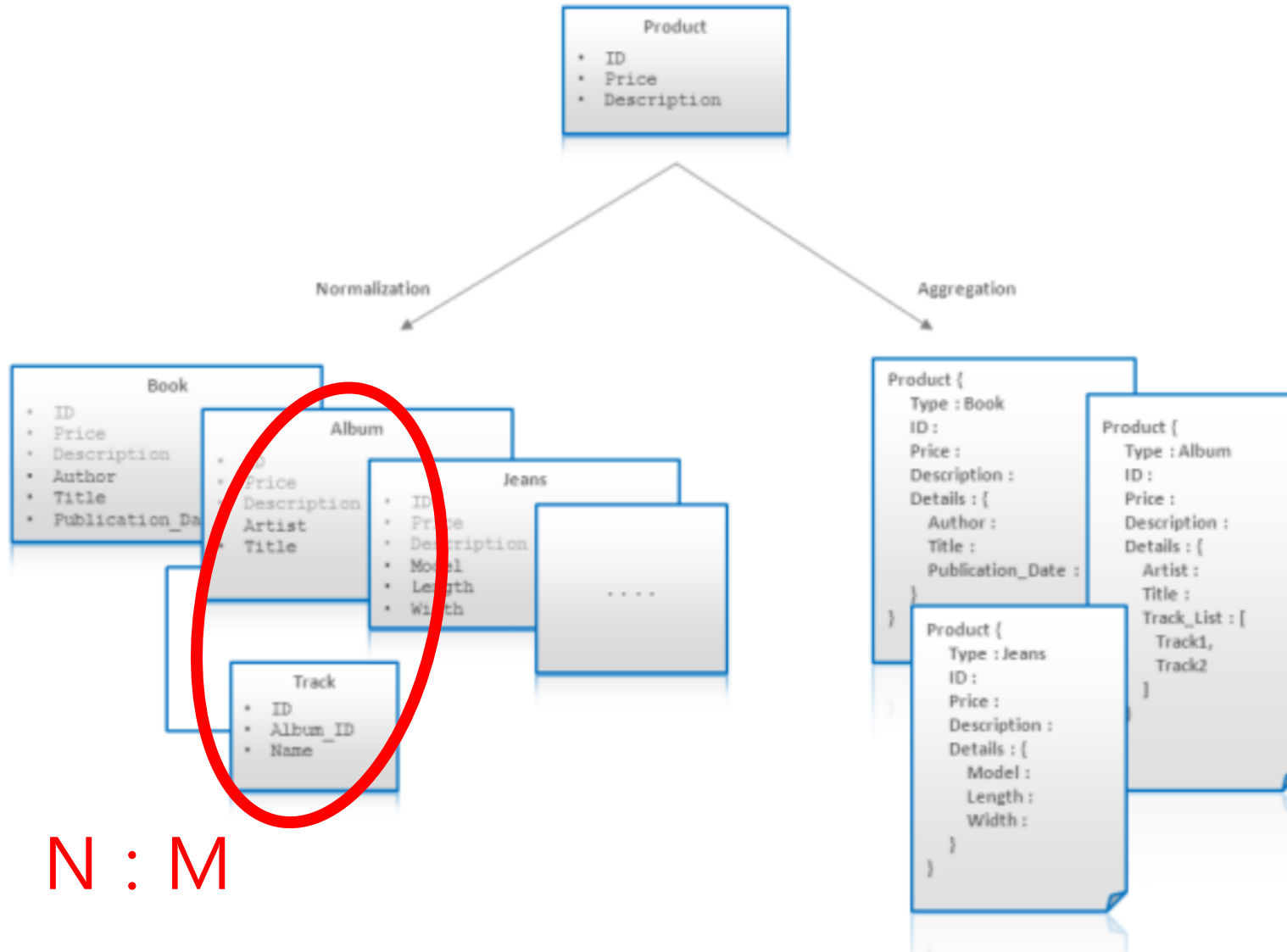


Aggregates – 예시(2)

1 : N
different attribute

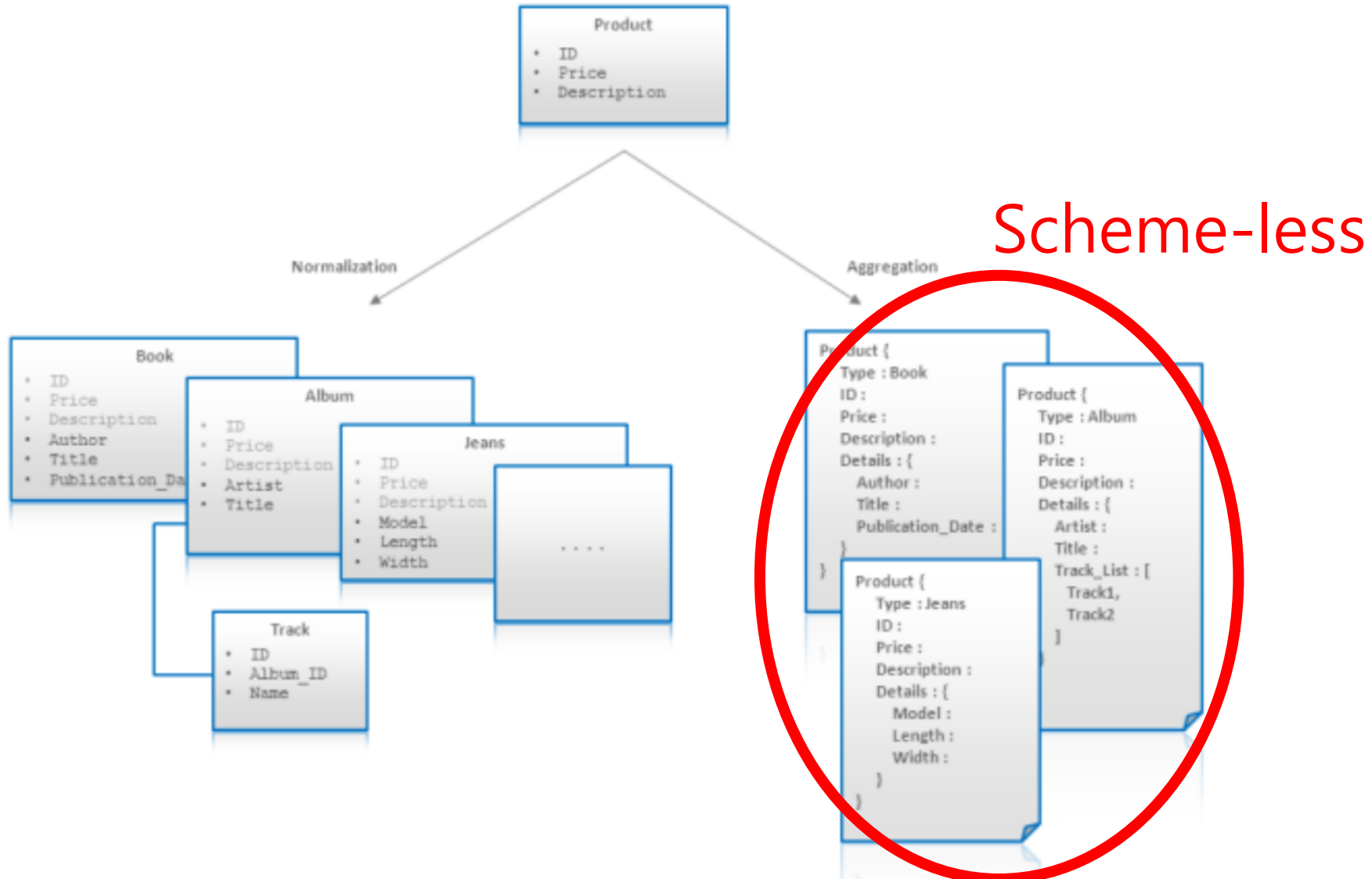


Aggregates – 예시(2)



N : M

Aggregates – 예시(2)



