* **Scalability**

ArangoDB는 여러 개의 데이터 모델을 지원하는 분산 데이터베이스이다. 따라서 많은 서버를 사용함으로써, 보통 하드웨어 기반으로, 수평적 크기 조정이 가능하다. 이러한 접근법은 용량의 증가만큼 성능을 향상 시켜줄 뿐 아니라, 복제본과 자동적인 fail-over을 통해 회복이 가능하다. 나아가, 우리는 요구에 맞게 자신의 용량을 dynamic 하게 키우고 줄일 수 있는 시스템을 만들 수 있다.

ArangoDB는 더 큰 서버들을 사용함으로써, 수직적 규모 조정도 가능하다. 더 많은 cpu 존재 시 서버는 자동적으로 더 많은 쓰레드를 사용하기 때문에 ArangoDB 에는 제한이라는 것이 없다. 그러나, 수직적 규모 조정은 서버 규모에 비해 비용이 선형적인 것 보다 더 빠르게 증가하면 회복력이나 dynamic 한 용량 조절이 달성될 수 없다는 단점이 있다.

**Architecture**

ArangoDB 의 클러스터 구조는 no single point of failure인 CP 마스터/마스터 모델이다. CP는 네트워크 파티션이 존재할 때, 디비가 내부 지속성보다 가용성을 중요시하는 것을 의미한다. 마스터/마스터란 클라이언트가 임의의 노드에 요청을 보낼 수 있고, 데이터베이스에 상관없이 같은 결과를 볼 수 있음을 뜻한다. No single point of failure 는 하나의 기계가 완전히 실패하더라도, 클러스터는 계속 요청을 받을 수 있음을 의미한다.

* **ArangoDB의 클러스터 구조**

: ArangoDB 클러스터는 네트워크 상에서 서로 대화하고, 각자 다른 역할을 맡은 많은 ArangoDB 인스턴스들로 구성되어 있다. 클러스터의 현재 환경설정은 Raft Consensus 프로토콜로 돌아가는 홀수 개의 ArangoDB 인스턴스 기반의 가용성이 높고 회복력이 좋은 key/value 저장 방식인 Agency 에 저장되어 있다. ArangoDB 클러스터의 다양한 변수들은 4가지 역할이 있다: Agents, Coordinators, Primary and Secondary DB servers.

1. **Agents(에이전트)**

: ArangoDB 클러스터에는 1개나 그 이상의 에이전트 들이 모여서 에이전시를 형성한다. 에이전시는 클러스터의 환경 설정을 저장하는 중심 장소이다. 전체 클러스터를 위한 동기화 작업을 제공하고, 리더 선발을 한다. 에이전시가 없이는 다른 구성 요소들이 작동할 수 없다.

밖에서는 보이지 않지만 에이전시는 클러스터의 심장이다. Fault tolerance는 에이전시에게 굉장히 중요하다. 그래서 Raft Consensus 알고리즘을 사용한다. 이 알고리즘은 ArangoDB 클러스터에서의 conflict 가 없는 환경설정 관리를 보장한다.

중심부에서는 에이전시가 커다란 환경설정 트리를 관리한다. 트리에서는 트랜잭션 읽기와 쓰기를 지원하며, 다른 서버들이 트리의 모든 변화에 있어 HTTP callback을 읽어드릴 수 있다.

1. **Coordinators**

코디네이터는 밖에서 접근가능해야 한다. 이것들이 실제로 클라이언트가 대화하는 대상이다. 쿼리 실행이나 Foxx 실행 같은 클러스터 업무들을 편성 한다. 코디네이터는 데이터가 어디 저장되었는지 알고, 사용자의 쿼리 등을 어디서 실행할 지 최적화 시켜준다. 코디네이터는 stateless하고 필요하다면 쉽게 셧다운되고 재부팅 된다.

1. **Primary DBserver**

프라이머리 서버는 데이터가 실제로 호스트 된 곳이다. 데이터 샤드를 호스팅하고, 프라이머리가 그 샤드에 대해 리더이거나 팔로워인 동기화 복제본을 사용한다.

프라이머리는 직접 접근은 안되지만 코디네이터를 통해 간접적으로 접근 가능하다. 코디네이터의 요청이 있을 시에는 전체 혹은 부분 쿼리를 실행하기도 한다.

1. **Secondaries**

세컨더리 서버는 프라이머리의 비동기 복제본이다. 만약 하나가 동기화 된 복제본만을 사용한다면, 세컨더리는 전혀 필요하지 않다. 각 프라이머리에 대해, 1개나 1개 이상의 세컨더리가 있을 수 있다. 복제본이 비 동기적이기 때문에, 복제본이 프라이머리의 성능을 방해하지 않는다. 반면에, 복제본은 살짝 뒤떨어질 수 있다. 세컨더리는 정상 클러스터 동작을 방해하지 않기 때문에 백업에 완벽하게 사용된다.

+ **Cluster ID**: 클러스터 안의 에이전시가 없는 ArangoDB 인스턴스들은 시작과 동시에 ID를 할당 받는다. ID를 노드로 사용하는 것은 클러스터에서 식별 가능하다.

+ **Sharding**: 위의 역할들은 데이터를 소위 샤드로 나눠 여러 프라이머리에 분산시킨다. 이 과정은 밖에서 보이지 않으며, 보통 다른 시스템에서 마스터/마스터 복제를 완성한다. ArangoDB 클러스터에서 당신은 아무 코디네이터에게 나 말을 건 후, 당신이 데이터를 읽거나 쓰면 자동적으로 그 데이터가 어디에 저장될 지 혹은 저장되어 있는 지를 알 수 있다. 샤드의 정보는 에이전시를 통해 코디네이터 들 사이에서 공유된다.

+ **환경 설정**: 이 구조는 매우 유연하고 서로 다른 상황에 적용되는 많은 환경설정을 허용한다.

1. 디폴트 환경설정은 각 기계에 정확히 하나의 코디네이터와 하나의 프라이머리를 돌린다. 고전적인 마스터/마스터 설정이며, 다른 노드 간의 완벽한 대칭이 있기 때문에, 클라이언트들은 아무 코디네이터에게 나 말을 걸고 같은 정보를 얻을 수 있다.
2. 서버보다 코디네이터의 개수가 더 많은 경우. Foxx는 코디네이터에서 돌아가기 때문에, 이 경우는 Foxx를 위한 Cpu가 많이 필요할 때 좋은 방법이다.
3. 데이터 저장 용량이 더 필요하고 쿼리 성능의 병목 현상을 줄이기 위해서는 코디네이터 보다 서버를 더 많이 배치한다.
4. 어플리케이션 서버(ex, node.js 서버)가 돌아가는 각 기계에 코디네이터를 배치하고, 에이전트와 서버가 따로 나머지 기계에 배치. 이 방법은 어플리케이션 서버와 디비 간의 네트워크 홉을 피하므로, latency를 줄인다.

* 중요한 것은 코디네이터 층의 규모 조정과 배치가 디비 층과 독립적이라는 것이다.

+ **복제**: ArangoDB 는 클러스터에서 2가지의 데이터 복제 방식을 제공한다: 동기화 와 비동기화. 각각 장단점을 가진다.

+ **자동 fail-over를 가지는 동기화 복제**: 동기화 복제는 각 샤드에 대해 작동한다. 각 콜렉션에 대해 각 샤드에 대해 얼마나 많은 카피본이 클러스터에 저장될 지 설정을 한다. 아무 때나, 카피본 중 하나가 리더로 선언되고, 나머지 다른 카피들은 팔로워가 된다. 이 샤드에 대한 쓰기 작업은 언제나 리더 카피를 가진 서버로 보내지고, 이 작업이 종료되고 코디네이터로 돌아가기 전에 이 변화는 모든 팔로워들에게 순서대로 복제된다. 읽기 작업은 리더 카피를 가진 서버에서 보여지고, 이것은 복잡한 트랜잭션 의미의 스냅샷을 제공한다.

