[DB] Index

Index란?
- Index 구현 자료구조
- Index 장단점
- Index의 종류

B-tree와 B+tree
- **Balanced Tree란?**
- B-tree란?
- B+tree란?
- B-tree VS B+tree
- B-tree/B+tree가 DB Index에 적합한 이유

Clustering Index VS Secondary Index
- Clustering Index
- **Secondary Index**

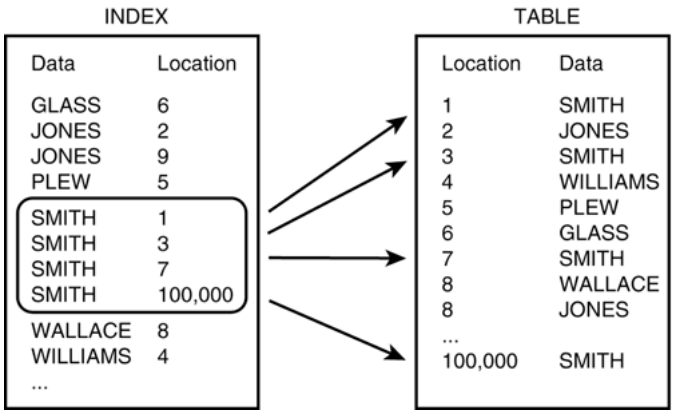
Full Table Scan와 Index Scan
- **Full Table Scan**
- Index Scan

Index 효과적으로 사용하는 방법
- Full Scan이 더 효과적인 경우
- 실무에서의 Index 고려 예시

참고
- MySQL vs MongoDB
- MySQL **InnoDB란?**
- 다중 컬럼 인덱스는 어떤 식의 구조를 가지는가?

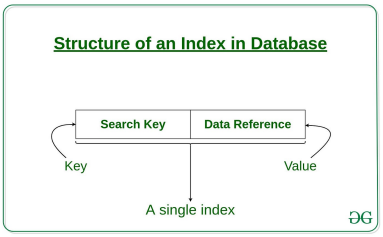
추가 질문
- 왜 DB 쿼리의 where 절에서 1=1를 사용하는가?

Index란?



http://knowtechstuffz.blogspot.com/2015/04/how-sql-indexes-work-internally.html

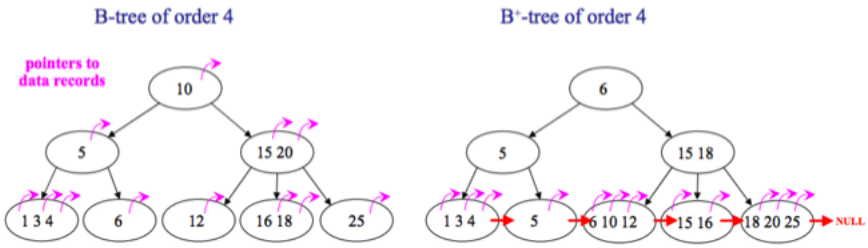
- 데이터베이스에서 테이블의 **검색 성능**을 높여주는 방법
- select 성능 향상
  (where, order by, join)
- 빠른 탐색을 위한 **정렬**, 해쉬되어 있음 (자료 구조에 따라 다름)
- (search-key, pointer) 구조로 별도 파일에 저장



  - search-key: Column값
  - pointer: 데이터의 물리적인 위치
- 보통 테이블 크기의 10% 정도의 저장 공간 차지
- Full Table Scan 없애줌

