1. Introduction to the Big Data

[용어 정의]

- Data: 컴퓨터에서 전자적으로 전송, 저장되는 정보

Structured Data (구조적 데이터)	Unstrucutured Data (비구조적 데이터)
행/열 구조와 관계형 DB로 표현 가능	행/열 구조 X, 관계형 DB 표현 X
숫자, 날짜, 문자열	사진, 음성, 영상, 이메일, 워드 파일, 스프레드시트
기업 데이터의 20% 차지	기업 데이터의 80% 차지
적은 저장용량 요구	많은 저장용량 요구
기존 솔루션으로 관리, 보호 쉬움	기존 솔루션으로 관리, 보호 어려움
정형화된 정의, 쉽게 구조화 가능한 DB 정보	미리 정의된 데이터 모델 X

- 소셜 네트워크 활성화로 비구조적 데이터의 수집량 급증, 빅데이터의 대부분 차지 → 관리할 수 있어야

- Database: 데이터의 구조적 집합 (데이터 연관성, 구조화됨) <-> file system(데이터 연관성 ↓)
- Data Model: DBMS에서 데이터가 어떻게 저장, 연결, 접근, 수정되는지 보여줌
    - · Relational Model: 데이터베이스를 관계의 집합으로 표현
    - · Hierarchial, Network, Object-Oriented, Object-Relational Model…
- DBMS: DB에 데이터 저장, 검색, 정의 및 관리 위한 **소프트웨어**
  (Database Management System)

- **Big Data**: 방대한 양의 비정형 데이터 집합 - 4V
    - · Volume: 데이터의 양이 방대함 (TB~ZB)
    - · Variety: 다양한 데이터 형태 포함. 구조적 데이터는 물론 비구조적, 반구조적 데이터까지 포함 (자유로운 형식, 다양한 종류)
    - · Velocity: 데이터의 생성, 수집 및 처리 속도가 빨라야 함
    - · Veracity: 데이터의 정확성과 타당성, 신뢰성 (품질이 좋은가)
      (검증을 통한 팩트체크)

[한국의 빅데이터 정책]

- 데이터의 개방과 재사용 촉진
- AI Hub 플랫폼: AI 학습 데이터 구축 확대 및 AI 개발 인프라 확보
- 공적 데이터와 사적 데이터 매핑의 연결 강화 목적

[빅데이터 분석학]

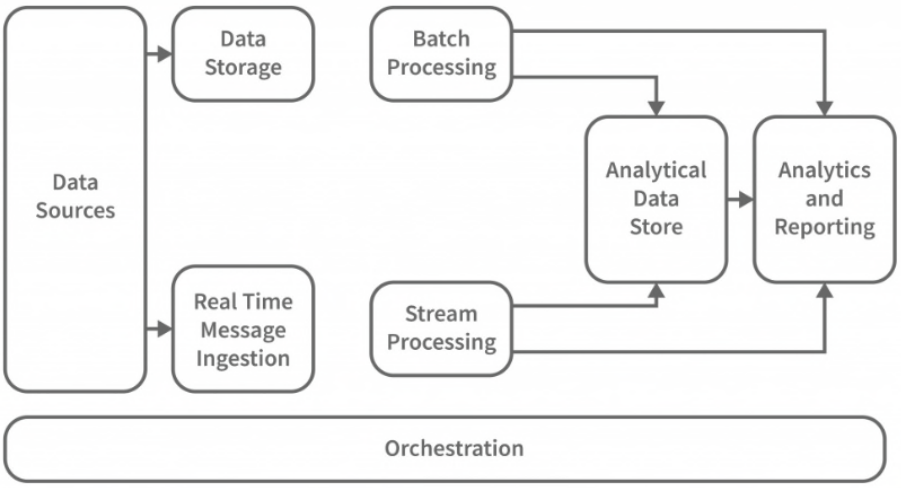
- 의미있는 insights를 추출하기 위한 프로세스
- 숨겨진 패턴, 알려지지 않은 연관성, 시장 트렌드, 고객 선호
- 효용성: 의사 결정, 부정 행위 방지, 비용 감소

[프레임워크]

- 소프트웨어 환경
- 소프트웨어 기능의 설계와 구현을 재사용
- 새로운 애플리케이션과 솔루션 개발에 사용

[빅데이터 프레임워크]

- 방대한 데이터를 실시간으로 처리하고, 낮은 비용과 내고장성(Fault Tolerance)을 충족해야 함

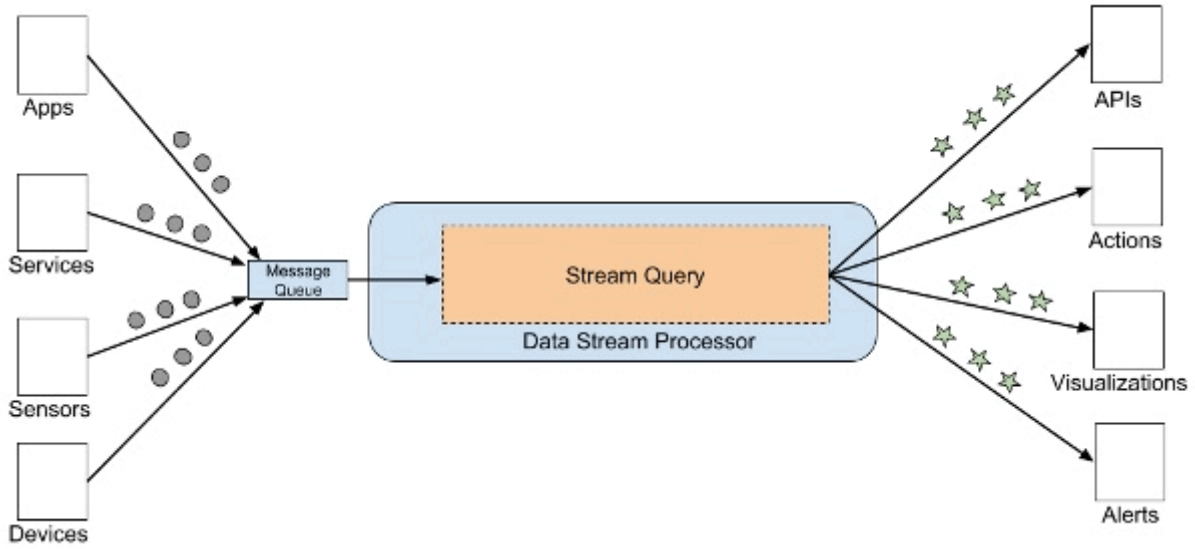


· 실시간 데이터 처리 필요성: 여러 데이터 소스에서 지속적으로 데이터 발생
→ 실시간 분석하여 패턴 분석 → 의사 결정 or 새로운 비즈니스 모델 생성

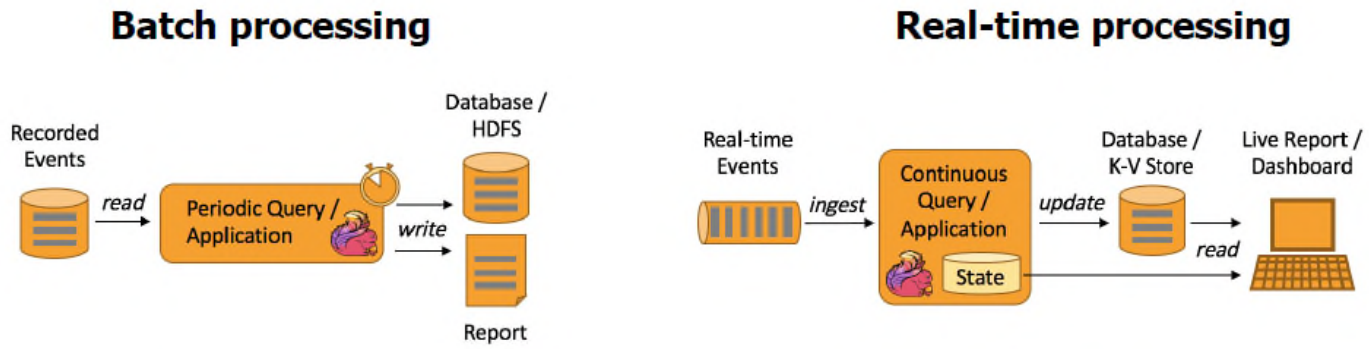
· 데이터 통합/분석 기법 요구사항: 데이터 필터링 및 정제(필요한 정보만),
데이터 일반화 및 정규화 (Unstructured → Structured)
이상치 탐지, 데이터 보강(불완전한 데이터, NULL 등)
데이터 통합, 데이터 분석, 데이터 시각화

