Database Systems
Lecture22 – NoSQL

Not only SQL

ACID
↓
기존 RDBMS에서 strong consistency를
희생한 프레임워크

Beomseok Nam (남범석)
bnam@skku.edu

Ch.8 Complex Data Types

Ch 8. Complex Data Types

- **Semi-Structured Data**
- Object Orientation
- Textual Data
- Spatial Data

Structured Data

- Data organized in a predefined format
  - E.g., database tables

instructor

ID	name	dept_name	salary
1	Alice	Comp. Sci.	20,000
2	Bob	Biology	10,000

department

dept_name	building
Comp. Sci.	Taylor
Biology	Taylor
Elec. Eng.	Watson

- First Normal Form (1NF)
  - Each column in a table contains atomic values

multiple value 못 가짐 → 하나의 value만

Unstructured Data

- Data that doesn't have a predefined structure or format

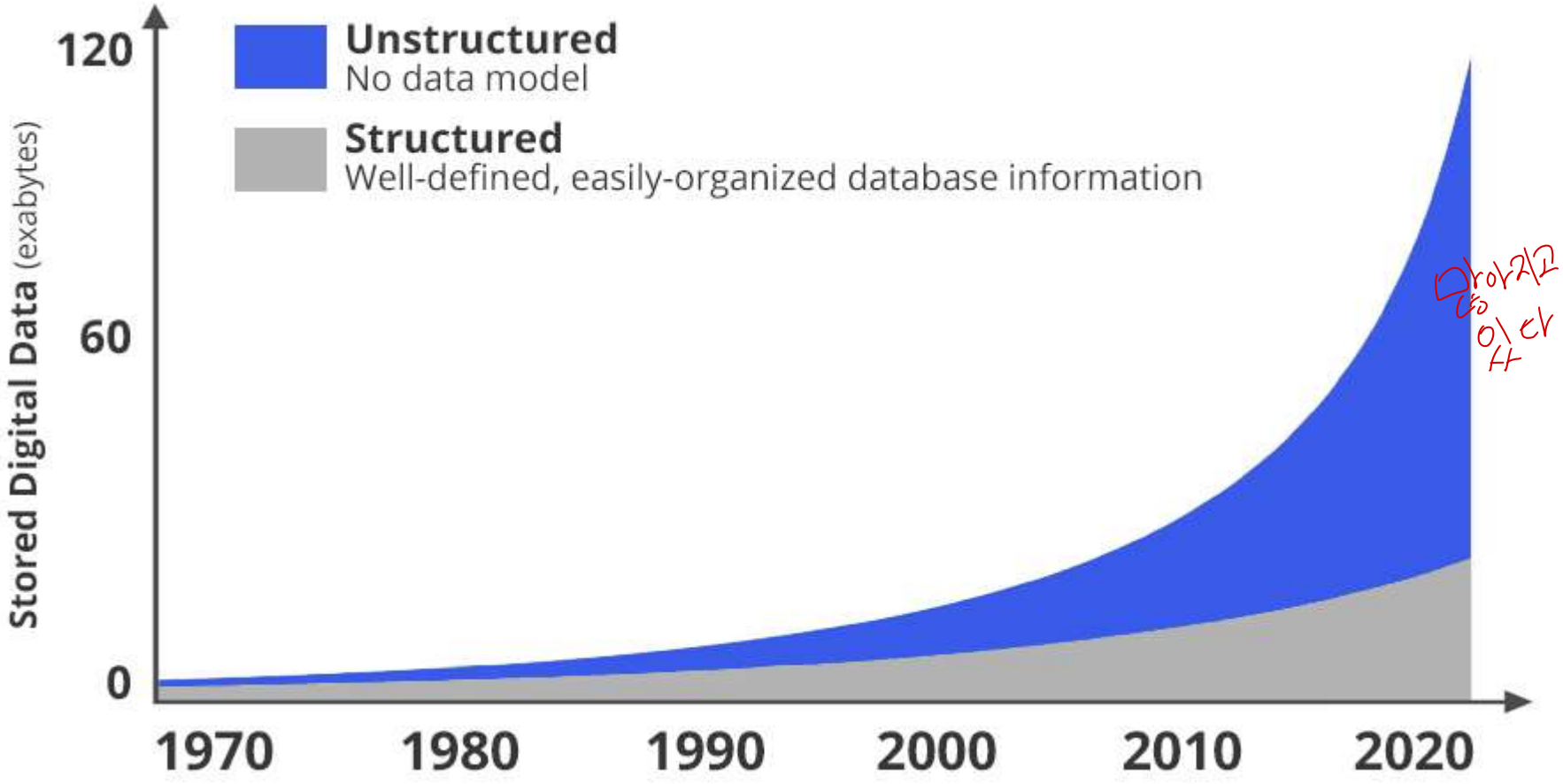
- **Text-based:** Emails, news articles, SNS posts, text files.
- **Non-textual:** Images, audio, video.

- **Challenges:** Lack of structure can make it difficult to search, sort, and analyze.

- **Benefits:** Provides a rich source of information and can be used for various applications like sentiment analysis, customer behavior analysis, and predictive analytics.

Explosion of Unstructured/Semi-Structured Data

- A poor fit for the legacy RDBMS



Graph Source: IDC

Semi-Structured Data </>

- Data that does not conform to a strict, fixed schema, but contains **tags** or **markers** to separate semantic elements and enforce hierarchies of records and fields.
- **Characteristics:**
  - **Self-describing:** contains metadata about its own structure.
  - **Flexible Schema:** Allows for variations in structure, new fields can be added easily.
  - **Hierarchy:** Can represent hierarchical relationships between data elements.
  - **Irregular or Incomplete:** Not all data elements may be present for every record.

Why is Semi-Structured Data Important?

- **Web Data:** Dominant format for data exchange on the internet.
- **Big Data:** Handles the variability and volume of modern data sources.

key-value pair

```
{
  "tweet_id": "RF_pattonoswalt_201311041239",
  "user": {   "username": "pattonoswalt",   "id": "pattonoswalt"  },
  "text": "My dong is super-friendly and loves getting rubbed by children. ",
  "created_at": "Mon Nov 04 12:39:00 +0000 2013",
  "hashtags": [   "CareerEndingTwitterTypos"  ]
}
```

관리하고 싶게 구조를 갖게 변환

XML

서버 간 데이터의 여러 파일포맷 ①

- XML uses tags to mark up text
- E.g.

```
<course>
        <course id> CS-101 </course id>
        <title> Intro. to Computer Science </title>
        <dept name> Comp. Sci. </dept name>
        <credits> 4 </credits>
</course>
```

any name이든 가능

- Tags make the data self-describing
- Tags can be hierarchical

nested 가능

JSON (JavaScript Object Notation)

- Example 1:

```
{
        "ID": ”1111",
        "name": {
                "firstname: "Albert",
                "lastname: "Einstein"
        },
        "deptname": "Physics",
        "children": [
                {"firstname": "Hans", "lastname": "Einstein" },
                {"firstname": "Eduard", "lastname": "Einstein" }
        ]
}
```

- Example 2:

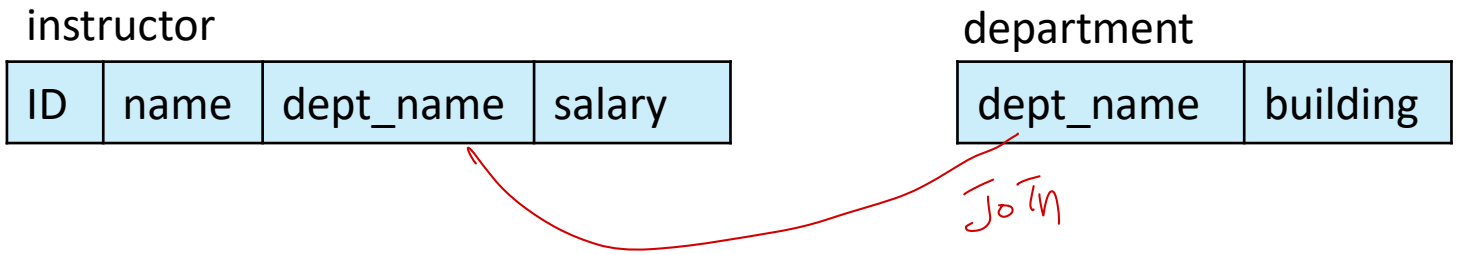
```
{
        "ID": "22222",
        "name": {
                ”Beomseok Nam"
        },
        "deptname": ”Computer Science",
      “e-mail”: “bnam@skku.edu”
}
```

Features of Semi-Structured Data Models

- **Multivalued data types** JSON: INF를 만족시킴 → 멀티밸류 사용가능
  - **Sets**, **multisets**
    - E.g.,: set of interests {‘soccer’, ‘cooking’, ‘game’, ‘rock’}
  - **Key-value map**
    - Store a set of key-value pairs
    - E.g., {(brand, Apple), (ID, MacBook), (size, 13), (color, silver)}
    - Operations on maps: *put*(key, value), *get*(key), *delete*(key)
  - **Arrays**
    - Widely used for scientific and monitoring applications

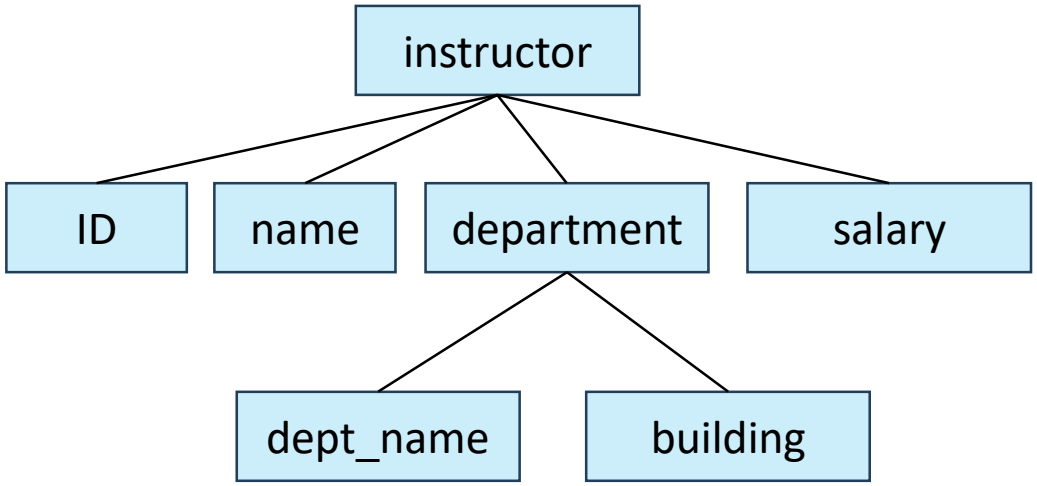
1st Normal Form vs. Nested Data

- 1st Normal Form RDBMS



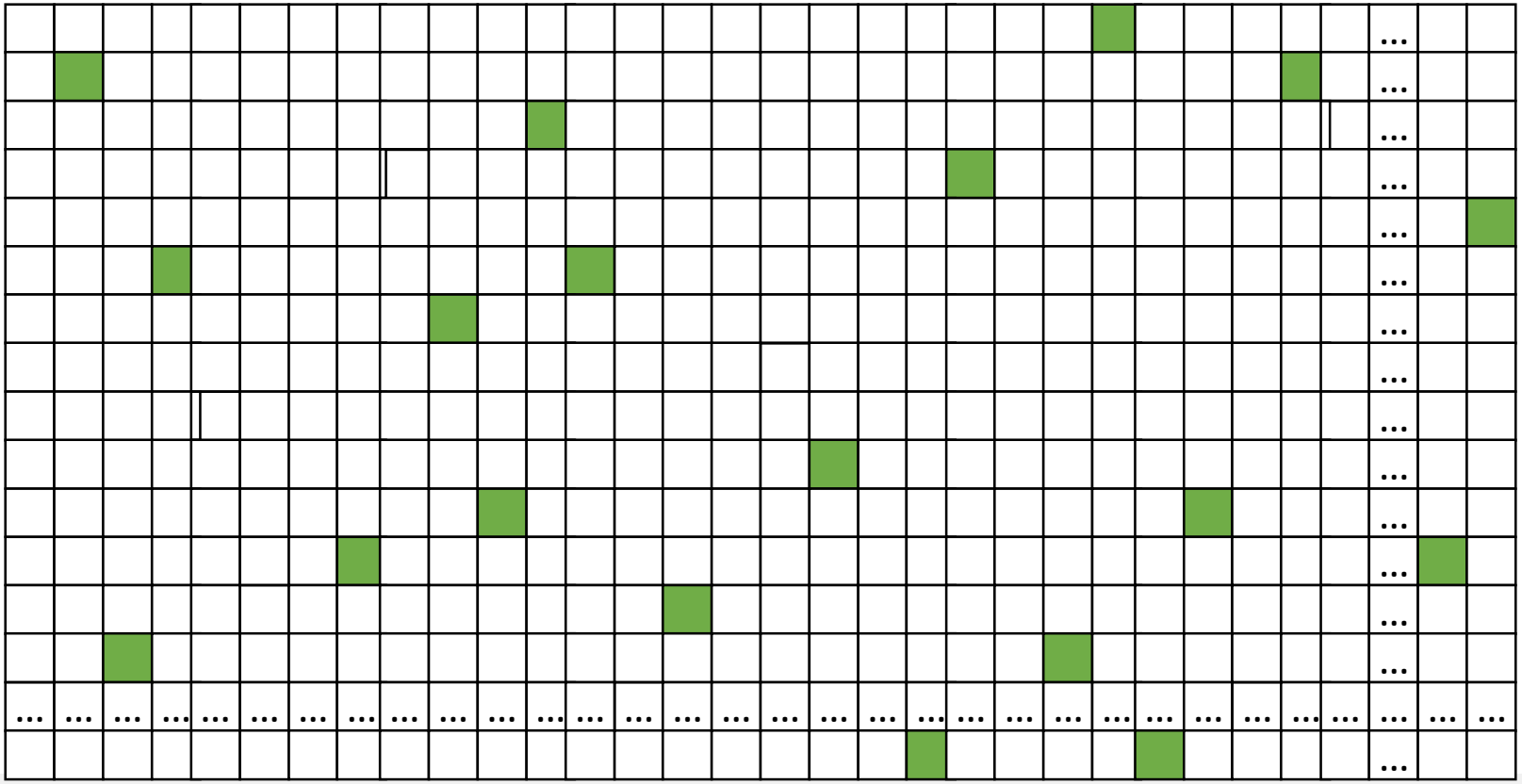
Vs.

- Nested Data (JSON)



Data Model for Semi-Structured Data

- Structured Table for Semi-Structured Data?
  - Does not allow schema changes
  - Too many columns
  - Sparse tables

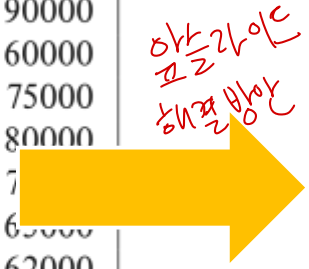


Data Model for Semi-Structured Data

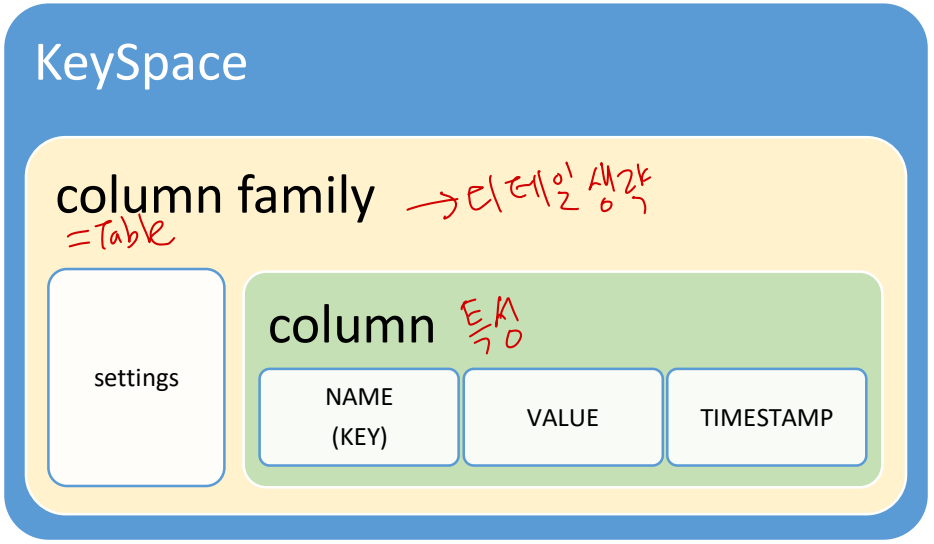
- Structured Table for Semi-Structured Data?
  - Does not allow schema changes
  - Too many columns
  - Sparse tables

Structured Table (Schema)

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	7
10101	Srinivasan	Comp. Sci.	6
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



Key-Value Stores (Schema-less)

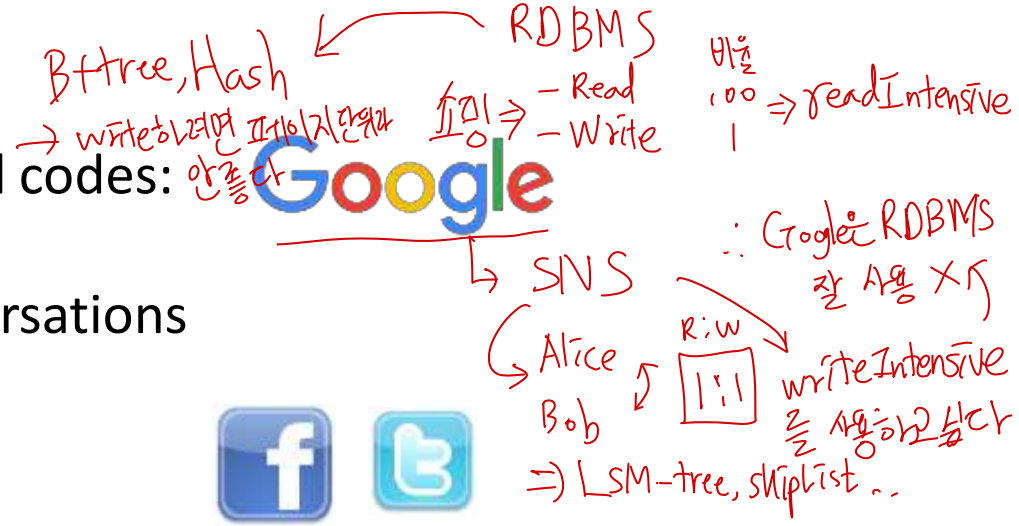


Key-Value Store (KVStore)

- KV-Stores seem very simple indeed
  - They are nothing but multi-threaded indexing systems
  - A simpler and more scalable "database"

- Interface
  - `put(key, value); // insert "value" associated with "key"`
  - `value = get(key); // read data associated with "key"`

- Examples
  - Google BigTable & internal codes:
    - Key: hangoutID
    - Value: Hangout conversations
  - Facebook, Twitter:
    - Key: UserID
    - Value: user profile (e.g., posting history, photos, friends, …)



When to use Key-Value Stores over RDBMS?