만약 한 샤드에 대해 팔로워 카피를 가진 서버가 fail 되면, 리더는 더 이상 그 팔로워에게 동기화를 하지 않는다. 3초의 짧은 타임아웃 이후 리더는 그 팔로워를 포기하고, out of sync 선언 후, 그 팔로워 없이 서비스를 지속한다….매정한 리더…… 그 팔로워 카피를 가진 서버가 복구되면, 자동으로 리더는 재동기화를 시킨다.

만약 한 샤드에 대해 리더 카피를 가진 서버가 fail 되면, 리더는 더 이상 어떤 요청도 받아들이지 못한다. 에이전시에 살아있음을 알리지 못하고, 에이전시의 Raft 리더에서 돌아가는 감시 프로세스는 15초 이상 신호를 받지 못하면 동기화 된 팔로워 중 하나를 리더로 승격시킬 수 있다. 이것은 에이전시 재 환경설정과, 이 샤드에 대해 코디네이터가 이제 다른 서버로 요청을 보내는 것을 포함한다. 서비스는 재개되고, 다른 복제본들은 자동적으로 새로운 리더와 동기화 된다. 기존 리더 카피를 가진 서버가 돌아오면, 그 서버는 팔로워 카피를 가졌다고 인식되어 새 리더의 데이터와 동기화 된다.

모든 샤드 데이터 동기화가 점진적으로 실행되고, 재동기화는 빠르게 진행된다. 이 기술은 서비스 방해 없이 디비 서버간 샤드 이동(리더와 팔로워)를 가능하게 한다. 그러므로, ArangoDB 클러스터는 특정 서버에서 다른 서버로 모든 데이터를 옮길 수 있고, 감시 하에 그 서버를 셧다운할 수 있다. 이것은 ArangoDB 클러스터를 서비스 방해 없이 축소하고, fault tolerance의 손실이나 데이터 손실을 축소한다. 나아가 수동적이거나 자동적으로 샤드의 분배 균형을 재조정한다.

모든 동작은 REST/JSON API나 web UI에 의해 가동된다. 모든 fail-over 동작은 ArangoDB 클러스터 내에서 완벽히 조정된다.

명백하게, 모든 요청에 대해 클러스터 내 네트워크 홉은 하나 이상이므로, 동기화 복제는 쓰기 동작의 지연을 증가시킨다. 그러므로 사용자는 복제 요인을 1로 설정할 수 있다. 이 말은 각 샤드에 대해 단 하나의 카피만 저장하고, 동기화 복제를 끄는 것이다. 데이터가 쉽게 복구되거나 중요하지 않다면 적합한 설정이다.

+ **자동 fail-over를 가지는 비동기화 복제**

: 비동기화 복제는 프라이머리와 세컨더리 디비 서버를 사용하여 조직화 된다는 점에서 다르다. 각 세컨더리는 비동기적 방식으로 투표를 통해 프라이머리에 있는 모든 데이터를 복제한다. 이 과정은 프라이머리의 성능에 아주 작은 임팩트를 가진다. 단점은 클라이언트에게 보내진 쓰기 동작의 컨펌과 데이터의 실제 복제본 사이의 딜레이가 있다는 것이다. 이 딜레이 동안 마스터 서버가 실패하면, 데이터가 손실될 수 있다.

그럼에도 불구하고, 우리는 이런 설정에서 자동 fail-over를 제공한다. 동기화와 대조되어, fail-over 관리는 ArangoDB 클러스터 밖에서 실행된다.

비동기화 복제본 없이는 2배 많은 인스턴스들이 필요하다. 동기화 복제본은 더 작거나 큰 인스턴스를 가질 수 있는 점에서 유연하지만, 하나가 fail 되면 데이터가 남아있는 것들 사이에서 재균형을 찾을 수 잇다. ….. 뭔개소리얔ㅋㅋㅋㅋㅋㅋ ( 마이크로서비스, zero 관리자, 아파치 메소스 생략)

**Data Model**

1. **Key/value 쌍**

Key/value 데이터 모델은 규모 조정이 가장 쉽다. ArangoDB에서, 도큐먼트 콜렉션이 항상 프라이머리 키 속성을 가지고, 더 이상의 인덱스 없이 하나의 간단한 key/value 쌍처럼 행동하도록 구현되었다.

이러한 구조에서 가능한 단 하나의 동작은 single key lookup과 key/value 쌍 삽입과 업데이트이다. 만약 키가 단지 샤딩 속성일 뿐이라면, 샤딩은 이 프라이머리 키와 관련해서 실행되고, 이런 동작들은 선형적으로 확장될 것이다. 만약 샤딩이 다른 샤드 키들을 사용하여 실행되면, 싱글 키 lookup은 모든 샤드에 대해 질문을 해야 하므로 선형적으로 확장되지 않는다.

1. **Document store**

샤딩된 콜렉션의 인덱스가 각 샤드의 로컬 인덱스과 같기 때문에, 추가 인덱스가 있더라고 같은 argument 가 적용된다. 그러므로 싱글 도큐먼트 동작은 특별한 샤딩 환경설정이 lookup이나 쓰기 동작을 힘들게 만들지 않는 한, 여전히 클러스터 사이즈와 함께 선형적으로 조정된다.

1. **Complex queries and joins**

AQL은 여러 콜렉션을 사용하거나, 추가 인덱스 사용과 같은 복잡한 쿼리를 허용한다. 만약 필요한 데이터가 각각 다른 기계에 있다면 통신이 많이 필요하기 때문에, 규모 조정은 힘들 수도 있다. AQL 실행 엔진은 클러스터에서 가장 효율적으로 결과를 조합하기 위한 데이터 파이프라인을 조직한다. 쿼리 최적화 머신은 클러스터 구조를 인지하고, 데이터가 무엇인지, 어디 저장되어 있는지, 어떻게 인덱싱 되어 있는지를 안다. 그러므로 쿼리의 어떤 부분이 클러스터에 어느 부분에서 돌아가야 하는지에 대한 결정을 내린다.

1. **Graph database**

GDB는 경험상 알 수 없는 길이의 path를 포함하는 그래프를 표현하기 좋다. 예를 들어 두 노드 사이의 가장 짧은 길이를 찾는 것이나, 특정 패턴에 맞는 모든 path를 찾는 것들이 해당된다.

그러나, 해당되는 노드들과 모서리들이 클러스터에 분산되어 있다면, 노드 사이 엄청 많은 통신이 필요하여 성능은 떨어진다. 좋은 성능을 위해서는 클러스터에 샤드 분배를 잘 하는 것이 중요하다. 대부분, 개발자와 사용자는 어떻게 그래프를 구조화 해야 좋은 지 알기 때문에 ArangoDB는 사용자가 지정할 수 있도록 한다. 보통 하나의 노드와 그 노드를 가리키는 모서리는 같은 클러스터에 저장한다.

* **Data Model and Modeling**

**Concepts**

1. **Database Interaction**

ArangoDB 는 클라이언트에게 도큐먼트를 제공하는 디비이다. 이 도큐먼트들은 TCP 연결을 통해 JSON을 사용해 전송되며, HTTP 프로토콜을 사용한다. REST API는 디비와 상호 작용하기 위해 제공된다.

웹 인터페이스는 Aardvark, 쉘은 Arangosh.

1. **Data Model**

ArangoDB에 저장할 수 있는 도큐먼트들은 비록 VelocyPack이라는 이진 형식으로 저장되어 있지만 JSON 형식을 따른다. 도큐먼트는 0개 이상의 속성을 포함하고, 각 속성은 값을 가진다. 값은 숫자, 문자열, null 등의 atomic 타입을 가지거나, 배열 등의 compound 타입을 가진다. 배열과 sub-object들은 이러한 모든 타입을 포함하며, 이러한 nested 데이터 구조들이 하나의 도큐먼트로 나타내질 수 있음을 의미한다.