· 데이터 보안, 개인정보 보호가 요구됨

· 스트리밍 데이터: 여러 데이터 소스로부터 실시간, 연속적으로 생성된 데이터
일반적으로 데이터 레코드가 KB 형태로 동시에 전송됨
고객 기록, 거래, SNS, 주식 시장, 동영상, GPS 정보 등

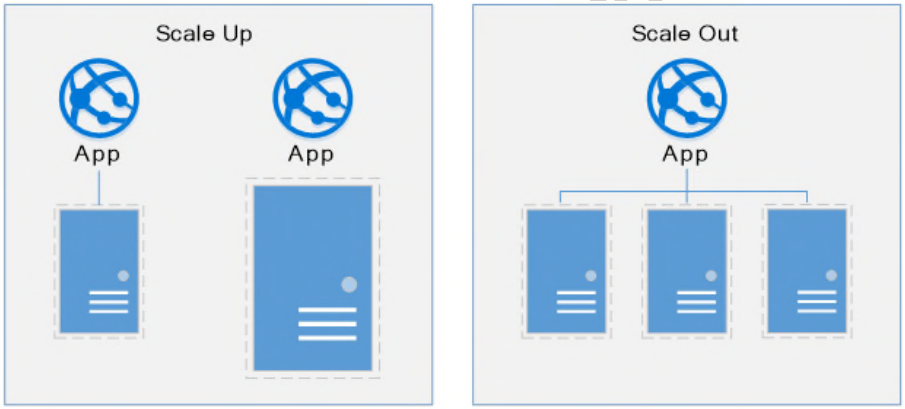


· 빅데이터 시스템: 대용량 데이터를 관리하기 위한 분산·병렬화된 시스템
기능 - 데이터 수집, 관리, 전송, 분석
배치(Batch) 데이터 처리 시스템 ↔ 실시간 데이터 처리 시스템



· 내고장성 시스템: 시스템은 오류 혹은 고장이 발생하더라도 작업을 수행할 수 있어야 함
(HW 오작동, SW 에러, 자원 부족, 데이터 변질 등)

· 비용 효율적 시스템: 비용 절감, Scale Up ↔ Scale Out (저용량 저장장치 여러개 분산)
빅데이터 시스템은 주어진 과업에 적합할 정도로 구성하여 비용 절감
고려사항: 데이터 자료형, 처리 시간, 처리량, 다른 시스템 요구사항
초기 구축 비용을 최소화하여 가격 경쟁력 있게 구성할 수 있음



· 기존 시스템과 호환성: 빅데이터 수집을 위한 기존 시스템과의 호환성 충족
새로운 시스템을 처음부터 개발 ↔ 기존 시스템에 기능 추가

· 오픈소스: 제한 없이 누구나 사용 가능, 대부분의 시스템에 해당

[MongoDB 환경 구축]

- mongod: MongoDB 서버 실행
- mongosh: MongoDB Shell 실행 (사용자 입력 명령어 해석하여 커널에 전달)

```
mongosh [database name] -u [user_name] -p '[password]'
```

- db.createUser({ // MongoDB 계정 생성

```
    user: "Admin",
    pwd: "password",
    roles: [{role: "read", db: "test"}, {role: "clusterAdmin", db: "myDB"}]
} )
```

2. Big Data Storage

[Centralized Storage]

- 하나의 기기에 내장된 DB에 데이터 저장 (중앙집중형)
- 해당 기기와 직접적으로 통신하여 정보 읽기/쓰기 진행함
- 관계형 DB에 주로 사용
  (RDB도 점차 분산형으로 바꾸는 추세)

[Relational Model]

- Relation = Table = Schema
- 연관된 데이터가 여러 테이블에 걸쳐 저장되며, 테이블 간 관계 형성에 의해 연결됨
- DBMS: 데이터베이스 생성과 관리를 위한 소프트웨어

- 단점
  - 유연성 부족: 구조화된 데이터에만 적합함 (비구조화 데이터 X)
  - 속도: 안정적인 데이터 보존 위한 설계 (급격한 데이터 증가 X)
  - 확장성 부족: 방대한 데이터 용량에 적합하지 않음
  - SQL 제한성: SQL을 이용한 특정 종류의 기본 쿼리문 구현 어려움

[Decentralized Storage]

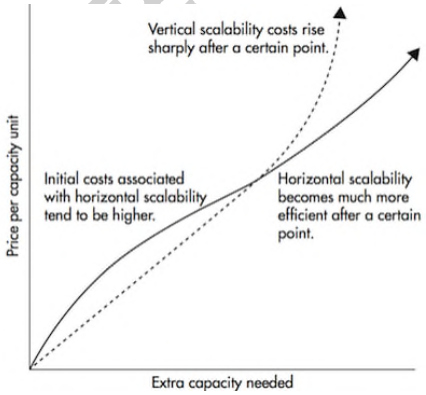
- 여러 기기에서 동시에 DB가 실행됨
- 사용자 입장에서 하나의 기기와 통신하는지, 여러 기기와 통신하는지 모름
- NoSQL DB에 주로 사용

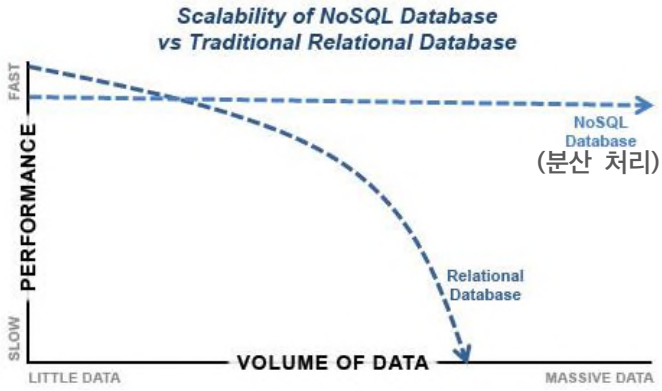
[NoSQL] → 기기 각각의 성능이 뛰어나지 않아도 됨

- "No SQL": 시스템에 SQL 사용 X, 대체 쿼리 언어를 사용함 (Schema-Free)
- "Not Only SQL": 시스템은 SQL과 함께 다른 기술과 쿼리 언어를 사용
- "Non-Relational": NoSQL 데이터베이스는 관계형 데이터베이스가 아님, "NoREL"

[NoSQL 특징]

- 유연함: NoSQL DB는 스키마를 사용 X or 느슨한 구조의 스키마 사용
  데이터의 스키마 구조를 정의할 필요 X
- 확장성: Scale-Up(Vertical Scaling)-단일 하드웨어 증설 / Scale-Out(Horizontal Scaling)-하드웨어 분산 (전체 총량의 성능 ↑)
  Horizontal Scaling 기법이 특정 시점 이후로 훨씬 저렴하고 효율적임 (초기 구축 비용 높기 때문)

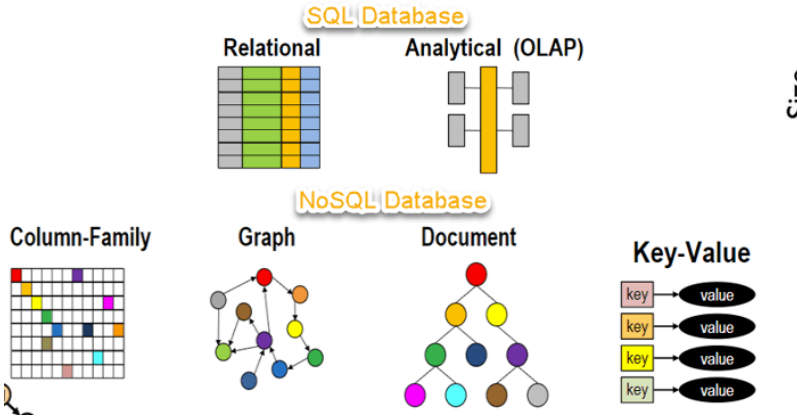


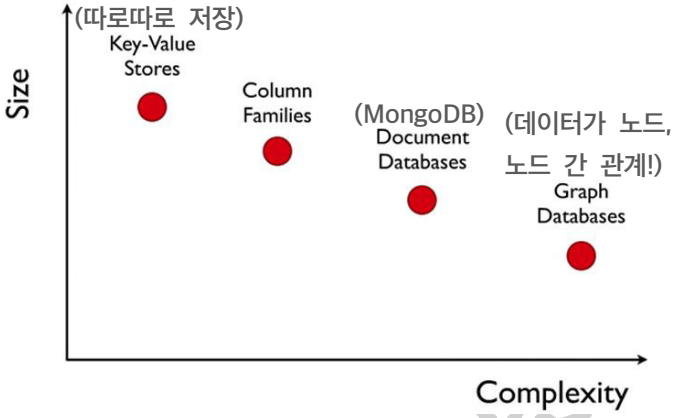


- 내고장성: 여러 기기의 조합이 단일 기기보다 내고장성 뛰어남, 시스템의 가용성을 향상시킴 (Decentralized)