- If workload is write-intensive
- If strong consistency is not required

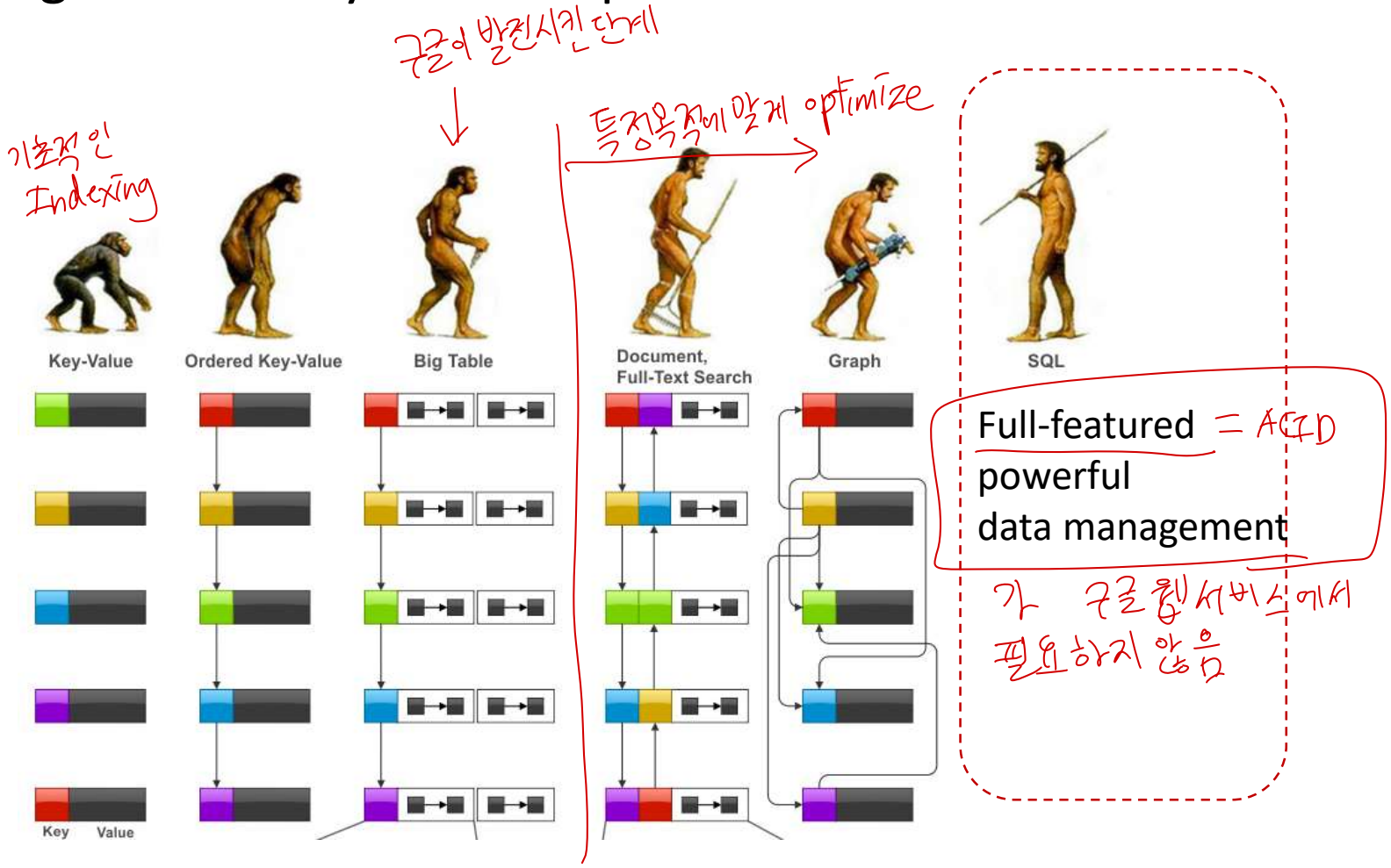
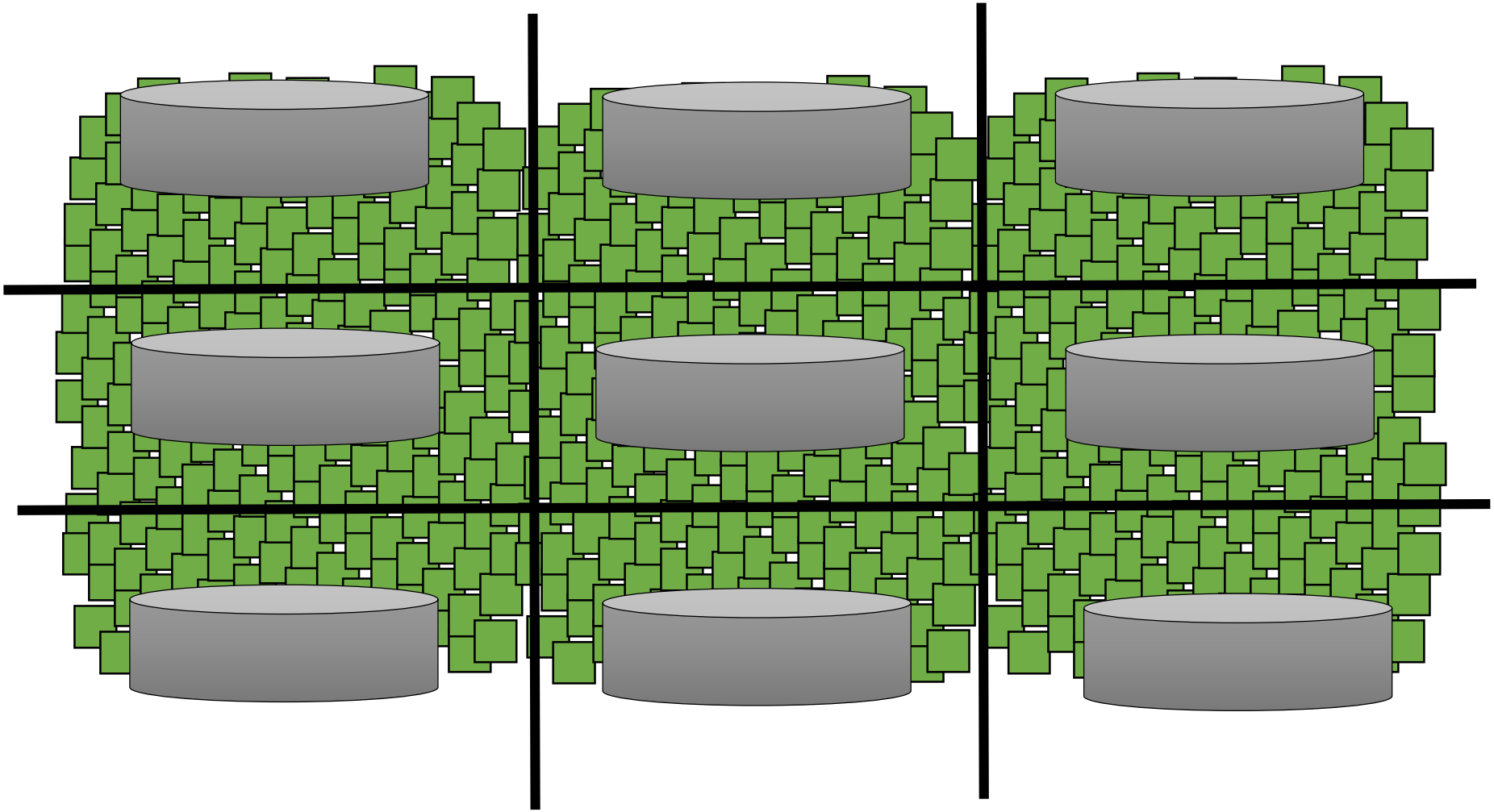


Image Source: From RDBMS to Key-Value Store: Data Modeling Techniques- Wishmitha S. Mendis

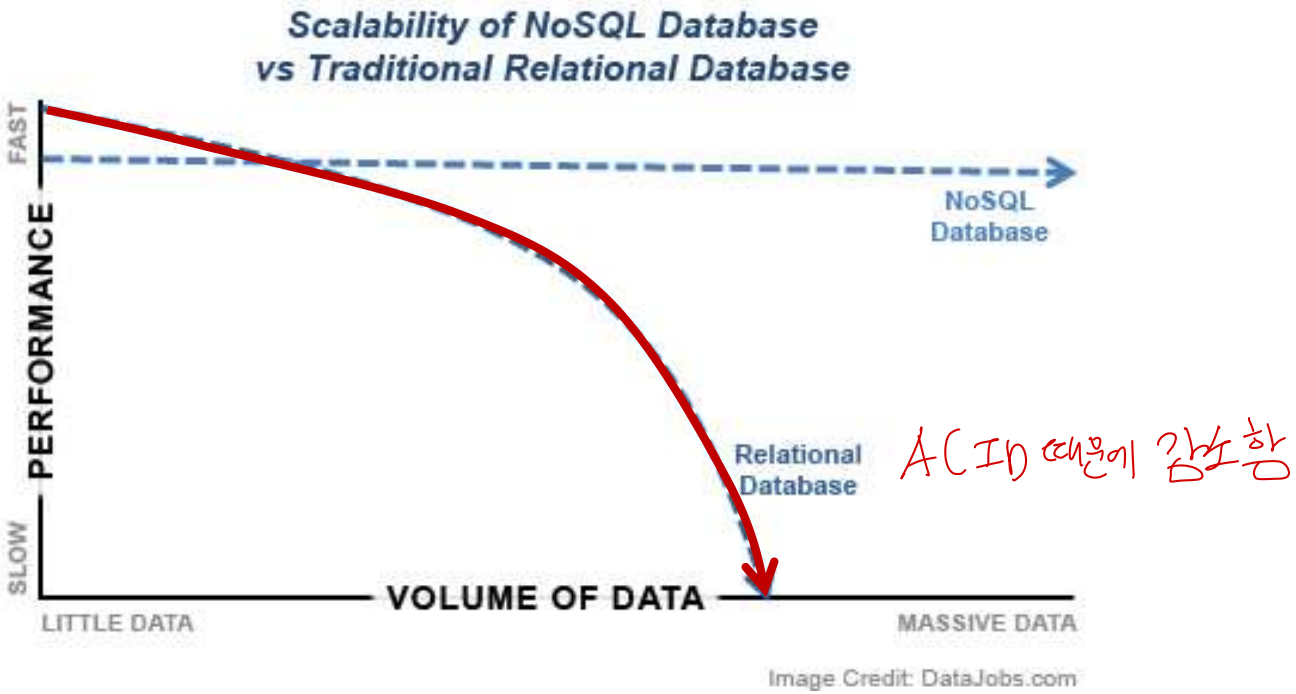
How to Manage Huge Datasets?

- Divide and Conquer! (하나에 담기엔 너무많다)