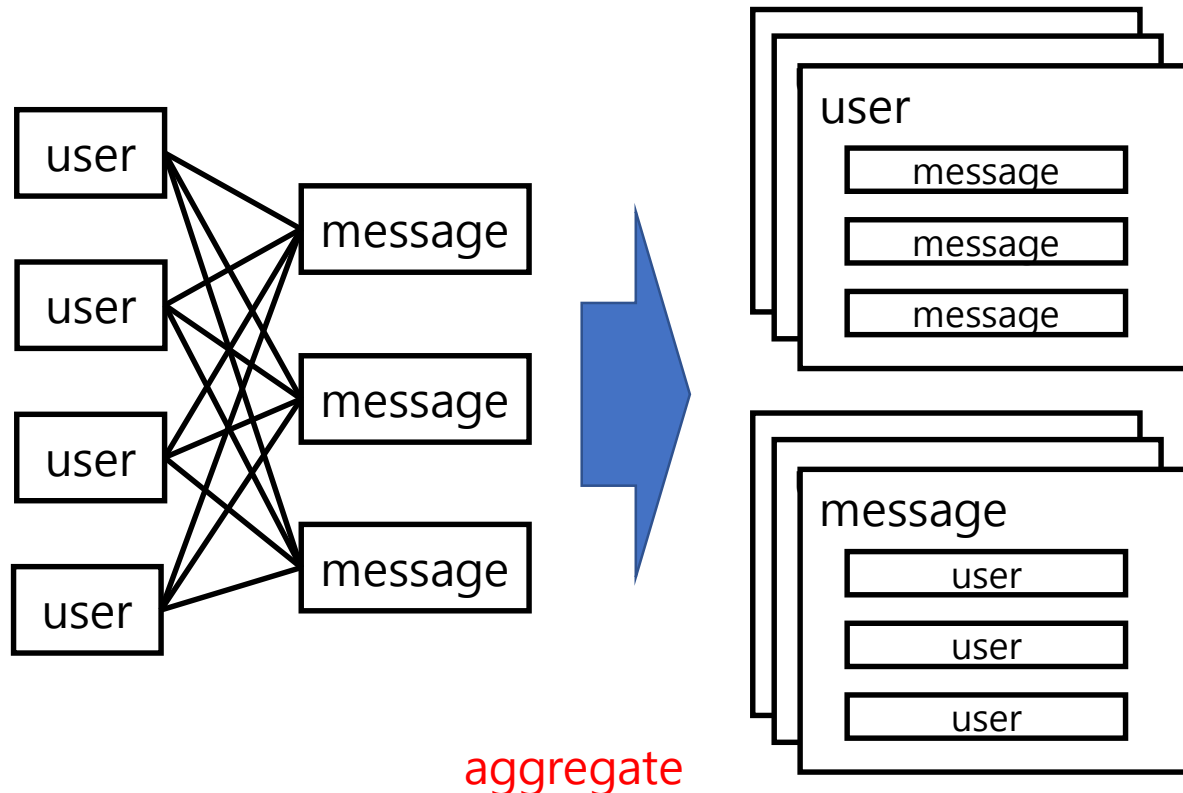
Application side JOIN

- NoSQL은 일반적으로 JOIN을 제약함
- JOIN을 하지 않도록 Denormalization과 aggregation
- 데이터 관계와 속성보다는 그 데이터를 어떻게 이용하는가에 따라 구조화
- 관계를 정의하고 normalization하여 join하는 관계형 모델링과 가장 큰 차이
- 대용량 데이터에 대한 빠른 응답성능, 확장성, 가용성을 최우선 목적으로 하므로 쿼리 타임 JOIN을 피하여 데이터 모델 구성

Application side JOIN

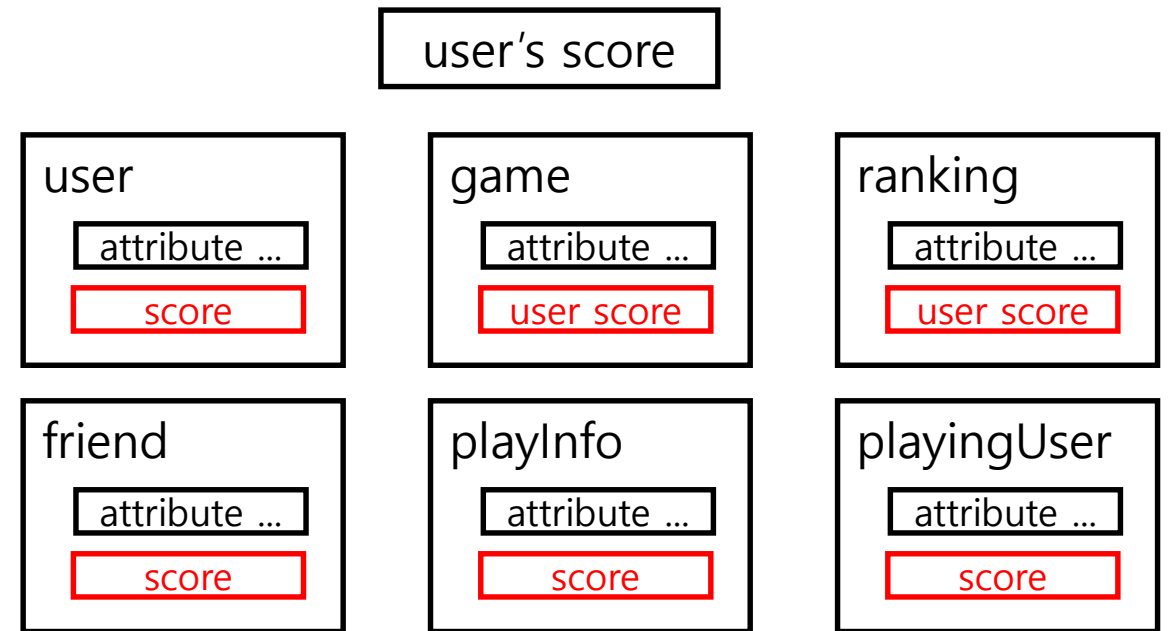
JOIN 대상 데이터를 Denormalization, aggregation 수행 시 문제가 발생할 수 있는 case

- JOIN 대상 데이터가 N:M관계를 가짐



many to many → one to many

- JOIN 대상 데이터가 수시로 변경

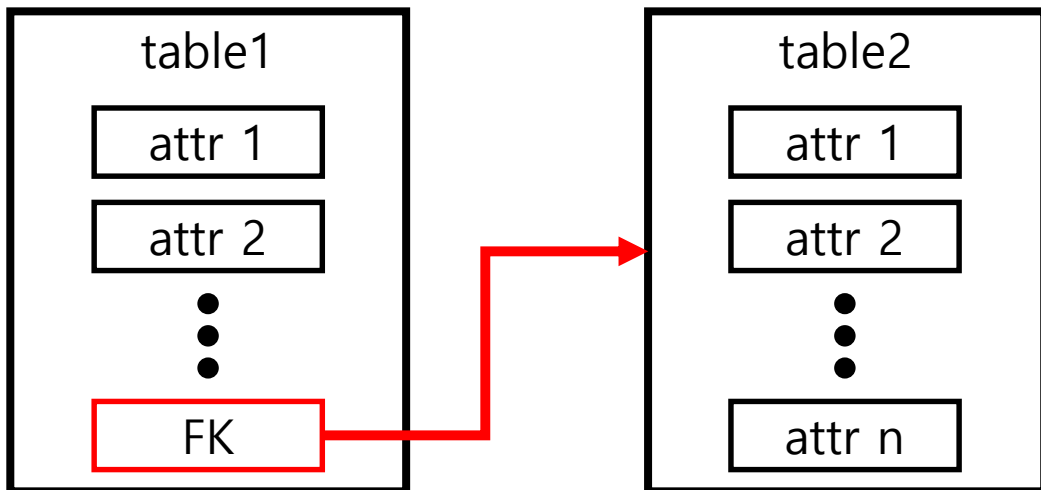


많은 entity에 중복 저장한 경우 일일이 다 찾아야 해서
오히려 엄청나게 많은 비용이 발생

차리리 쿼리 타임 JOIN을 수행하는게 더 낫다

Application side JOIN

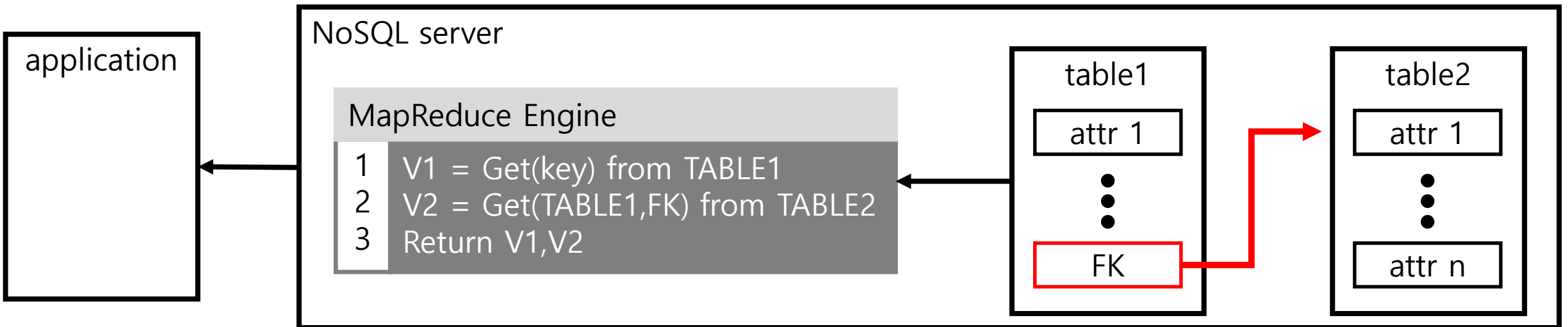
- 어쩔 수 없이 성능상의 문제로 Join을 수행해야 한다면, application 레벨에서 로직 코드로 처리
- Join 필요한 테이블 만큼 I/O효율을 포기하고 스토리지 효율 높임



```
application
1  V1 = Get(key) from TABLE1
2  V2 = Get(TABLE1,FK) from TABLE2
3  Return V1,V2
4
5
6
7
```

