02. 2주차 스터디 내용

1. Replica Set

1.1 레플리카 셋 구성하기

1.1.1 레플리카 셋이란?

- 고가용성 환경을 위해 필요한 기술
- 자체적인 복제 기능
- 3대 이상의 구성원을 가지는 것을 권장. 홀수로 노드를 유지하는 것을 권장

1.1.2 레플리카 셋의 구성원

- Primary
- Secondary
- Arbiter

1.1.3 레플리카 셋의 선거

- 구성원의 과반이 동의 하는 것
- 투표권은 최대 7대까지
- 선출된 Primary 보다 최신의 데이터를 가진 Secondary가 있으면 해당 구성원을 Primary에 맞춰 Rollback
- WriteConcern 옵션을 조정해 쓰기 작업에 대한 Rollback을 방지

1.2 레플리카 셋 구성 명령

1.2.1 초기 구성하기

- `rs.initiate()` 명령을 통해 레플리카 셋 초기 구성
- Primary에서 한번만 실행

```
> rs.initiate( {
   _id : "rs0",
   members: [
      { _id: 0, host: "mongodb01:27017" },
      { _id: 1, host: "mongodb02:27017" },
      { _id: 2, host: "mongodb03:27017" }
   ]
})
{
  "ok" : 1,
  "$clusterTime" : {
    "clusterTime" : Timestamp(1609920364, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  },
```

```
  "operationTime" : Timestamp(1609920364, 1)
}
```

- _id: 레플리카 셋 이름
- host: 구성원들의 ip 또는 도메인 주소

1.2.2 레플리카 셋 확인

- `rs.conf()` 명령으로 확인 가능
- `rs.isMaster()` 명령으로 Primary 조회 가능

1.3 레플리카 셋 세부 설정

1.3.1 `rs.conf()` 명령으로 알 수 있는 것들

rs.conf()

Aa 필드	설명
_id	구성원 멤버를 구분하는 값으로 0부터 시작하고, 멤버가 추가될 때 마다 1씩 증가해서 설정해야 한다.
arbiterOnly	기본값은 false. 구성원을 arbiter로 설정하는 경우 true로 설정한다.
buildIndexes	priority가 0이 아닐때 기본값은 true. Primary를 따라 인덱스를 백업할 것인지를 정한다.
hidden	기본값은 false. true로 바꾸면 드라이버가 구성원에 대한 명령을 내릴수 없고, isMaster() 명령으로 구성원의 정보를 확인할 수도 없다.
priority	장애시 primary 선출을 위한 선거가 열렸을때 해당 구성원이 Primary로 선출될 상대적인 가능성을 의미한다. 값이 0인 경우 절대 Primary가 될 수 없다.
tags	태그 도큐먼트를 넣어서 복제 구성원을 그룹에 따라 나눌수 있다. 태그를 이용하면 WriteConcern 시 이용할 수 있다.
slaveDelay	기본값은 0. Secondary가 Primary로부터 동기화 할 때 일정시간 딜레이를 가지고, 설정된 시간만큼 데이터를 유지한다. 딜레이 설정을 하게되면 priority 값은 0이 되어야 한다.
votes	투표권은 기본적으로 구성원당 1표씩 가지고 있다. 이 값에 따라 선거 발생시 해당 구성원의 투표수가 결정된다.

- `rs.reconfig(<option>)` 명령을 통해 변경 가능

1.3.2 새로운 구성원 추가

- `rs.add()` : Secondary 추가

```
> rs.add(
  {
    _id: <int>,
    host: <string>,
    arbiterOnly: <boolean>,
    buildIndexes: <boolean>,
    hidden: <boolean>,
```

```
    priority: <number>,
    tags: <document>,
    slaveDelay: <int>,
    votes: <number>
  }
)
```

- `rs.addArb()` : Arbiter 추가

```
> rs.addArb("mongodb0a:27017")
{
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1611125667, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1611125667, 1)
}
```

1.3.3 구성원 제거

- 멤버 제거 절차
  1. 제거할 멤버의 mongod을 중지
  2. 프라이머리에 접속
  3. `rs.remove("mongod03:27017")` or `rs.remove("mongod3.example.net")`