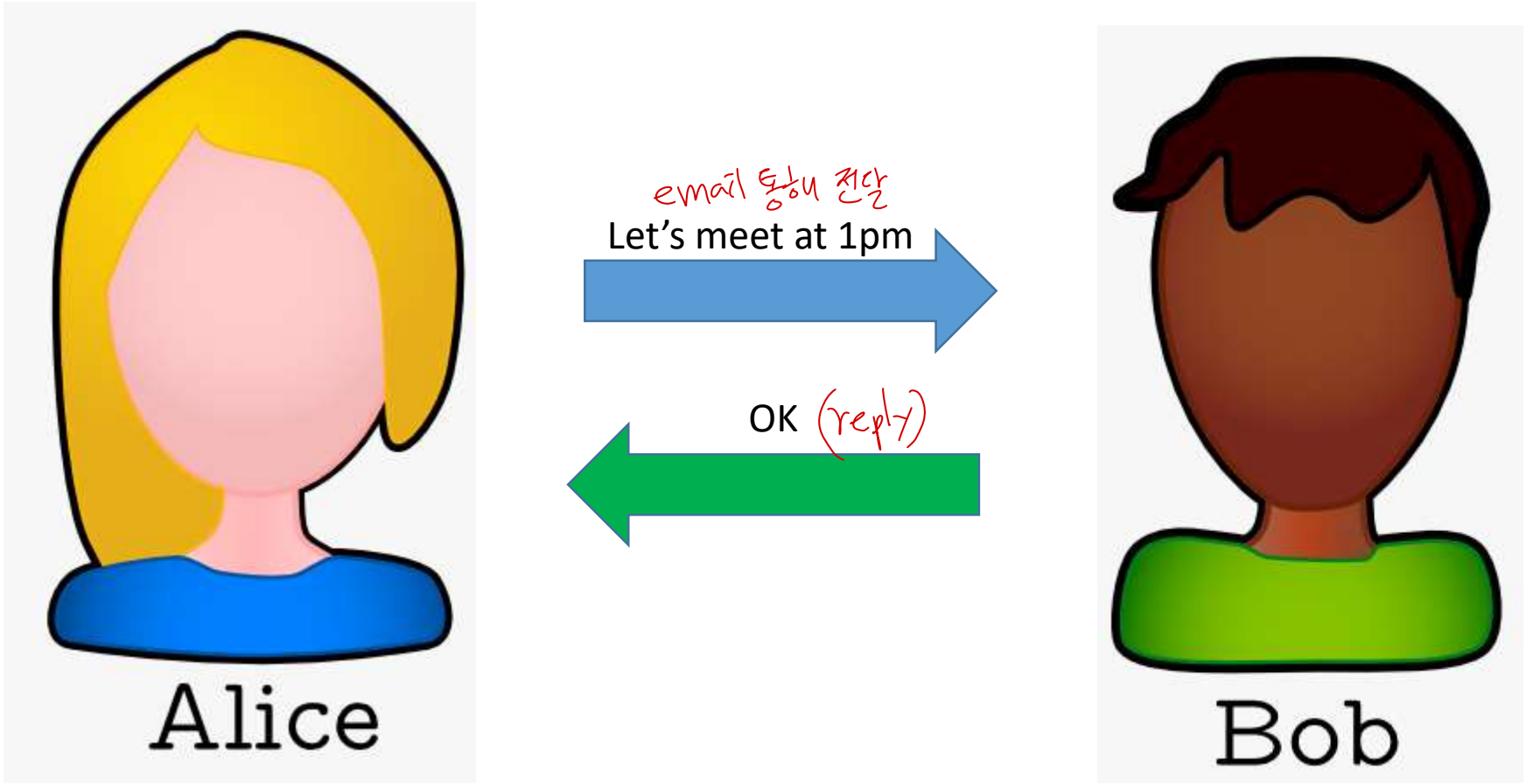


Distributed and Parallel DBMS

- Not scalable because “consistency” is hard to achieve in large scale systems

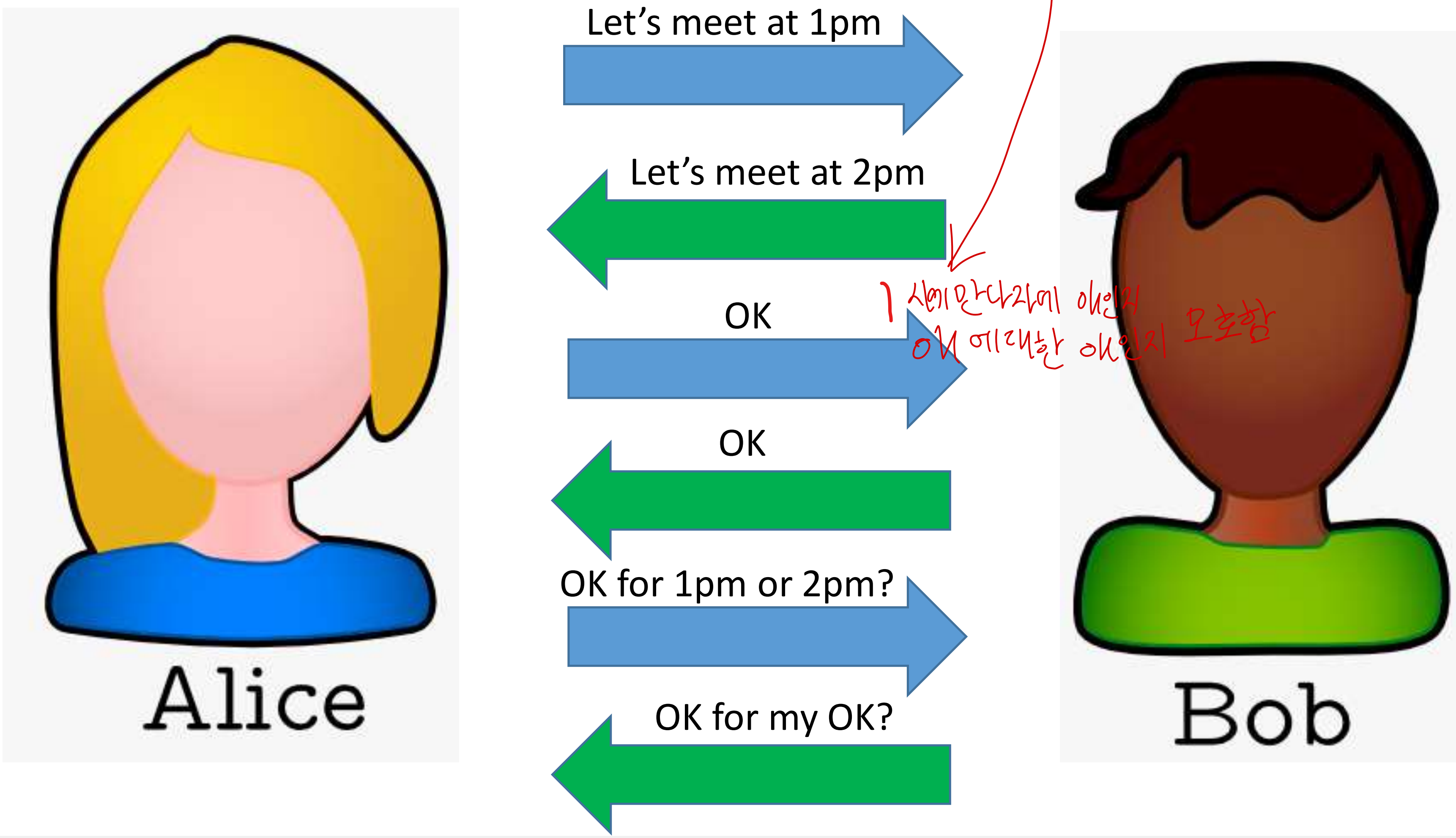


Distributed Consensus is a Hard Problem



Distributed Consensus is a Hard Problem

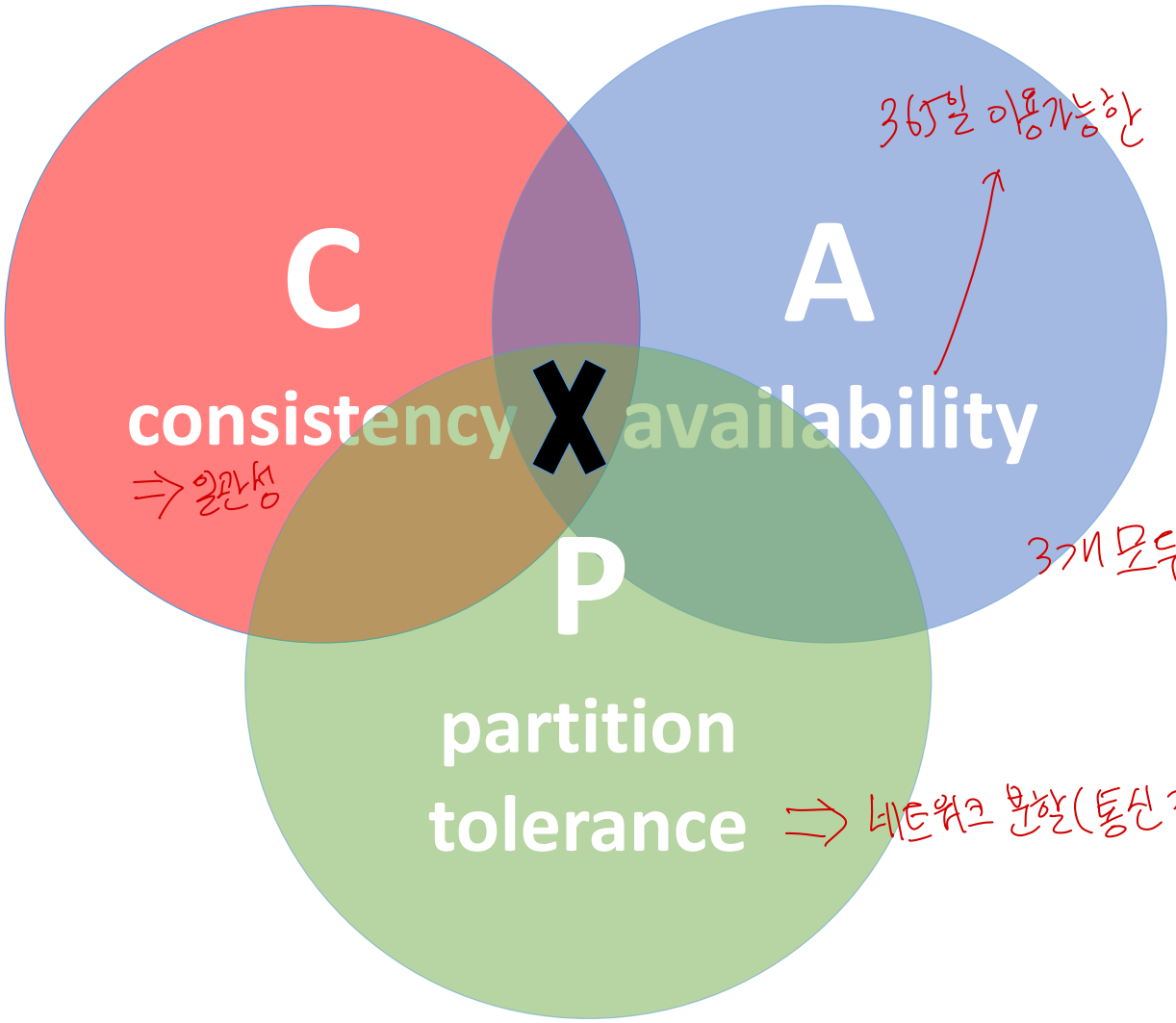
Synchronization clock이 없어서
메세지들간의 정보가 모호해진다



Agreement with Unreliable Message Channel

- In distributed systems, network is unreliable
- Then, distributed consensus becomes even harder.
  - Alice → Bob : Let's meet at noon on the 2nd floor
  - Alice ← Bob : Ok!
  - Alice (What if Bob doesn't know that I received his message?)
  - Alice → Bob : I received your message, so it's ok.
  - Bob (What if Alice doesn't know that I received her message?)
  - …

CAP Theorem



3가지 이용가능한

⇒ 일관성

3개 모두 고려하는 것은 힘듦

⇒ 네트워크 분할(통신 장애) 에도 시스템 동작

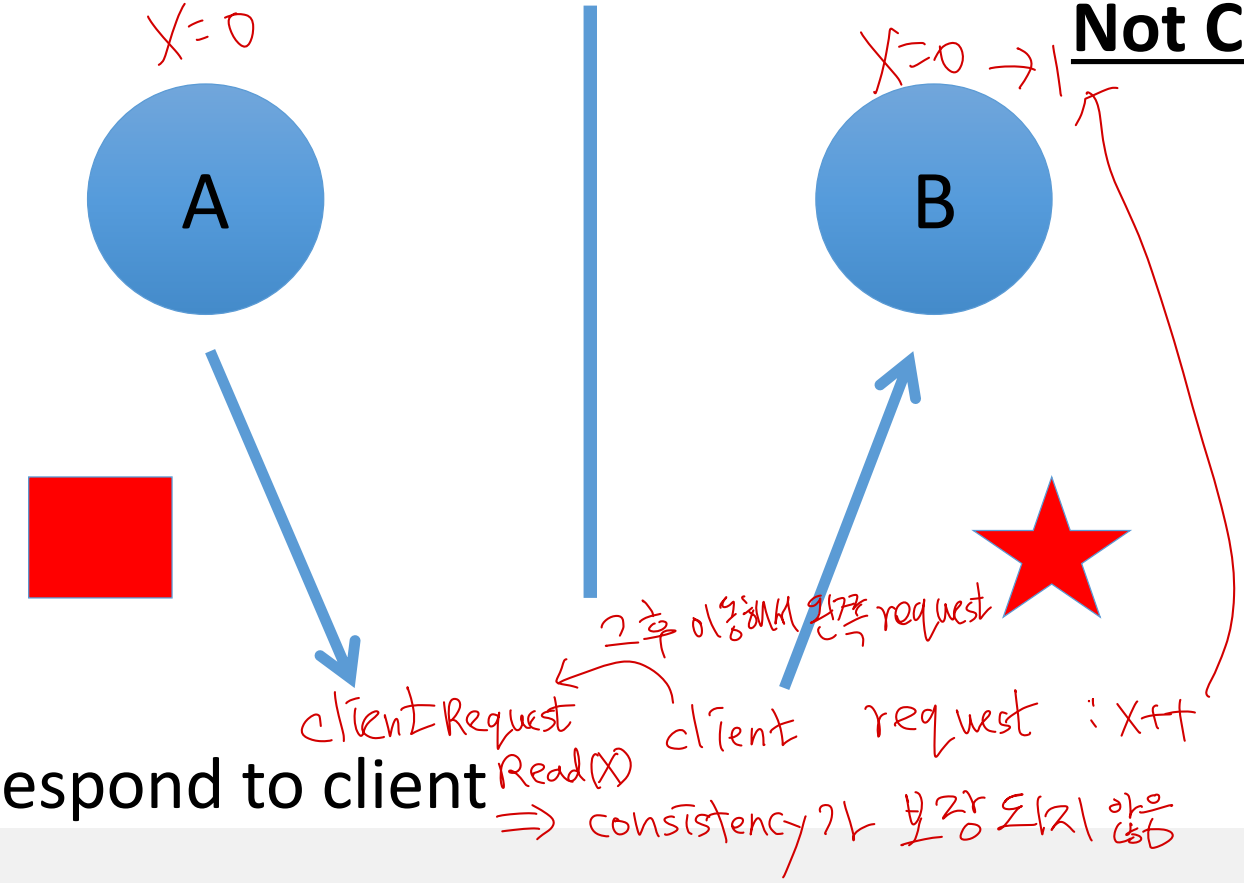
CAP Theorem

- **C**onsistency:
  - All nodes should see the same data at the same time
- **A**vailability:
  - Node failures do not prevent survivors from continuing to operate
- **P**artition-tolerance:
  - The system continues to operate despite network partitions
- A distributed system can satisfy any two of these guarantees at the same time **but not all three**

CAP Theorem: AP Systems sacrifice C

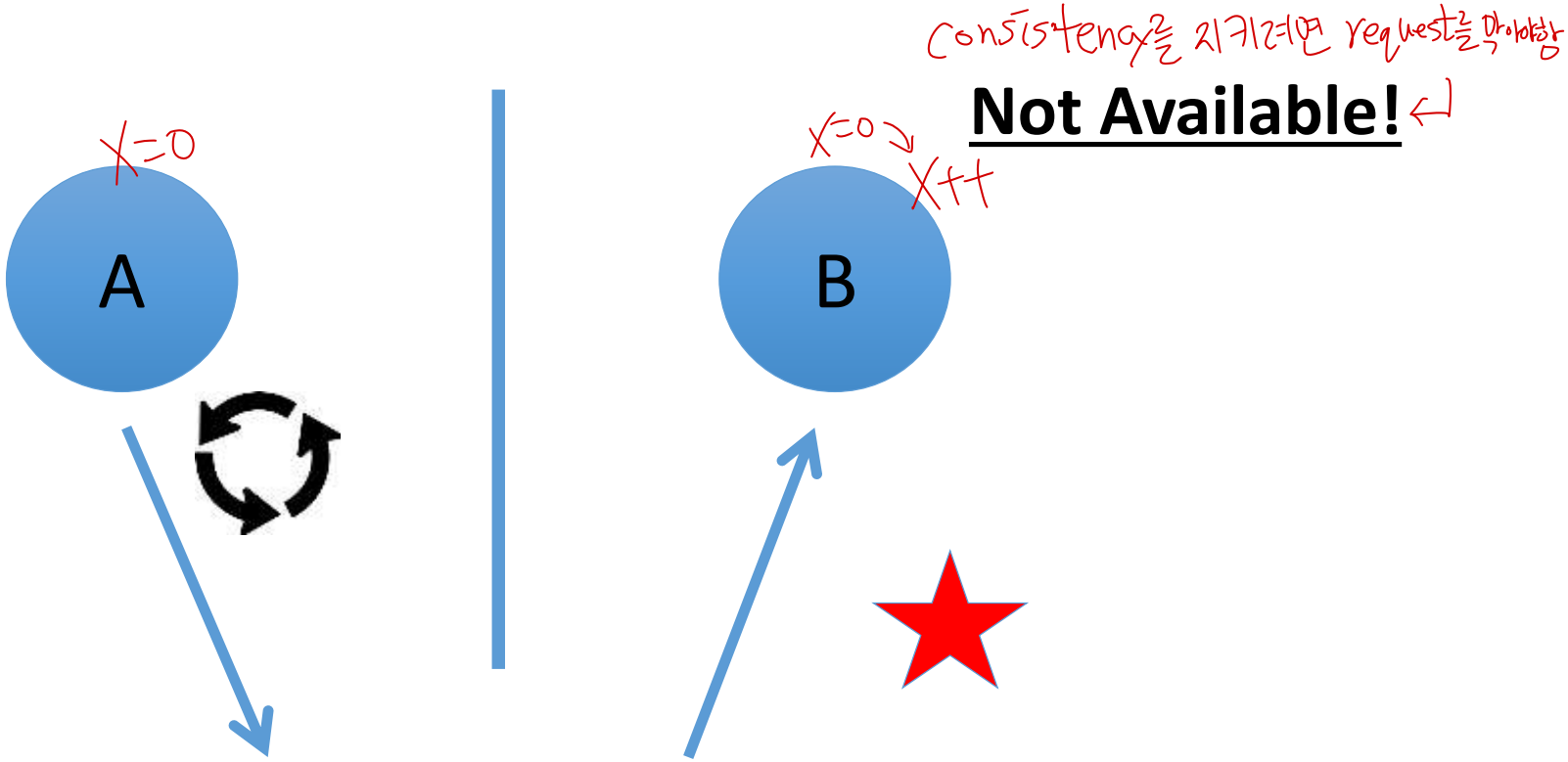
- To make a distributed system ***available*** when a network ***partition occurs***, ***consistency*** must be given up.

partition은 전제조건



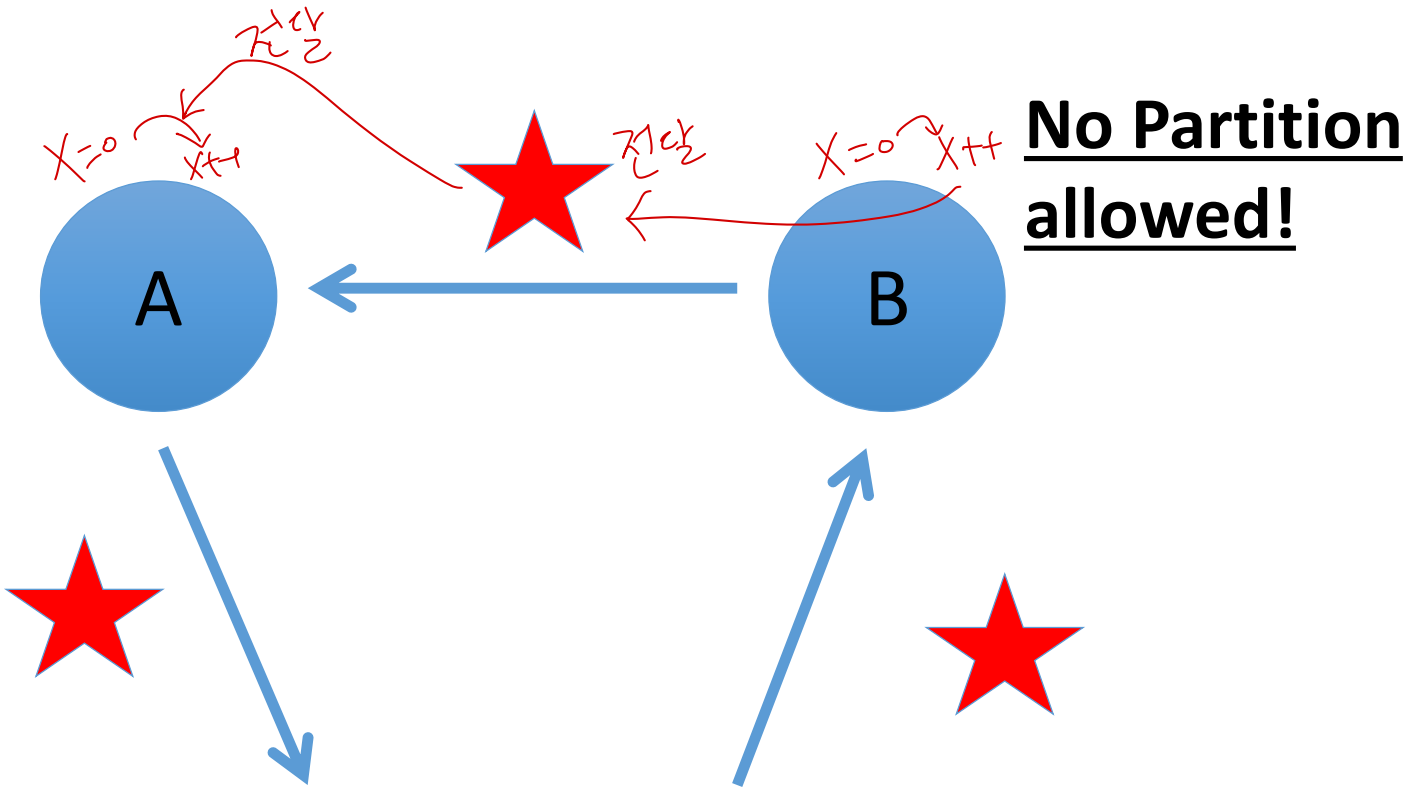
CAP Theorem: CP Systems sacrifice A

- To make a distributed system ***consistent*** when a network ***partition occurs***, one of the partitions must not be ***available.***