도큐먼트들은 컬렉션으로 그룹 지어진다. 컬렉션은 0개 이상의 도큐먼트를 포함한다. 컬렉션은 관계형에서 테이블로, 도큐먼트는 열로 비교된다. 차이점은, 관계형에서는 테이블에 저장하기 전에 행을 정의해줘야 했다는 것이다. 이러한 정의는 스키마라고도 알려져 있다. , ArangoDB 는 스키마가 없고, 이것은 도큐먼트가 어떠한 속성들을 가질 수 있는 지 정의하지 않아도 된다는 뜻이다. 모든 싱글 도큐먼트는 완전히 다른 구조를 가질 수 있고 여전히 하나의 컬렉션에 다른 도큐먼트들과 같이 저장될 수 있다. 실제로 한 컬렉션의 도큐먼트들 사이에는 공통 분모를 가지지만, 디비 그 자체는 특정 데이터 구조를 강요하지 않는다.

컬렉션에는 두 종류가 있다: 도큐먼트 컬렉션과 엣지 컬렉션. 엣지 컬렉션 또한 도큐먼트를 저장하지만 도큐먼트 사이의 관계를 표현하는 from, to라는 두 개의 특별한 속성을 가진다. 보통, 도큐먼트 컬렉션에 저장된 두 도큐먼트(정점) 들은 엣지 컬렉션에 저장된 도큐먼트(엣지)에 의해 연결된다. 이것은 ArangoDB의 그래프 데이터 모델이다.

컬렉션은 디비 안에 존재한다. 하나 이상의 디비가 존재한다. 컬렉션이나 도큐먼트 안의 데이터가 서로 고립되어 있기 때문에, 디비 들은 보통 여러 세입자를 둔다. 디폴트 데이터베이스 \_system은 제거 불가능 하므로 특별하다. 디비 사용자들은 이 데이터 베이스 안에서 관리되고, 그들의 관리 권한은 모든 디비에서 유효하다.

1. **Data Retrieval**

쿼리 : 도큐먼트를 새로운 데이터를 계산하거나 존재하는 도큐먼트를 삭제, 조정하기 위해 특정 기준을 바탕으로 도큐먼트를 필터링한다. 쿼리는 간단하기도 하면서, 그래프를 탐색하거나 여러 컬렉션을 사용해 join 하는 등 복잡해지기도 한다.

커서 : 한 덩어리 보다는 처리 가능한 작은 단위를 얻을 수 있도록 쿼리의 결과를 반복

인덱스 : 검색 속도를 높혀준다.

**Handling Databases**

자바 스크립트를 사용하여 ArangoDB에서 디비를 관리하는 법 소개. - ArangoDB와의 연결을 하기위해서는 db.\_useDatabase()를 사용해 명시적으로 현재 연결중인 데이터베이스를 바꿔주어야 하고, 명시한 그 이후부터의 액션/연결은 해당 데이터베이스와 되게 된다. - 데이터베이스가 바뀌었다면, 클라이언트 드라이버는 현재 데이터베이스의 이름을 저장해 두어야한다. arangodb의 연결은 어떠한 상태 정보(HTTP 요청/결과 데이터)도 저장하지 않기에 필요하다

**Working with databases**

**+ Database Methods**

* **Name**

db.\_name() : 현재 디비의 이름을 문자열로 리턴

require(“@arangodb”).db.\_name();

* **Id**

db.\_id() : 현재 디비 아이디를 문자열로 리턴

require(“@arangodb”).db.\_id();

* **Path**

db.\_path() : 현재 디비의 파일시스템 경로를 문자열로 리턴

require(“@arangodb”).db.\_path();

* **IsSystem**

db.\_isSystem() : 현재 사용중인 디비가 시스템 디비인지 리턴.

* **Use Database**

db.\_useDatabase(name) : name을 가진 디비로 현재 사용 디비를 바꿈

* **List Databases**

db.\_databases() : 모든 디비의 리스트 리턴 (\_system에서만 사용가능)

* **Create Database**

db.\_createDatabase(name, options, users) : name을 갖는 디비 생성. Option은 현재 아무 의미 없음. 미래를 위한 것. User는 초기 사용자를 생성. 구체적으로 user object 여야 함. User object는 4가지 속성가짐.

1. Username : 필수, 이름 문자열
2. Passwd : 디폴트는 빈 문자열, 비번 문자열
3. Active : 불린형, 디폴트는 true, 사용자 계정 활성화 나타냄
4. Extra : 선택, ArangoDB에서 해석은 안됨

- 초기 user 설정 없이 데이터베이스를 만들어도, 생성된 이후 user의 정보를 추가, 변경할 수 있다.

require("@arangodb/users").save(username, password, true);

require("@arangodb/users").update(username, password, true);

require("@arangodb/users").remove(username);

다음과 같이 할 수도 있다.(\_system 데이터베이스에서만 가능)

db.\_createDatabase("newDB", [], [{ username: "newUser", passwd: "123456", active: true}])

* **Drop Database**

db.\_dropDatabase(name) : name을 가지는 디비 버림. \_system에서만 사용가능. - 데이터베이스의 물리적인 모든 것이 삭제된다. 자기 자신은 drop 불가

**Notes about databases**

각 디비는 각각의 시스템 컬렉션을 가지며, 디비가 생성될 때 설정되어야 한다. 그래서 디비 생성은 좀 시간이 걸린다.

복제는 디비 당 설정된다. 새로운 디비가 생성된 이후에는 그 디비에 복제본 로깅이나 적용하려면 명시적으로 설정해야한다.

**Javascript Interface to Collection**

Arangodb의 모든 콜렉션은 고유한 식별자와 이름이 있다. arangodb는 식별자를 통해 콜렉션들을 구별하는데 arangodb는 그 식별자들을 control할 수는 없다. 사용자가 콜렉션의 이름을 지정할 수 있게 하기위해 각각의 콜렉션은 사용자에 의해 정의된 고유한 이름을 가지게 된다. 사용자는 이름을 통해 콜렉션에 접근해야한다.

- 콜렉션 생성 => db.\_create(collection-name)

- 콜렉션 접근 => db.\_collection(collection-name)

- 콜렉션이 없을 경우 생성, 있을 경우 해당 콜렉션 접근 => db.collection-name

**Collection methods**

**Drop** : 콜렉션을 삭제한다. 시스템 콜렉션을 삭제하기 위해서는 isSystem set을 반드시 true설정 해주어야 한다.

arangosh> col = db.example;

[ArangoCollection 14631, "example" (type document, status loaded)]

arangosh> col.drop();

arangosh> col;

[ArangoCollection 14631, "example" (type document, status deleted)]

arangosh> col = db.\_example;

[ArangoCollection 14634, "\_example" (type document, status loaded)]

arangosh> col.drop({ isSystem: true });

arangosh> col;

[ArangoCollection 14634, "\_example" (type document, status deleted)]

**Truncate**

- 콜렉션의 인덱스를 제외한 모든 데이터를 지운다.

arangosh> col = db.example;

arangosh> col.save({ "Hello" : "World" });

arangosh> col.count();

arangosh> col.truncate();

arangosh> col.count();

**Properties**

- collection.properties()를 통해 콜렉션의 속성을 가져오거나 설정할 수 있다. 콜렉션의 모든 속성이 리턴된다.

arangosh> db.example.properties(); -> 실행 결과 확인

arangosh> db.example.properties({ waitForSync : true }); -> Change a property

**Figures**

- collection.figures() -> 콜렉션의 figure들을 리턴하여 준다.