- 많은 대기업이 전통적인 스키마 기반 DBMS에서 NoSQL DB로 전환중 (대부분의 NoSQL DB가 오픈소스)

[NoSQL DB 유형]



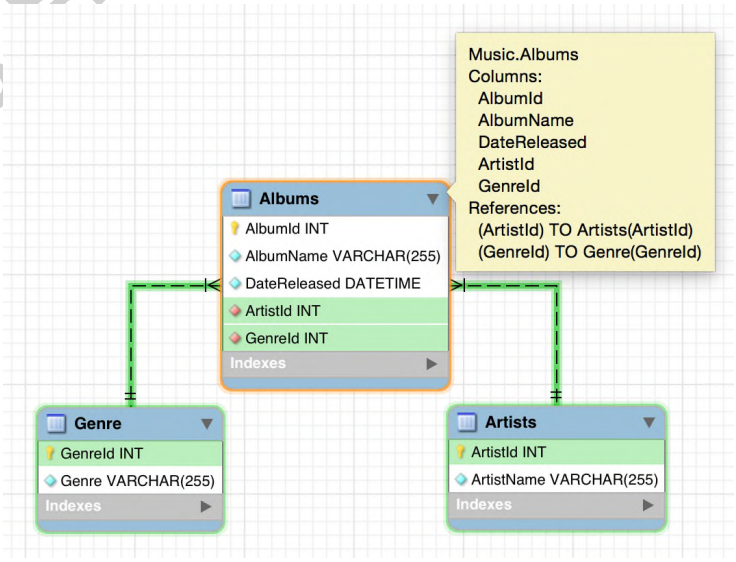


- **Key-Value Store DB:** 데이터 저장하기 위해 단순한 key / value 값을 사용하는 NoSQL DB
  대다수 언어가 데이터 저장하는 구조, 확장성 ↑ // 딕셔너리, 해쉬, 연관 배열 등
  Key-Value DBMS: Redis, Oracle NoSQL DB, Project Voldemort, Aerospike...

- **Document Store DB:** 데이터 저장 위해 Document 기반 모델 사용하는 DB
  (Key-Value 쌍 여러 개를 하나의 Document에 저장)
  Key-Value DB와 달리 Document DB의 Value는 반구조화된 데이터로 이루어짐
  각 Record(Row)와 함께 연관된 데이터를 하나의 Document에 저장
  각 Document는 쿼리문을 사용할 수 있는 반구조화된 데이터 저장 (XML, JSON)

(RDB와 유사하게 만들 수 있음, Key-Value는 X)
- **Document Store vs. Relational DB**

```
{
    '_id' : 1,
    'artistName' : { 'Iron Maiden' },
    'albums' : [
        {
            'albumname' : 'The Book of Souls',
            'datereleased' : 2015,
            'genre' : 'Hard Rock'
        }, {
            'albumname' : 'Killers',
            'datereleased' : 1981,
            'genre' : 'Hard Rock'
        }, {
            'albumname' : 'Powerslave',
            'datereleased' : 1984,
            'genre' : 'Hard Rock'
        }, {
            'albumname' : 'Somewhere in Time',
            'datereleased' : 1986,
            'genre' : 'Hard Rock'
        }
    ]
}
```

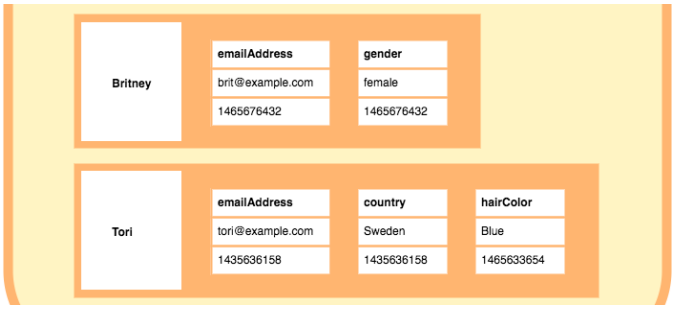


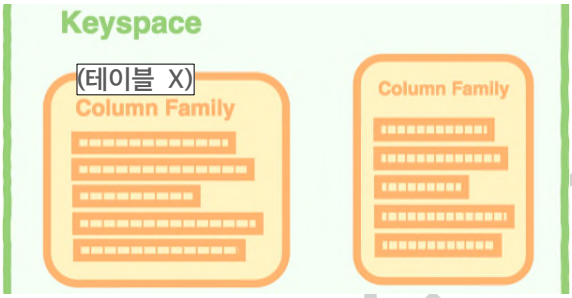
· Relational DB에서 PK, FK 필드를 통해 3개의 서로 다른 테이블을 연관시켜야 함

· Tables: 모든 데이터를 주어진 Entity에 따라 하나의 Document에 저장함 (스키마 구조 자유로움)
· Schemas: 어떠한 두 개의 Document도 서로 다른 구조와 자료형의 데이터를 저장 가능
· Scalability: Horizontal Scale 용이함 (Scale-Out, 하드웨어 분산)
· Relationships: 하나의 Record(Row)와 연관된 모든 데이터는 같은 Document 안에 저장됨

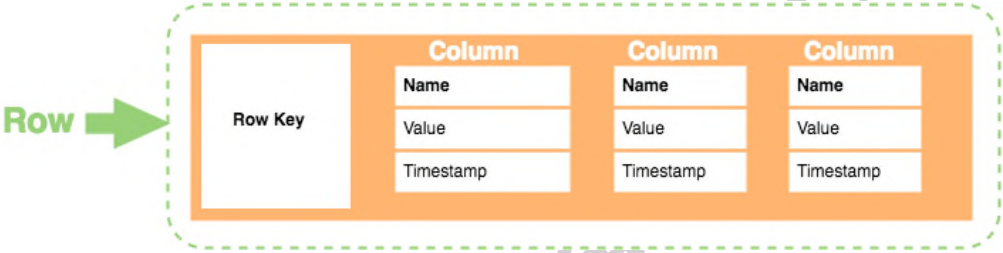
웹 애플리케이션, 사용자 생성 콘텐츠, 카탈로그 데이터, 네트워킹/컴퓨팅 분야에 활용 가능
Document Store DBMS: MongoDB, DocumentDB, CouchDB, MarkLogic, OrientDB

- **Column Store Database:** Column 기반의 모델을 사용하는 데이터를 저장하는 DB
  하나의 Column Family는 여러 개의 Rows 포함
  각 Row는 서로 다른 개수의 Column을 보유할 수 있음
  (서로 다른 Column 이름, 자료형 등을 가질 수 있음)
  각 Column은 위치한 Row에 포함됨
  (관계형 DB처럼 모든 행에 걸쳐있지는 않음. 각 Column은 name/value 조합과 timestamp 지님)





- **Column Store Database 구조 (각 Row 마다)**

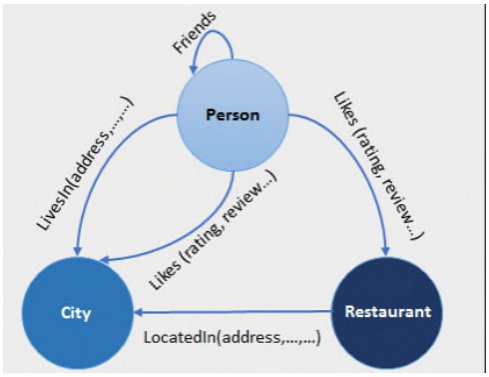


· Row Key: 각 Row는 고유값 가짐 → 고유 식별자
· Column: 각 Column은 하나의 Name, Value, Timestamp를 가짐
· Column-Name: name/value 조합 중 Name에 해당
· Column-Value: name/value 조합 중 Value에 해당
· Timestamp: 데이터가 추가된 시점의 날짜와 시각을 포함