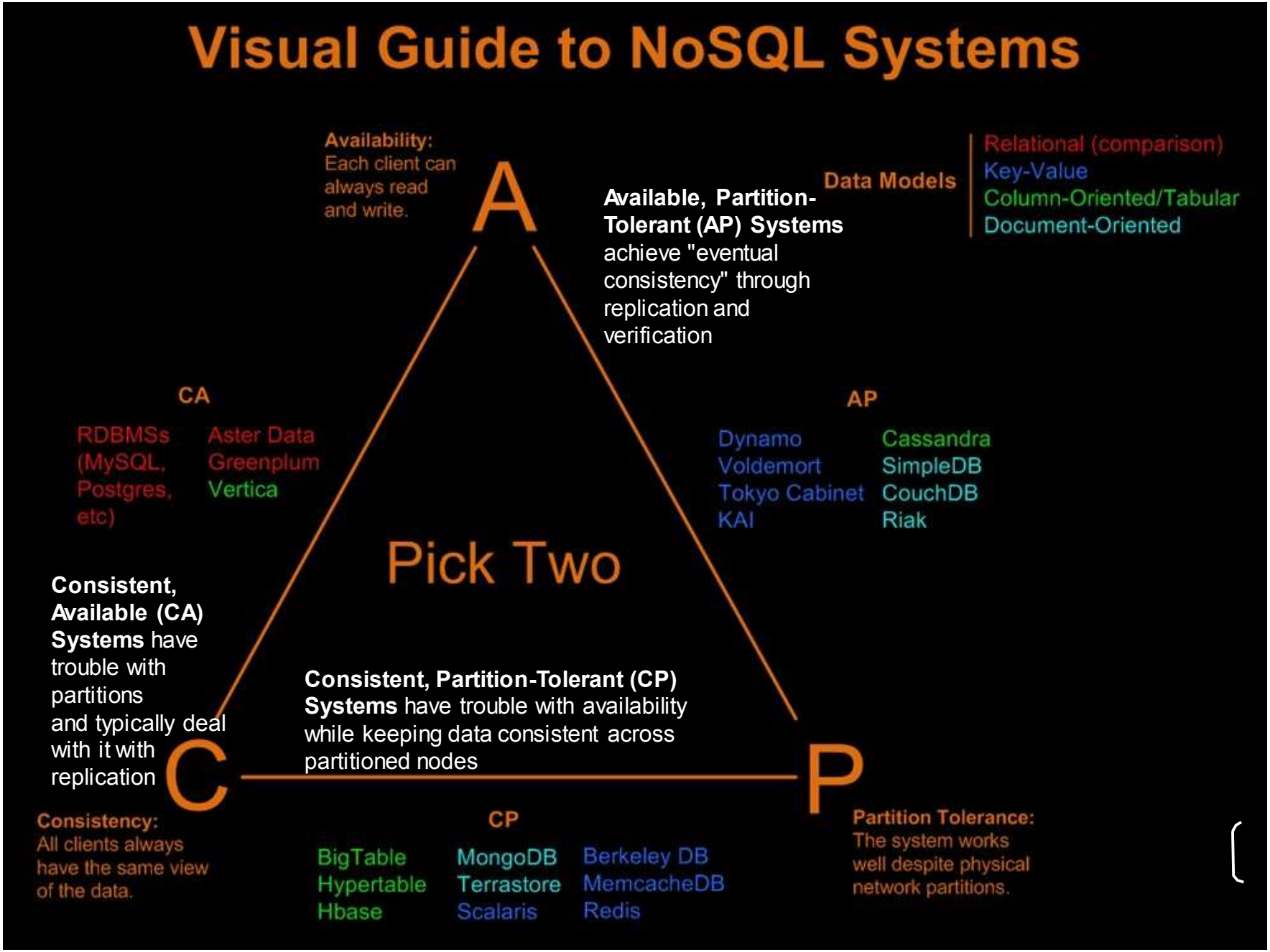
CAP Theorem: CA Systems sacrifice P

- To make a distributed system ***consistent*** and ***available*** there must not be partition.



A gets updated from B

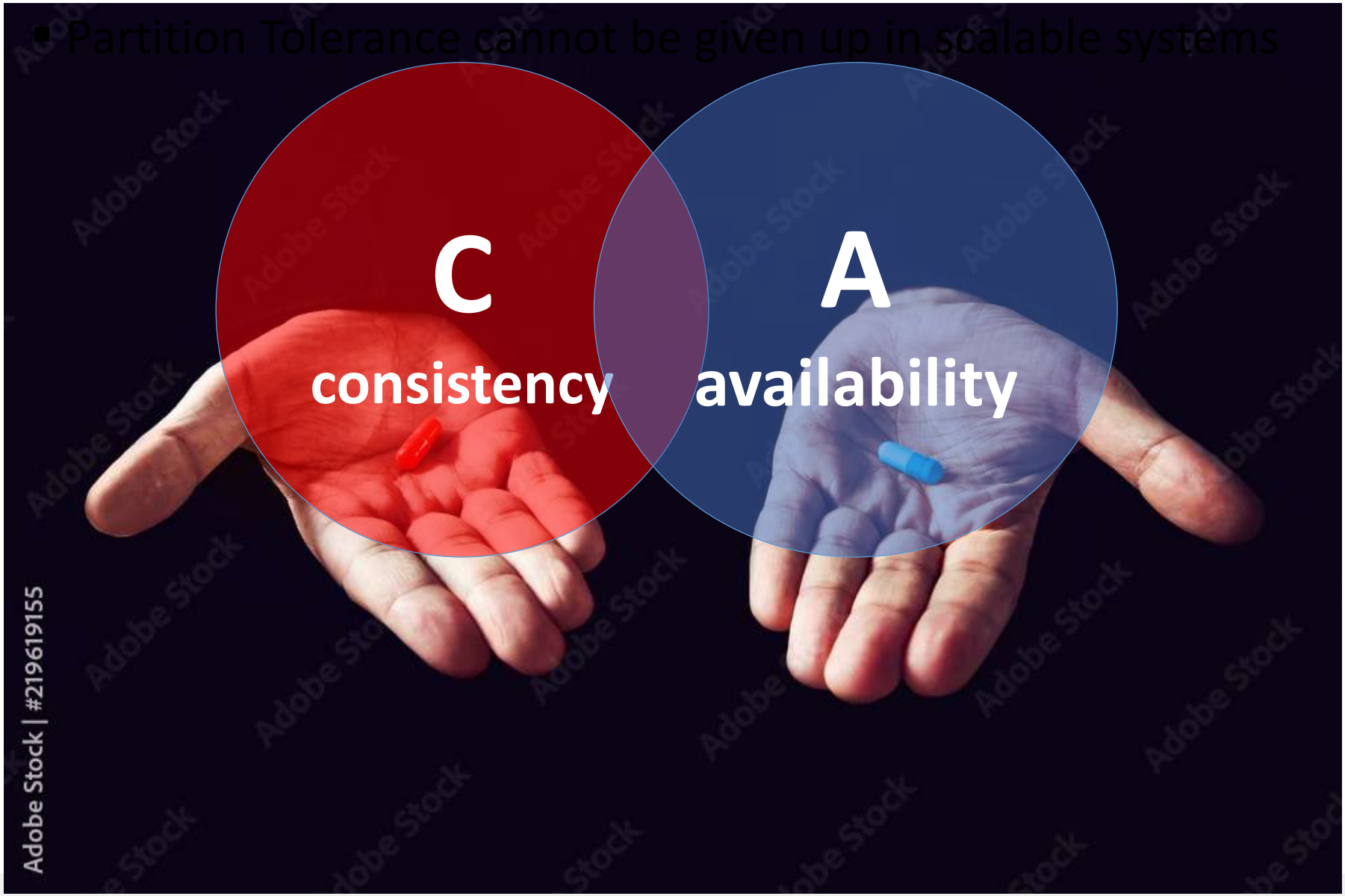
Visual Guide to NoSQL Systems



http://blog.nahurst.com/visual-guide-to-nosql-systems

Which one would you sacrifice?

- Partition Tolerance cannot be given up in scalable systems



Sacrificing Availability → Financial Penalty

Consistency 중시!

삼성화재

시스템 성능 개선을 위해
홈페이지 서비스가 일시 중단 중입니다.
10월 10일(화) 낮 12시에 다시 오픈할 예정입니다.
이용에 불편을 드려 대단히 죄송합니다.

시스템 중단기간 중, 아래의 자동차보험 업무에 한하여
제한적으로 서비스가 제공되며, 그 외의 경우는 홈페이지 오픈 후
다시 방문하셔서 이용해 주세요.

중단기간 2017년 9월 29일(금) 밤 10시 ~ 10월 10일(화) 낮 12시

제주특별자치도 홈페이지 및 패밀리사이트

서비스 일시 중단 안내

카카오뱅크는 점검 중

더욱 안정적인 서비스를 제공하기 위해
시스템 점검을 진행하고 있습니다.
점검이 완료된 후 다시 접속해주세요.

- 점검대상 카카오뱅크 전체 서비스
- 점검시간 2020년 6월 28일(일) 02:00 ~ 09:00 (7시간)

서비스 점검 사전 안내

안정적인 서비스 제공을 위해 시스템 점검 예정입니다.
점검 이후 자동 로그인이 해제되니 참고해 주세요.

계정 확인 : 프로필 사진 → MY 원티드

10월 17일 오전 1시 – 5시 (약 4시간 소요 예상)

계정 확인하기

NAVER

잠시 후 다시 확인해주세요!

지금 이 서비스와 연결할 수 없습니다.
문제를 해결하기 위해 열심히 노력하고 있습니다.
잠시 후 다시 확인해주세요.

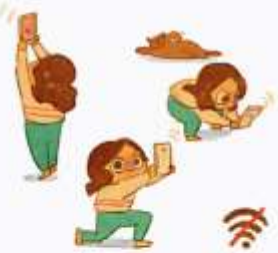


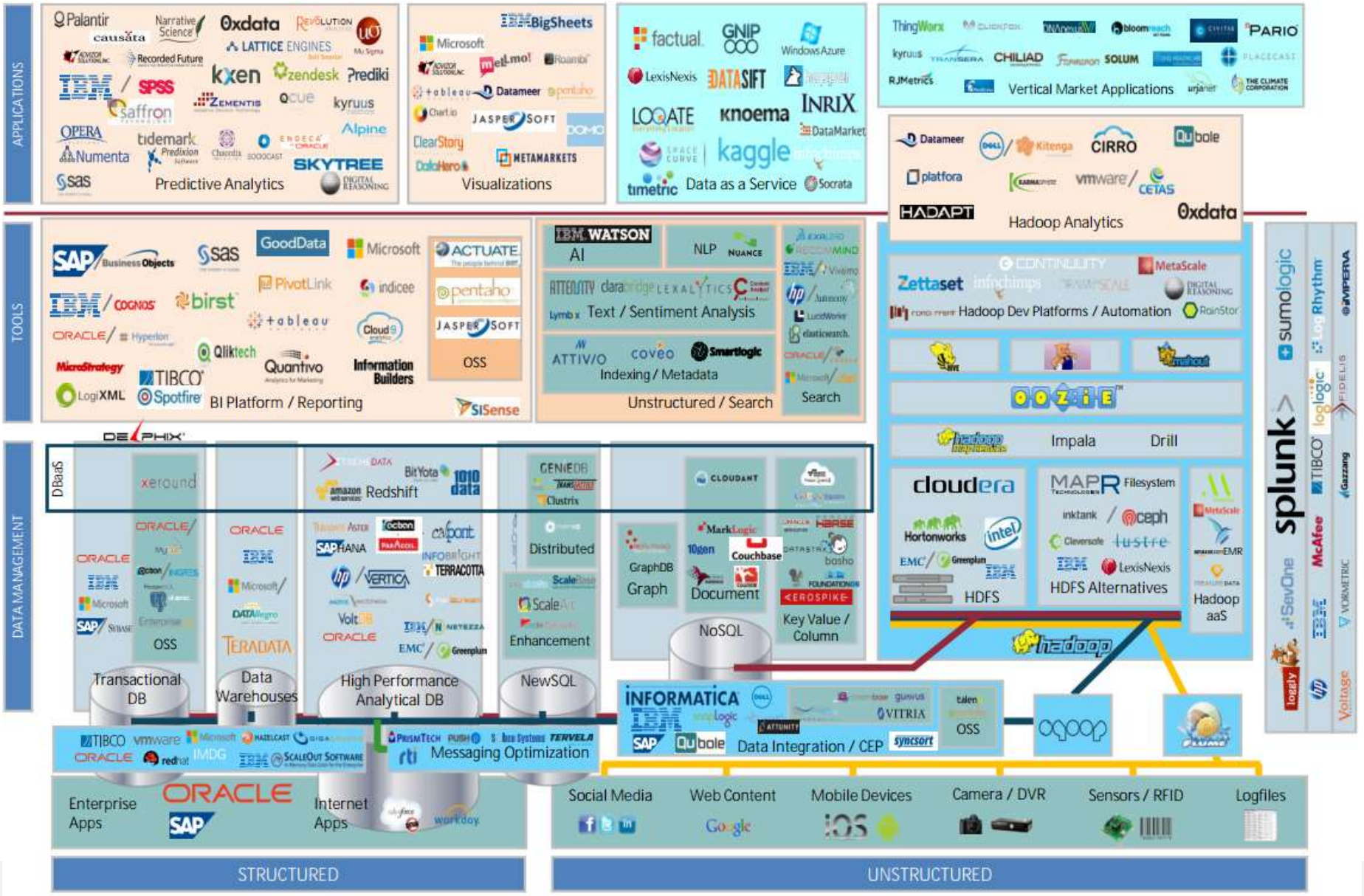
Illustration by 여일

이전 페이지 | 네이버쇼핑 홈

Eventual Consistency : B2C (Business to Consumer)

- Facebook example:
  - Bob finds an interesting story and shares with Alice by posting on her Facebook wall
  - Bob asks Alice to check it out
  - Alice logs in her account, checks her Facebook wall but finds nothing is there.
- What can they do?
  - Nothing. They use the SNS for free
- Eventual consistency is not for B2B

So many data processing engines out there…



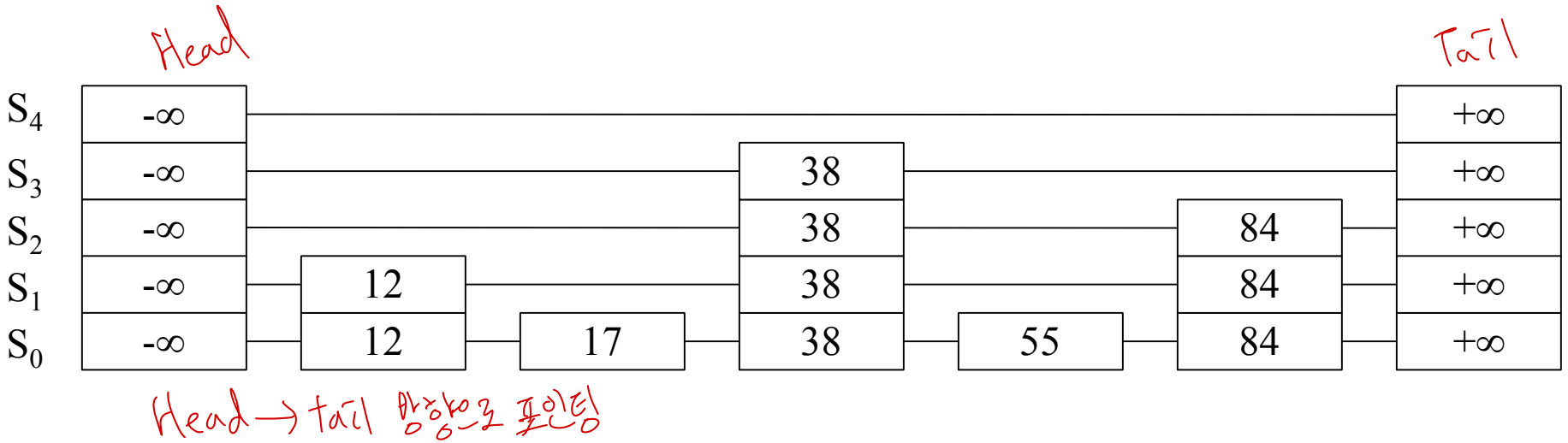
Skip Lists

Skip Lists (a.k.a Skiplist)

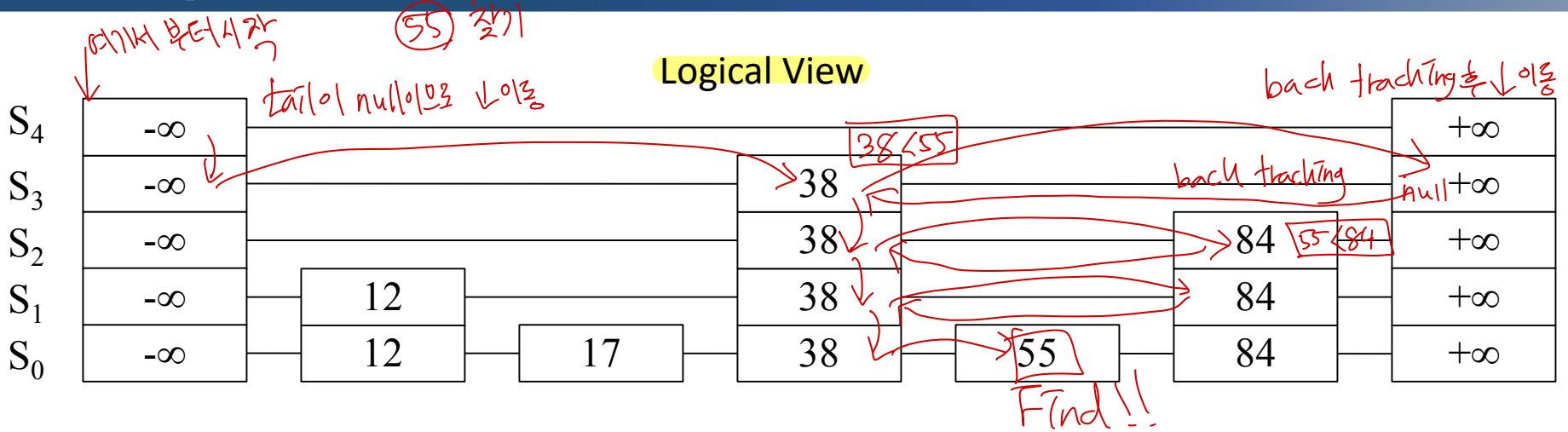
- Binary Search Trees: O(log n)
  - If, and only if, the tree is “balanced”
- Insertion of partially sorted data is optimally bad for binary search trees.
- Skip Lists make use of some randomization to maintain O(log n) search and update times
  - These are on average figures
- Analysis of skip lists is interesting
  - Worst case is potentially very bad
  - We will not cover the analysis here.