1.4 복제 아키텍처

- 세컨더리는 프라이머리에서 OpLog를 가져온 다음 OpLog를 재생해서 데이터를 동기화
- 세컨더리 멤버는 프라이머리 뿐만 아니라 다른 세컨더리 멤버의 OpLog를 재생할 수도 있음

1.4.1 OpLog

- Operation Log의 약자
- 복제를 위해서만 사용
- 컬렉션의 레코드 형태로 저장
- 저널로그와 주체가 다르다. OpLog는 MongoDB 엔진이 처리, 저널로그는 스토리지 엔진이 처리

1.4.2 OpLog 구조

- OpLog는 데이터베이스 서버에 'oplog.rs' 라는 컬렉션으로 기록
- 항상 모든 필드가 존재하는 것은 아니며, 필요에 따라 생성되는 필드가 존재

ts(Timestamp): 저장순서를 결정. 동기화를 잠시 중단하거나 재시작할 때 기준.
t(Primary Term): 레플리카 셋의 Primary 선출하는 투표가 실행시 증가.
h(Hash): OpLog의 도큐먼트는 Primary 멤버에서 실행된 데이터 변경작업을 의미, 각각의 작업에는 OpLog의 해시 값을 이용해서 식별자가 할당되는데 이 식별자를 h 필드에 저장.
v(Version): 도큐먼트의 버전을 의미.
op(Operation Type): i(Insert),d(Delete),u(Update),c(Command),n(No Operation) 등 오퍼레이션 종류를 저장. n은 단순 정보 저장.
ns(Namespace): 데이터가 변경된 컬렉션의 네임스페이스가 저장.
o(Operation): op필드에 저장된 오퍼레이션 타입별로 실제 변경된 정보가 저장
o2(Operation 2): op필드가 u인 경우에만 o2 필드가 존재. 업데이트 될 대상 도큐먼트의 _id 정보를 저장.

1.4.3 local 데이터베이스

- MongoDB의 기본 데이터베이스 중 하나로 설치하면 생성됨
- oplog.rs를 포함하여 해당 노드의 DB 스스로 만을 위한 컬렉션이 저장
- local 데이터베이스 안의 내용들은 Secondary로 복제되지 않음
- 모니터링 데이터, 백업 이력 등 복제하지 않아도 되는 데이터를 생성, 관리할 수 있음
- Secondary 멤버들도 local 데이터베이스에는 데이터 입력, 수정 및 삭제가 가능

1.4.4 초기 동기화(Initial Sync)

- 처음 설치 후 비어있는 데이터베이스를 레플리카 셋에 투입하면 이미 투입되어 있는 멤버들로 부터 모든 데이터를 일괄적으로 가져옴
- 데이터가 있는 서버의 경우 이 과정을 무시
- 부트스트랩(bootstrap): 데이터가 있는 DB를 레플리카 셋에 투입하는 것을 의미
- 초기 동기화는 단일 쓰레드로 진행되기 때문에 시간이 오래 걸림
- 초기 동기화의 경우 중간에 멈췄다가 다시 시작하는 경우 처음부터 다시 시작해야함
- 초기 동기화 방법에는 'Auto'와 'Manual' 2가지 방법이 있음.