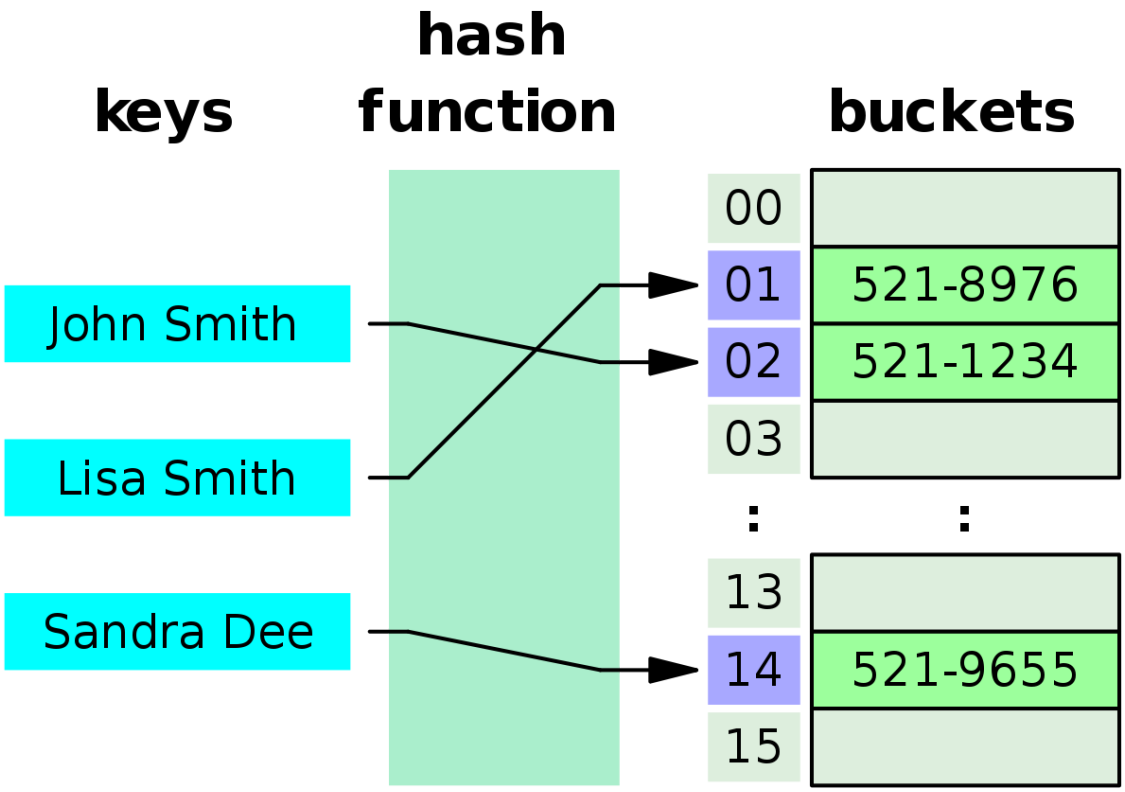
Index 구현 자료구조

- **B-tree / B+tree**



https://stackoverflow.com/questions/870218/what-are-the-differences-between-b-trees-and-b-trees

- Hash
  - 동등 비교 최적화



- Bitmap
  - B-tree 단점 보완 (공간 낭비 / not, null, or 연산 성능 낮음)

상품테이블

상품 ID	1	2	3	4	5	6	7	8	9	10
상품명	KA387	RX008	SS501	WDGS	CY690	K1EA	PO782	TQ84	UA345	OK455
색 상	GREEN	GREEN	RED	BLUE	RED	GREEN	BLUE		BLUE	RED

BLUE	0	0	0	1	0	0	1	0	1	0
GREEN	1	1	0	0	0	1	0	0	0	0
RED	0	0	1	0	1	0	0	0	0	1
NULL	0	0	0	0	0	0	0	1	0	0

상품_색상_BITMAP_IDX

http://www.gurubee.net/lecture/3264

유형	장점	단점	주로 활용되는 상황	시간복잡도
B-tree B+tree	- 범위 쿼리에 효과적 (데이터가 정렬되어있음)	- 데이터 갱신 느림	- 일반적인 RDBMS - 데이터 값이 다양한 경우	검색: O(log n) 삽입/삭제: O(log n)
Hash	- 동등 연산(==)에 효과적 - 비교적 적은 용량	범위 쿼리 불가능	정확한 일치 검색이 필요한 경우	검색/삽입/삭제: O(1) (* Hash Collision이 없는 경우)
Bitmap	적은 용량 차지	데이터 갱신 시 모든 비트맵 인덱스를 변경해줘야함	- 데이터 값의 종류가 적고, 동일한 데이터가 많은 경우 (enum) - 쿼리에 OR 연산자가 많은 경우	검색/삽입/삭제: O(n/W) W: bit 수

Index 장단점

장점

- 검색 성능 향상

단점

- 추가 저장 공간 필요 (약 10%)
- 자주 변경되는 컬럼(Update, Insert, Delete)은 성능 저하
  (B-tree로 구현된 index 재구성이 필요하기 때문)