Column Store DBMS: Bigtable, Cassandra, HBase, Vertica, Druid

- **Graph Database:** 데이터를 표현하고 저장하기 위해 그래프 모델을 사용하는 DB
  Relational Model을 대체
  (Relational DB - 데이터를 미리 정의된 스키마를 통한 고정된 구조의 테이블에 저장
  Graph DB - 미리 정의된 스키마 X, 모든 스키마는 입력된 데이터를 단순히 반영함)
  Graph DB는 연결된 데이터를 다루는 데 적합
  Nodes: 데이터 저장 / Arrows: 노드 간의 관계를 나타냄

  SNS, 실시간 상품 추천, 네트워크 다이어그램, 부정 감지, 접근 관리 등에 활용
  Graph DBMS: Neo4j, Blazegraph, GraphBase



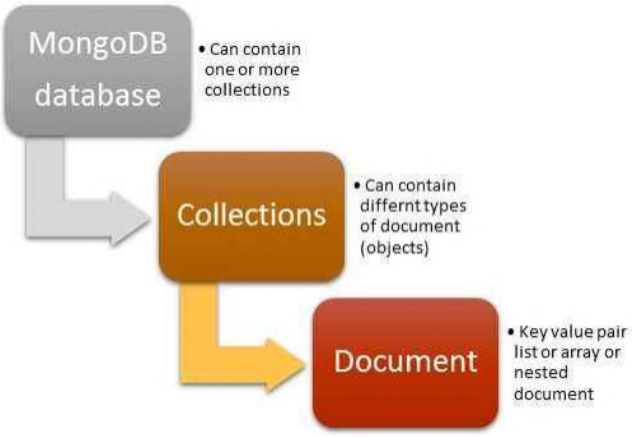
3. MongoDB Overview

[빅데이터 프로세스]

- 데이터 수집 → 데이터 정제, 변형 → 모델 트레이닝 → 데이터 테스트 → 수정 및 향상
  (MongoDB)

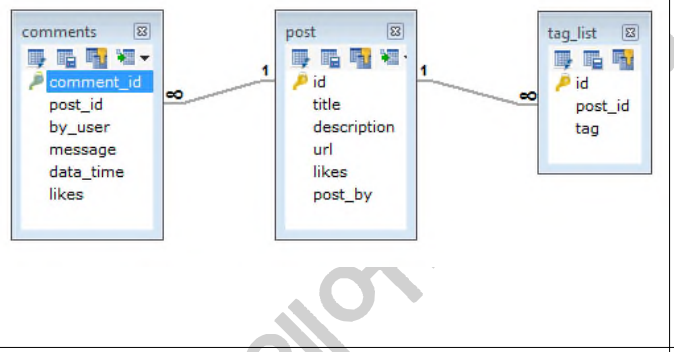
[MongoDB 구조]

- 일반적인 Relational DB 모델: 하나의 테이블이 여러 개의 열(Row) 가짐
- MongoDB 구조: 하나의 Collection이 여러 개의 Document 가짐



RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

- 빅데이터에 적합함: Unstructured Data
  · ex) 블로그/웹사이트에서 사용할 DB 설계

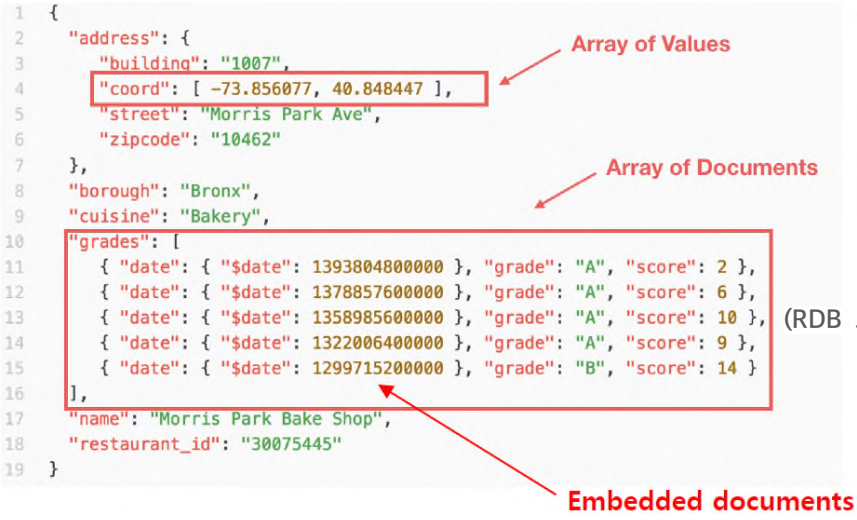


RDBMS 스키마에서 최소 3개의 테이블 필요함

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

MongoDB Collection 1개 (Aggregate Orientation)

- Embedded Document



(RDB Join 유사)

[MongoDB Document 패턴]

- 1 : 1 패턴

Students

Student_id	Student_name
2023299001	Jeong-Hun Kim
2023299002	Jae-Seong Yeon
2023299003	Jong-Pil Park

Personal information

Registration number	Phone	Student_id
990101-X	01012341234	2023299001
990615-X	01015152020	2023299002
990930-X	01019194848	2023299003

```
{
    "_id": "2023299001",
    "name": "Jeong-Hun Kim",
    "Personal information":
    {
        "Registration number": "990101-X",
        "Phone": "01012341234"
    }
}
```

Embedded documents

- 1 : N 패턴

Students

Student_id	Student_name
2023299001	Jeong-Hun Kim
2023299002	Jae-Seong Yeon
2023299003	Jong-Pil Park

Grade

Lecture	Score	Student_id
Database	A+	2023299001
Database	B+	2023299002
Data structure	A+	2023299001

① Embedded document (strong association)

```
{
    "_id": "2023299001",
    "name": "Jeong-Hun Kim",
    "Personal information":
    {
        "Registration number": "990101-X",
        "Phone": "01012341234"
    },
    "Grades":[
        {"lecture": "Database", "score": "A+"},
        {"lecture": "Data structure", "score": "A+"}
    ]
}
```

(하나의 Document에 배열 형태로 저장)

② Linked document (weak association)

```
{
    "_id": "2023299001",
    "name": "Jeong-Hun Kim",
    "Personal information":
    {
        "Registration number": "990101-X",
        "Phone": "01012341234"
    }
}
```

```
{"_id": "001", "lecture": "Database", "score": "A+",
"student_id": "2023299001"}
{"_id": "002", "lecture": "Data structure", "score": "A+",
"student_id": "2023299001"}
```

(RDB Join 모방 별도의 Document)

- N : M 패턴

Students

Student_id	Student_name
2023299001	Jeong-Hun Kim
2023299002	Jae-Seong Yeon
2023299003	Jong-Pil Park

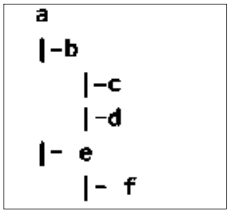
Take class

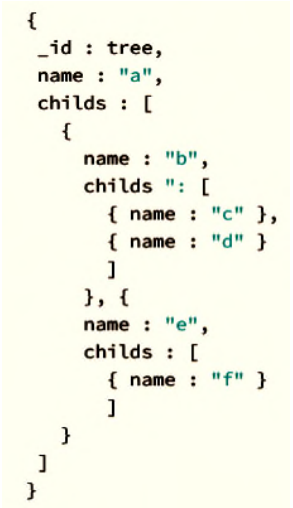
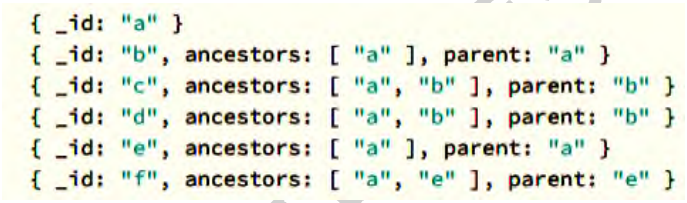
Lecture