1.4.4 수동 초기 동기화

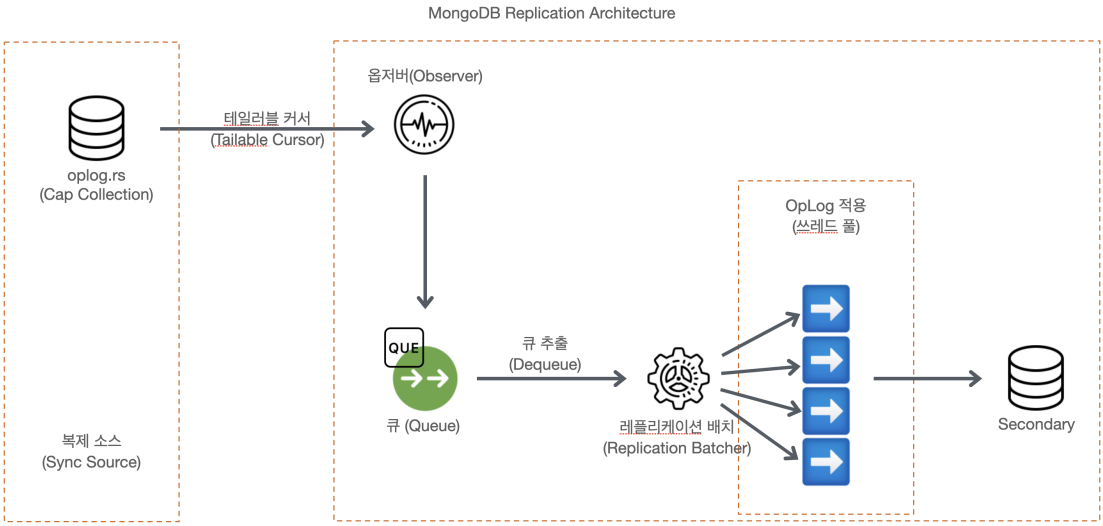
- 다른 레플리카 멤버의 데이터 파일을 복사한 후 레플리카 셋에 투입
- 데이터 파일을 복사하기 위해 해당 멤버의 DB를 종료한 상태에서 복사를 진행.
- LVM 스냅샷 백업을 사용하는 것도 가능

1.4.5 자동 초기 동기화

- 사용자가 특별히 해줄것은 없음
- 초기 데이터를 복제하고 복제하는 동안 변경된 OpLog를 적용한 후 인덱스를 생성

1.4.6 실시간 복제

- 언젠가는 Primary와 Secondary의 데이터가 동기화 되는 것을 최종 일관성이라고 표현



- MongoDB 복제 아키텍처
  1. MongoDB가 처리한 모든 데이터 변경 내용을 Cap 컬렉션 구조의 oplog.rs 컬렉션에 저장
  2. 테일러블 커서를 통해 최신 데이터 전송한다.
  3. Secondary의 백그라운드 쓰레드(Observer)의해 큐에 일정 개수의 OpLog를 담는다.
  4. 리플리케이션 배치 쓰레드는 큐에서 일정 개수의 OpLog를 가져와 OpLog 적용 쓰레드에 맞게 작업량을 나눈 다음 작업을 요청한다.
     - 테일러블 커서: Cap 컬렉션에서 지원하는 리눅스의 Tail 이 화면에 보여주는 것처럼 큐 방식으로 동작하는 커서
     - Observer: Secondary 멤버의 OpLog 수집을 위한 백그라운드 쓰레드. 수집하여 큐에 쌓는 역할만 한다.
     - 리플리케이션 배치: 큐에서 OpLog를 가져와 적용 쓰레드 개수에 맞게 작업량을 나눈 다음 작업을 요청
     - OpLog 적용 쓰레드(Applier): 기본값으로 16개를 사용하고 쓰레드는 각각 5,000개의 OpLog의 아이템을 담으 수 있는 자체 캐시 메모리를 갖고 있다. 최대 512MB의 메모리를 사용할수 있게 제한되어 있으며, 16개의 OpLog 적용 쓰레드 전체 8GB의 자체 캐시 메모리를 사용할 수 있다.
- Primary 멤버는 모든 멤버의 복제 상태를 가지고 있다.

1.4.7 OpLog 컬렉션 크기 설정

- oplog.rs의 컬렉션 크기에 따라 OpLog를 얼마나 담을 수 있느냐가 결정되며, 세컨더리가 허용 가능한 지연시간이 결정된다.
- OpLog의 크기를 명시하지 않으면 Disk Free space에서 5%정도를 OpLog의 크기로 결정

1.4.8 OpLog의 정보 확인

- `db.getReplicationInfo()` JSON 포맷의 결과
- `db.printReplicationInfo()` 단순 텍스트 포맷의 결과

2. MongoDB의 스토리지 엔진

2.1 WiredTiger 스토리지 엔진

2.1.1 wiredTiger란?