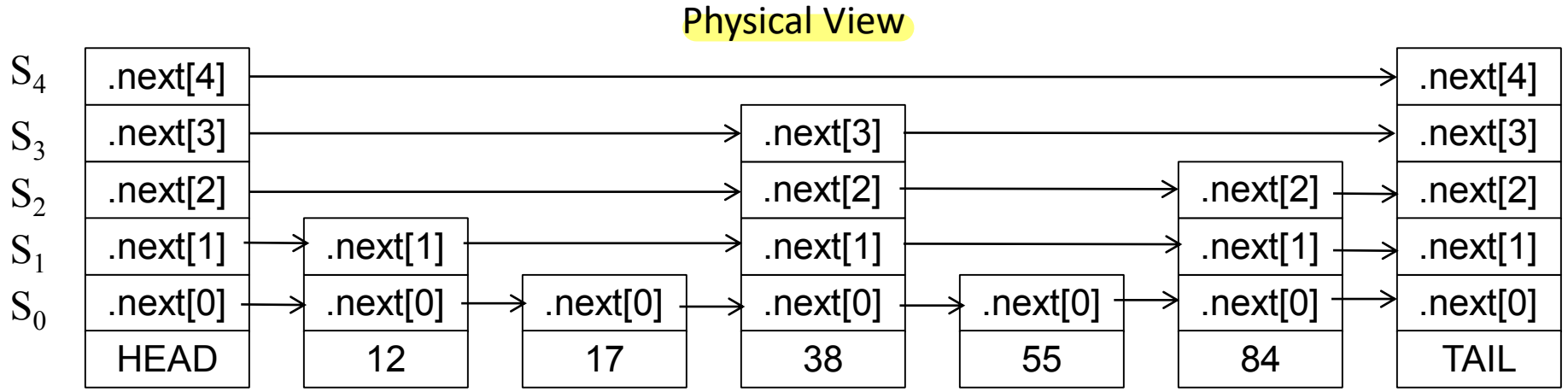
Skip Lists (SkipList)

- A Skip List is a series of lists: $\{S_0, S_1, \ldots, S_h\}$ <Invariant 불변> ; 상위노드가 하위노드의 sub set
  - S_0 contains all the elements
  - S_i contains a randomly chosen subset of elements from S_{i-1}
  - Each list S_i is sorted
- We introduce two special elements: -∞ & +∞
  - Each list S_i contains these, S_h contains only these.



Skip Lists


Logical View
S_4
S_3
S_2
S_1
S_0
-∞
12
17
38
55
84
+∞
Find!!


Physical View
S_4
S_3
S_2
S_1
S_0
.next[4]
.next[3]
.next[2]
.next[1]
.next[0]
HEAD
12
17
38
55
84
TAIL

Skip List - Informal

- We set the lists up so that:
  - An item that is in S_i is in S_{i+1} with probability 1/2 (or 1/4, 1/8, etc)
- So S_1 has about n/2 items in it
  - S_2 has about n/4 items
  - S_i has about $n/(2^i)$ items
- The height of the skip list is about $\log_2(n)$

S_{i+1}/S_i 값 ↓ 1/2 즉, 1보다 더작은 분수임

$\frac{n}{2^H} = 1$ 일때까지 올라감 $\Rightarrow H = \log_2 n$

$\therefore$ 시·복 : $O(\log n)$

최상위레벨에서 노드 n개를 skip 했다면 리프노드에선 $n \times 2^{\log_2 n}$ 개를 skip한 것임

NOTE:

- Unlike a binary search tree, this "halving" property is not enforced
  - It is approximate

Skip List Operations

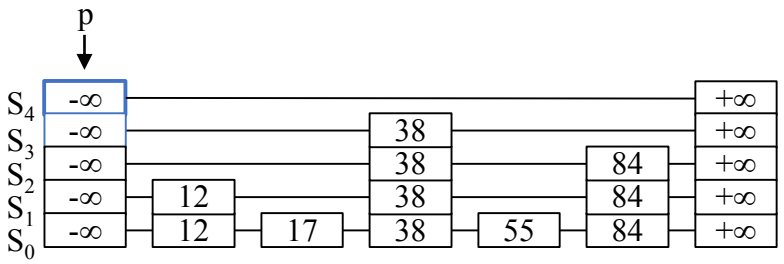
- We use a positional abstraction
- Operations
- After(p) – position after p on the same level
- Before(p) – position before p on the same level
- Below(p) – position below p in the same tower
- Above(p) – position above p in the same tower

Searching

Simple: binary search (modified)

- The upper level lists provide something like a “thumb index”

- Algorithm for searching for the key k in a skip list
  - Returns p = reference to the key in the list which is equal or larger than k



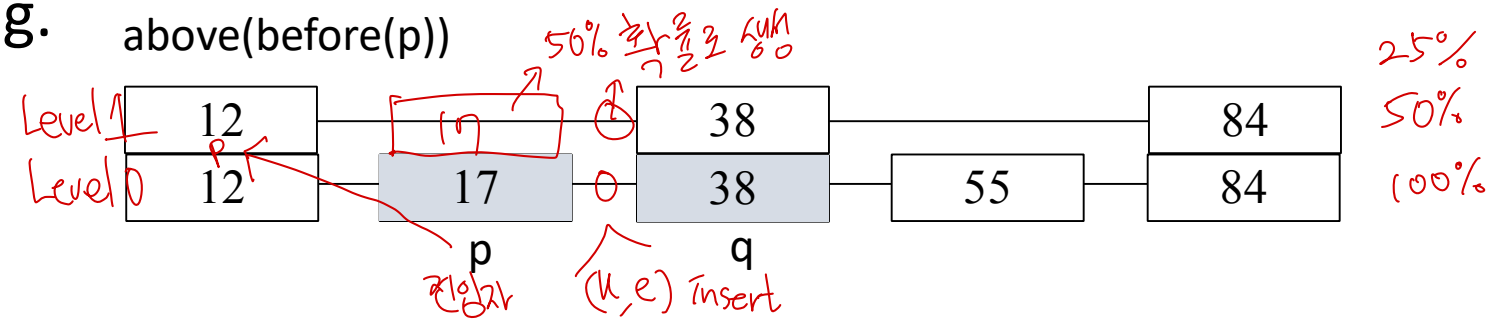
p ← first element of S_{max_height}
while below(p) ≠ **null do**
p ← below(p) {drop down}
while key(after(p)) ≤ k **do**
p ← after(p) {scan forward}

Inserting

- Uses insertAfterAbove(p,q,(k,e)) which inserts (k,e) after p and above q.

```
p ← SkipSearch(k)
q ← insertAfterAbove(p,null,(k,e))
while random() < ½ do
        while above(p) = null do
                p ← before(p) {scan backwards}
        p ← above(p)            {jump up}
        q ← insertAfterAbove(p,q,(k,e))
```

- E.g.



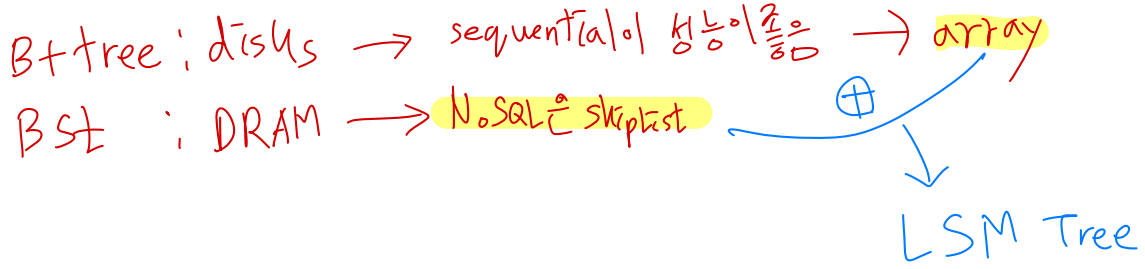
Deleting

- Removal of an item is easier than insertion
- We simply find the item (SkipSearch(k))
  - Then we remove this from the list S_0 지우고 연결 새로 (링크드리스트와같)
  - We recursively remove it from the lists S_i above this
    - This is simply deletion from a linked list, which is simple...

note

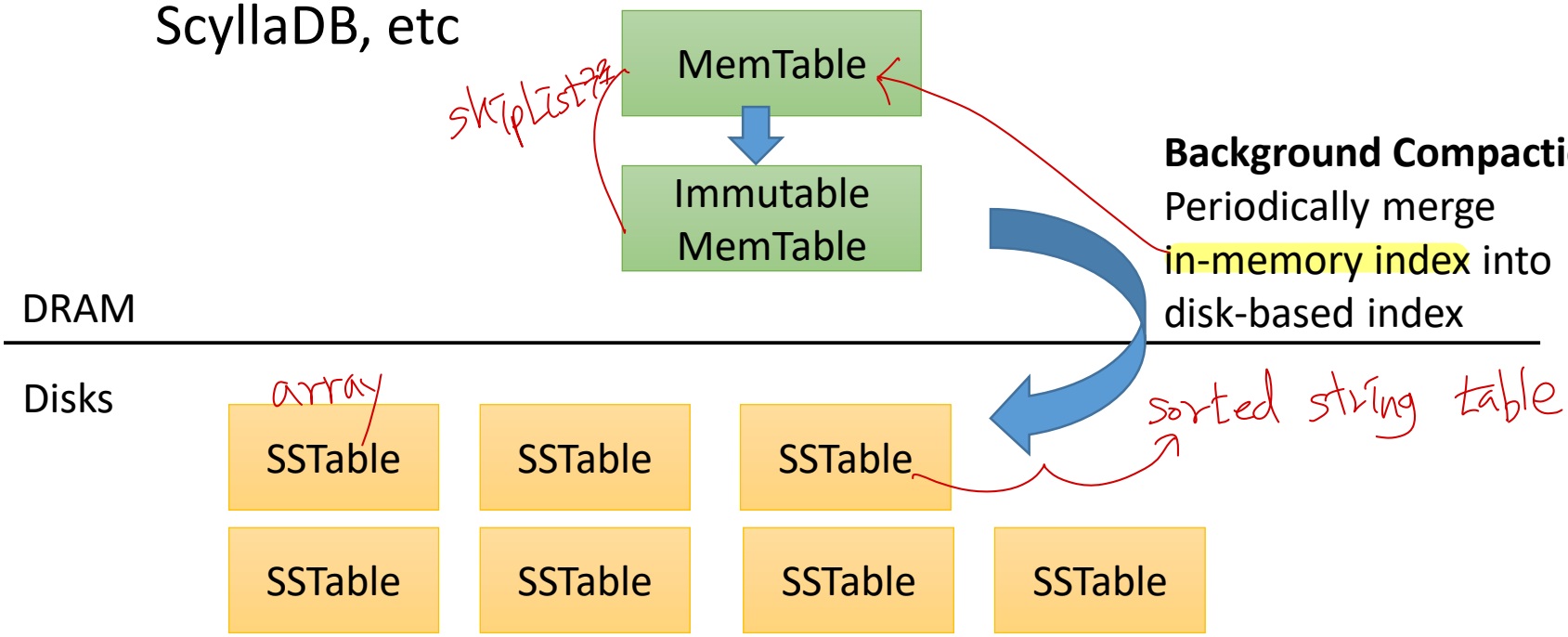
- The *above* and *before* methods can be eliminated
  - We can always insert, search and delete using a simple scan-forward-and-down regime
- Storage overhead is therefore just a (key,next,down) triple for each tower element...

Chap. 24 Log-Structured Merge Tree

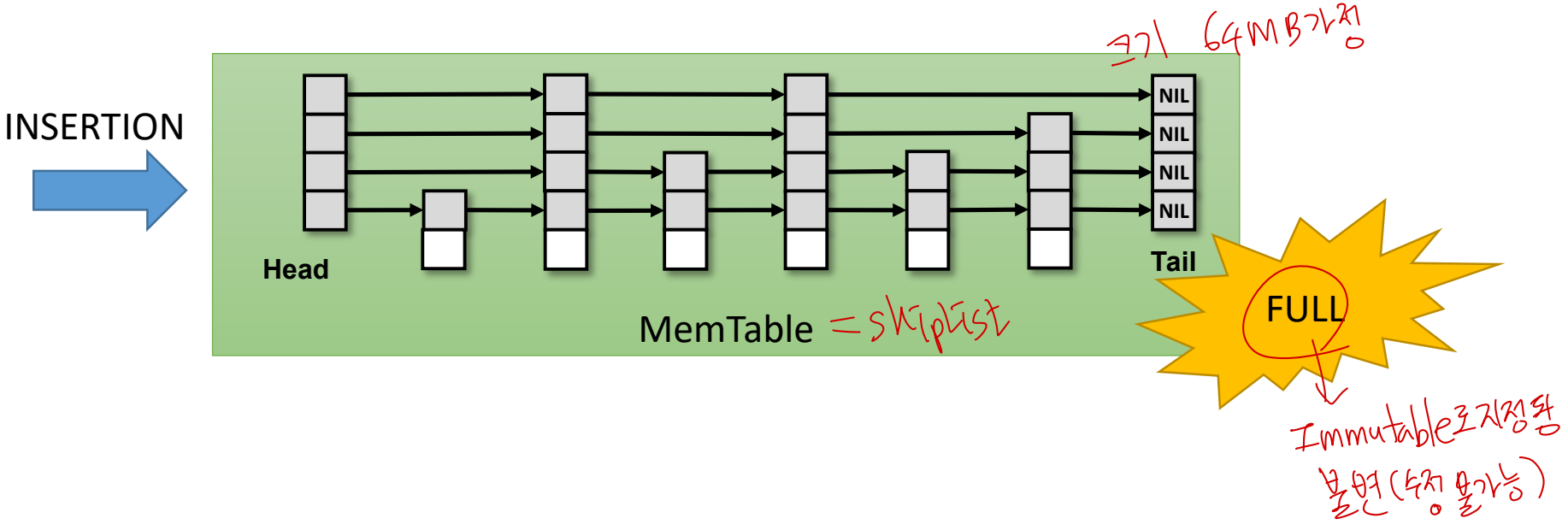


Log-Structured Merge-Tree

- Designed for Write-intensive workloads
  - Fast insertion
  - Moderate search performance
- Widely used in various key-value stores
  - BigTable, Hbase, Cassandra, RocksDB, WiredTiger, InfluxDB, ScyllaDB, etc