- arangosh> db.demo.figures() -> 실행 결과 확인

**Load**

- collection.load() -> 콜렉션을 불러온다(status가 loaded로 바뀜)

arangosh> col = db.example;

[ArangoCollection 14722, "example" (type document, status loaded)]

arangosh> col.load();

arangosh> col;

[ArangoCollection 14722, "example" (type document, status loaded)]

**Revision**

- collection.revision() -> 콜렉션의 revision id를 리턴하여 준다.(string형)

- revision id는 document가 변경되었을 때 변경되며, 이것을 통해 변경 여부를 확인 할 수 있다.

**checksum**

- collection.checksum(withRevisions, withData) -> 콜렉션안의 데이터의 체크섬을 계산하여준다.

**Unload**

- collection.unload() -> 콜렉션을 메로리로부터 unload한다. 모든 쿼리가 끝날때까지 지연된 후 실행된다.

arangosh> col = db.example;

[ArangoCollection 7363, "example" (type document, status loaded)]

arangosh> col.unload();

arangosh> col;

[ArangoCollection 7363, "example" (type document, status unloaded)]

**Rename**

- collection.rename(new-name) -> 콜렉션의 이름을 재지정함.

**Database methods**

**Collection**

- db.\_collection(collection-name) -> 한 개의 콜렉션이나 null을 리턴

arangosh> db.\_collection("demo");

[ArangoCollection 93, "demo" (type document, status loaded)]

-> 이름을 가지고 콜렉션 확인

arangosh> db.\_collection(123456);

[ArangoCollection 123456, "demo" (type document, status loaded)]

-> id를 가지고 콜렉션 확인

arangosh> db.\_collection("unknown");

null

-> 없는 콜렉션 일 경우

**Create**

- db.\_createEdgeCollection(collection-name) -> 엣지 콜렉션 생성

-

**All Collection**

- db.\_collections() -> 데이터 베이스의 모든 콜렉션을 리턴하여 준다.

**Collection Name**

- db.collection-name

**Drop**

- db.\_drop(collection) : 콜렉션과 그 모든 인덱스, 데이터를 drop

- db.\_drop(collection-identifier) : 식별자에 해당하는 콜렉션의 인덱스 데이터를 drop. 식별자에 해당하는 것이 없어도 에러가 일어나지 않는다.

- db.\_drop(collection-name) : collection-name에 해당하는 콜렉션과 그 모든 인덱스를 drop. 해당하는 것이 없어도 에러가 일어나지 않는다.

- db.\_drop(collection-name, options) : system 콜렉션을 drop하기 위해서 isSystem 속성을 반드시 true로 해야한 drop가능하다

Drops a collection:

arangosh> col = db.example;

[ArangoCollection 14574, "example" (type document, status loaded)]

arangosh> db.\_drop(col);

arangosh> col;

[ArangoCollection 14574, "example" (type document, status loaded)]

Drops a collection identified by name:

arangosh> col = db.example;

[ArangoCollection 14577, "example" (type document, status loaded)]

arangosh> db.\_drop("example");

arangosh> col;

[ArangoCollection 14577, "example" (type document, status deleted)]

Drops a system collection:

arangosh> col = db.\_example;

[ArangoCollection 14580, "\_example" (type document, status loaded)]

arangosh> db.\_drop("\_example", { isSystem: true });

arangosh> col;

[ArangoCollection 14580, "\_example" (type document, status deleted)]

**Truncate**

- db.\_truncate(collection) -> 콜렉션의 인덱스를 제외하고 모든 것을 삭제함

Truncates a collection:

arangosh> col = db.example;

arangosh> col.save({ "Hello" : "World" });

arangosh> col.count();

arangosh> db.\_truncate(col);

arangosh> col.count();

Truncates a collection identified by name:

arangosh> col = db.example;

arangosh> col.save({ "Hello" : "World" });

arangosh> col.count();

arangosh> db.\_truncate("example");

arangosh> col.count();

**Documents**

**Basics and terminology**

- JSON 객체들이며, 각 객체들은 nested되어 있고 list들을 가지고 있다.

- 각각의 document는 고유 키가 있어 식별자로 사용된다.

- 같은 데이터 베이스에서 고유한 식별자를 통해 각각의 document들이 handle된다.

- 같은 document의 다른 revision은 document revision을 통해 구별될 수 있다.

- **\_id** : document handle

- **\_key** : document primary key

- **\_rev** : document revision

- \_key는 document 생성시 설정 가능, \_id, \_key는 document생성시 자동적으로 설정됨

- \_rev는 ArangoDB에 의해 자동적으로 관리된다.

- document key는 collection안에서는 고유적이다.

- key option을 통해서 key 생성 규칙을 지정할 수도 있다

Document Revision(MVCC token – document각각의 revision)

- string값으로 정수형 숫자를 포함하며, 하나의 document에 대한 document revision들의 list에 대해 고유하다

- 같은 시간에 생성되고 똑같다고 하여도, 하나의 shard는 서로 다른 2개의 revision이 다른 \_rev string을 가지도록 한다.

- 각각의 클러스터의 시간이 조금씩 다르기 때문에, 서로 다른 shard나 콜렉션의 시간 기준은 비교가능하게 보장해 주지 않는다.

Multiple Document in a single command

- 3.0버젼부터 하나의 명령어를 가지고 여러 document에 적용할 수 있도록 지원되었다.

- multiple document operation은 multiple document에 대해서만 사용할 수 있다.

**Collection methods**

**All**

- collection.all() -> 컬렉션의 모든 document를 커서 형식으로 리턴.

- toArray, hasNext, next, limit, skip을 통해 결과에 접근할 수 있다.

Use toArray to get all documents at once:

arangosh> db.five.save({ name : "one" });

arangosh> db.five.save({ name : "two" });

arangosh> db.five.save({ name : "three" });

arangosh> db.five.save({ name : "four" });

arangosh> db.five.save({ name : "five" });

arangosh> db.five.all().toArray();

arangosh> db.five.save({ name : "one" });

arangosh> db.five.save({ name : "two" });

arangosh> db.five.save({ name : "three" });

arangosh> db.five.save({ name : "four" });

arangosh> db.five.save({ name : "five" });

arangosh> db.five.all().limit(2).toArray();

**Query by example**

- collection.byExample(example) -> document에서 example에 맞는 콜렉션을 커서 형식으로 리턴한다.

example을 { "a" : { "c" : 1 } } 로 했을 시(value 형식),

{ "a" : { "c" : 1 }, "b" : 1 } 은 match 되고, { "a" : { "c" : 1, "b" : 1 } } 은 match 되지 않는다

example을 { "a.c" : 1 }로 했을 시(path 형식), { "a" : { "c" : 1 }, "b" : 1 } 과 { "a" : { "c" : 1, "b" : 1 } } 둘 다 match 된다.

next를 루프 방식으로 표현 한 예)

arangosh> db.users.save({ name: "Gerhard" });

arangosh> db.users.save({ name: "Helmut" });

arangosh> db.users.save({ name: "Angela" });

arangosh> var a = db.users.byExample( {"name" : "Angela" } );

arangosh> while (a.hasNext()) print(a.next());

**First Example**

- collection.firstExample(example) -> document에서 exmaple에 맞는 콜렉션을 리턴하여 준다.

(커서방식아님)

**Any**

- collection.any() -> 아무 컬렉션 하나 리턴

**Count**

collection.count() -> 컬렉션의 개수 리턴

**Document**

- collection.document(object) -> object에 해당하는 collection의 document를 리턴하여 준다.

\_id와 \_key값에 대하여 먼저 나온 것에 대한 값이 올바르지 않을 경우 error.