- WiredTiger 스토리지 엔진은 내부적인 Lock 경합 최소화를 위해서 Hazard-Pointer나 Skip-List와 같은 많은 신기술을 채택하고 있으며, 최신 RDBMS들이 가지고 있는 MVCC와 데이터 파일 압축, 암호화 기능등을 갖추고 있다.
- WiredTiger 스토리지 엔진의 가장 자주 쓰는 기본적인 파라미터를 몇 가지
  - **dbPath**: 데이터 파일이 저장되는 경로. MongoDB는 저널로그나 OpLog가 모두 이 디렉토리의 하위에 저장된다. 다른 디스크나 파티션에 저장하고 싶을때는 심볼릭링크를 사용하는 것이 좋음.
  - **directoryPerDB**: 데이터베이스 단위로 디렉토리를 생성할 지 dbPath에 모두 생성할 지 결정한다. true로 설정하면 데이터베이스 별로 생성한다.
  - **journal**: 서버의 저널로그(=트랜잭션 로그)를 활성화할 것인지 결정.
    - **journal.enabled**를 false로 하면 저널로그를 디스크에 기록하지 않음.
    - **journal.commitIntervalMs** 옵션은 디스크에 동기화 간격을 결정. MongoDB는 트랜잭션 단위로 저널 로그를 디스크에 동기화 하지 않기 때문에 해당 옵션에서 설정한 밀리초 단위로 주기적으로 저널로그를 디스크에 동기화한다.

2.1.2 WiredTiger 스토리지 엔진의 저장 방식

WiredTiger 스토리지 엔진은 3가지 타입의 저장소를 가지고 있다.

- **레코드(도큐먼트) 스토어:** 일반적인 RDBMS가 사용하는 저장 방식. 테이블의 레코드를 한꺼번에 같이 저장하는 방식. B-Tree 알고리즘 사용
- **컬럼 스토어:** 대용량 분석(OLAP 또는 DW)용도로 주로 사용. 테이블의 레코드와 상관없이 각 컬럼 패밀리로 데이터 파일을 생성하므로 데이터파일의 크기가 작고 속도가 빨라 대용량 분석에 적합
- **LSM(Log Structured Merge Tree) 스토어:** HBase나 카산드라와 같은 NoSQL에서 자주 사용하는 저장 방식으로, 데이터 읽기보다는 쓰기에 집중한 저장방식. B-tree로 동작하지 않고 순차 파일형태로 데이터를 저장. 메모리의 저장 가능한 크기의 조각으로 데이터 파일을 관리하는데 메모리의 저장 가능한 한계를 넘어서면 디스크에 저장.

현재 MongoDB에서는 레코드 스토어 방식의 저장 방식만 사용한다.

2.1.3 WiredTiger 스토리지 엔진의 내부 동작 방식

- WiredTiger 스토리지 엔진은 다은 DBMS와 동일하게 B-Tree 구조의 데이터 파일과 서버 크래시로부터 데이터를 복구하기 위한 저널로그(WAL, Write Ahead Log)를 가지고 있음.
- Oracle의 Redo처럼 파일 내에서 로테이션 되는 방식이 아니라 새로운 로그파일을 생성하고, 지난 사용하지 않는 파일은 자동으로 삭제하는 방식.
- 저널로그의 아카이빙이나 크기는 MongoDB의 설정 파일로 제어할 수 없으며, 다음과 같이 WiredTiger의 설정 옵션을 MongoDB에 명시해야 함

```
storage:
  ...
  engine: wiredTiger
      engineConfig:
          cacheSizeGB: 10
          configString: "log=(archive=true, enabled=ture, file_max=100MB, path=/log/journal)"
      collectionConfig:
          blockCompressor: snappy
```

  - **enabled:** 저널로그를 활성화 할 것인지를 설정. true 활성, false 비활성
  - **archive:** 기본값으로 체크포인트 이전의 저널로그를 자동 삭제하는데, 이 것을 삭제하지 않고 아카이빙하려면 true로 설정
  - **file_max:** 저널로그의 최대 크기
  - **path:** 저널로그를 다른 경로에 저장하고자 할 때 설정
- 사용자가 쿼리를 실행하면 WiredTiger 스토리지 엔진은 블록 매니저를 통해서 필요한 데이터 블록을 디스크에서 읽어 공유 캐시에 적재하여 쿼리를 처리

2.2 wiredTiger의 구성

2.2.1 공유 캐시

- WiredTiger는 내부에 공유 캐시를 가지고 있음.
- 4.2 버전부터는 OS 페이지 캐시를 사용하던 MMAPv1 스토리지가 완전히 배제.
- 공유 캐시의 최적화는 MongoDB의 처리 성능에 있어 매우 중요. WiredTiger 스토리지 엔진의 처리가 원활하지 못한경우 공유 캐시 사용량 그래프의 변화를 보는 경우가 많다.