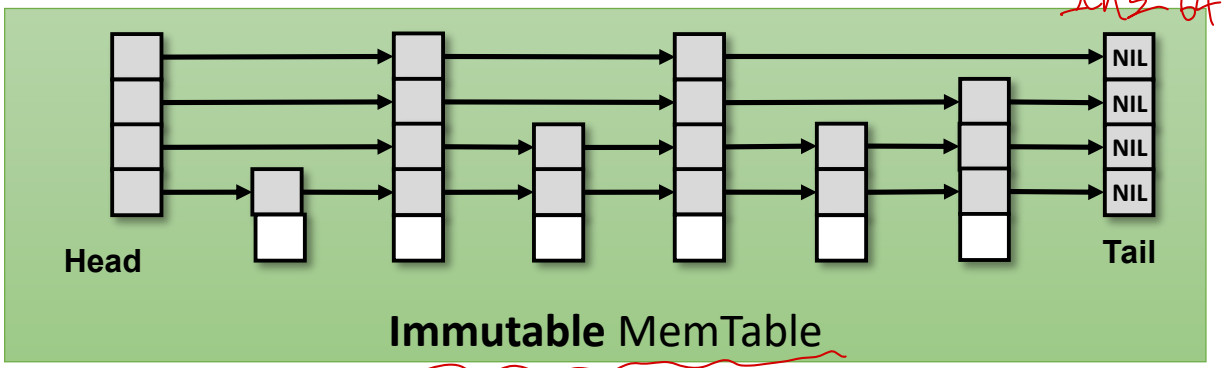
Log-Structured Merge-Tree



DRAM

Disks

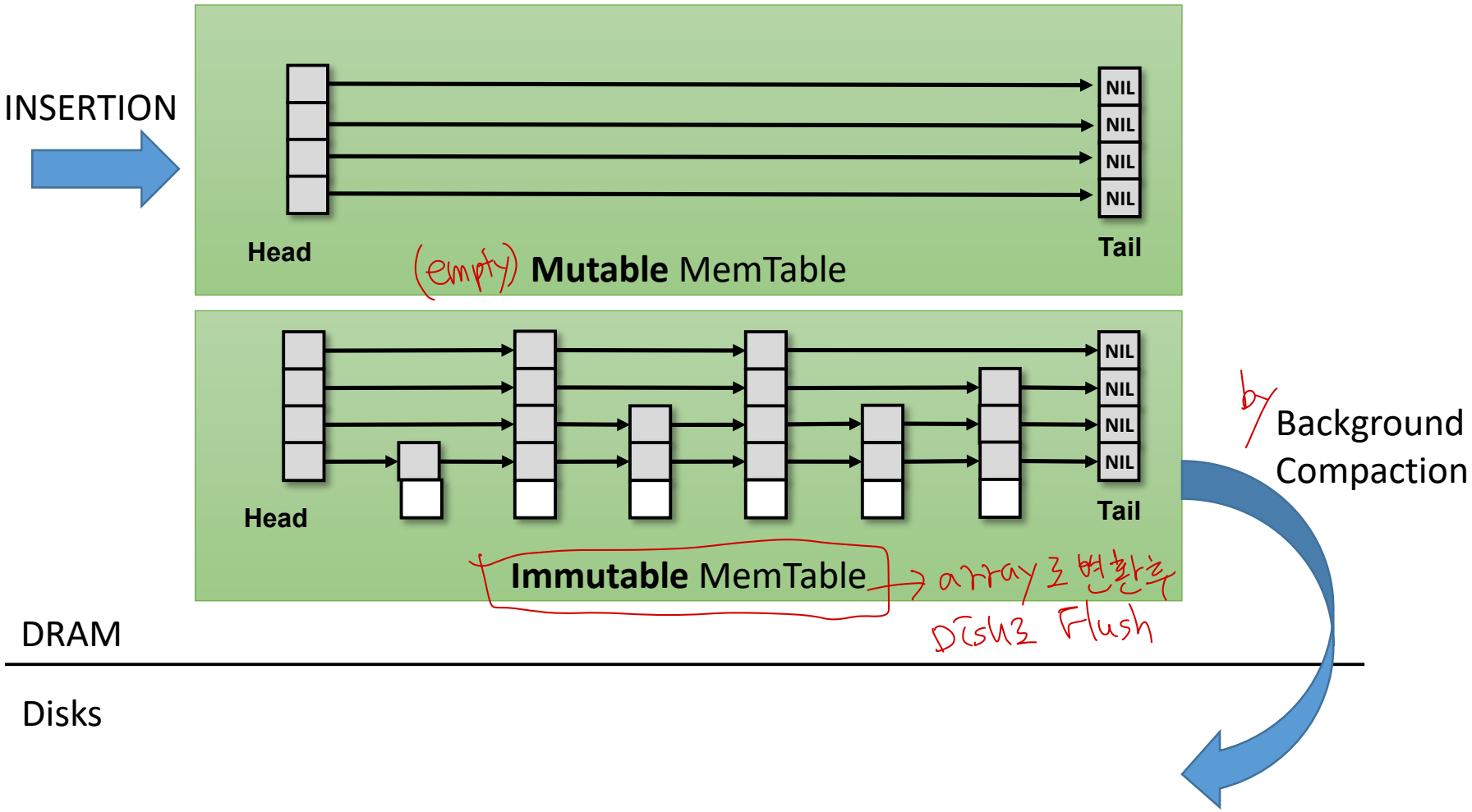
Log-Structured Merge-Tree



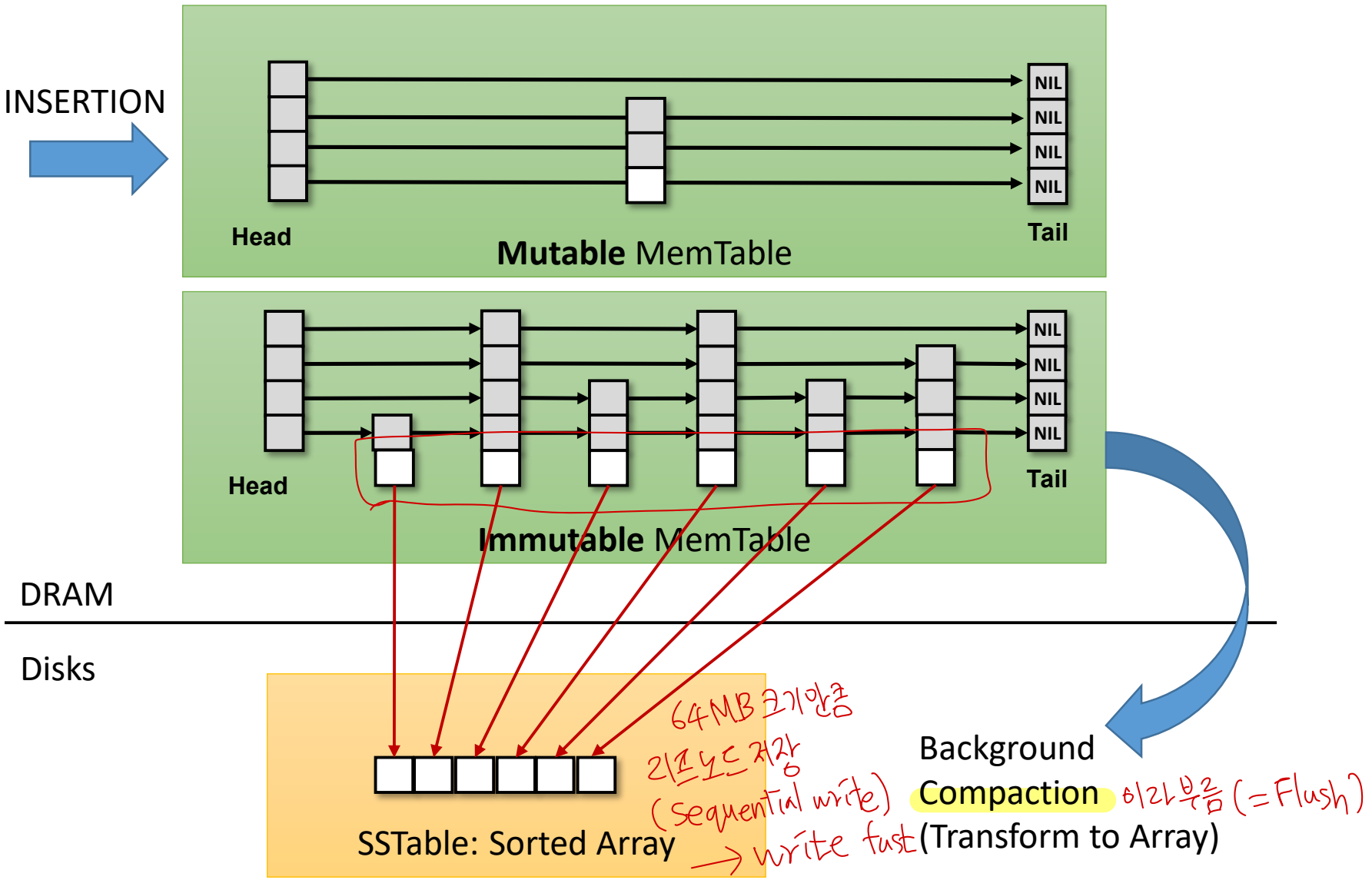
DRAM

Disks

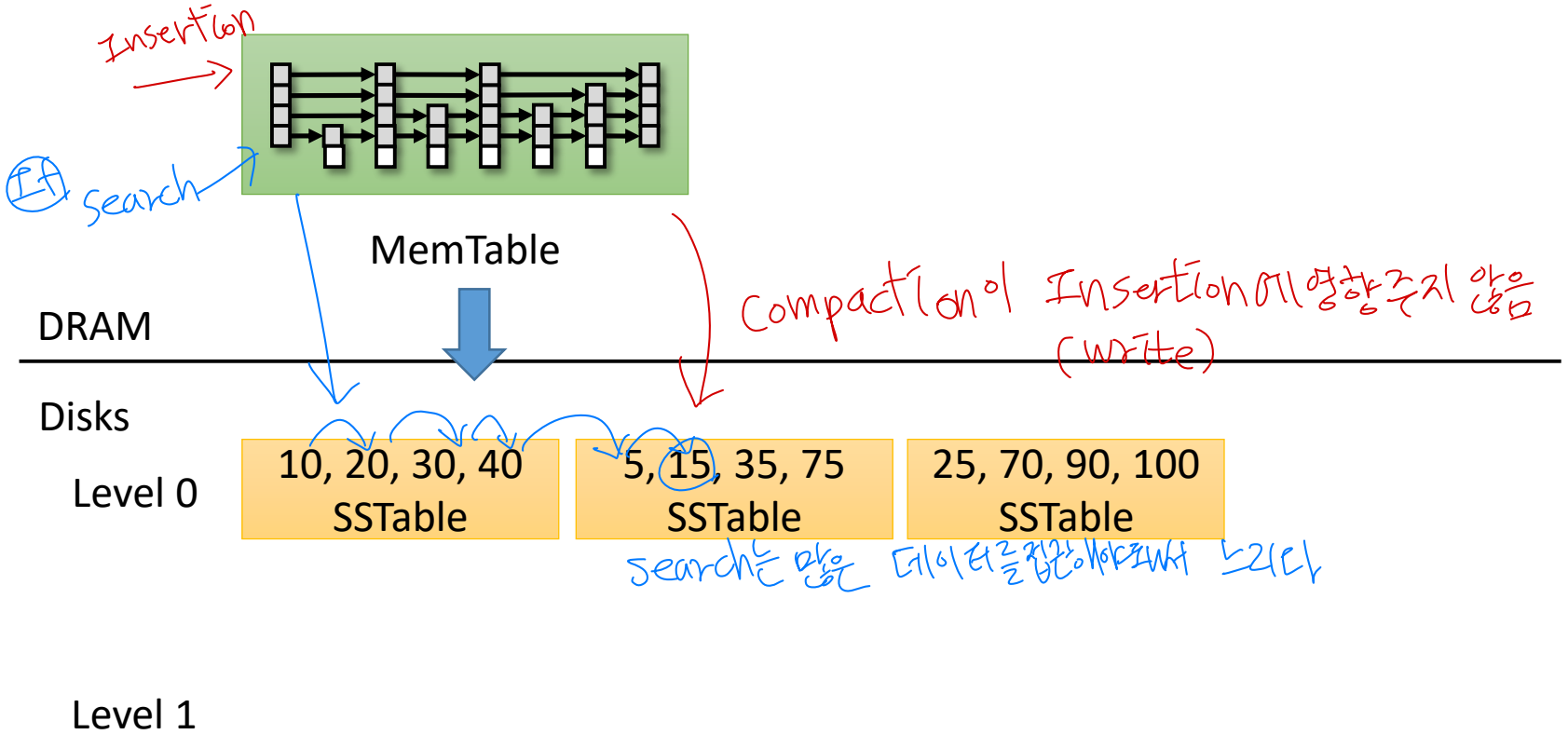
Log-Structured Merge-Tree



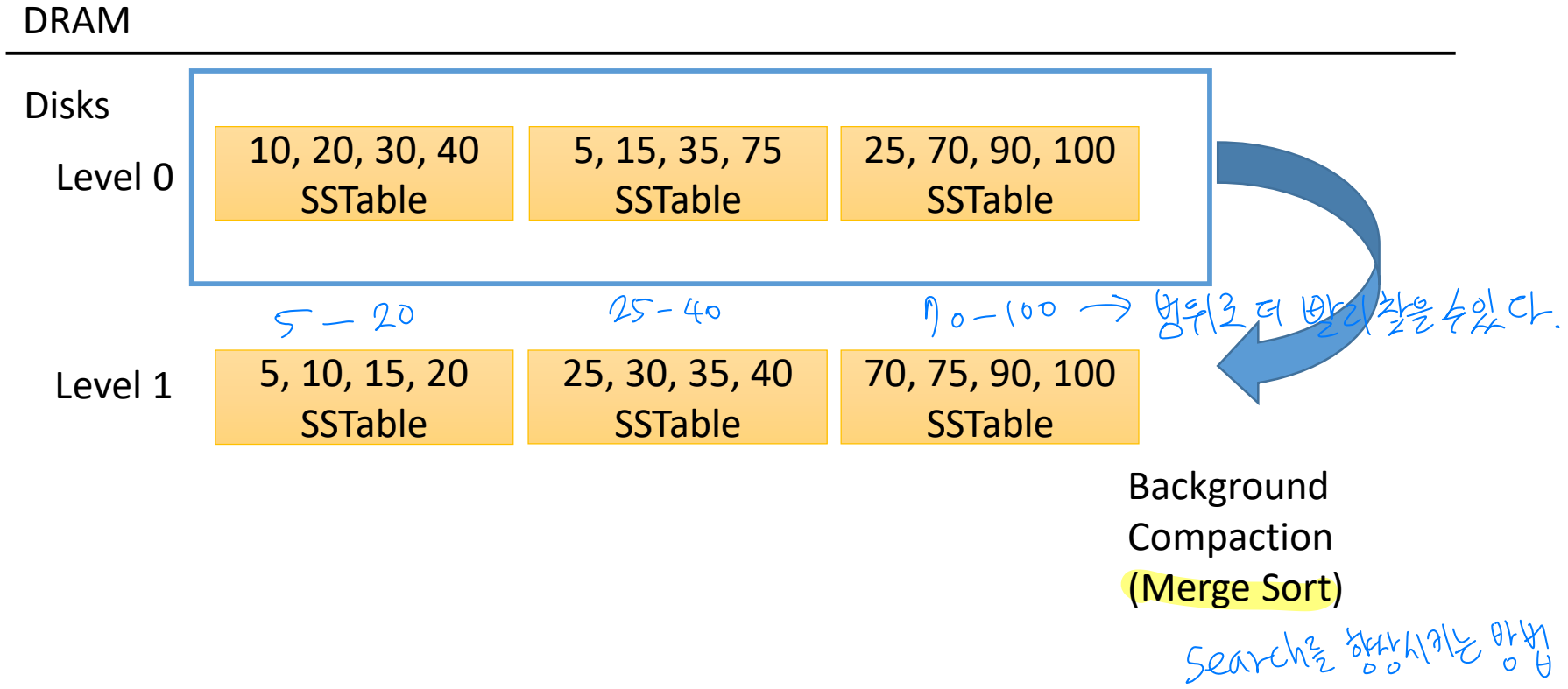
Log-Structured Merge-Tree



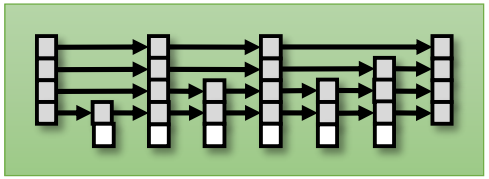
Log-Structured Merge-Tree



Log-Structured Merge-Tree



Log-Structured Merge-Tree



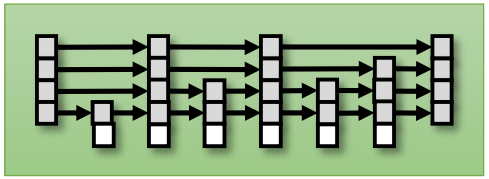
MemTable

DRAM

Disks

Level 0	8, 31, 40, 95 SSTable	2, 7, 43, 71 SSTable	16, 22, 27, 85 SSTable
Level 1	5, 10, 15, 20 SSTable	25, 30, 35, 40 SSTable	70, 75, 90, 100 SSTable

Log-Structured Merge-Tree



MemTable

DRAM

Disks

Level 0

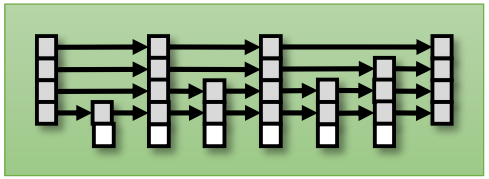
8, 31, 40, 95 SSTable	2, 7, 43, 71 SSTable	16, 22, 27, 85 SSTable

Level 1

5, 10, 15, 20 SSTable	25, 30, 35, 40 SSTable	70, 75, 90, 100 SSTable

Background
Compaction
(Merge Sort)

Log-Structured Merge-Tree



MemTable

DRAM

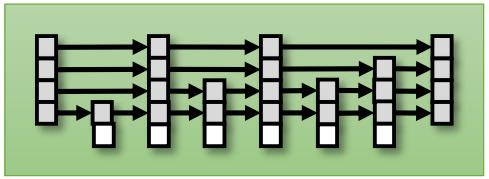
Disks

Level 0

Level 1

2, 5, 7, 8 SSTable	10,15,16,20 SSTable	22,25,27,30 SSTable	31,35,40,43 SSTable	70,71,75,85 SSTable	90,95,100 SSTable

Log-Structured Merge-Tree



MemTable

DRAM

Disks

Level 0 SST SST

If there are too many SSTs in a level, merge sort with the next level

데이터 너무많아져서 최적이지 않음

Level 1 SST SST SST SST SST SST SST SST SST SST

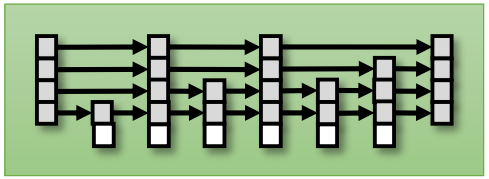
Victim될 array을 정하고

Level 1과 Level 2의 뽑아내는 array끼리 merge sort

Level 2

Background Compaction (Merge Sort)

Log-Structured Merge-Tree



MemTable

DRAM

Disks

Level 0 SST SST

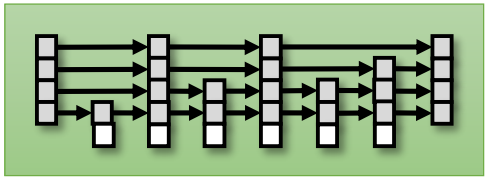
Level 1 SST SST SST

Level 2 SST SST SST SST SST SST SST

Same key-values are written multiple times
→ Write Amplification Problem

=single Insert가 multiple Disk I/O발생

Log-Structured Merge-Tree



MemTable

DRAM

Disks

Level 0 SST SST

Level 1 SST SST SST

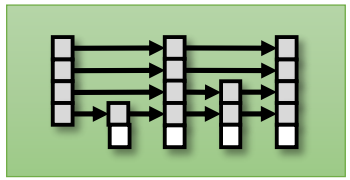
Level 2 SST SST SST SST SST

...