- collection.document(document-handle)

- collection.document(array)

- collection.document(document-key)

**Exists**

- collection.exists(object) -> \_id, \_key를 통해 해당 하는 document가 존재하는 경우 \_id, \_key, \_rev를 리턴하고 없을 경우에는 false를 리턴한다. revision mismacth는 error를 낸다.

- false가 나온 경우에는 무조건 없는 document라는 의미

**Insert**

- collection.insert(data) -> 컬렉션에 주어진 데이터를 가진 document를 생성한다. \_id, \_rev는 무시되고 자동적으로 생성된다. \_key가 명시되어있지 않으면 자동적으로 생성된다.

- collection.insert(data, options) -> 옵션 파라미터는 object이여야만 하고 다음과 같은 옵션을 특정한다.

- waitForSync : false일 시 전체를 동기화, true일 시 해당하는 것만 동기화 적용

silent : true일시, 메소드는 어느것도 리턴하지 않는다.

returnNew : true 일시, new attrubute가 추가되어 리턴된다.

**Replace**

- collection.replace(selector, data) -> selector에 해당하는 document를 data로 변경한다.

- collection.replace(selectorarray, dataarray, options)

- waitForSync : false일 시 전체를 동기화, true일 시 해당하는 것만 동기화 적용

overwrite :

returnNew :

returnOld :

silent : 어떤 메소드도 리턴하지 않음

**Update**

- collection.update(selector, data) : selector에 해당하는 document에 data를 추가하여 준다.

- collection.update(selector, data, options)

- replace와 동일한 옵션 + keepNull, mergeObjects(중요, 예제로 확인할것)

keepNull : false일시 null값은 적용되지 않음, true시 null값도 적용이 됨.

mergeObjects :

**Remove**

- 삭제

Remove By Example

- collection.removeByExample(example) : path, value 에 해당하는 document 삭제

**Collection type**

- collection.type() -> 컬렉션의 타입을 리턴하여 준다.

- db.\_version() -> 데이터베이스의 버전을 리턴하여 준다.

**Edge**

- edge-collection.edges(vertex) -> vertex안의 모든 edge들을 찾는다.

arangosh> db.\_create("vertex"); //vertex 컬렉션 생성

arangosh> db.\_createEdgeCollection("relation"); //relation edge 컬렉션 생성

arangosh> var myGraph = {}; //배열(?) 생성

arangosh> myGraph.v1 = db.vertex.insert({ name : "vertex 1" }); //vertex컬렉션에 v1 document 생성

arangosh> myGraph.v2 = db.vertex.insert({ name : "vertex 2" }); //vertex컬렉션에 v2 document 생성

arangosh> myGraph.e1 = db.relation.insert(myGraph.v1, myGraph.v2, //relation 컬렉션에 e1 생성

........> { label : "knows"});

arangosh> db.\_document(myGraph.e1);

arangosh> db.relation.edges(myGraph.e1.\_id);

- edge-collection.inEdges(vertex) -> to가 vertex인 edge들 출력

- edge-collection.outEdges(vertex) -> from이 vertex인 edge들 출력

**Naming Conventions**

**Database names**

- 데이터 베이스 이름은 반드시 a부터 z, 0-9, \_, - 문자들로 구성되어야 한다.(아스키 문자가 아니면 불가)

- 무조건 문자로 시작되어야 하며

- 최대 길이는 64바이트

- 대소문자를 구별한다

**Collection names**

- a to z, 0 to 9, -, \_ 로 구성되어야 하며, 아스키 문자가 아니면 불가능

- 문자로 시작되어야 하며, 시스템 컬렉션은 \_로 시작되어야 한다. 시스템 컬렉션은 유저에 의해 사용되어서는 안된다.

- 최대 길이 64바이트

- 대소문자 구별

**Document keys**

키는 string. 숫자로만 이루어진 key는 안되며 하나라도 string이 존재해야 한다.

- 적어도 1바이트 이상, 254바이트 이하, empty key는 명시할 수 없다.

- key의 이름으로는 a to z, 0 to 9 그리고 \_()+,=;$%#@! 등을 사용할 수 있다.

- key의 value로는 !@#^%$등을 사용할 수 없다.

- 컬렉션에서 key의 이름은 고유해야한다.

- 대소문자 구별

- document생성시 필수적인 key들을 명시하지 않았다면 자동적으로 생성한다.

- 자동적으로 만들어 지는 불투명한 값으로 다루어야 하고 형식에 의존해선 안된다.

- 현재 생성되는 키의 포맷은

- numeric 값들은 생성된 시간에 따라 표현

**Attribute names**

- 시스템 attribute의 경우 \_로 시작되며 대부분 특별한 목적을 위해 사용된다

- \_id : document‘s handle을 위해

- \_key : document에 대해 유저가 지정한 키

- \_rev : document의 revision number

- \_from, \_to : 엣지 컬렉션의 추가 정보

- 길이에 제한이 없지만, 제한적이게 사용할 것

- 대소문자 구별

- empty name으로 된 attribute는 허락되지않는다.

**\* indexing**

**인덱스**

- 인덱스된 attribute를 쿼리로 사용할 경우 document에 대한 빠른 접근을 허락한다. Arangodb는 자동적으로 시스템 attribute를 인덱스 하기 때문에 유저는 새로운 인덱스를 추가하가지 않아도 된다.

- 유저 정의 인덱스는 collection level에서 생성될 수 있다.

- \_id, \_key는 컬렉션의 primary key의 역할을 하고, \_from, \_to는 엣지 콜렉션에서 자동적으로 엣지 인덱스가 된다.

- \_id를 가지고 유저 정의 인덱스는 불가능하고, 그 외의 \_key, \_rev, \_from, \_to는 가능하다.

arangodb는 다음과 같은 인덱스를 제공한다

**primary index**

- 컬렉션 안의 document들의 key를 hash방식으로 인덱싱한다. 같은 \_key와 \_id를 가지고서 탐색을 할시에 자동적으로 AQL 쿼리에서 사용된다.

db.collection.document("<document-key>");

db.\_document("<document-id>");

- 컬렉션의 primary index는 없애거나 변화시킬 수 없고, 유저 정의 primary index를 생성하는 방식도 없다.

**Edge Index**

- 모든 엣지 컬렉션은 자동적으로 생성되는 엣지 인덱스를 가지고 있다. 이는 \_from과 \_to속성을 가지고 도큐먼트에 대한 빠른 접근을 제공한다.

- 엣지 인덱스는 엣지 컬렉션안에서 같은 \_from과 \_to를 가지고 있는 것을 찾는 AQL에서 사용된다

db.collection.edges("<from-value>");

db.collection.edges("<to-value>");

db.collection.outEdges("<from-value>");

db.collection.outEdges("<to-value>");

db.collection.inEdges("<from-value>");

db.collection.inEdges("<to-value>") -> 엣지 찾기

- 엣지 인덱스는 hash index방식을 사용한다.

- 엣지 인덱스는 삭제되거나 변경될 수 없다.

**hash index**

- 특별한 attribute 값을 가지고서 도큐먼트에서 빠른 탐색이 가능하다.

- 해쉬 인덱스는 하나 또는 여러개의

- unique hash index : unique index에 의해 다루어지는 컬렉션의 모든 도큐먼트는 다른 key value를 가져야 한다. 인덱스 attribute를 가지고 있지 않은 도큐먼트나 인덱스 attribute로 null을 가지고 있는 도큐먼트에 대해서도 인덱스 될 수 있다. 중복되지 않은 엣지가 생성되게 사용될 수 있다.

- unique, sparse hase index : 모든 콜렉션 도큐먼트들은 고유한 인덱스를 위해 다른 attribute 값들을 가져야 한다. 적어도 1개의 인덱스 attribute가 설정되지 않거나 null인 도큐먼트들은 인덱싱 되지 않는다. 이 종류의 인덱스는 컬렉션내에 도큐먼트 중 중복되는 키가 없는 것을 보장하기 위해 사용된다.