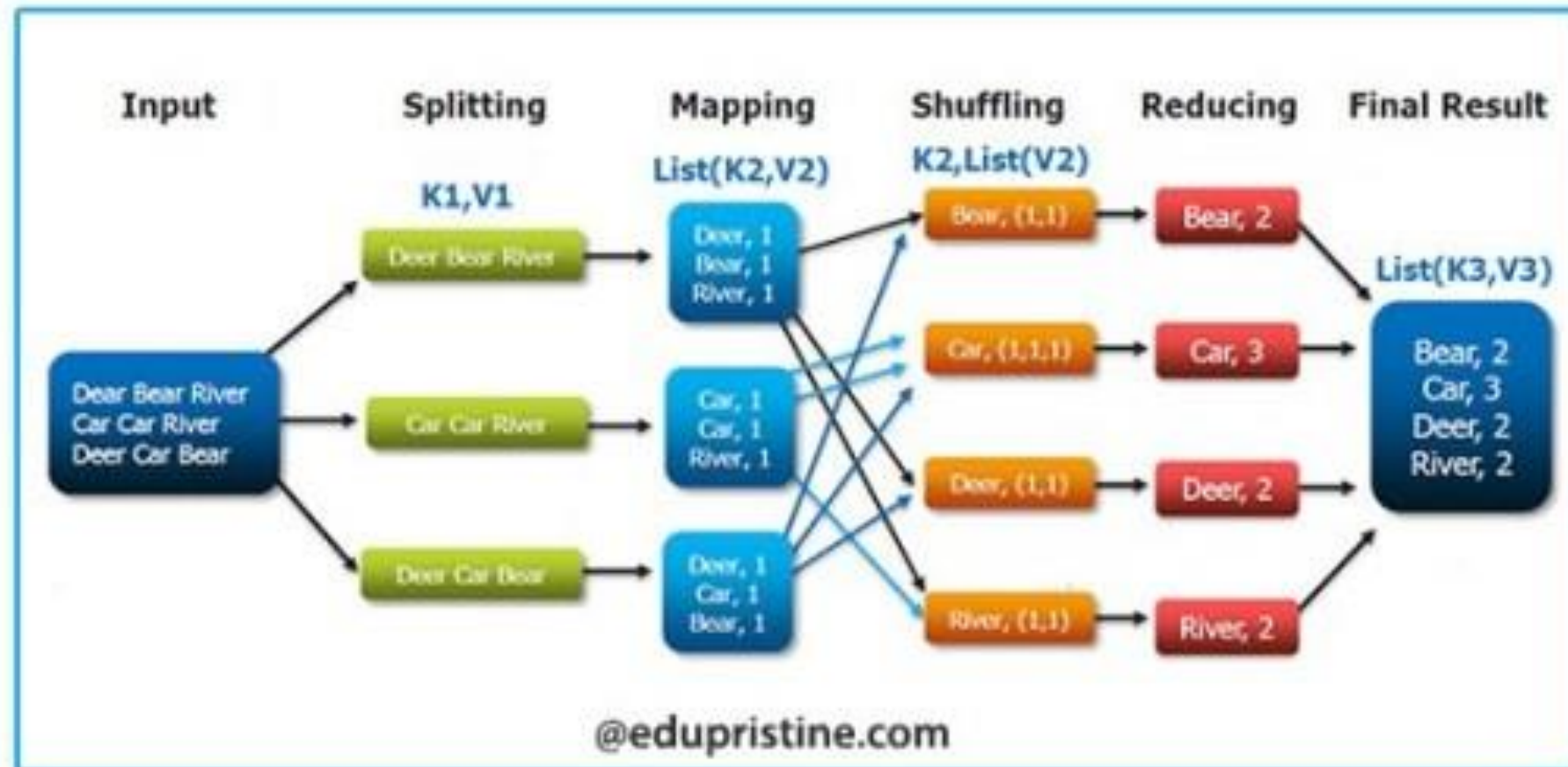
server side JOIN – MapReduce

- MongoDB 등의 일부 NoSQL DB는 stored procedure 지원
→ MapReduce
- application side에서 하던 join연산을 NoSQL 서버에서 수행
- I/O 효율은 올라가지만 NoSQL 서버부담 가중



server side JOIN – MapReduce

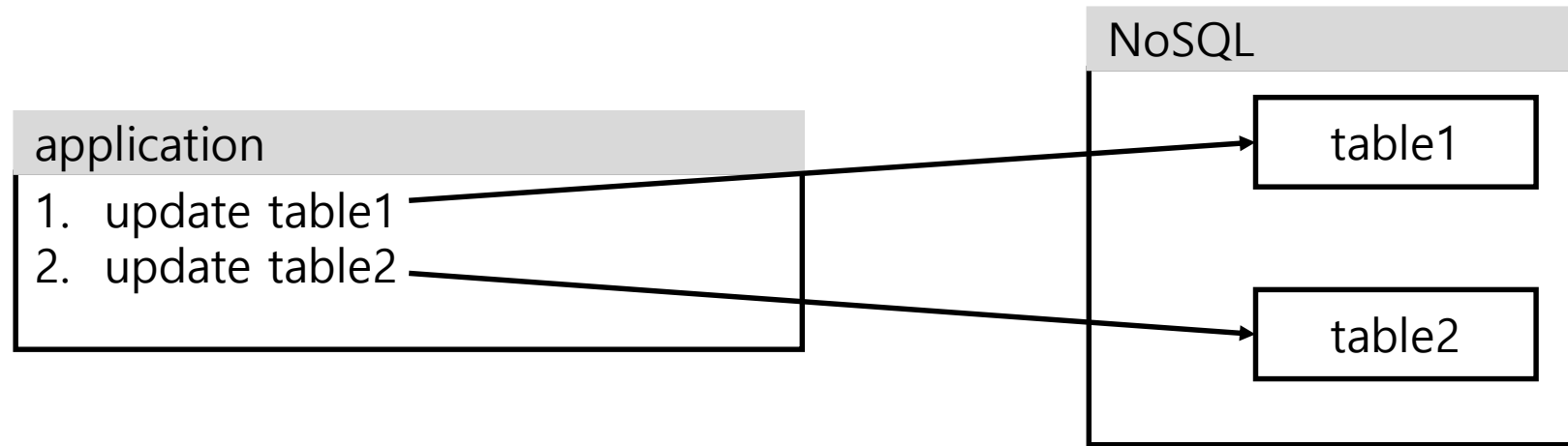
- map reduce



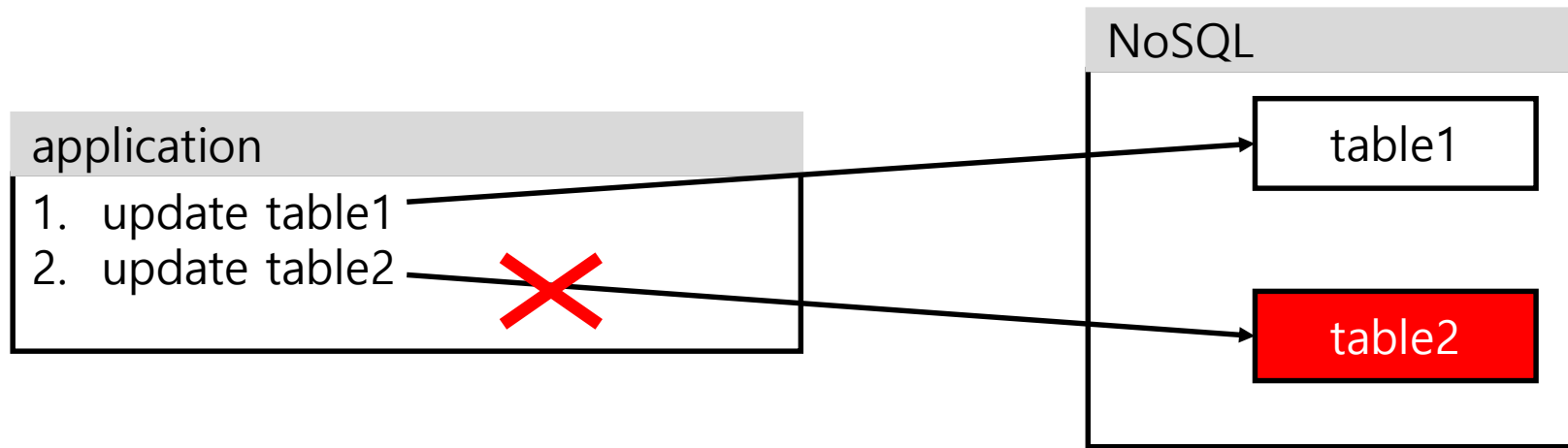
일반적인 NoSQL 모델링 스킴

- Atomic Aggregates
- Enumerable Keys
- Dimensionality Reduction
- Index Table
- Composite Key Index
- Aggregation with Composite Keys
- Inverted Search – Direct Aggregation