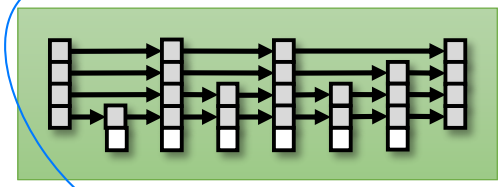
Level N SST SST SST SST SST SST SST SST SST

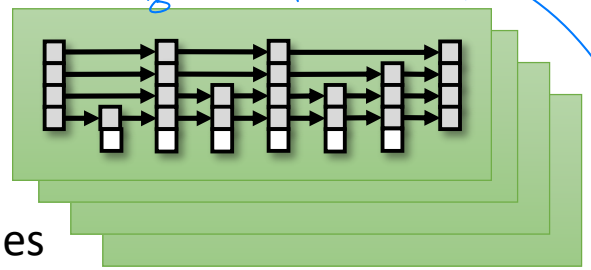
This is what LSM-Tree should be like

Log-Structured Merge-Tree



Mutable MemTable





Immutable MemTables

Out of Memory Error

DRAM

Disks

Too Many Level 0 SSTs

Level 0 SST SST SST SST SST SST SST SST

Level 1 SST SST SST

Level 2 SST SST SST SST SST

...

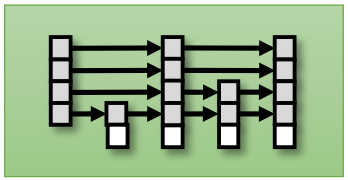
Level N SST SST SST SST SST SST SST SST SST

But, In Reality

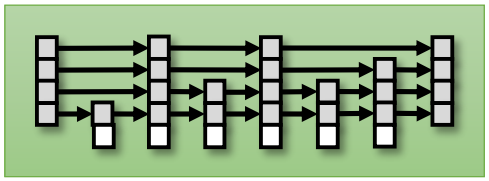
[1] Memory writes are very fast

[2] Merge sort on disks is slow

Log-Structured Merge-Tree



Mutable MemTable



Immutable MemTables

Out of Memory Error

DRAM

Disks

Too Many Level 0 SSTs

Level 0 SST SST SST SST SST SST SST SST

Level 1

Level 2

If there are too many MemTables or Level 0 SSTables, Block Incoming Writes!!

This problem is known as Write Stall Problem

...

Level N SST SST SST SST SST SST SST SST SST


mongoDB®

RDB ACID to NoSQL BASE

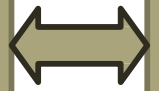
RDBMS 원칙

Atomicity

Consistency

Isolation

Durability



MongoDB 원칙

Basically

Available (CP)

Soft-state
(State of system may change over time)

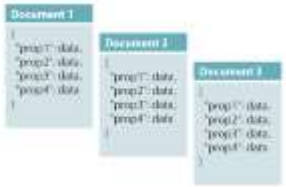
Eventually consistent
(Asynchronous propagation)

NoSQL Databases

Graph database

neo4j

GraphLab



Document-oriented

JSON

mongoDB

CouchDB
relax

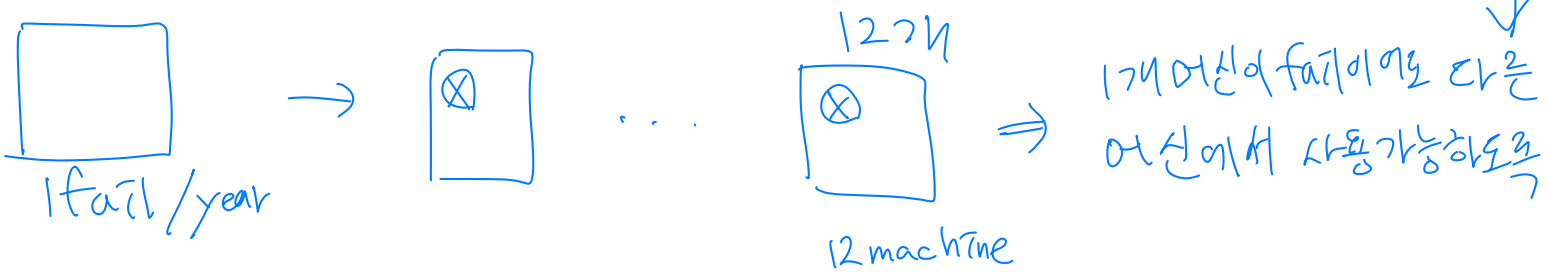
Column family

Cassandra

APACHE HBASE

Key Value Storage Systems

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are **partitioned** across multiple machines
- Queries are routed by the system to appropriate machine
- Records are also **replicated** across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are **consistent**



MTTF
Mean Time to Fail
말에1번 실패

What is MongoDB?

- Developed by 10gen
  - Founded in 2007
- A document-oriented, NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - In practice, this means you can store hashes with any keys and values that you choose
      - Keys are stored as strings
      - Document Identifiers (_id) will be created for each document, field name reserved by system
    - Application tracks the schema and mapping
    - Uses BSON format
      - Based on JSON – B stands for Binary

Functionality of MongoDB

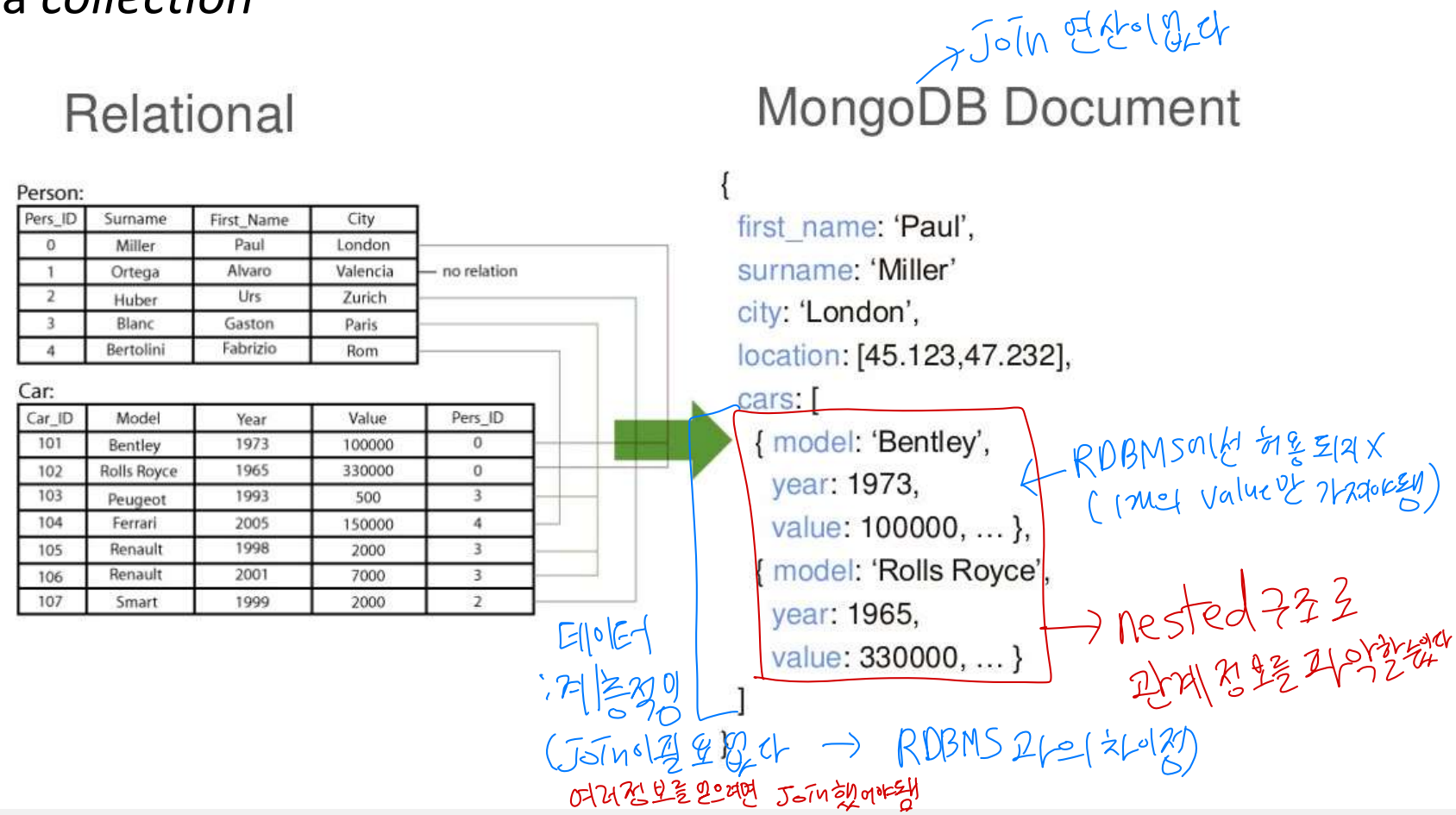
- Dynamic schema
  - No DDL
- Document-based database
- Secondary indexes (값을 탐색할때 유리함)
- Query language via an API
- Atomic writes and fully-consistent reads (consistency를 보장하지 X)
  - If system configured that way
- Master-slave replication with automated failover (replica sets) (실패해도 살아있게 복제)
- Built-in horizontal scaling via automated range-based partitioning of data (sharding)
- No joins nor transactions

Why use MongoDB?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
  - No ERD diagram
- Not well suited for heavy and complex transactions systems

Data Model

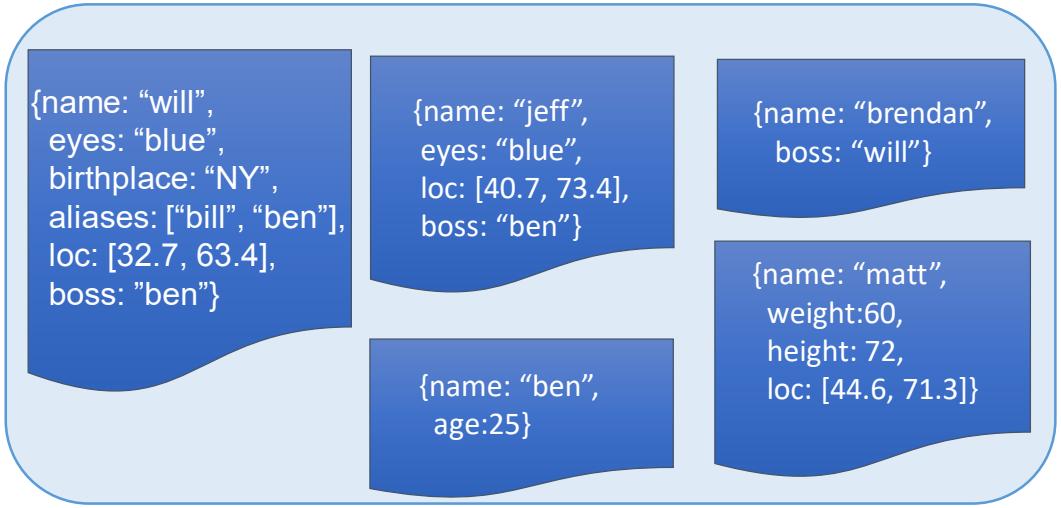
- Stores data in form of BSON (Binary JSON) *documents*
- Group of related *documents* with a shared common index is a *collection*



RDB Concepts to NO SQL

- MongoDB does not need any pre-defined data schema
- Every document could have different data

- Schema Free



- RDBMS vs MongoDB

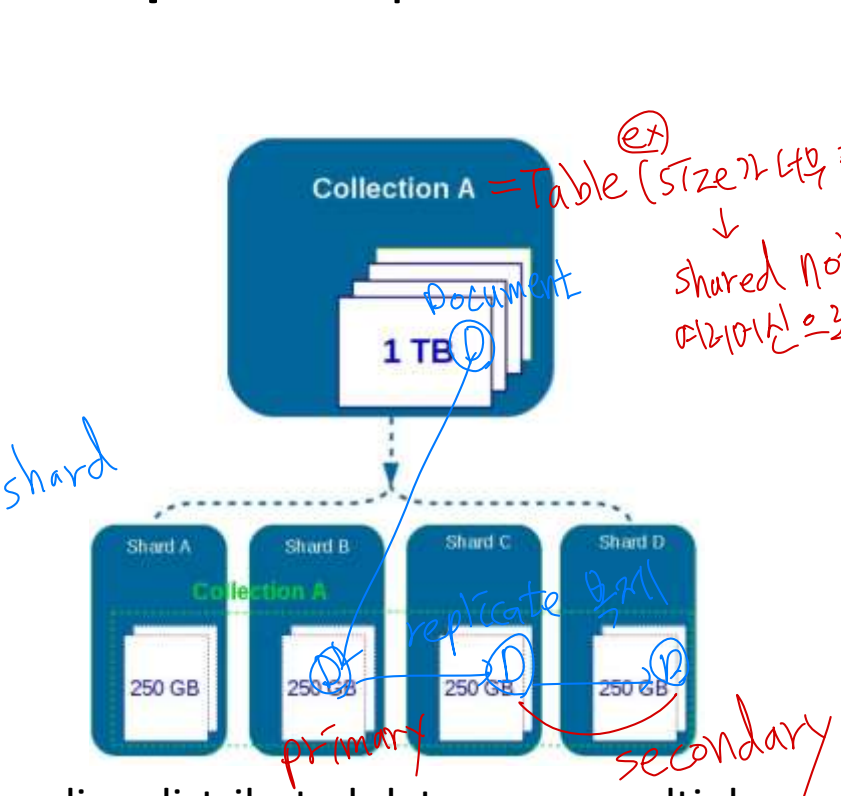
RDBMS		MongoDB
Database	⇒	Database
Table	⇒	Collection
Row	⇒	Document (JSON, BSON)
Column	⇒	Field
Index	⇒	Index
Join	⇒	Embedded Document
Partition	⇒	Shard

HBase Cassandra or Column Family

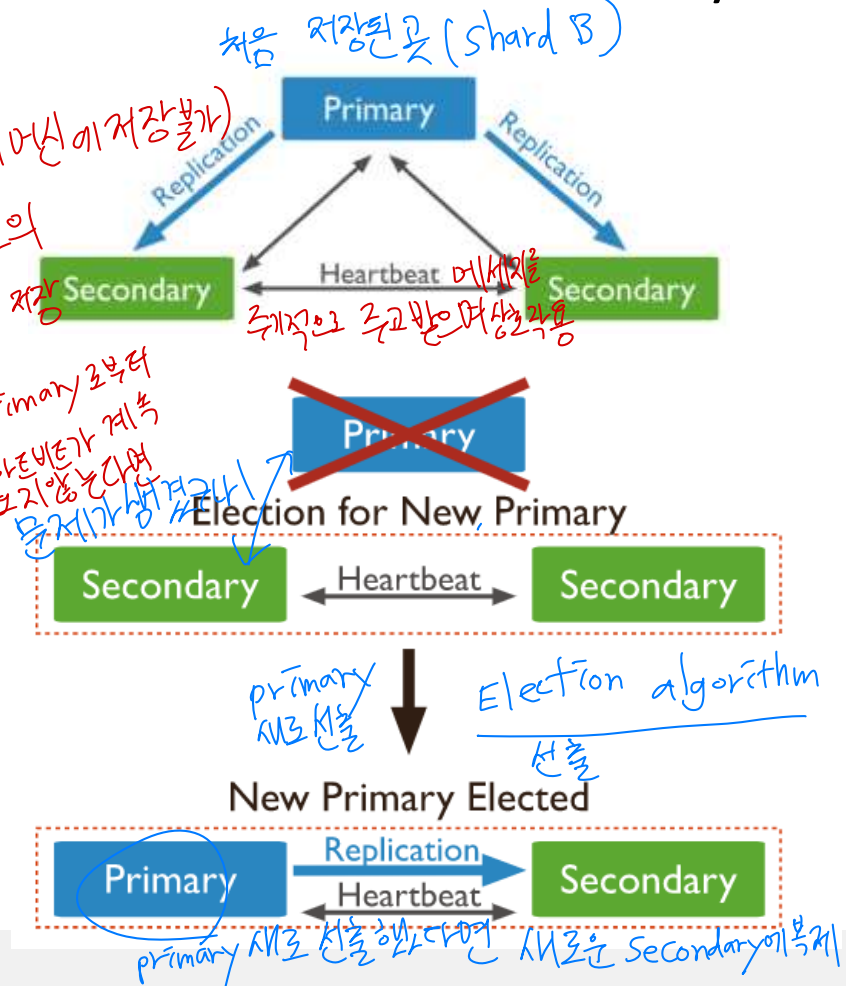
- A MongoDB instance may have zero or more ‘databases’
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have one or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.