- non-unique hash index : 컬렉션의 모든 도큐먼트는 인덱싱 되며, sparse하지 않다. 인덱스 attribue가 없거나 null인 도큐먼트 들도 인덱싱된다.

- non-unique, sparse hash index : null이 아닌 제대로 된 인덱스 attribute의 값을 가진 도큐먼트에 대해서만 인덱스 된다.

|  |  |  |  |
| --- | --- | --- | --- |
|  | not set, null index | unique key-value | 특성 |
| unique | o | o | edge의 중복 생성 제한 |
| unique, spaarse | x | o | key의 중복 생성 제한 |
| non-unique | o | x | unique 제약 위반 않음 |
| non-unique, sparse | x | x |  |

- unique hash index에서의 탐색, 삽입, 변경, 삭제 등의 분할 상관 복잡도는 O(1)이다.

**Skiplist Index**

- skiplist는 정렬된 인덱스 구조이다. 특정한 attribute값을 가지고 빠르게 도큐먼트를 찾을 수 있으며 ,

- 쿼리에 의해 모든 인덱스 attribute가 주어진 경우, 또는 leftmost prefix가 명시된 경우. lookup, range, 정렬 등에 사용된다.

- 정렬에 skiplist 인덱스를 사용하려면 인덱스 정의에 나타나는 순서와 동일한 순서로 index attribute를 조회의 SORT 절에 지정해야한다.

- Skiplist 인덱스는 항상 오름차순으로 만들어 지지만 오름차순 또는 내림차순으로 인덱싱 된 atrribute에 액세스하는 데 사용할 수 있다. 그러나 조합 된 인덱스의 경우 SORT 절에 지정된 단일 쿼리의 정렬 순서는 모두 오름차순이거나 또는 모두 내림차순 이어야한다

- value1, value2 순서대로 skiplist index를 한다고 하면,

SORT value1 ASC, value2 ASC (and its equivalent SORT value1, value2 )

SORT value1 DESC, value2 DESC

SORT value1 ASC (and its equivalent SORT value1 )

SORT value1 DESC

- 위의 소트방식은 인덱스 순서를 따르지 않을 수 있어 추가적인 단계가 더 필요하다.

SORT value1 ASC, value2 DESC

SORT value1 DESC, value2 ASC

SORT value2 (and its equivalent SORT value2 ASC )

SORT value2 DESC (because first indexed attribute value1 is not used in sort clause)

- unique skiplist index, unique/sparse skiplist index, non-unique skiplist index, non-unique/sparse skiplist index -> hash index와 동일

**Persistent Index**

- persistent index는 지속송이 있는 정렬된 인덱스이다. 도큐먼트가 저장되거나 업데이트 될 때 인덱스가 디스크에 쓰이기에, 서버를 다시 시작하거나 처음 load할 때 컬렉션으로부터 다시 만들 필요가 없다. 그래서 load시간이 줄어든다.

- 컬렉션의 유일한 인덱스로 사용될 수 없다. 켈렉션의 메모리안에 primary 인덱스와 여러 잠재된 인덱스 들이 더 있을수 있기 떄문이다.(엣지 인덱스나 엣지 콜렉션)

- primary key의 포인터만 저장하고 값은 저장하지 않는다. 인덱스의 value lookup을 통해 문서를 검색하려면

실제 도큐먼트를 가져오기 위한 O(1)이 추가적으로 필요하다.

- 모든 쿼리의 인덱스 attribute가 주어지거나 leftmost prefix가 지정되어야 사용할 수 있다.

**Geo Index**

- 유저는 컬렉션의 하나 또는 여러 attribute에 대해 추가적인 geo 인덱스를 만들 수 있다.

- geo 인덱스는 지구표현의 장소를 빠르게 찾는데 사용됩니다.

**fulltext index**

- 도큐먼트의 단어나 접두사 등을 찾기 위해 사용한다.

- single attribute에 대해서만 생성되며 모든 도큐먼트에 대해 적용가능하다.

- 토큰화를 통해 진행되며, lower-case form으로 인덱스 되며, complete match query와 prefix query를 지원한다. and, or, not등도 지원

- attribute가 설정되고 문자열 값을 포함하는 문서만 인덱스 되고, 단어가 적어도 최소한의 길이를 가져야만 한다.

**Indexing attributes and sub-attributes**

- nested된 attribute도 인덱스 된다. top-level의 attribute는 name만 필요하며 여러 필드에 결합 된 인덱스를 만들려면 필드 배열에 멤버를 더 추가해야한다.

- To index sub-attributes, specify the attribute path using the dot notation:

// { name: {last: "Smith", first: "John" } }

db.posts.ensureIndex({ type: "hash", fields: [ "name.last" ] })

db.posts.ensureIndex({ type: "hash", fields: [ "name.last", "name.first" ] })

**Indexing array values**

- 인덱스 attribute에 배열이 포함되어 있으면 ArangoDB는 기본적으로 전체 배열을 인덱스 값으로 저장한다. 인덱스를 통해 배열의 개별 멤버에 액세스하는 것은 불가능하다. attribute를 인덱스 할 시 배열인 attribute는 인덱스 되지 않는다.

- 각각의 배열 멤버에 대해 인덱스 하려고 한다면, special array index가 필요하다.

collection.ensureIndex()를 사용하면 배열 인덱스는 set up 된다. 해쉬나 skiplist 인덱스를 배열로 인덱스를 하려면 attribute name을 \*로 확장해야하고 필터링 쿼리에서 IN 연산을 해 줘야한다.

posts라는 컬렉션의 tag attribute에 hash 배열인덱스를 만든다.

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*]" ] });

db.posts.insert({ tags: [ "foobar", "baz", "quux" ] });

-[\*]를 반드시 써줘야 하는 것은 아니다. 배열 인덱스의 사용을 보여주기 위함이다.

FOR doc IN posts

FILTER 'foobar' IN doc.tags

RETURN d

FOR doc IN posts

FILTER 'foobar' IN doc.tags[\*]

RETURN doc

- array index를 사용하지 않은 필터링

FILTER doc.tags ANY == 'foobar'

FILTER doc.tags ANY IN 'foobar'

FILTER doc.tags IN 'foobar'

FILTER doc.tags == 'foobar'

FILTER 'foobar' == doc.tags

- subattribute도 배열 값으로 할수 있다.(해봐야 함)

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*].name" ] });

db.posts.insert({ tags: [ { name: "foobar" }, { name: "baz" }, { name: "quux" } ] });

- subattribute에 대한 배열 index

FOR doc IN posts

FILTER 'foobar' IN doc.tags[\*].name

RETURN doc

-ArangoDB는 인덱스 속성 당 하나의 [\*] 연산자로 배열 인덱스 생성을 지원한다.

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*].name[\*].value" ] }); -> 불가

- 배열 값은 배열 인덱스에 삽입되기 전에 자동으로 중복 제거된다. 배열에 동일한 값은 하나만 존재 가능하며 이미 존재 하는 데이터를 넣으려 하면 에러가 일어나며, 존재하지 않는데 2개를 넣는 경우에 대해서는 에러가 나지 않는다.

db.posts.insert({ tags: [ "foobar", "bar", "bar" ] }); -> bar가 없었다면 가능, 있었다면 불가능

- 배열로 선언된 attribute에는 배열로 넣어야만 한다. 다음것들은 인덱스 되지 않는다.