  일반적으로 RAM의 50%, 최소 256MB를 캐시 메모리로 사용. MongoDB가 사용할 캐시 사이즈를 구하는 방법

  > Total RAM 4GB, (0.5 x (4GB – 1GB) = 1.5GB
  > Total RAM 1.25GB, (0.5 x (1.25GB – 1GB)) = 128MB < 256MB

  전체 RAM의 사이즈가 1.25GB를 가진 경우 계산식에 의해 나오는 값이 256MB보다 작기 때문에 그냥 256MB를 사용한다. 디스크의 인덱스나 데이터 파일을 메모리에 캐시하여 빠르게 쿼리를 처리할뿐만 아니라, 데이터의 변경을 모아서 한번에 디스크로 기록하는 쓰기 배치 기능도 가지고 있다.
- 공유 캐시는 MongoDB 서버를 재시작하지 않고도 크기 조정이 가능합니다.

```
db.adminCommand({"setParameter": 1,
                 "wiredTigerEngineRuntimeConmfig": "cache_size=25G"})
```

- 일반적으로 RDBMS는 데이터 페이지 이미지를 그대로 캐시로 적재하는데, 그래서 캐시에 적재된 B-Tree 노드를 통해 자식 노드를 찾아 갈때도 B-Tree상의 주소를 사용하는데, 공유 캐시에 적재되어도 똑같은 구조를 가지고 있으며, 트리를 찾아가는 과정에서 계속 페이지와 실제 메모리상의 주소로 맵핑(별도의 해시맵)을 참조하게 된다. 하지만 WiredTiger 스토리지 엔진은 디스크의 데이터 페이지가 캐시로 공유될때 메모리에 적합한 트리 형태로 재구성하기 때문에 별도의 맵핑 없이 메모리 주소(C/C++ Pointer)를 이용해 바로 검색이 가능하기 때문에 맵핑 테이블의 경합이나 오버헤드가 없다.
- 일반적인 RDBMS에서는 하나의 데이터 페이지 내에 저장된 레코드들의 인덱스를 별도로 관리한다. 이는 Table Full Scan을 하지 않지않고 필요한 데이터만 빠르게 가져오기 위한 것인데 WiredTiger 스토리지 엔진은 레코드 인덱스를 별도로 관리하지 않고 저장하며, 데이터 페이지를 공유 캐시의 메모리에 적재할 때 레코드 인덱스를 새롭게 생성해서 매모리에 적재한다. 이런 과정에 의해 디스크에서 메모리로 적재하는 과정은 RDBMS보다 느리게 동작하지만, 캐시에 적재된 후 필요한 레코드를 검색하고 변경하는 작업은 RDBMS보다 빠르게 동작한다.

2.2.2 Hazard Pointer

- wiredTiger 스토리지 엔진에서 캐시에 메모리를 올리고, 사용 후 필요없게 되어 캐시에서 제거하는 방식이 이 Hazard Pointer를 이용
- wiredTiger 스토리지 엔진에서 “사용자 쓰레드”는 사용자의 쿼리를 처리하기 위해서 wiredTiger 캐시를 참조하는 쓰레드이며, “이빅션 쓰레드(Eviction Thread)”는 캐시가 다른 데이터 페이지를 읽어 들일 수 있도록 빈 공간을 만드는 역할을 담당한다.
- 모든 “사용자 쓰레드”는 wiredTiger 캐시의 데이터 페이지를 참조할 때, 먼저 Hazard Pointer에 자신이 참조하는 페이지를 등록합니다. 쿼리가 처리되는 동안 “이빅션 쓰레드”는 캐시에서 제거해야할 데이터 페이지를 골라 캐시에서 삭제하는 작업을 하게되는데 이때 제거해도 될만한 페이지(자주 사용되지 않는 페이지)를 골라서 먼저 하자드 포인터에 등록되어 있는지 확인하고, 등록되어 있는 페이지는 무시하고, 등록되어 있지 않는 페이지는 캐시에서 삭제한다.
- wiredTiger 스토리지 엔진 사용할 수 있는 Hazard Pointer의 기본값은 1000개이며, configString에 hazard_max 옵션 값을 설정하여 변경이 가능하다. 최소 15부터 1000개 이상 설정할 수 있다.

```
storage:
  ...
  engine: wiredTiger
    engineConfig:
      cacheSizeGB: 10
      configString: "hazard_max=2000"
```