Lecture_id	Lecture_name
001	Database
002	Information theory
003	Computer network

One-way	Two-way
[Students] {"_id": "2023299001", "name": "Jeong-Hun Kim", "class": ["001", "002", "003", ...]} {"_id": "2023299002", "name": "Jae-Seong Yeon", "class": ["001", 003", ...]} {"_id": "2023299003", "name": "Jong-Pil Park", "class": ["002", ...]} [Lecture] {"_id": "001", "name": "Database"} {"_id": "002", "name": "Information theory"} {"_id": "003", "name": "Computer network"}	[Students] {"_id": "2023299001", "name": "Jeong-Hun Kim", "class": ["001", "002", "003", ...]} {"_id": "2023299002", "name": "Jae-Seong Yeon", "class": ["001", 003", ...]} {"_id": "2023299003", "name": "Jong-Pil Park", "class": ["002", ...]} [Lecture] {"_id": "001", "name": "Database", "students": ["2023299001", "2023299002", ...]} {"_id": "002", "name": "Information theory", "students": ["2023299001", "2023299003"]} {"_id": "003", "name": "Computer network", "students": ["2023299001", "2023299002"]}

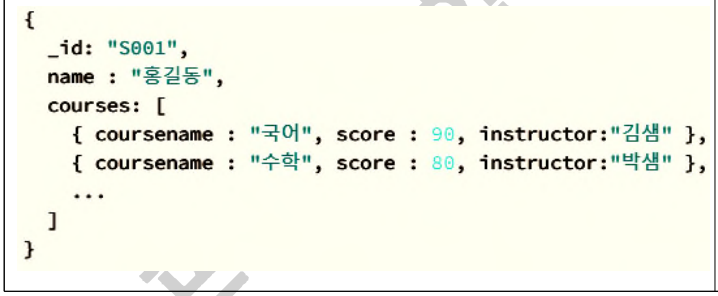
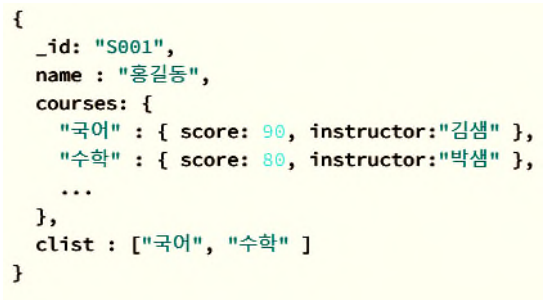
- 트리 패턴



Embedded Tree	Linked Document
	
전체 트리를 하나의 Document에 포함 액세스 속도 ↑, But 트리 커지면 복잡도 ↑	트리 구조를 분리된 Document에 포함 특정 노드 쿼리 유용, But 트리 업데이트 어려움

- Dynamic Field 패턴

· 필드 이름에 Key 대신 Data 저장, 저장공간 절약
But, Linked Document 구성 ↓, Document 전체 관리 ↓

기본 Document 구조	Dynamic Field 패턴
	

- Embedded Document vs. Linked Document

(Document 분산되어 독립적)

Embedded Document	특성	Linked Document
강함	**연결 관계**	약함
X	**일관성**	일관성 요구될 때 사용
사이즈 작을 때 사용	**Document 크기**	사이즈 클 때 사용
수정 적을 때 사용	**수정 빈도**	수정 잦을 때 사용
High Risk	**위험성**	.

[MongoDB 구성요소]

- DB: 여러 Collection에 대한 물리적인 컨테이너, 각 DB는 파일 시스템에서 고유의 파일 집합을 가짐
- Collection: MongoDB 도큐먼트의 그룹, RDBMS의 테이블과 유사

- DB 생성: use *DATABASE_NAME* → 새로 생성된 DB일 경우 물리적 할당 X, 가상 공간
  (DB가 이미 존재하면, 기존 DB를 리턴함)

- 현재 DB: db

- DB 목록: show dbs
  (가상의 DB이기 때문)
  (DB가 비어있을 경우, 목록에 표시되지 않음)
-
- DB Collection Document 삽입: db.mycollection.insert({name: "Jeoung-Hun Kim"})
  (삽입하면 공간 할당, 물리적 존재하여 목록 표시됨)
  (기존에 없던 Collection에 Document 삽입하면, Collection 자동 생성)

- 현재 DB 삭제: db.dropDatabase()

- Collection 생성: db.createCollection("newcollection", *options*)

- Collection 목록: show collections

- Collection 삭제: db.*COLLECTION_NAME*.drop()

[Capped Collection]

- Collection 용량을 제한함 (Document 개수 / 내부 데이터 용량, 원형 Queue 구조)

- Capped Collection 생성: db.createCollection("cappedCollection", {capped: true, size: 10000}

- Document 개수 제한(max): db.createCollection("capped", {capped: true, size: 10000, max: 1000})

- Capped 여부 확인: db.cappedCollection.isCapped()

- 기존 Collection Capped 전환: db.runCommand({"convertToCapped": "collectionName", size:10000})

- Capped Collection 옵션 변경: db.runCommand({"collMod": "collectionName", cappedSize: 10000})
  db.runCommand("collMod": "collectionName", cappedMax: 500))

[Collation]

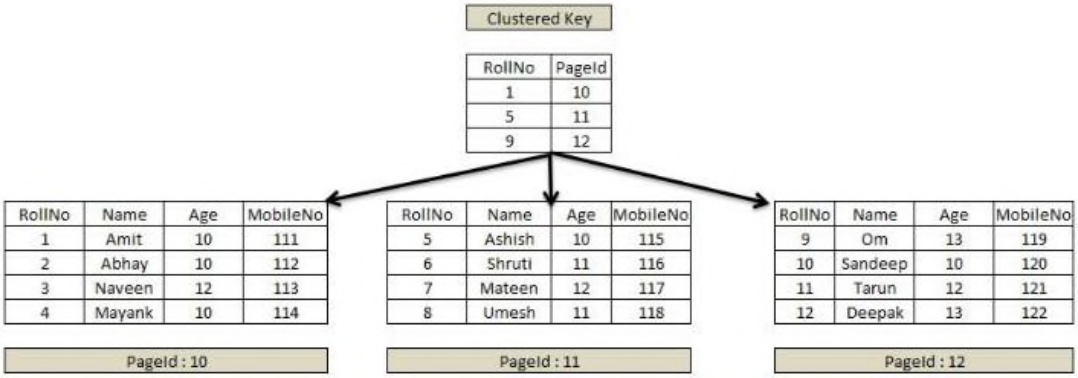
- String을 어떻게 비교하고 정렬할지 정의한 규칙 (MongoDB 기본: UTF-8)
- db.char_test.find({}, {_id:0}).sort({col:1})
- 옵션: locale(각 언어 기준) / caseLevel(대문자, 악센트) / caseFirst(대문자 우선) / strength(비교 레벨)
  numericOrdering(Numeric String을 문자 or 숫자로 비교할지 정함) / Alternate(공백, 점 무시할지)
  maxVariable(최대 비교 글자 수) / backwards(비교 방향) / normalization(텍스트 정규화 필요시)

[Time Series Collection]

- Time Series Data: 시간에 따른 변화를 분석하여 insights를 얻는 일련의 데이터
- Time Series Collection은 효율적으로 Time Series Data를 저장함
- 이점: Time Series Data 작업 복잡도 ↓, 쿼리 효율성 ↑, 디스크 사용량 ↓, 탐색 위한 I/O ↓

[Clustered Collection] → 페이징

- 그룹화된 인덱스로 생성되는 Collection, 이미 정렬된 상태로 물리적 저장



- 이점: 쿼리 속도 ↑ (Secondary Index 필요 X, 삽입 / 수정 / 삭제 / 탐색 등), 용량 절감

[Document 명령어]

- Document 삽입: db.artists.insert({artistname: "John Lande"}) // deprecated ↓
  (Document는 {key: value} or {field: value} 구조)

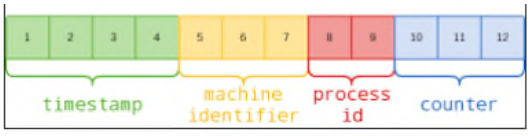
- Document 여러 개 삽입: db.artists.insert(

```
[
 {artistname: "The Kooks", genre: "pop"},
 {artistname: "Bastille", genre: "classic"},
 {artistname: "Gang of Four", genre: "rock"}
]
)
```

- Document 1개 삽입 (insertOne): db.*COLLECTION_NAME*.insertOne(document)

- Document 여러개 삽입 (insertMany): db.*COLLECTION_NAME*.insertMany(documents)