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*]" ] });

db.posts.insert({ something: "else" });

db.posts.insert({ tags: null });

db.posts.insert({ tags: "this is no array" });

db.posts.insert({ tags: { content: [1, 2, 3] } });

- 인덱스 배열에 null값 지정이 가능하다. ([null] 방식으로)

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*]" ] });

db.posts.insert({tags: null}) // Will not be indexed

db.posts.insert({tags: []}) // Will not be indexed

db.posts.insert({tags: [null]}); // Will be indexed for null

db.posts.insert({tags: [null, 1, 2]}); // Will be indexed for null, 1 and 2

- 인덱스 배열을 sparse로 선언하여도 인덱스 부분의 배열에 영향을 주지 않는다. sparse버젼에 대해서도 배열에는 null값 지정이 가능하다. ([null] 이렇게 넣어야함)

db.posts.ensureIndex({ type: "hash", fields: [ "tags[\*]", "name" ], sparse: true });

db.posts.insert({tags: null, name: "alice"}) // Will not be indexed

db.posts.insert({tags: [], name: "alice"}) // Will not be indexed

db.posts.insert({tags: [1, 2, 3]}) // Will not be indexed

db.posts.insert({tags: [1, 2, 3], name: null}) // Will not be indexed

db.posts.insert({tags: [1, 2, 3], name: "alice"})

// Will be indexed for [1, "alice"], [2, "alice"], [3, "alice"]

db.posts.insert({tags: [null], name: "bob"})

// Will be indexed for [null, "bob"]

- ==. !=, >, <, >=, <=, ANY, ALL, NONE 등은 현재 배열 인덱스에 사용할 수 없다.

**Vertex centric indexs**

- 그래프의 가장 중요한 인덱스는 에지 컬렉션의 \_from 및 \_to 속성을 인덱싱하는 에지 인덱스이다. 주어진 edge를 시작하거나 도착하는 모든 vertex에 빠르게 액세스 할 수 있으므로 그래프의 모든 이웃을 빠르게 찾을 수 있다.

- 정점 중심의 인덱스를 만들기 위해서는 간단히 이렇게

db.edges.ensureIndex({"type":"skiplist", "fields": ["\_from", "timestamp"]});

- 다음과 같은 쿼리를 적용하면

FOR v, e, p IN 1..1 OUTBOUND "V/1" edges

FILTER e.timestamp ALL >= "2016-11-09"

RETURN p

- 최근에 생성된 time stamp의 개수가 적을 경우 결과가 빠를 것이다.

**which index to use when**

- ArangoDB는 각 컬렉션의 \_key 특성을 자동으로 인덱싱한다. attribute에 대해서 따로 인덱스 해줄 필요는 없다. 도큐먼트의 \_id 특성은 \_key 특성에서 파생되므로 암시적으로 인덱싱된다.

- 또한 ArangoDB는 모든 에지 콜렉션에서 \_from 및 \_to에 대한 인덱스를 자동으로 작성하므로 오고 가는 연결을 효율적으로 의미할 수 있다.

**Index types**

- 사용자는 하나 이상의 도큐먼트 attribute에 인덱스를 정의할 수 있다.

- hash index : 검색 쿼리에 모든 인덱스 attribute가 제공되는 경우에 빠르게 액세스 할 수 있다. 인덱스는 ==연산에만 사용되며 range 쿼리는 지원하지 않아 정렬을 할 수 없다.

- skiplist index : 인덱스 들을 정렬되게 유지하기에. 비교, 범위 정렬들의 쿼리에 사용될 수 있다. For low selectivity attributes, skiplist indexes will be more efficient than non-unique hash indexes

- persistent index : sorted skiplist 인덱스와 비슷하게 작동한다. 단, 모든 인덱스 값은 디스크에 유지되므로 서버를 다시 시작하거나 인덱스를 다시 로드 할 때 메모리에서 다시 작성할 필요가 없다. persistent 인덱스의 연산은 로그 복잡성을 갖지만 실제로 인덱스를 사용하여 도큐먼트를 페치하려면 기본 인덱스에 추가 왕복이 필요할 수 있기 때문에 작업에 skiplist 인덱스의 연산보다 더 높은 상수 계수가 있을 수 있다.

- geo index : 좌표 값을 통해 도큐먼트를 찾는다. 도큐먼트에 대한 좌표를 latitude, longitude를 이용해 잡아줄 수 있다.

**sparse vs non-sparse index**

- sparse 인덱스에는 인덱스 속성 중 하나 이상이 설정되지 않았거나 값이 null 인 도큐먼트가 포함되어 있지 않다.

- 이러한 도큐먼트는 sparse index에서 제외되므로 non-sparse index보다 적은 수의 도큐먼트가 포함될 수 있다. 이렇게 하면 인덱싱 속도가 빨라지고 인덱싱 된 attribute가 컬렉션의 모든 도큐먼트가 아닌 일부 도큐먼트에서만 발생하는 경우, 메모리 사용이 줄 수 있다. sparse 인덱스는 존재하지 않거나 선택적인 attribute가 인덱싱되는 경우 non-unique hase index의 충돌 수를 줄입니다.

**Index Utilization**

- 대부분의 경우 Arangodb는 쿼리에서 컬렉션당 하나의 인덱스를 사용한다. 쿼리는 여러 필터를 or로 조합하여 컬렉션당 2개 이상의 인덱스를 사용할 수 있으며, and연산으로 컬렉션당 하나의 인덱스를 사용할 수 있다.

- 동일한 콜렉션의 서로 다른 attribute에 대해 멀티 인덱스를 작성하면 인덱스를 선택할 때 쿼리 Optimizer가 더 많은 선택을 할 수 있다. 다른 attribute에 대해 여러 개의 인덱스를 만들면 여러 속성에 대한 FILTER 조건을 사용하여 다른 쿼리의 속도를 높일 수 있다.

- ArangoDB의 primary 인덱스, 엣지 인덱스, hash 인덱스는 자동으로 selectivity estimate를 제공한다. 인덱스 selectivity estimate는 웹 인터페이스에 제공된다 getIndexes () 는 값을 반환하고 explain() 는 주어진 결과물을 출력한다.

**Troubleshooting(문제해결)**

- 주어진 AQL 쿼리를 실행하는 데 사용할 인덱스와 사용할 인덱스에 대해 의심이가는 경우 쿼리 view의 웹 인터페이스에서 explain버튼을 클릭하거나 ArangoShell에서 explain()을 사용하면 된다.

var query = "FOR doc IN collection FILTER doc.value > 42 RETURN doc";

var stmt = db.\_createStatement(query);

stmt.explain();

- explain() 명령어는 쿼리실행에 대한 계획을 JSON표현으로 반환해 준다. JSON 형식으로 결과물이 나오는 것은 코드적으로 사용하기 위한 것이다. 사람이 읽을수 있고 보다 간결하게 쿼리를 설명하기 위해 다음과 같은 툴이 있다.

var query = "FOR doc IN collection FILTER doc.value > 42 RETURN doc";

require("@arangodb/aql/explainer").explain(query);

- 쿼리가 인덱스를 사용하지 않는다고 하면, 다음과 같이 하면 된다.

1. attribute 이름의 철자가 올바른지 점검, attribute의 이름을 잘못 적용하여도 에러를 발생하지 않기에

2. 쿼리에 사용된 컬렉션에 대한 getIndexes()의 반환 값을 확인하고 해당 인덱스가 쿼리의 필터 조건에 사용 된 attribute에 실제로 있는지 확인합니다.

3. 인덱스가 있지만 쿼리에서 사용되지 않으면 인덱스의 타입이 잘못되었을 수 있다. 예를 들어 해시 인덱스는

비교(==)에만 사용되지만 <, <=,>,> =과 같은 다른 비교 유형에는 사용되지 않는다. 또한 모든 인덱스 attribute가 쿼리의 FILTER 조건에 사용되는 경우에만 hash 인덱스가 사용됩니다. 적어도 첫 번째 attribute 가 FILTER 조건에서 사용되는 경우에만 skiplist 인덱스가 사용된다. 추가적으로 skiplist index attribute를 쿼리에 지정하면 더 많은 도큐먼트를 필터링 할 수도 있다.