Atomic Aggregation

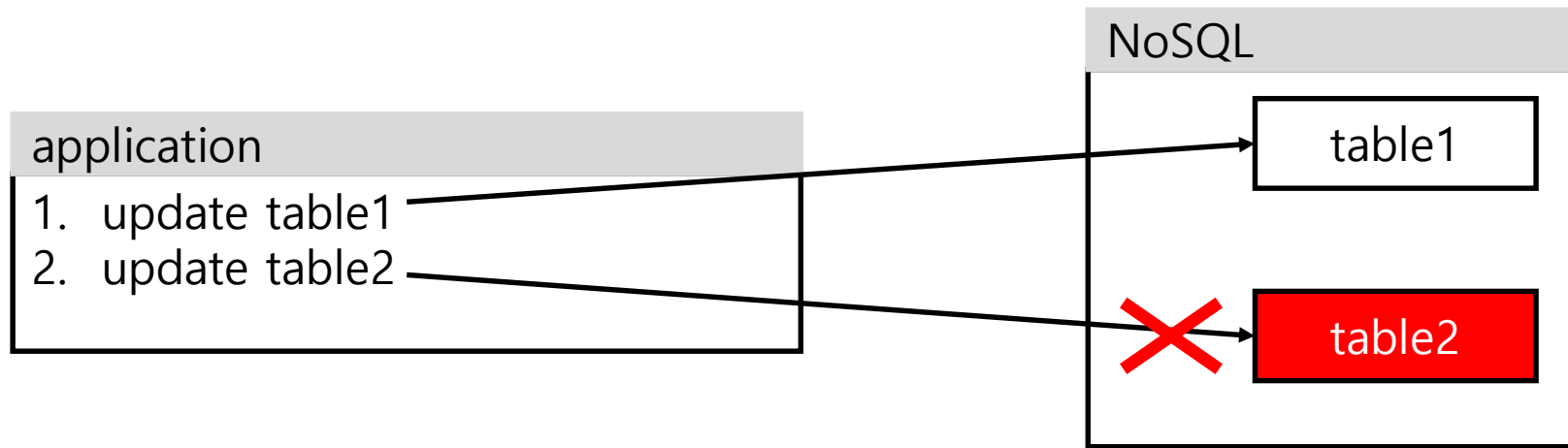


Atomic Aggregation



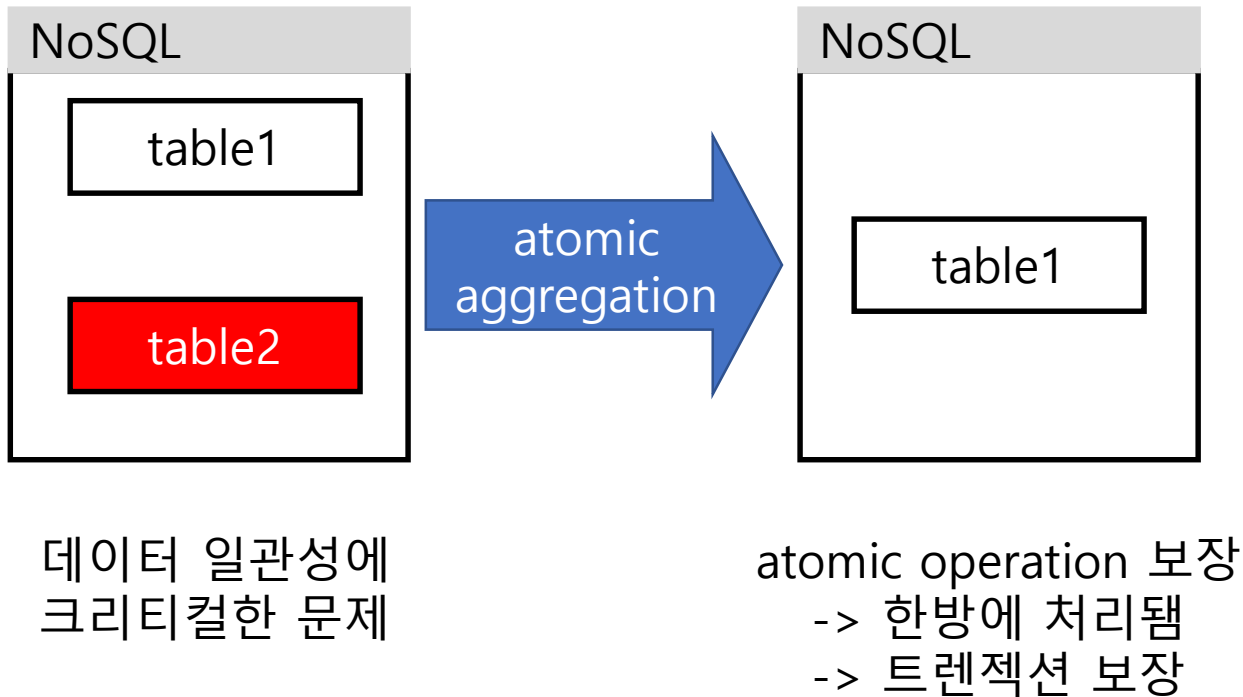
application layer에서의 장애로 인해 일부 테이블이 업데이트가 되지 않은 경우

Atomic Aggregation



NoSQL layer에서의 장애로 인해 일부 테이블이 업데이트가 되지 않은 경우

Atomic Aggregation

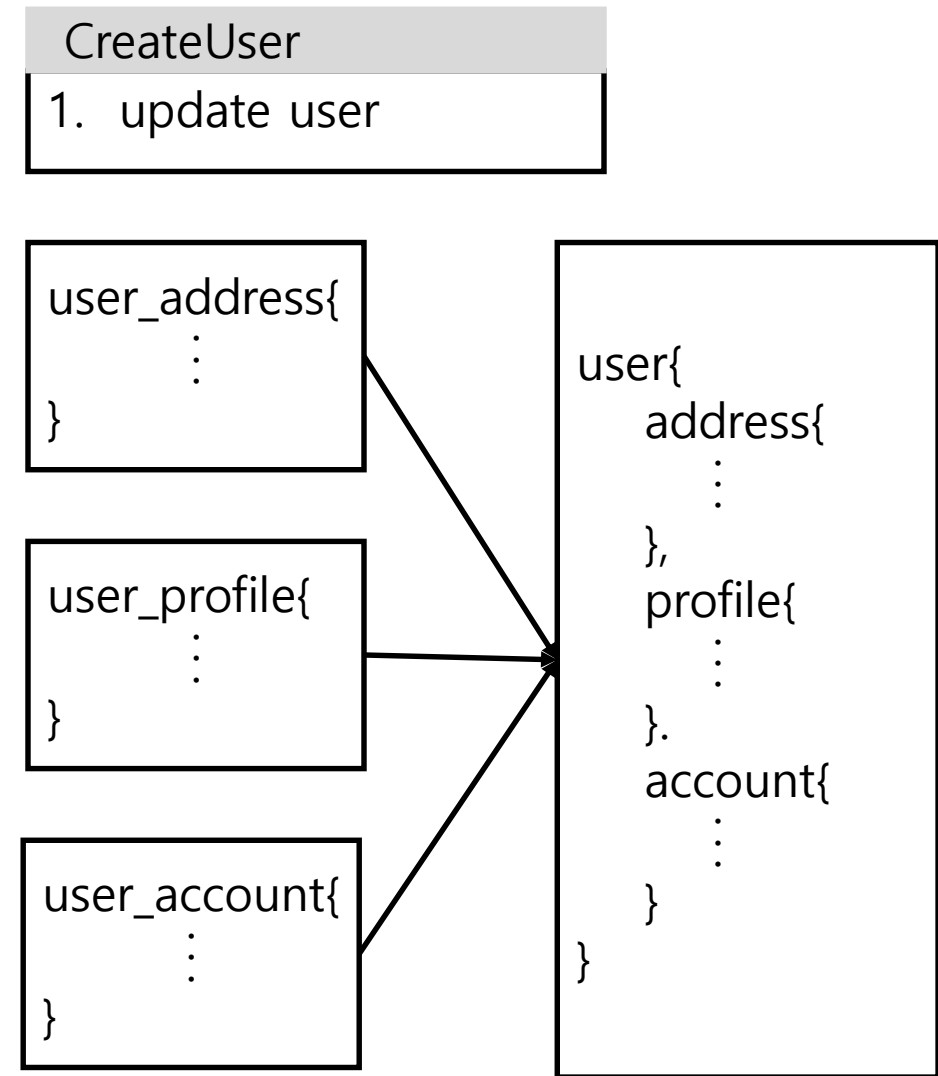
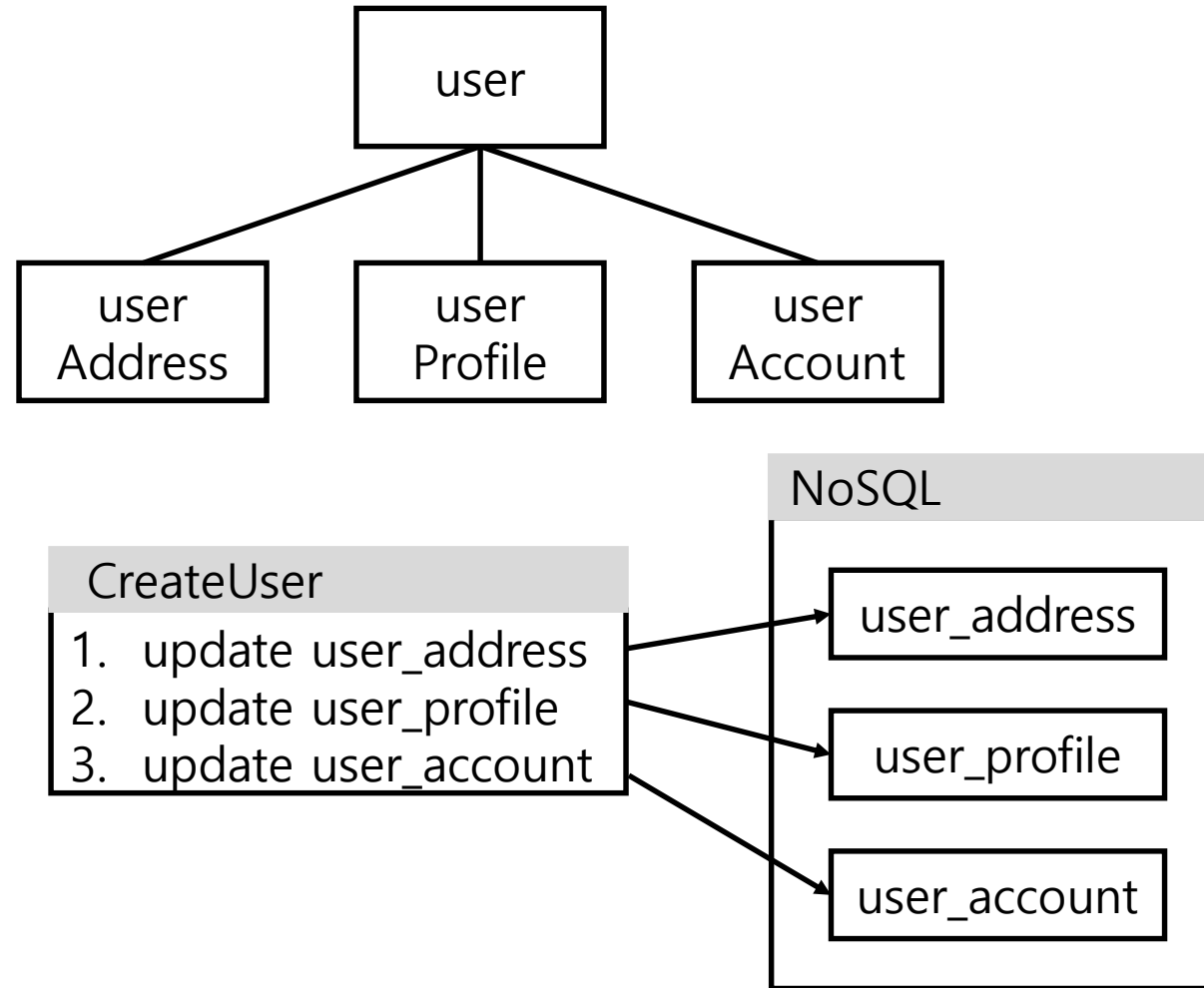