- "_id" 필드: Unique Key 역할, 지정하지 않으면 자동 설정

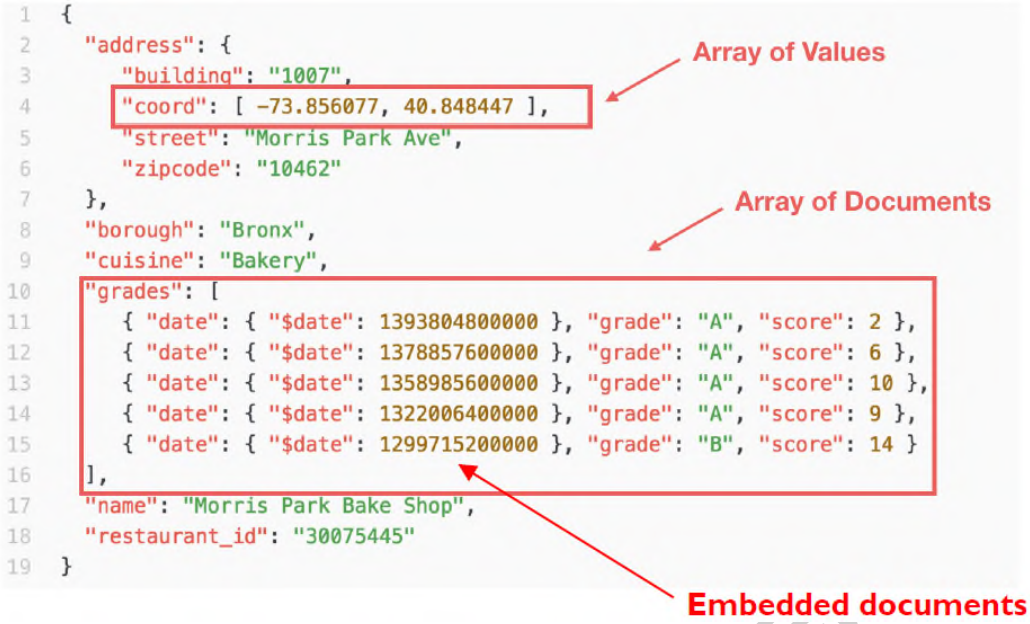


- Document 쿼리: db.musicians.find()
  (SQL SELECT * FROM musicians 유사함)

- Document 쿼리를 형식에 맞게 출력: db.musicians.find().pretty()