2.2.3 Skip-List

- wiredTiger 스토리지 엔진에서 Lock-Free를 구현하기 위한 또 다른 기술이 스킵 리스트 자료 구조

- 스킵 리스트는 wiredTiger 엔진에서 RDBMS의 Undo 같은 역할을 함. 다른점이 있다면 데이터 페이지의 레코드를 직접 변경하지 않고, 변경된 데이터를 스킵 리스트에 추가.
- 사용자가 쿼리가 데이터를 읽을 때에는 변경 이력이 저장된 스킵 리스트를 검색해서 원하는 시점의 데이터를 가져간다. 이렇게 변경된 내용을 직접 데이터 페이지에 덮어쓰지 않고 별도의 리스트로 관리하는 이유는 쓰기 작업 처리를 빠르게 하기 위함.
- 변경된 내용을 스킵 리스트에 추가하기만 하면 되는데 스킵 리스트에 추가하는 작업은 매우 빠르고, 사용자의 응답 시간도 빨라진다. 이런 방식 때문에 wiredTiger 스토리지 엔진은 여러 쓰레드가 하나의 데이터 페이지를 동시에 읽거나 쓸 수가 있어서 동시 처리 성능이 매우 향상되는 것

2.2.4 캐시 이빅션(Cache Eviction)

- 공유 캐시를 사용하는데 있어, 새로운 디스크 데이터 페이지를 읽어서 공유 캐시에 적재할 수 있도록 빈 공간을 항상 적절히 유지해야 하는데, 이렇게 빈 공간을 유지하기 위해 wiredTiger 스토리지 엔진은 이빅션 모듈을 가지고 있으며, 이를 “이빅션 서버(Eviction Server)”라고도 부른다.
- 이빅션 서버는 백그라운드 쓰레드로 실행되는데 공유 캐시에 적재된 데이터 페이지 중에서 자주 사용되지 않는 데이터 페이지 위주로 공유 캐시에서 제거하는 작업을 한다. 이 과정에서 공유 캐시 스캔이 빈번히 발생하며 B-Tree의 브랜치 노드는 자주 접근되는 비율에 관계없이 공유 캐시에 유지하려는 경향이 있다.
- 이전에는 SSD를 사용해도 이빅션이 데이터 페이지의 읽기 성능을 따라가지 못해서 캐시가 가득차는 경우가 발생했는데, 이러한 현상은 버전업이 되면서 점차 나아졌고 현재는 MongoDB에서 SSD를 사용하는 것이 최적의 저장매체로 사용을 권장하고 있다.