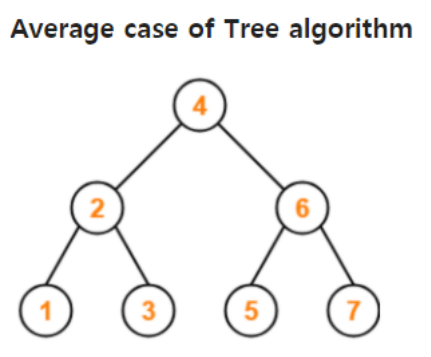
Index의 종류

- Clustering index
- Secondary index

B-tree와 B+tree

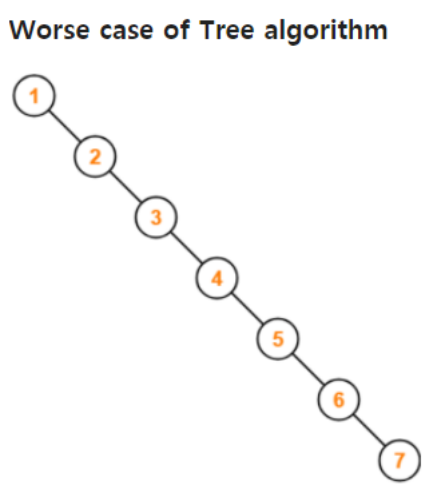
💡 **이진 탐색 트리**

- 평균: O(log N)



https://helloinyong.tistory.com/296

- 최악: O(N)

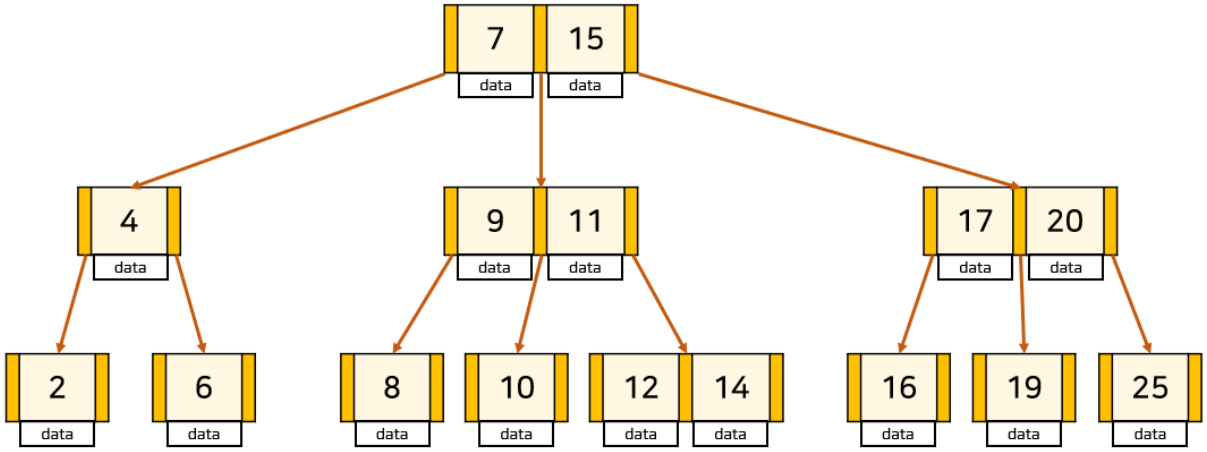


https://helloinyong.tistory.com/296

Balanced Tree란?

- 균형잡힌 트리
- 트리가 한쪽으로 치우치지 않게, 특정 규칙에 따라 트리 재정렬
- 이진 탐색의 최악의 복잡도 개선
- 종류: B-tree, AVL-Tree, RedBlack-Tree 등

B-tree란?



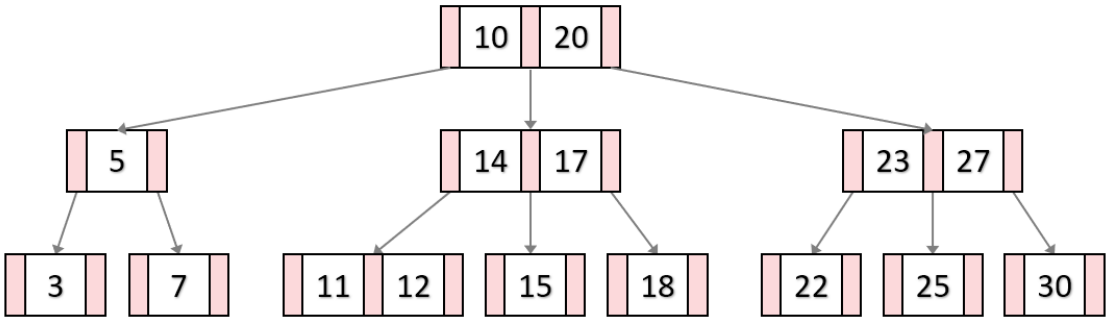
https://velog.io/@tlsdnxkr/CS-DB-인덱스-인덱스-자료구조

- **하나의 노드에 여러 데이터** 저장 가능
- Root - Branch - Leaf 형태
- 모든 노드는 Key로 항상 정렬 되어있음 (이진 탐색 트리 특성)
- Root로부터 Leaf까지 거리 일정
- 모든 노드가 (key, data)로 이루어져있음
- 탐색, 삽입, 삭제: O(log N)
- 노드의 최대 개수는 정해져 있음