4. MongoDB Basic Queries

[Embedded Document 구조]




```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

[Embedded Document - 예시용 Collection 및 Document]

- db.inventory.insertMany([

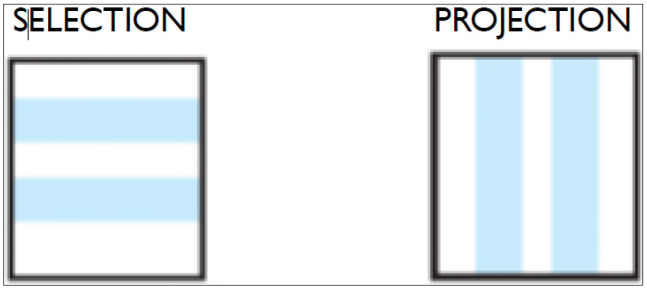
```
{item: "journal", qty: 25, size: {h: 14, w: 21, uom: "cm"}, status: "A"},
{item: "notebook", qty: 50, size: {h: 8.5, w: 11, uom: "in"}, status: "A"},
{item: "paper", qty: 100, size: {h: 8.5, w: 11, uom: "in"}, status: "D"},
{item: "planner", qty: 75, size: {h: 22.85, w: 30, uom: "cm"}, status: "D"},
{item: "paper", qty: 45, size: {h: 10, w: 15.25, uom: "cm"}, status: "A"}
```

  ])

[find() 메소드]

- MongoDB Collection에서 데이터를 쿼리하기 위해 사용
- db.*COLLECTION_NAME*.find(query, projection)

- 쿼리 종류



- Document 쿼리: db.inventory.find()
  (SQL SELECT * FROM inventory 유사함)

- Document 쿼리를 형식에 맞게 출력: db.inventory.find().pretty()

[쿼리 비교 연산자]

이름	기능
$eq	지정한 값과 일치하는 값을 가진 Document를 찾음
$gt	지정한 값보다 큰 값을 가진 Document를 찾음
$gte	지정한 값보다 크거나 같은 값을 가진 Document를 찾음
$in	지정한 배열 범위 내부에 있는 값을 가진 Document를 찾음
$lt	지정한 값보다 작은 값을 가진 Document를 찾음
$lte	지정한 값보다 작거나 같은 값을 가진 Document를 찾음
$ne	지정한 값과 일치하지 않는 값을 가진 Docuemnt를 찾음
$nin	지정한 배열 범위에 있지 않은 값을 가진 Document를 찾음

- $eq: db.inventory.find({qty: {$eq: 25}}) == db.inventory.find({qty: 25})
  (SQL SELECT * FROM inventory WHERE qty = 25 유사함)

- $gt, $gte, $lt, $lte: db.inventory.find({qty: {$gt: 25}})
  db.inventory.find({qty: {$gte: 25}})
  (SQL SELECT * FROM inventory WHERE qty > 25 (qty >= 25) 유사함)

- $ne: db.inventory.find({qty: {$ne: 25}})
  (SQL SELECT * FROM inventory WHERE qty != 25 유사함)

- $in: db.inventory.find({status: {$in: ["A", "D"]}})
  (SQL SELECT * FROM inventory WHERE status in ("A", "D")

[쿼리 논리 연산자]

이름	기능
$and	주어진 조건 배열을 모두 만족하는 Document를 찾음
$not	주어진 조건을 만족하지 않는 Document를 찾음
$nor	주어진 조건 배열을 모두 만족하지 않는 Document를 찾음
$or	주어진 조건 배열을 하나라도 만족하는 Document를 찾음

- $and: db.inventory.find({$and: [{status: "D"}, {qty: {$lte: 75}}]})
  == db.inventory.find({status: "D", qty: {$lte: 75}})
  (SQL SELECT * FROM inventory WHERE status = "D" AND qty <= 75

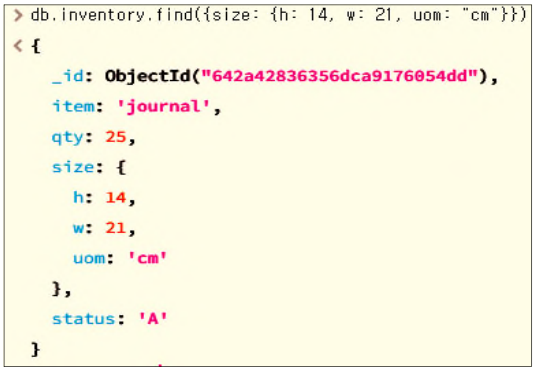
- $or: db.inventory.find({$or: [{status: "A"}, {qty: {$lt: 30}}]})
  (SQL SELECT * FROM inventory WHERE status = "A" OR qty <= 30

순서!

- $not: db.inventory.find({qty: {$not: {$gt: 75}}})

[Query Embedded Document]

{item: "journal", qty: 25, size: {h: 14, w: 21, uom: "cm"}, status: "A"},
{item: "notebook", qty: 50, size: {h: 8.5, w: 11, uom: "in"}, status: "A"},
{item: "paper", qty: 100, size: {h: 8.5, w: 11, uom: "in"}, status: "D"},
{item: "planner", qty: 75, size: {h: 22.85, w: 30, uom: "cm"}, status: "D"},
{item: "paper", qty: 45, size: {h: 10, w: 15.25, uom: "cm"}, status: "A"}



- db.inventory.find({size: {h: 14, w: 21, uom: "cm"}})

- Embedded Document에 대한 **값 전체 일치 여부**를 확인하기 위해서는 **Field 순서**를 비롯하여 **해당 Document의 Value와 정확히 일치해야** 한다. → Dot Notation 사용 권장 (쌍따옴표)

- db.inventory.find({size: {w: 21, h: 14, uom: "cm"}}) : 필드 순서가 달라 X
- db.inventory.find({size: {h: 21, w: 14}}) : 누락된 필드가 있어 X

- Dot Notation: db.inventory.find({"size.uom": "in"})
  db.inventory.find({"size.h": 14, "size.w": 21})
  db.inventory.find({"size.h": {$lt: 15}, "size.uom": "in", status: "D"})

[Projection]

- 쿼리문에서 일치하는 Document에 대해 리턴할 Field를 결정 (0: 리턴 X, 1: 리턴 O)

- db.inventory.find({ }, {qty: 1}) → _id 필드와 qty 필드 출력, 나머지는 X
- db.inventory.find({ }, {qty: 1, _id: 0}) → qty 필드만 출력, _id 필드와 나머지는 X
- db.inventory.find({ }, {qty: 0}) → qty 필드만 출력 X, 나머지는 모두 출력

[Query Operator with Projection]

- db.inventory.find({item: "paper"}, {size: 0, _id: 0})
  → "item" 필드 값이 "paper"인 Document에서 _id, size 필드를 제외한 나머지 필드 출력
  (SQL SELECT item, qty, status FROM inventory WHERE item = "paper" 유사함)

[Document 업데이트]

- db.inventory.update({item: "paper"}, {$set: {item: "paperless"}}) // deprecated ↓
  → 기본적으로 MongoDB는 값이 일치하는 첫 번째 Document만 업데이트함 updateOne() updateMany()

- db.inventory.update({item: "paper"}, {$set: {item: "paperless"}}, {multi: true})
  → 값이 일치하는 모든 Document를 업데이트함
  (SQL UPDATE inventory SET item = "paperless" WHERE item = "paper" 유사함)

[Document 삭제]

- db.inventory.remove({item: "journal"}) // deprecated ↓
- deleteOne(), deleteMany()

- db.inventory.remove({item: "notebook"}, 1) : 값이 일치하는 첫 번째 Document만 삭제함

- db.inventory.remove({ }) : 모든 Document를 삭제함

[Basic Queries 실습 - Projection]

- "restaurants" Collection에서 모든 Document에 대해 restaurant_id, name, borough, cuisine 필드 출력
  → db.restaurants.find({ }, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1})

- "restaurants" Collection에서 모든 Document에 대해 _id 필드 제외한 …, zip code(Nested) 필드 출력
  → db.restaurants.find({ }, {_id: 0, restaurant_id: 1, name: 1, borough: 1, "address.zipcode": 1})

[Basic Queries 실습 - Comparison Operators]

- borough가 "Queens"에 속하는 식당의 restaurant_id, name, borough, cuisine 필드 출력
  → db.restaurants.find({borough: "Queens"}, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1})

- borough가 "Queens"에 속하지 않는 식당의 restaurant_id, name, borough, cuisine 필드 출력
  → db.restaurants.find({borough: {$ne: "Queens"}}, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1})

[Basic Queries 실습 - Embedded Document 쿼리]

- "grades.grade" 값이 "A"인 식당의 restaurant_id, name, borough, cuisine 필드 출력
  → db.restaurants.find({"grades.grade": "A"}, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1})
  → db.restaurants.find({grades: {$elemMatch: {grade: "A"}}}, {restaurant_id: 1, …, cuisine: 1})

- "American" cuisine을 제공하지 않고 grades.score 값이 70 초과인 식당 출력
  → db.restaurants.find({cuisine: {$ne: "American"}, "grades.score": {$gt: 70}})
  → db.restaurants.find({cuisine: {$not: {$regex: "American"}}, "grades.score": {$gt: 70}})

[Basic Queries 실습 - 쿼리 연산자 $or]

- borough가 "Staten Island", "Queens", "Bronx", "Brooklyn" 중 하나에 속하는 식당의 …, cuisine 필드 출력
  → db.restaurants.find({borough: {$in: ["Staten Island", "Queens", "Bronx", "Brooklyn"]}}, {…, cuisine: 1})
  → db.restaurants.find({$or: [{borough: "Staten Island"}, …, {borough: "Brooklyn"}]}, {…, cuisine: 1})

- borough가 "Staten Island", "Queens", "Bronx", "Brooklyn"에 속하지 않는 식당의 …, cuisine 필드 출력
  → db.restaurants.find({borough: {$nin: ["Staten Island", "Queens", "Bronx", "Brooklyn"]}}, {…, cuisine: 1})
  → db.restaurants.find({$nor: [{borough: "Staten Island"}, …, {borough: "Brooklyn"}]}, {…, cuisine: 1})

[Basic Queries 실습 - 쿼리 연산자 $and]

- "American" cusine 제공하지 않고 "grades.score" 70 초과하고 borough가 "Brooklyn" 속하지 않는 식당
  → db.restaurants.find({cuisine: {$ne: "American"}, "grades.grade": {$gt: 70}, borough: {$ne: "Brooklyn"}})
  → db.restaurants.find({cuisine: {$not: {$regex: "American"}}, …, borough: {$ne: "Brooklyn"}})

[Basic Queries 실습 - Final Task]

- "Hamburgers" cuisine 제공하고 "grades.grade" 값이 "A"이고 borough가 "Manhattan", "Queens", "Staten Island", "Bronx"에 속하지 않는 식당의 _id 필드 제외한 restaurant_id, …, cuisine 필드 출력
  → db.restaurants.find({cuisine: "Hamburgers", "grades.grade": "A", borough: {$nin: ["Manhattan", "Queens", "Staten Island", "Bronx"]}}, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id:0})

5. MongoDB Intermediate Query (Manipulating Data)

[Array of Values에 대한 쿼리 - 예시용 Collection 및 Documents]

- db.inventory2.insert([
  {item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [14, 21]},
  {item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [14, 21]},
  {item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [14, 21]},
  {item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [22.85, 30]},
  {item: "paper", qty: 45, tags: ["blue"], dim_cm: [10, 15.25]}
  ])

[Array of Values에 대한 쿼리]

- db.inventory2.find({tags: ["red", "blank"]})
  → 2번째 Docuement // 배열 Element의 순서까지 완전히 일치하는 Document만 리턴함

- db.inventory2.find({tags: "red"})
  → 1~4번째 Document // 배열에 "red" Element 포함하는 모든 Document 반환. 너무 많은 데이터 쿼리

[Array of Values의 인덱스에 따른 쿼리]

- db.inventory2.find({"dim_cm.1": {$gt: 25}}) // zero-based indexing
  → 4번째 Document // dim_cm 필드의 Value인 배열의 두 번째 Element 값이 25보다 큰 Document 리턴

[Array of Documents에 대한 쿼리 - 예시용 Collection 및 Documents]

- db.inventory3.insert([
  {item: "NORWAY", instock: [{warehouse: "A", qty: 5}, {warehouse: "C", qty: 15}]},
  {item: "notebook", instock: [{warehouse: "C", qty: 5}]},
  {item: "paper", instock: [{warehouse: "A", qty: 60}, {warehouse: "B", qty: 15}]},
  {instock: [{warehouse: "A", qty: 40}, {warehouse: "B", qty: 5}]},
  {item: null, instock: [{warehouse: "B", qty: 15}, {warehouse: "C", qty: 35}]}
  ])

[Array of Documents에 대한 쿼리]

- db.inventory3.find({"instock": {warehouse: "A", qty: 5}})
  → 1번째 Document, "instock" 배열의 Element 순서까지 완전히 일치해야 리턴함

- db.inventory3.find({"instock": {qty: 5, warehouse: "A"}})
  → "instock" 배열의 Element 순서가 일치하지 않아 리턴값 X

- db.inventory3.find({"instock.qty": {$gte: 20}}) → 3, 4, 5번째 Document
  → "instock" 배열에 20 이상 값의 "qty" 필드를 가진 Embedded Document를 가진 모든 Document 반환

- db.inventory3.find({"instock.0.qty": 5}) → 1, 2번째 Document
  → "instock" 배열의 첫 번째 Embedded Document 요소에 "qty" 필드 존재하고 값이 5인 Document 반환

- db.inventory3.find({"instock.0.qty": {$lte: 20}}) → 1, 2, 5번째 Document
  → "instock" 배열의 첫 번째 Embedded Document 요소에 "qty" 필드 있고 값 20 이하인 Document 반환

[Element 연산자]

이름	기능
$exists	지정한 필드가 존재하는 Document를 찾음
$type	특정 필드가 지정한 자료형인 Document를 찾음

[Array of Documents에 대한 쿼리 - 예시용 Collection 및 Documents]

- db.inventory3.insert([
  {item: "NORWAY", instock: [{warehouse: "A", qty: 5}, {warehouse: "C", qty: 15}]},
  {item: "notebook", instock: [{warehouse: "C", qty: 5}]},
  {item: "paper", instock: [{warehouse: "A", qty: 60}, {warehouse: "B", qty: 15}]},
  {instock: [{warehouse: "A", qty: 40}, {warehouse: "B", qty: 5}]},
  {item: null, instock: [{warehouse: "B", qty: 15}, {warehouse: "C", qty: 35}]}
  ])

[Equality Filter]

- 필드의 값이 null이거나 해당 필드가 아예 존재하지 않는 Document를 찾음

- db.inventory3.find({item: null})
  → 4번째 Document ("item" 필드 없음), 5번째 Document ("item" 필드 값이 null)

- db.inventory3.find({item: {$ne: null}})
  → 1, 2, 3번째 Document // "item" 필드가 존재하고 그 값이 null이 아닌 Document 반환

[Existence Check: $exists]

- 지정한 필드가 존재하는 Document를 찾음

- db.inventory3.find({item: {$exists: false}})
  → 4번째 Document ("item" 필드 없음)

- db.inventory3.find({item: {$exists: true}}, {_id: 0, item: 1})
  → 1, 2, 3번째 / 5번째 Document (값은 null이지만 "item" 필드 존재), "item" 필드만 출력

[BSON: $type]

- 특정 필드가 지정한 자료형(BSON)인 Document를 찾음
- 구조화되지 않아 자료형을 예측할 수 없는 데이터를 다룰 때 유용함

Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary Data	5	"binData"	
Undefined	6	"undefined"	Deprecated
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated
Javascript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit Integer	18	"long"	
Decimal128	19	"decimal	New in Version 3.4
Min Key	-1	"minKey"	
Max Key	127	"maxKey"	

[$type - 예시용 Collection 및 Documents]

- db.grades.insertMany([
  {"_id": 1, name: "Alice King", classAverage: 87.33333}, // double
  {"_id": 2, name: "Bob Jenkins", classAverage: "83.52"},
  {"_id": 1, name: "Cathy Hart", classAverage: "94.06"},
  {"_id": 1, name: "Drew Williams", classAverage: NumberInt("93")}, // 32-bit Integer

[$type - 특정 자료형 반환]

- db.grades.find({classAverage: {$type: "string"}})
  → 2, 3번째 Element, "classAverage" 필드의 자료형이 string인 Document를 찾음

- db.grades.find({classAverage: {$type: 2}})
  → 2, 3번째 Element, $type에 해당하는 값이 2인 경우 매칭되는 자료형이 string임

- db.grades.find({classAverage: {$type: "number"}})
  → 1, 4번째 Element, $type은 숫자에 대한 alias도 지원함: Double, 32-bit Integer, 64-bit Integer, Decimal

[$type - 특정 배열에 속하는 자료형 반환]

- db.grades.find({classAverage: {$type: [2, 1]}})
- db.grades.find({classAverage: {$type: ["string", "double"]}})
  → 1, 2, 3번째 Element, "classAverage" 필드의 자료형이 string 혹은 double인 Document를 찾음

[Evaluation 연산자]

이름	기능
$expr	쿼리문 내부에서 aggregation(집계) 표현식을 사용하게 함
$jsonSchema	지정된 JSON 스키마에 대한 Document의 유효성을 검증함
$mod	필드의 값에 대한 나머지 연산과 함께 지정된 결과가 존재하는 Document를 선택함
$regex	지정된 정규식 표현에 해당되는 Document를 선택함
$text	텍스트 검색 수행함
$where	JavaScript 표현식을 만족하는 Document를 매칭함

[Evaluation 연산자 - 예시용 Collection 및 Document]

- db.inventory.insertMany([
    {item: "journal", qty: 25, size: {h: 14, w: 21, uom: "cm"}, status: "A"},
    {item: "notebook", qty: 50, size: {h: 8.5, w: 11, uom: "in"}, status: "A"},
    {item: "paper", qty: 100, size: {h: 8.5, w: 11, uom: "in"}, status: "D"},
    {item: "planner", qty: 75, size: {h: 22.85, w: 30, uom: "cm"}, status: "D"},
    {item: "paper", qty: 45, size: {h: 10, w: 15.25, uom: "cm"}, status: "A"}
  ])

[$regex - 논리 연산자(정규식)]

- 쿼리문에서 string에 대한 패턴을 매칭하는 정규식 표현 기능을 제공함

- 양식:
  - {<field>: {$regex: /pattern/, $options: '<options>'}}
  - {<field>: {$regex: 'pattern', $options: '<options>'}} 구분!
  - {<field>: {$regex: /pattern/ <options>}}

- db.inventory.find({item: {$regex: "paper"}})
  → 3, 5번째 Element, "item" 필드의 값에 "paper" 키워드를 포함한 Document를 선택함

[$regex - 시작 글자, 끝 글자로 검색]

- db.inventory.find({item: {$regex: "^note"}})
  → 2번째 Element, "item" 필드의 값이 "note" 키워드로 시작하는 Document를 선택함

- db.inventory.find({item: {$regex: "nal$"}})
  → 1번째 Element, "item" 필드의 값이 "nal" 키워드로 끝나는 Document를 선택함

- db.inventory.find({item: {$regex: "PAPER", $options: 'i'}})
  → 3, 5번째 Element, $options: 'i'를 사용하면 대소문자 구분하지 않음

- db.inventory.find({status: "A", $or: [{qty: {$lt: 30}}, {item: {$regex: "^p"}}]})
  → 1, 5번째 Element, "status" 값이 "A"이고, "qty" 값이 30보다 작거나 "item" 값이 "p" 키워드로 시작

[Intermediate Queries 실습 - restaurants Collection 구조]