Aggregation하고 뭐가 다른가?

Aggregation
1:N이나 N:M에서의 JOIN을 없애기 위한 용도

Atomic Aggregation
트랜잭션 보장을 통한 데이터 일관성 보장

Atomic Aggregation



Index Table

- NoSQL은 기본적으로 RDB처럼 index기능을 제공하지 않음
- value의 필드를 이용하여 Search하면 Full scan해야함
- value의 필드로 search가 아예 불가능할 수도 있음
- 별도의 index table을 만들어서 index기능 대신 사용할 수 있음
- 이 방식은 일반적으로 상당히 많이 사용하는 방식임

Index Table

music table				
music_id				
	name	artist	album	category(index)

특정 카테고리의 음악만을 리스팅 해야 한다면?

music category index table				
category				
	music_id	name	artist	album

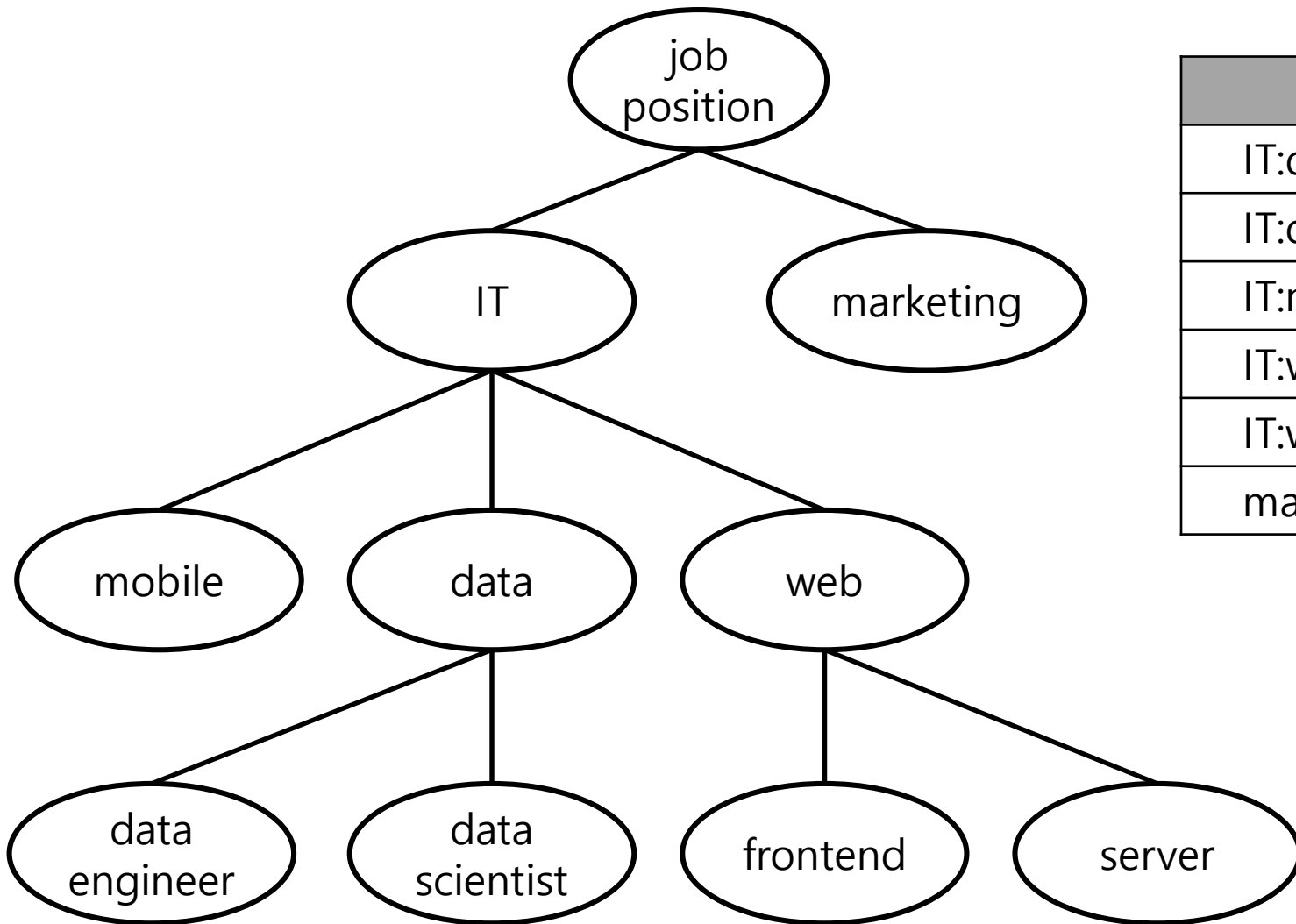
Index Table시 고려사항

- Cassandra 등에서는 Secondary index 기능이 있음
- 경우에 따라 Secondary index가 성능이 좋은 경우가 있고, index table이 성능이 좋은 경우가 있음
- secondary index 또는 index table를 사용할 계획이면 각 쿼리를 벤치마킹 후 의사결정 해야 함

Composite Key

- RDB의 복합키와 같은 개념
- RDB는 여러개의 column을 복합키로 바인딩하지만 NoSQL은 Delimiter를 이용하여 복수의 key를 바인딩
- Ordered Key-Value store의 경우 sorting이나 grouping 가능

Composite Key



key	value
IT:data:data_engineer	
IT:data:data_scientist	
IT:mobile	
IT:web:frontend	
IT:web:server	
marketing	

Composite Key

SELECT Values WHERE job_category = "IT:"

SELECT Values WHERE sub_category = "IT:data"

job_category:sub_category:position

key	value
IT:data:data_engineer	
IT:data:data_scientist	
IT:mobile	
IT:web:frontend	
IT:web:server	
marketing	

Composite Key의 고려사항

- key설계 시 분산시스템에서 한 쪽 노드에만 몰리는 현상 주의
- NoSQL은 N개의 노드로 이루어진 클러스터
- Key를 기준으로 N개의 서버에 나눠져서 저장
- 부하가 한쪽으로만 몰리지 않게 key를 설계해야 함

Inverted Search Engine

- value의 내용을 key로 하고, key를 value로 넣는 방법
- 검색엔진에서 많이 씀 : bot이 검색해서 문서 내용을 색인화
- value의 단어를 key로 하기 때문에 효과적인 검색이 가능

key	value
bcho,tistor.com/nosql	nosql,cassandra,riak
bcho.tistory.com/cloud	amazon,azure,google
facebook.com/group/serverside	amazon,google,riak
highscalability.com/bigdata	nosql,riak
www.basho.com/riak	riak



key	value
amazon	bcho.tistory.com/cloud facebook.com/group/serverside
nosql	bcho,tistor.com/nosql highscalability.com/bigdata
riak	bcho,tistor.com/nosql facebook.com/group/serverside highscalability.com/bigdata www.basho.com/riak