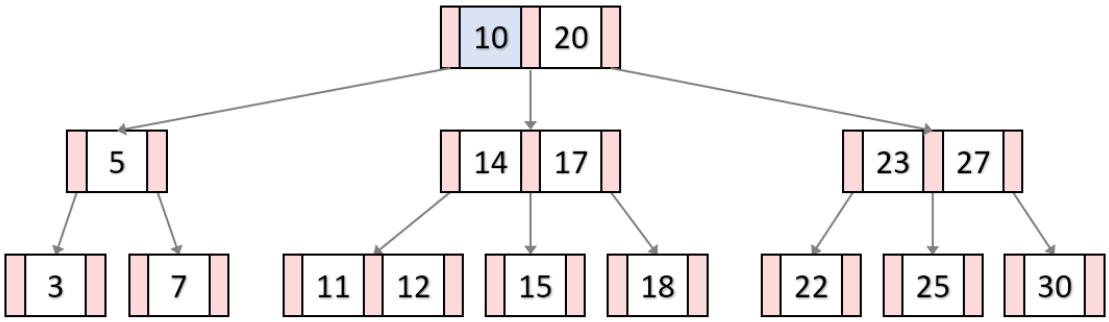
▼ [참고] 탐색/삽입/삭제 과정

1. **탐색**

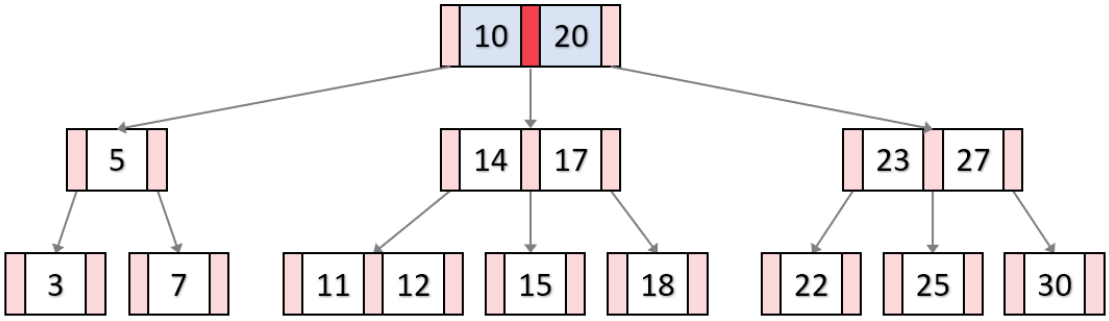
1 18 검색 시작, 루트노드의 key를 순회하면서 검색 시작



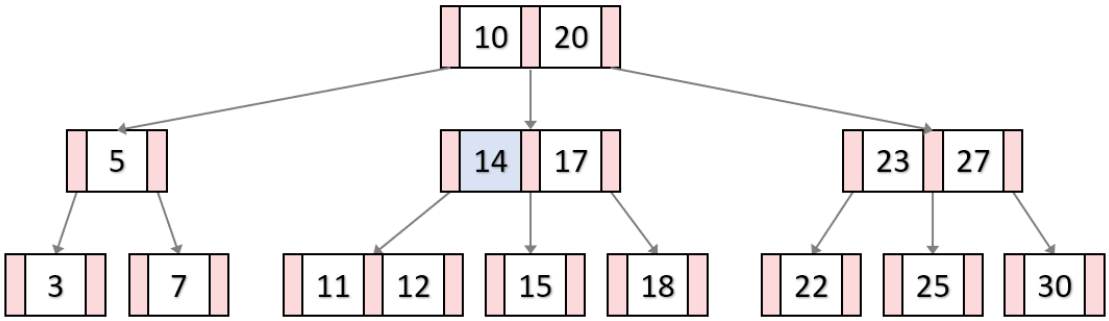
2 18은 10보다 크기 때문에 다음 key를 검사



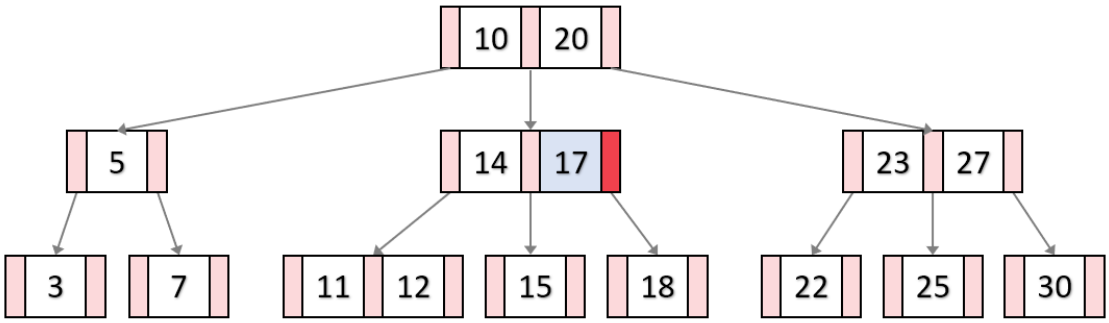
3 18은 10보다 크고, 20보다 작기 때문에 10과 20 사이의 자식 노드로 이동