```
"address": {
   "building": "1007",
   "coord": [ -73.856077, 40.848447 ],
   "street": "Morris Park Ave",
   "zipcode": "10462"
},
"borough": "Bronx",
"cuisine": "Bakery",
"grades": [
   { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
   { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
   { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 }
,
   { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
   { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
],
"name": "Morris Park Bake Shop",
"restaurant_id": "30075445"
```


[Intermediate Queries 실습 - Dot Notation]

- "restaurants" Collection에서 "grades.grade" 값이 "A"인 식당의 restaurant_id, …, cuisine 필드를 출력
  → db.restaurants.find({"grades.grade": "A"}, {_id: 0, restaurant_id: 1, …, cuisine: 1})

- "American" cuisine을 제공하지 않고 grades.score 값이 70 초과인 식당을 출력
  → db.restaurants.find({cuisine: {$ne: "American"}, "grades.score": {$gt: 70}})

[Intermediate Queries 실습 - Array of Values에 대한 쿼리]

- "address.coord" 필드에서 배열 요소인 latitude = -73.85이고, longitude = 40.84인 모든 식당을 출력
  → db.restaurants.find({"address.coord": [-73.85, 40.84]})

- "American" 또는 "Chinese" cuisine을 제공하고, "grades.score"이 60보다 크고, latitude가 -74 미만
  → db.restaurants.find({cuisine: {$in: ["American", "Chinese"]}, "grades.score": {$gt: 70},
  "address.coord.0": {$lt: -74}})

[Intermediate Queries 실습 - Array of Documents에 대한 쿼리]

- "grades" 배열의 두 번째 요소가 "grade" 필드 값 "A"를 포함하고 "score"이 9, ISODate가 …인 식당
  → db.restaurants.find({"grades.1.grade": "A", "grades.1.score": 9,
  "grades.1.date": ISODate ("2014-08-11T00:00:00.000+00:00")})

- "grades" 배열의 8번째 요소가 "score" 값 30을 초과하는 식당의 restaurant_id, name, score 출력
  → db.restaurants.find({"grades.7.score": {$gt: 30}}, {restaurant_id: 1, name: 1, "grades.score": 1})

[Intermediate Queries 실습 - 정규식 표현]

- "name" 필드에 "Pizza" 키워드를 포함하는 식당 출력
  → db.restaurants.find({name: {$regex: "pizza"}})

- "borough" 필드가 "Staten Island" 또는 "Queens"이며, "name" 필드가 "Wen" 세 글자로 시작하는 식당
  → db.restaurants.find({borough: {$in: ["Staten Island", "Queens"]}, name: {$regex: "^Wen"}})

[Intermediate Queries 실습 - Existence Check]

- 모든 Document의 "address" 필드가 "street" 필드를 포함하고 있는지 아닌지 확인
  → db.restaurants.find({"address.street": {$exists: false}})

[Intermediate Queries 실습 - $type]

- 모든 Document의 "address.coord" 필드 값이 배열 형식인지 확인
  → db.restaurants.find({"address.coord": {$not: {$type: "array"}}})

[Intermediate Queries 실습 - 복합]

- AirBnb DB의 "reviews" Collection에서 "weekly_price" 필드가 존재하고,
  "name" 필드가 "Pr" 키워드로 시작하며, "review_scores.review_scores_rating" 값이 80 이상인 Document
  → db.reviews.find({weekly_price: {$exists: true}, name: {$regex: "^Pr"},
  "review_scores.review_scores_rating": {$gte: 80}})