2.2.5 Checkpoint

- 체크포인트는 모든 데이터베이스에 존재하는 개념으로 데이터 파일과 트랜잭션 로그가 동기화 되는 시점을 의미하며, WiredTiger 엔진은 버전 3.6부터 MongoDB는 60초 간격으로 체크포인트를 생성(스냅샷 데이터를 디스크에 기록).
- 사용자 데이터에 대해 60초 간격으로 또는 2GB의 저널 데이터가 기록된 경우 중 먼저 발생하는 체크포인트를 생성.
- 사용자 요청을 빠르게 처리하면서 커밋된 트랜잭션의 영속성을 보장하기 위해서 트랜잭션 로그(WAL, Write Ahead Log = 저널로그)를 먼저 기록하고, 데이터 파일에 기록하는 것은 트랜잭션에 상관없이 뒤로 미뤄 작업을 한다.
- 체크포인트는 DBMS가 서버가 크래시되거나 재구동 되었을때, 복구를 시작할 시점을 결정하는 기준이 된다. 체크포인트의 간격이 길면 복구 시간이 길어지게 되고, 또 반대로 너무 짧으면 쿼리를 처리하는 성능이 떨어짐.
- Oracle이나 MySQL의 INNODB가 Fuzzy 방식의 체크포인트를 사용하는 것과는 다르게 MongoDB의 wiredTiger 스토리지 엔진은 Sharp 체크포인트 방식을 사용. 평상시에는 디스크 쓰기가 없지만, 체크포인트가 실행되는 시점에 한 번에 모아서 더티 페이지를 기록하는 방식.
- wiredTiger의 체크포인트는 절대 기존의 B-Tree 구조를 덮어 쓰지 않는다. 새로운 leaf 페이지가 추가될때 새로운 공간을 할당 받고, 새로운 브랜치 노드가 구성되면 root 노드를 삭제하지 않고 새로운 브랜치 노드가 기존 leaf부터 새로 생성된 leaf를 바라보는 B-Tree 구조가 생성되며, 만약 사

용하지 않는 leaf가 있다면 해당 부분은 처내고 전체적으로 다시 B-Tree 구조를 완성한다. 새로운 B-Tree구조가 완성되면 기존의 B-Tree를 삭제하고 사용하던 공간을 반납.

2.2.6 MVCC

- wiredTiger 스토리지 엔진은 REPEATABLE-READ, READ_COMMITTED, SNAPSHOT 세가지의 격리 수준을 제공하는데, SNAPSHOT 수준의 격리 수준을 기본으로 선택하고 있다.
- wiredTiger 스토리지 엔진의 SNAPSHOT 격리 수준은 REPEATABLE-READ와 동일한 수준이다.
- 예전 3.x 버전에서는 트랜잭션을 엄격하게 유지하지는 않았지만, 4버전 부터는 RDBMS 수준의 트랜잭션을 지원.

2.2.7 데이터 블록(페이지)

- wiredTiger 스토리지 엔진은 고정된 크기의 블록을 사용하지 않는다. 하지만 하나의 페이지가 너무 커지는 것을 방지하기 위해 최대 크기에 대해서는 제한을 두고 있다.
- B-Tree의 브랜치 노드와 leaf노드의 페이지 크기를 다르게 설정할 수 있으며, 또 컬렉션 별로, 인덱스 별로 다르게 설정할 수도 있다. 하지만 컬렉션 별로 미세하게 설정을 변경하게 되면 관리의 어려움이 생기고 운영 비용을 높이는 결과를 가져올 수도 있다.
- 분석이나 대량의 Insert 작업이 많다면 페이지를 크게, 소규모의 데이터에 Random access가 잦고 빈번하게 Update가 발생한다면 가능한 페이지 크기를 작게 설정하는 것이 좋음.

2.2.8 운영체제 캐시(페이지 캐시)

- wiredTiger 스토리지 엔진은 Cached IO를 기본 옵션으로 사용하고 있음. wiredTiger 스토리지 엔진이 참조하고자 하는 데이터 페이지는 리눅스 페이지 캐시에 있는 데이터를 다시 자신의 공유 캐시에 복사하는 것.
- OS의 페이지 캐시에도 있고, wiredTiger 스토리지 엔진에도 같은 데이터가 있는데 이것을 더블 버퍼링이라고 하며, RDBMS에서는 이러한 더블 버퍼링 문제를 해결하기 위해 Direct IO 방식을 주로 사용.
- Cached IO를 사용하는 경우 리눅스 서버가 디스크 데이터 쓰기를 전담해서 처리합니다. 이때 페이지 캐시 처리에서 가끔식 문제를 발생하기도 하지만, 성능적인 부분에서는 Cached IO가 Direct IO보다 빠른 속도를 보여줌.