Sharding and Replication

- **Shard** (mongod) is a Mongo instance to handle a subset of original data.
- **Replication** provides redundancy and increases data availability

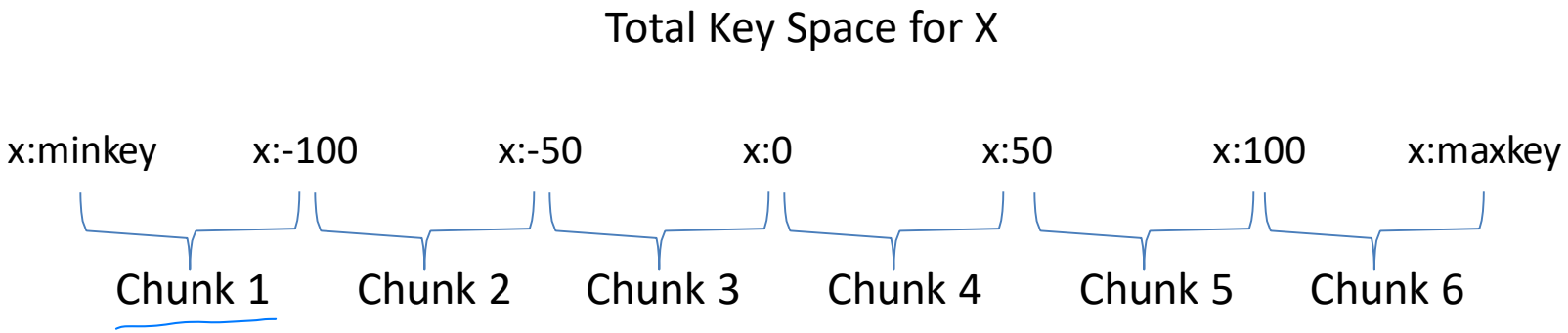


Sharding distributed data across multiple mongod's



Range Sharding

- Shard Key: Single or compound field in schema used for data partitioning
- Partitions are called *chunks*. Two strategies:
  - [1] Range based: Shard Key Values are partitioned into ranges

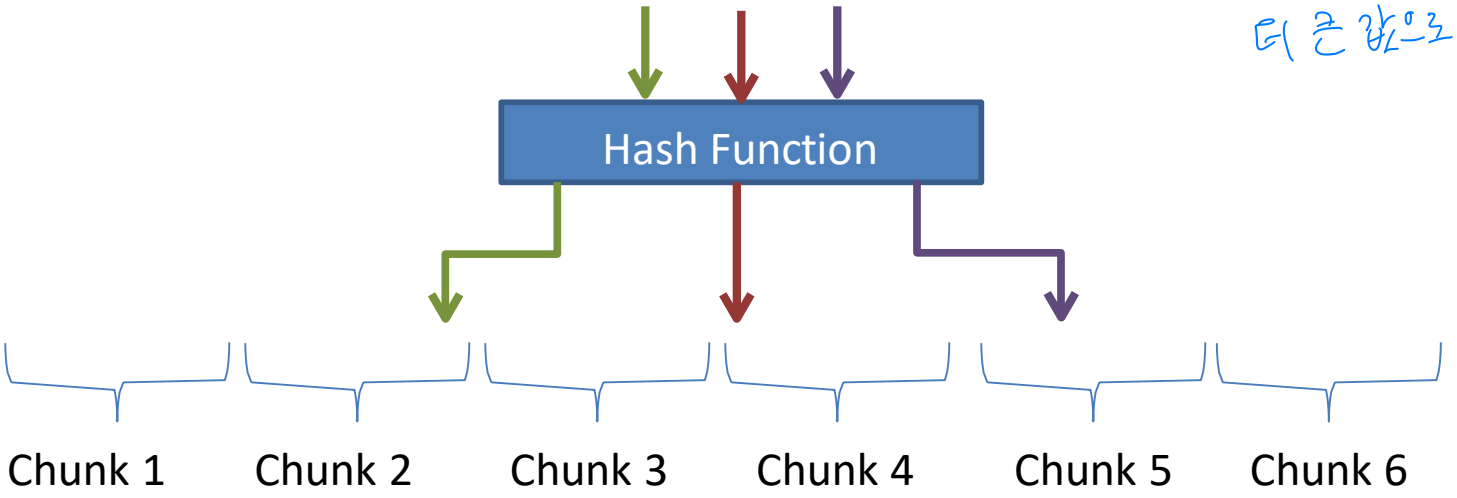


physical File : chunk
chunk의 logical name : shard

Hash Sharding

- [2] Hash based: Hash of shard key values are partitioned into ranges

Hash % ?
? 값은 머신 수보다 더 큰 값으로!



- Range Queries are efficient for the first strategy
- Hash Scheme leads to better data balancing

↳ load balancing이 좋다
골고루 분배

⇒ 해시를 사용하면 range query를 사용할 때 100 < x < 200 모든 shard를 탐색해야함
해시는 지역성이 없어서

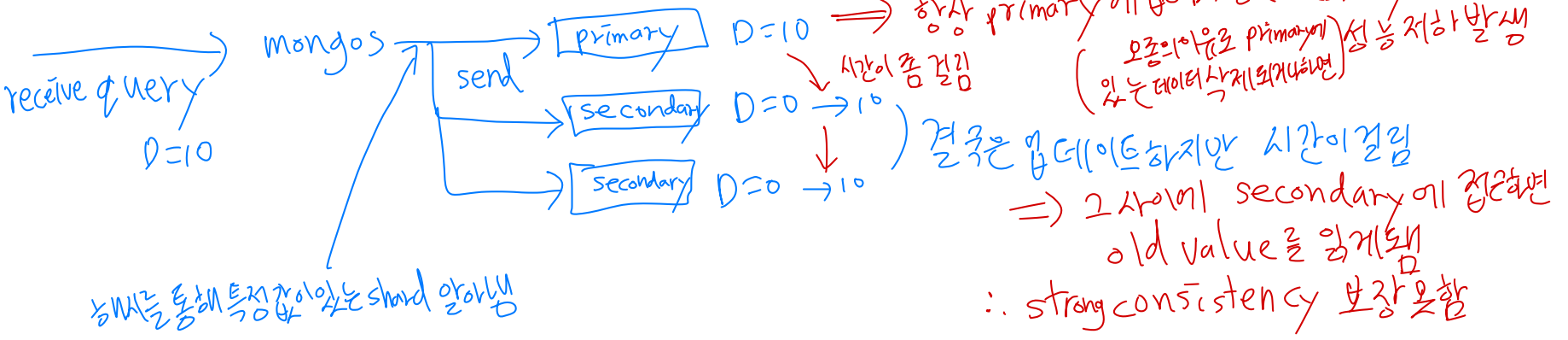
Balancing

- Splitting: Background process which splits when a chunks grows beyond a threshold
- Balancing: Migrates chunks among shards if there is an uneven distribution

특정 shard에 데이터가 몰린다면 몇개의 key를 주변 shard로 줘서 데이터 분포의 균형을 맞춘다.

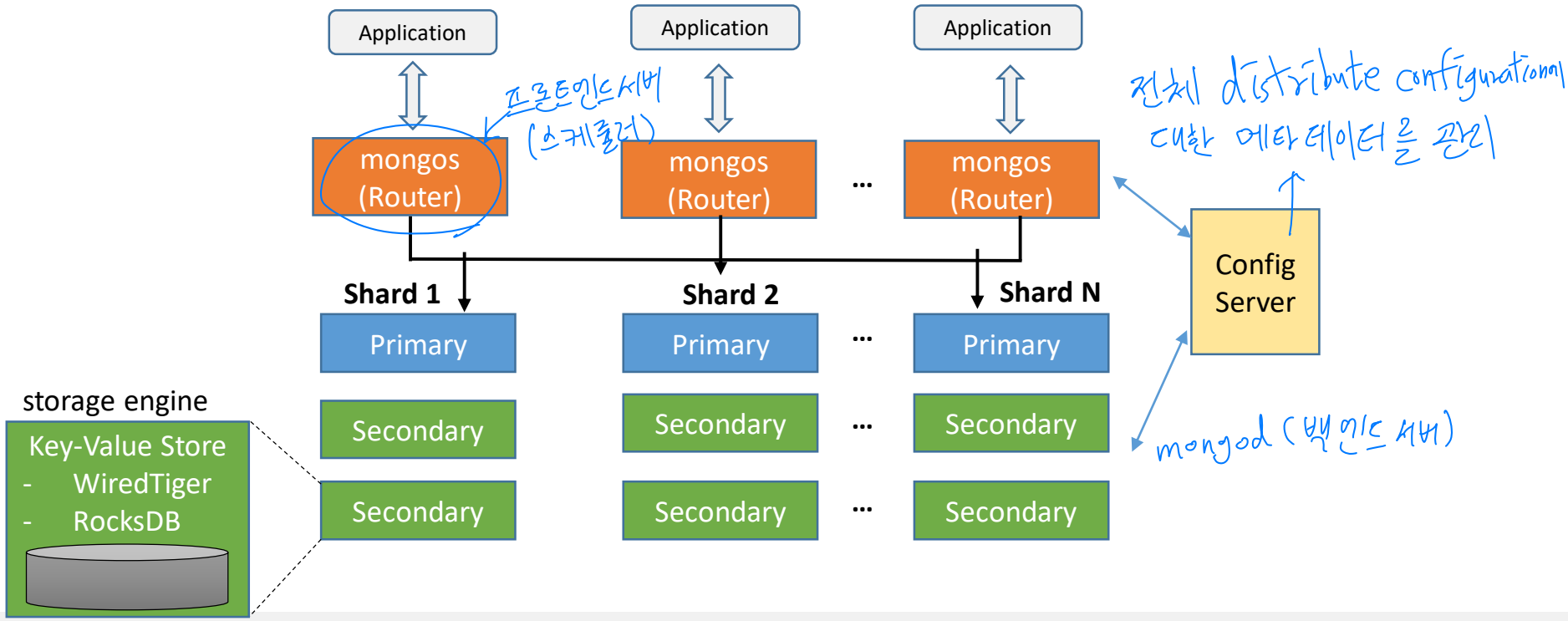
Read Preference with Replication

- Determine where to route read operation
- Default is primary. Possible options are secondary, primary-preferred, etc.
- Helps reduce latency, improve throughput
- Reads from secondary may fetch stale data

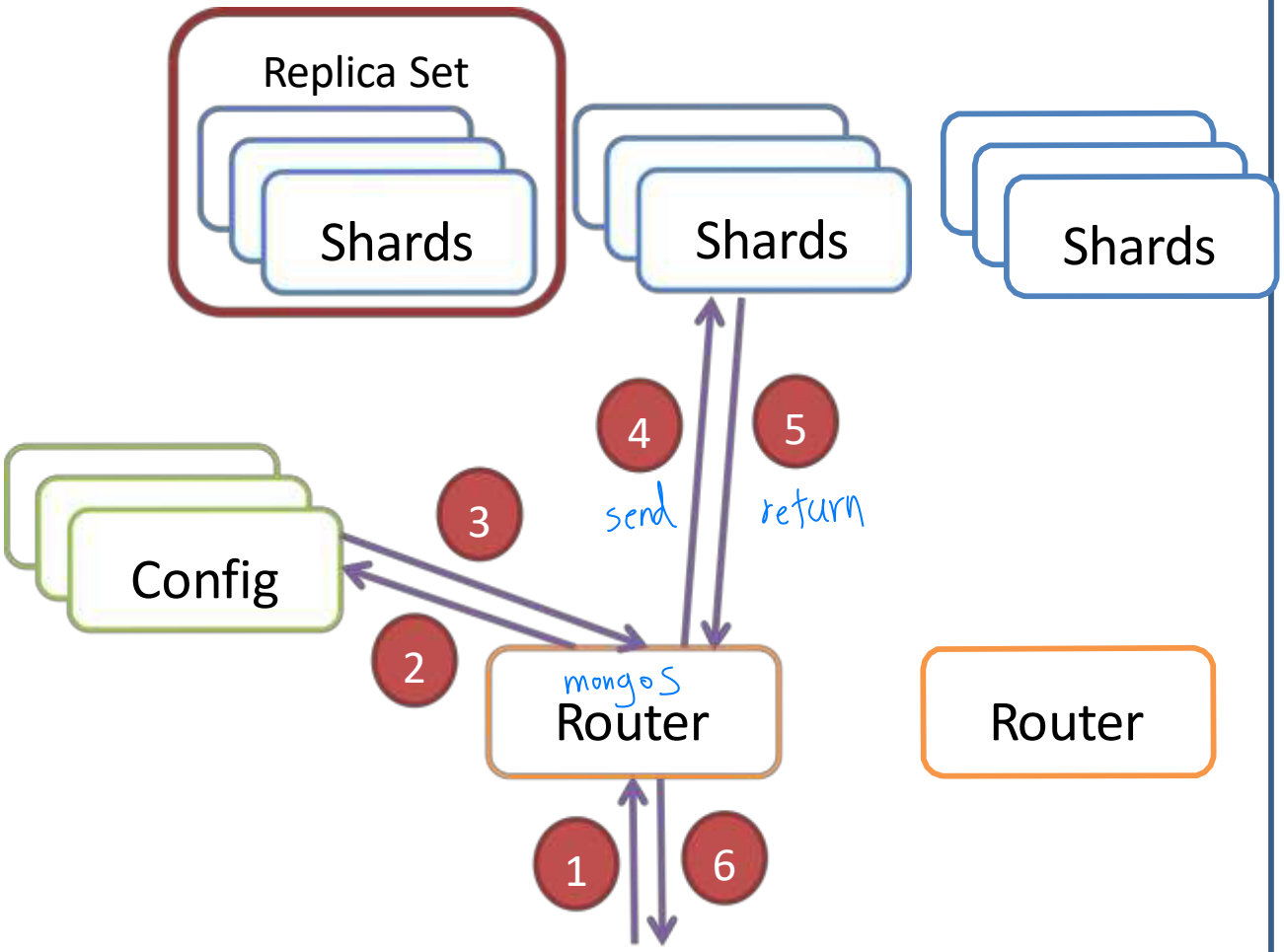


MongoDB Architecture

- **Mongos** is a query router to shards
  - Analogous to a database router.
  - Decides how many and which mongods should receive the query
  - Mongos collates the results, and sends it back to the client.
- **Config Server** is a Mongo instance which stores metadata



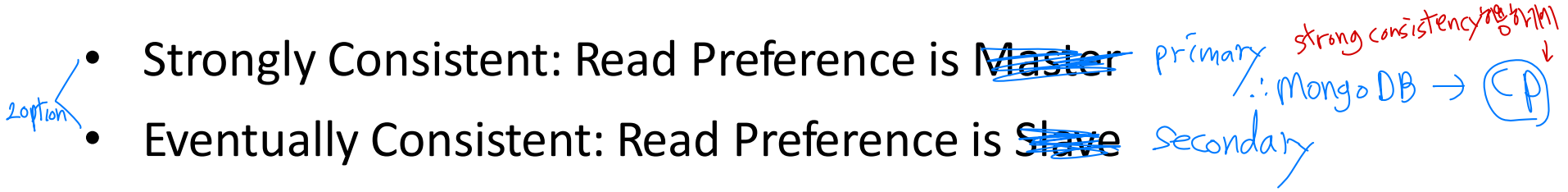
Typical MongoDB Query Processing



- Shards: mongod servers store the data
- Multiple shard servers form a *replica set*
- Replica set maintain same replica of data
- Routers: mongos interfaces with clients and routers operations to appropriate shards
- Config: Stores collection level metadata.

Consistency

- Strongly Consistent: Read Preference is ~~Master~~
- Eventually Consistent: Read Preference is ~~Slave~~
- CAP Theorem: Under partition, MongoDB becomes write unavailable thereby ensuring consistency

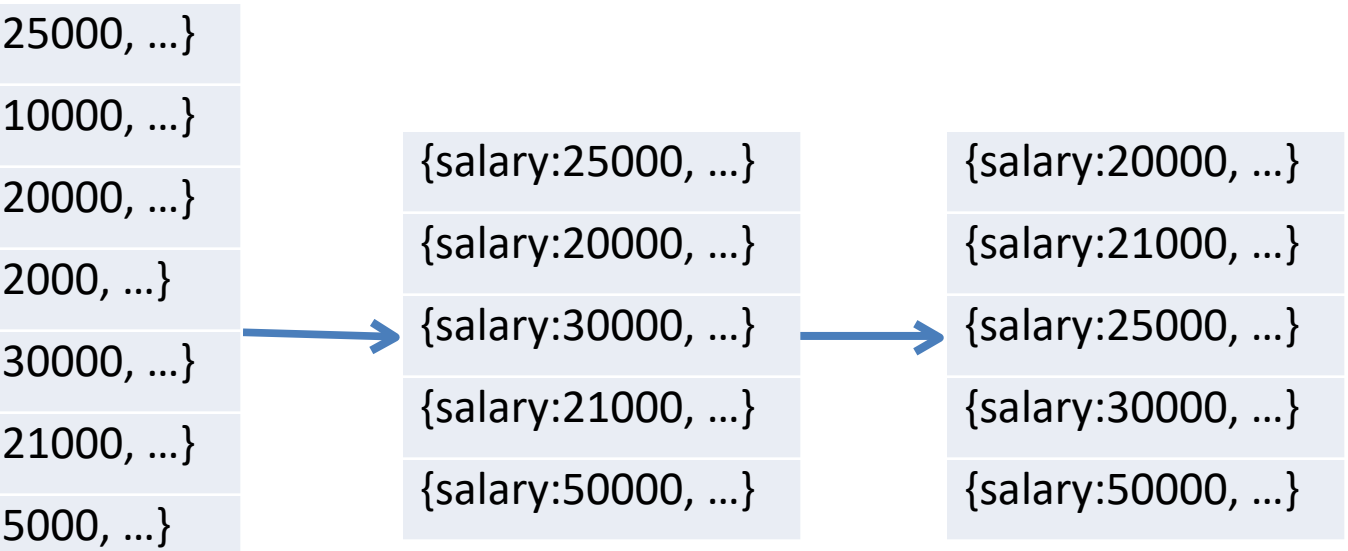


Query

- Query all employee names with salary greater than 18000 sorted in ascending order

```
db.users.find({salary:{$gt:18000}, {name:1}}).sort({salary:1})
```

Collection | Condition | Projection | Modifier



Query Operators

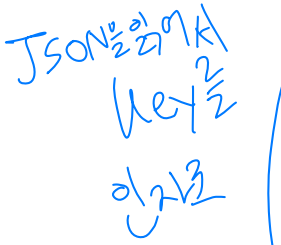
Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

https://docs.mongodb.org/manual/reference/operator/query/

Insert

- Insert a row entry for new employee Sally

```
db.users.insert({
        name: “sally”,
        salary: 15000,
        designation: “MTS”,
        teams: [ “cluster-management” ]
        })
```



[]: array

Update

- All employees with salary greater than 18000 get a designation of Manager

Update Criteria
Update Action
Update Option

```
db.users.update(
    {salary:{$gt:18000}},
    {$set: {designation: “Manager”}},
    {multi: true}
)
```

- Multi option allows multiple document update

Delete

- Remove all employees who earn less than 10000

Remove Criteria

```
db.users.remove(
    {salary:{$lt:10000}},
)
```

- Can accept a flag to limit the number of document removal