4. 인덱스 된 attribute를 함수의 변수 또는 임의의 표현식으로 사용하면 attribute의 인덱스가 사용되지 않을 수 있습니다. 예를 들어 다음 쿼리는 value에 대한 인덱스를 사용하지 않는다.

FOR doc IN collection FILTER TO\_NUMBER(doc.value) == 42 RETURN doc

FOR doc IN collection FILTER doc.value - 1 == 42 RETURN doc

이 경우 연산의 한쪽에 인덱스 attribute만 존재하도록 쿼리를 다시 작성해야하며, 또는 도큐먼트의 양을 제한하기 위해 추가 필터와 인덱스를 사용해야한다.

**Working with Indexes**

**Index identifiers and Handles**

인덱스 핸들은 디비에서 인덱스를 유니크하게 식별한다. 문자열이고, /로 구분된 컬렉션 이름과 인덱스 식별자로 구성되어 있다. 인덱스 식별자는 ArangoDB로 자동 생성된 숫자 값으로 되어있다.

인덱스 핸들이나 인덱스 식별자를 이용한 컬렉션의 특정 인덱스 접근

db.collection.index("<index-handle>");

db.collection.index("<index-identifier>");

db.\_index("<index-handle>");

**collection methods**

* getIndexes() : 컬렉션의 모든 인덱스 리스트

arangosh> db.test.ensureHashIndex("hashListAttribute",

........> "hashListSecondAttribute.subAttribute");

arangosh> db.test.getIndexes();

* Index 생성

Hash : ensureHashIndex

Skiplist : ensureSkiplist

fulltext, geo1 (1개의 속성), geo2 (2개의 속성)

sparse, unique 설정 가능

arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a" ], sparse: true });

arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a", "b" ], unique: true });

* Index drop

db.\_dropIndex(index) : 인덱스가 없으면 false 리턴, 있으면 삭제하고 true 리턴

db.\_dropIndex(index-handle) : 핸들러로 인덱스 삭제

arangosh> db.example.ensureIndex({ type: "skiplist", fields: [ "a", "b" ] });

arangosh> var indexInfo = db.example.getIndexes();

arangosh> indexInfo;

arangosh> db.\_dropIndex(indexInfo[0])

arangosh> db.\_dropIndex(indexInfo[1].id)

arangosh> indexInfo = db.example.getIndexes();

* Index 가 사용되었는지 다시 확인

arangosh> var explain = require("@arangodb/aql/explainer").explain;

arangosh> db.example.ensureIndex({ type: "skiplist", fields: [ "a", "b" ] });

arangosh> explain("FOR doc IN example FILTER doc.a < 23 RETURN doc", {colors:false});

**Hash Indexes**

: 복잡도 O(1)로 도큐먼트의 위치를 찾아내는 쿼리에 사용. 해쉬 인덱스가 unique 하다면, 두개의 도큐먼트는 같은 속성 값을 가질 수 없다.

Uniqueness를 위반하면 도큐먼트를 새로 생성하거나 업데이트 할 수 없다. 인덱스가 sparse로 선언되면, 인덱스 속성값이 정해져 있지 않거나 null 일경우, 도큐먼트는 인덱스에서 제외되며 uniqueness 체크가 수행되지 않음.

**Accesing Hash Indexes from shell**

1. Unique Hash Indexes

unique = true , unique constriant

field1,.. fieldn을 path 속성으로 사용하여 unique hash index 생성. 적어도 하나의 path 속성이 주어져야 함. 인덱스는 디폴트로 non-sparse로 설정됨.

컬렉션에 모든 documents들은 무조건 인덱스로 구분되어야 한다. 새 도큐먼트를 생성하거나 원래의 도큐먼트를 업데이트하는 것은 속성의 uniqueness가 위반되면 실패한다.

Sparse = true , sparse한 인덱스를 만듬

인덱스가 성공적으로 생성되면, 인덱스 식별자가 리턴된다.

존재하지 않는 속성은 null. Sparse 인덱스에서는 null값을 가진 모든 도큐먼트는 인덱스에서 제외된다. 이러한 도큐먼트들은 uniqueness 체크에 포함되지 않는다. Non-sparse 인덱스에서는 모든 도큐먼트들이 null 이든 아니든 인덱스에 포함되며 uniqueness 체크에도 포함된다.

인덱스가 성공적으로 생성되면, 인덱스 식별자는 인덱스 부가 설명과 함께 리턴된다.

arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a", "b.c" ], unique: true });

arangosh> db.test.save({ a : 1, b : { c : 1 } });

arangosh> db.test.save({ a : 1, b : { c : 1 } });

arangosh> db.test.save({ a : 1, b : { c : null } });

arangosh> db.test.save({ a : 1 });

결과



1. Non-unique Hash indexes

Non-unique는 unique 옵션 걸지 않음

나머지는 unique hash index와 동일

1. Hash array indexes

Fields: [“field1[\*]”,…, “fieldn[\*]”] -> 배열로 선언

나머지는 위와 동일

arangosh> db.test.ensureIndex({ type: "hash", fields: [ "a[\*]" ] });

arangosh> db.test.save({ a : [ 1, 2 ] });

arangosh> db.test.save({ a : [ 1, 3 ] });

arangosh> db.test.save({ a : null });

**Ensure uniqueness of relations in edge collections**

\_from 과 \_to 를 사용하여 세컨더리 인덱스를 생성하는 것이 가능함. Unique 옵션을 걸어 두 개의 필드를 같이 인덱스를 결합시키면 중복된 관계 생성을 방지한다.

예를 들어, verts라는 도큐먼트 컬렉션은 verts/A, verts/B, verts/C 라는 도큐먼트 핸들을 가진 정점들을 포함할 수도 있다. 이러한 도큐먼트들의 관계는 예를 들어 edges라는 이름의 엣지 콜렉션에 저장될 수도 있다. 이제 verts/A가 한번 이상의 엣지에 의해 verts/b 와 연결되지 않게 한다. unique하고 non-sparse한 해쉬 인덱스를 통해 만들 수 있다.

db.edges.ensureIndex({ type: "hash", fields: [ "\_from", "\_to" ], unique: true });

edges에 {\_from: “verts/A”, \_to:”verts/B”} 라는 엣지를 생성하는 것은 딱 한번만 용납된다. A에서 B로 가는 다른 엣지를 저장하려고 한다면 서버에서 unique constraint violated 에러와 함께 거절된다. 이것은 업데이트도 적용된다.

B에서 A로의 관계 추가는 가능하고, A에서 A나, B에서 B도 추가가능하다.

**Skiplists**

도큐먼트의 1개 이상의 속성으로 스킵리스트 인덱스를 정의하는 것은 가능하다. 이 스킵리스트는 주어진 범위내에서 도큐먼트의 위치를 찾아내는 쿼리에 사용된다. 스킵리스트가 유니크하게 선언되면, 두개 이상의 도큐먼트는 같은 속성값을 가질 수 없다.

Uniqueness가 위반되면 새로 도큐먼트를 만들거나 도큐먼트를 업데이트 할 수 없다. 스킵리스트가 sparse로 선언되면, 도큐먼트는 인덱스에서 제외될 것이고, 인덱스 속성이 정해지지 않았거나 null 값인 경우 uniqueness 체크가 이루어지지 않을 것이다.

**Accessing skiplist indexes from the shell**

* Unique skiplist index

collection.ensureIndex({ type: "skiplist", fields: [ "field1", ..., "fieldn" ], unique: true })

* Skiplist를 unique하게

Non-sparse가 default ( sparse : true 추가)

Uniqueness와 sparse의 관한 설정은 hash 와 같음

* Non-unique skiplist index