Enumerable Keys

- RDB의 sequence와 같은 기능을 제공
(특정 NoSQL DB만 해당 / 특히 Graph DB)
- 자동으로 key의 카운터를 올려서 순차적으로 연속된 키 부여
- data traverse 기능 제공

Hierarchy Modeling Techniques

- Tree Aggregation
- Adjacency Lists (인접 리스트)
- Materialized Paths
- Nested Sets
- Nested Documents Flattening: Numbered Field Names
- Nested Documents Flattening: Proximity Queries
- Batch Graph Processing

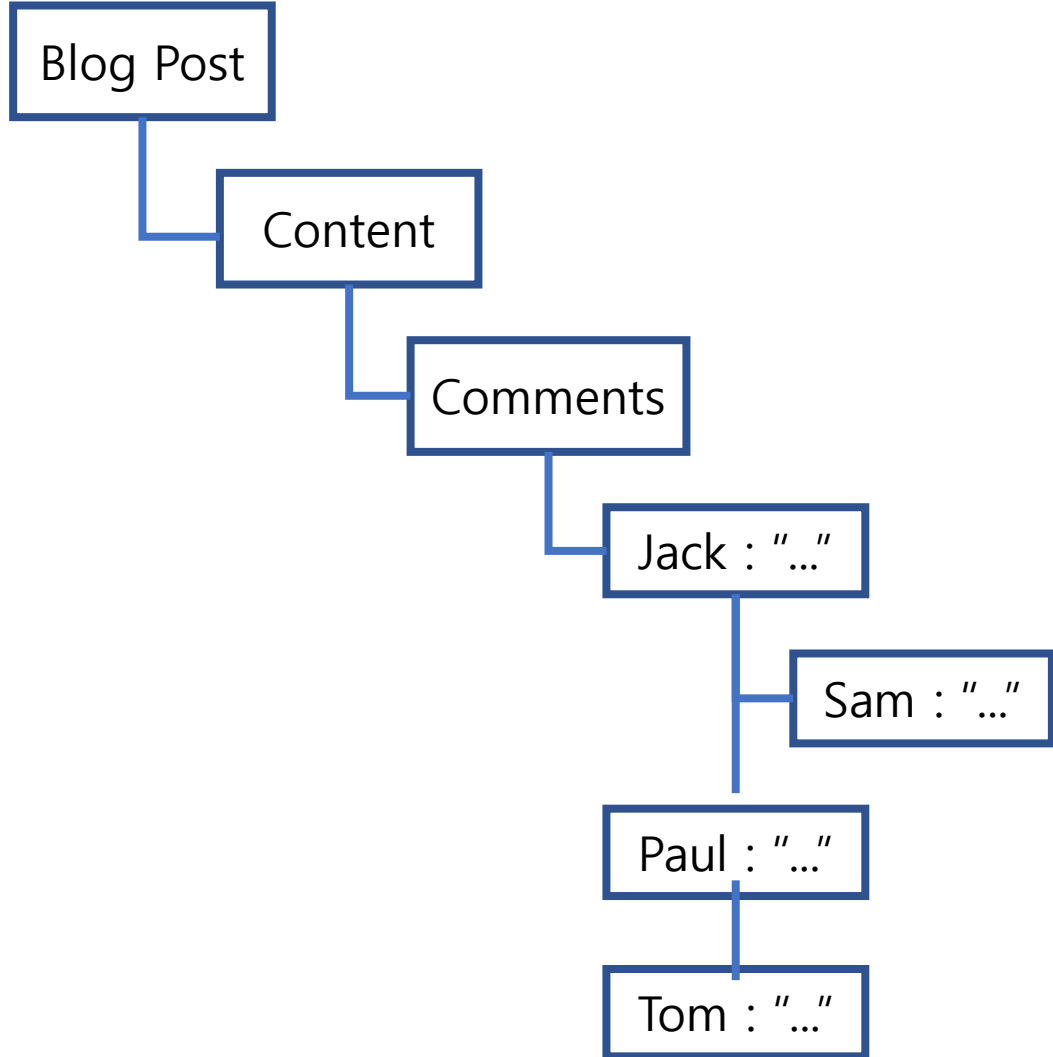
계층적 데이터 구조 모델링 패턴

- NoSQL의 기본적인 데이터 모델은 row, column을 가지고 있는 테이블 구조의 저장구조를 가짐
- 하지만 요구사항에 따라 tree와 같은 계층형 데이터 구조를 저장해야 할 경우가 있음, NoSQL은 이런 구조의 저장이 힘들
- RDB는 계층형 데이터 구조를 위하여 기능적으로 지원하며 데이터 모델링을 통해 저장 가능
- NoSQL에서 계층형 구조를 저장하는 기법은 RDB의 모델링 기법을 참고하여 구현

Tree Aggregation

- Tree 구조자체를 하나의 Value에 저장하는 방식
- JSON이나 XML등의 문서형 구조로 계층 정의
- Document DB의 정수
- Tree변경이 크지않고 변경이 많이 없는 경우 적합

Tree Aggregation

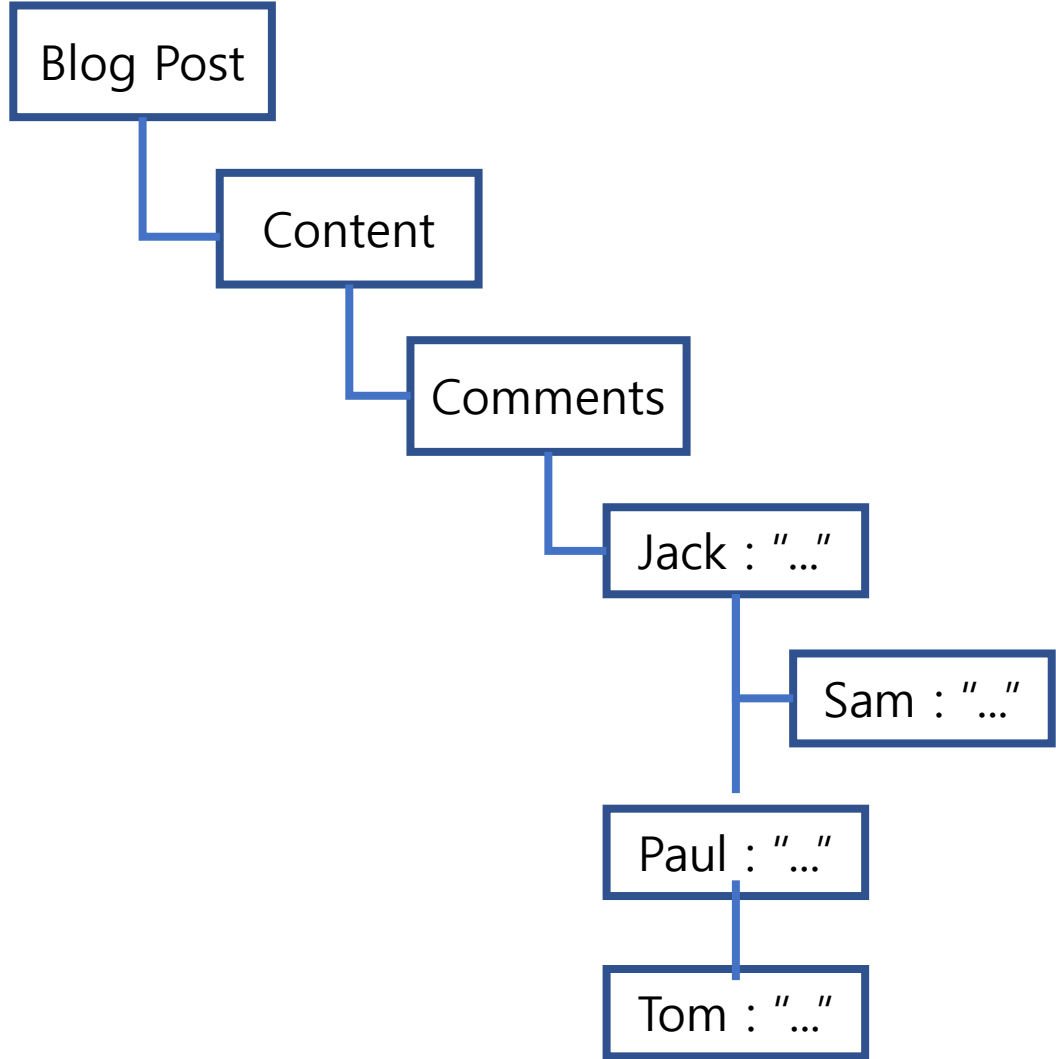


```
{
  "content1" : {
    "post" : "...",
    "comments" : [
      {
        "message" : "...",
        "by" : "Jack"
        "replies" : [
          {
            "by" : "Sam",
            "message" : "..."
          }
        ]
      }, {
        "message" : "...",
        "by" : "Paul"
      }, {
        "message" : "...",
        "by" : "Tom"
      }
    ]
  }
}
```


Adjacent Lists

- adjacent list는 자료구조에서의 tree에 해당하는 구조를 사용
- tree의 각 node에 parent node의 포인터와 child node의 포인터를 저장하는 방식
- Tree 검색 시, child node 포인터와 parent node 포인터를 이용하여 순회함 -> Tree traversing
- 특정 node를 알면 상위node 하위node 자유롭게 traversing 가능