2.2.9 압축

- wiredTiger 스토리지 엔진의 가장 큰 장점 중 하나는 압축 기능. RDBMS와 비교 했을때 크게 3가지 장점이 있다.
  - **가변 사이즈의 페이지 사용:** 페이지가 가변 사이즈이기 때문에 압축 결과를 그대로 디스크에 기록하기때문에 RDBMS들처럼 페이지를 스플릿하여 여러번 압축하는 과정을 가질 필요가 없다.
  - **데이터 입출력 레이어에서의 압축지원:** wiredTiger의 블록 매니저가 디스크 데이터 페이지를 읽어오면서 공유 캐시에 적재할 때 압축을 해지한 상태에서 적재하고, 공유 캐시에서 디스크로 기록되는 페이지에 대해서만 압축을 실행해서 저장한다.
  - **다양한 압축 알고리즘 지원:** 4.2버전부터 Zstd를 지원하지 시작했으며, zlib과 Snappy를 지원하고 있다. 압축률이 높을수록 공간 절약이 되지만 압축하고 해제할 때 속도는 성능에 영향을

미친다.

압축률과 속도

Property	압축률	압축속도 (MB/sec)	압축해제 속도 (MB/sec)
zstd	2.9	330	940
zlib	2.7	95	360
Snappy	2.1	480	1600

- wiredTiger 스토리지 엔진에서는 압축 알고리즘 뿐만 아니라 컬렉션과 인덱스 그리고 저널로그에 대해서 각각 압축을 적용할 것인지 결정할 수 있다. 저널 로그와 데이터 파일은 Snappy로, 인덱스는 Prefix 압축을 적용해서 많이 사용함.
- Prefix 압축이란 인덱스 키에서 왼쪽 부분의 중복영역을 생략하는 압축 방식. 프리픽스 압축은 블록 압축과 다르게 공유 캐시에 적재된 상태에서도 압축상태를 유지한다. 프리픽스 압축은 데이터를 읽을 때, 항상 완전한 키 값을 얻기 위해서 조립과정을 거치는데, 페이지 첫번째에 있는 인덱스 키부터 재조립 과정을 거치게 되면 효율이 떨어지기도 함. 압축률이 좋으면 인덱스를 읽는 속도가 떨어질수도 있고, 인덱스를 역순으로 읽을때는 더 심해진다.

2.2.10 암호화

- 엔터프라이즈 버전에서만 암호화 기능을 사용할 수 있는데, 커뮤니티 버전의 소스 코드에도 암호화 처리를 위한 인터페이스가 그대로 남아 있기 때문에 간단한게 플러그인 형태의 모듈을 개발한다면 커뮤니티 버전에서도 데이터 암호화 기능을 사용할 수 있다.

2.3 그 밖의 MongoDB의 스토리지 엔진의 종류

- **MMAPv1**: 처음 몽고DB가 나왔을때 사용하던 스토리지 엔진으로 3.0부터 WiredTiger 스토리지 엔진이 나온 이후 사용하지 않음
- **In-Memory**: WiredTiger의 변형으로 디스크에 기록하지 않고 Memory에만 보관하는 스토리지 엔진. 엔터프라이즈 버전에서만 사용가능하나 서버의 코드를 조금만 커스터마이징하면 커뮤니티 버전에서도 사용 가능.
- **RocksDB**: 페이스북에서 LevelDB를 커스터마이징하여 개선한 스토리지 엔진
- **TokuDB**: Percona에서 개발중인 스토리지 엔진

2.3.1 스토리지 엔진 혼합 사용

- 하나의 MongoDB 서버(인스턴스)에서 동시에 여러 스토리지 엔진을 사용할 수는 없습니다. 하지만 레플리카 셋에서 프라이머리 노드는 WiredTiger 엔진을 사용하고, 세컨더리에서는 RocksDB 같은 다른 스토리지 엔진을 사용하는 것은 가능합니다. 샤드 클러스터 역시 마찬가지 입니다. 1번 샤드에서는 WiredTiger를 사용하고, 2번에서는 RocksDB를 사용하는 것이 가능합니다. 프라이머리를 In-Memory 엔진으로 구성할 경구 DB가 내려가면 모든 데이터와 oplog가 사라지기 때문에 레플리카 셋의 세컨더리를 디스크 기반의 WiredTiger로 저장하면 일부 멤버는 데이터를 가지고 있는 환경을 구성할 수 있습니다. 근래에는 SSD의 성능이 매우 좋아져서, SSD기반의 WiredTiger 스토리지 엔진을 사용한 구성을 많이 합니다.