Adjacent Lists



blog_post

- content_key
- parent
- child
- value

content_key	parent	child	value
Content1	NULL	Comments	"..."
Comments	Content1	comment1...4	4
comment1	Comments	comment4	"...",Jack,1
comment2	Comments	NULL	"...",Paul,0
comment3	Comments	NULL	"...",Tom,0
comment4	Comment1	NULL	"...",Sam,0

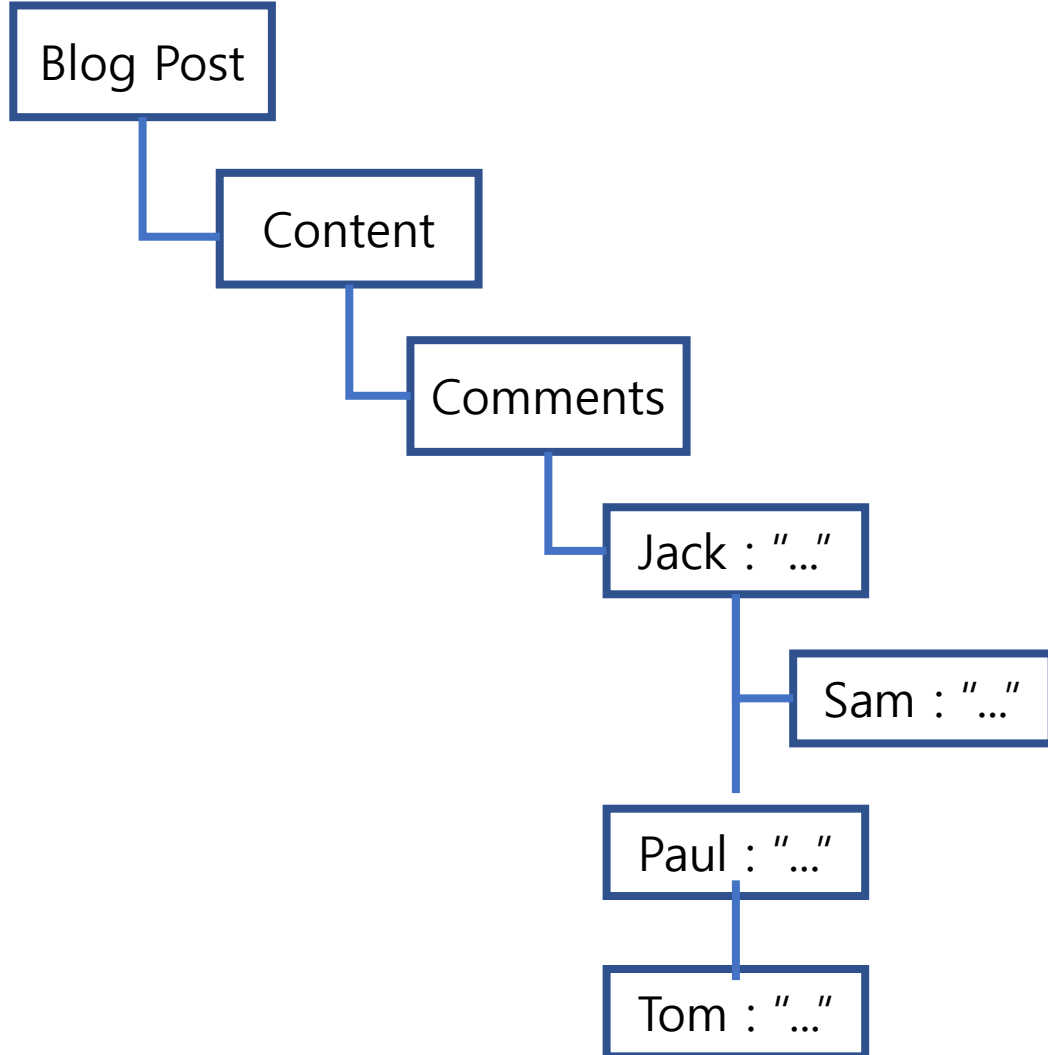
Adjacent List시 고려사항

- traversing시 하나의 노드를 이동할때마다 포인터를 들고 와야 하므로 그 때마다 쿼리를 날려서 IO부담이 증가된다
- Tree 크기가 크다면 다른 방법을 사용하는 것이 좋다
- 구현의 용이성 VS 쿼리 IO trade-off

Materialized Path

- 가장 단순하면서 확실한 방법
- 그냥 각 노드마다 그 노드까지 가는 전체경로를 key로 저장
- 구현은 좀 번거롭지만 효율적인 저장 방식
- search시 Regular Expression을 사용할 수 있다면, 특정 노드의 하위 노드들만 가져오는 등 다양한 응용이 가능

Materialized Path



key(path)	value
content1	" ... "
content1/comments	4
content1/comments/comment1	" ... "
content1/comments/comment2	" ... "
content1/comments/comment3	" ... "
content1/comments/comment1/comment1	" ... "

Query

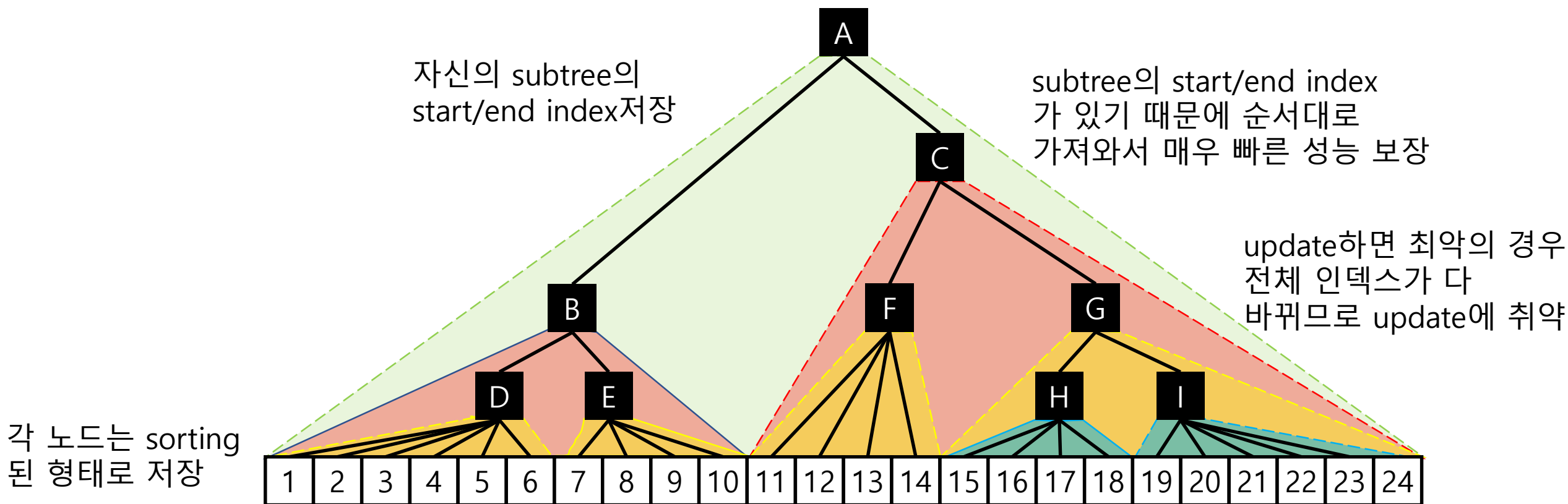
values :=
Get(RegExp("content1/comments/*"))

Materialized Path시 고려사항

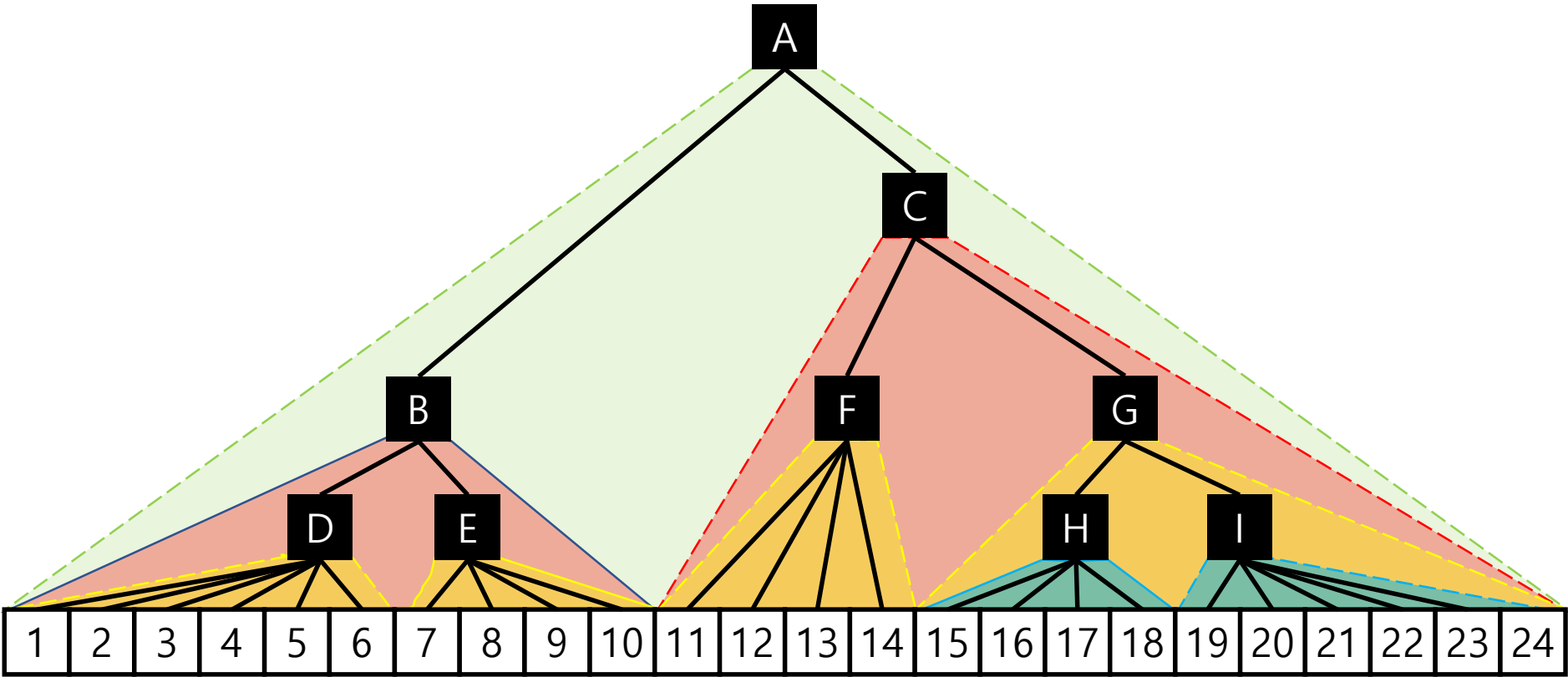
- Regular Expression이 없는 일반 Key-value store나 Ordered Key-value store에서는 적용이 힘들
- MongoDB 같은 DocumentDB에서는 Regular Expression을 지원하므로 효과적으로 사용이 가능함

Nested Sets

- Node가 포함하는 모든 child node에 대한 범위정보 가짐
- 업데이트가 거의 없는 대규모 tree 저장에 유용함



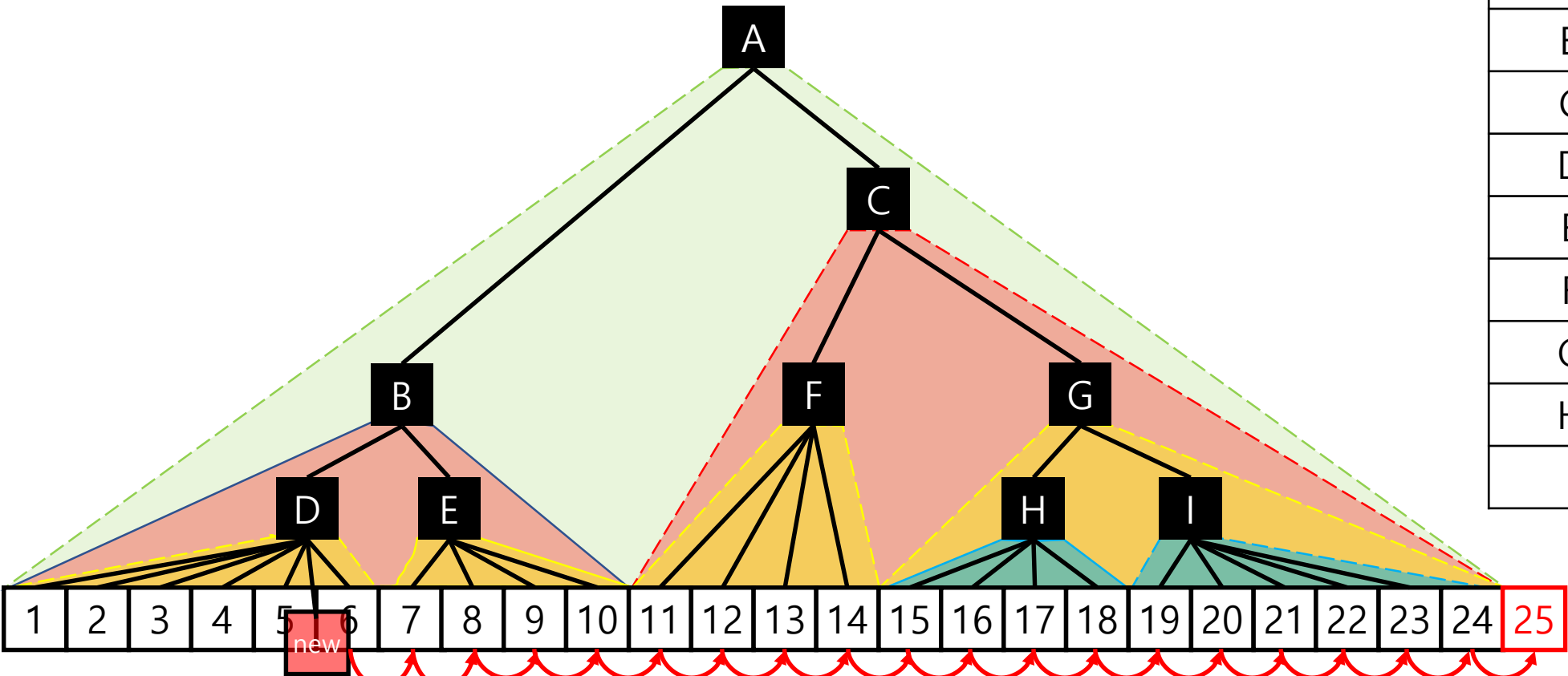
Nested Sets



node	start	end
A	1	24
B	1	10
C	11	24
D	1	6
E	7	10
F	11	14
G	15	24
H	15	18
I	19	24

Nested Sets

Update : D 밑에 기존 index 5와 6사이에
값이 추가 된다면?



node	start	end
A	1	24→25
B	1	10→11
C	11→12	24→25
D	1	6→7
E	7→8	10→11
F	11→12	14→15
G	15→16	24→25
H	15→16	28→19
I	19→20	24→25

정리해봅시다

NoSQL 데이터모델링

1. 도메인 모델 파악
2. 쿼리결과 디자인
3. 모델링 패턴을 이용한 데이터 모델링
4. 기능 최적화
5. NoSQL 선정 및 테스트
6. 선정된 NoSQL에 최적화 및 하드웨어 테스트
7. 운영테스트

(1) 도메인 모델 파악

- 데이터 entity와 관계 파악
- ERD 그려서 entity와 관계를 도식화
- 여기까지는 RDB의 방법도 크게 다르지 않음

(2) 쿼리 결과 디자인

- 도메인 모델을 토대로 app내에서의 쿼리 결과값을 먼저 디자인
- 쿼리 결과값들을 바탕으로 쿼리 정의
- 디자인한 쿼리 결과를 기반으로 테이블 정의

(3) 모델링 패턴 이용한 데이터 모델링

- 일반적인 경우 sorting, grouping, 특히 join기능을 제공 안함
- Get / Put으로만 데이터를 처리할 수 있는 형태로 모델링
- Key 설계가 가장 중요함
- sorting는 ordered K/V store를 통해 해결
- grouping은 계층형 모델링 패턴을 통해 해결
- join은 app side join이나 server side join등을 이용

(4) 기능 최적화

- 첨부 파일 :
 - 포스팅에 의존적이며 변경이 적고, 개수가 많지 않기 때문에 하나의 필드에 모아서 저장하는 것이 나옴
- 분류에 따른 포스팅 출력 :
 - 현재는 포스팅 순서대로만 출력 가능
 - 포스팅에 분류 필드를 별도로 넣고, 필드에 따라 where문으로select할 수 있어야 함 (Secondary Index)

(5)NoSQL 선정

- 모델링한 데이터 구조를 효과적으로 실행할 수 있는 NoSQL 검토
- NoSQL 특성 분석 및 부하테스트, 안정성/확장성 테스트 수행
- 경우에 따라서는 여러개의 NoSQL을 복합하여 사용

(6) NoSQL DB 및 하드웨어 최적화

- 선정한 NoSQL에 맞게 데이터 모델 최적화
- 해당 NoSQL에 맞는 applicaiton interface 설계
- 하드웨어 디자인(물리노드, 스케일 아웃 등)

(7) 운영 최적화

- 결국 데이터 모델링 또한 운영까지 신경써야 함
- 전체적인 workflow 고려
- 운영조건, 비용 및 요구사항에 맞춰서 테스트 해야함
- 운영 시, 부하를 최대한 분산시키고 fail-over/fail-back 고려
- 효율적으로 데이터를 모니터링 할 수 있는 방법까지 고려



운행을 해보야함

결론

- NoSQL을 사용하는 시스템 개발 시, 데이터 모델링이 80% 이상
- 선택한 NoSQL과 도메인, app 특성에 맞게 설계
- 하드웨어도 같이 고려해야 함
- 실버블렛은 없음
 - 하나로 다 하려고 하지 말고 여러 NoSQL이나 RDB등을 적절하게 혼용