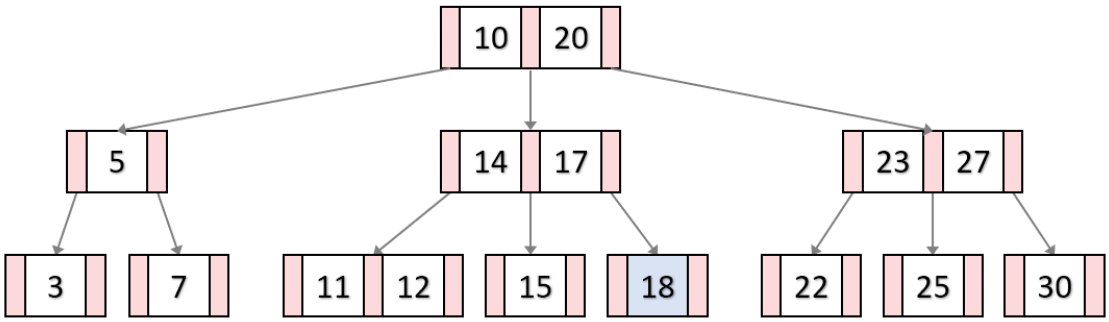
4 자식노드에서 다시 검색 시작. 18은 14보다 크기 때문에 다음 key를 검사



5 18은 노드의 가장 마지막 key인 17보다 크기 때문에 노드의 가장 마지막 자식노드로 이동



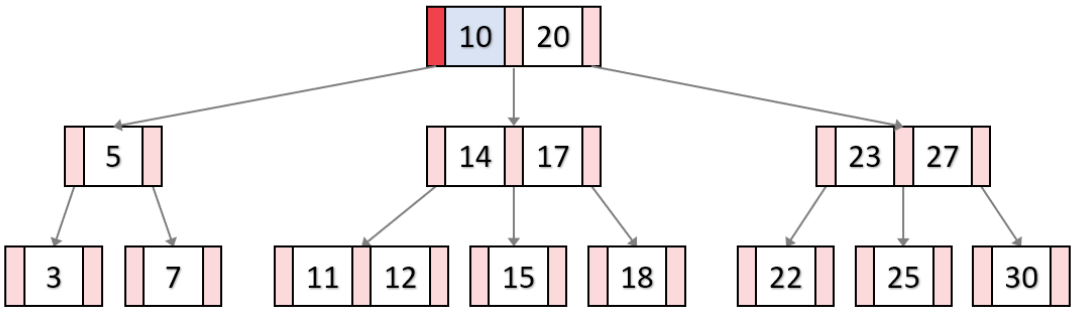
6 18 검색 완료



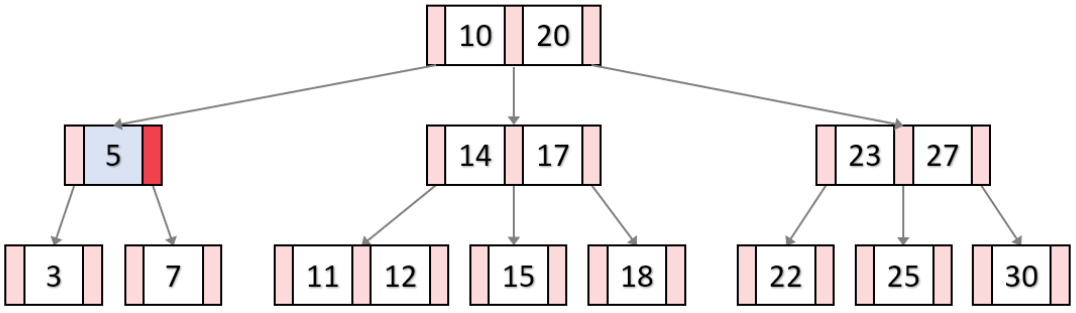
2. 삽입

   a. 분할 일어나지 않는 경우

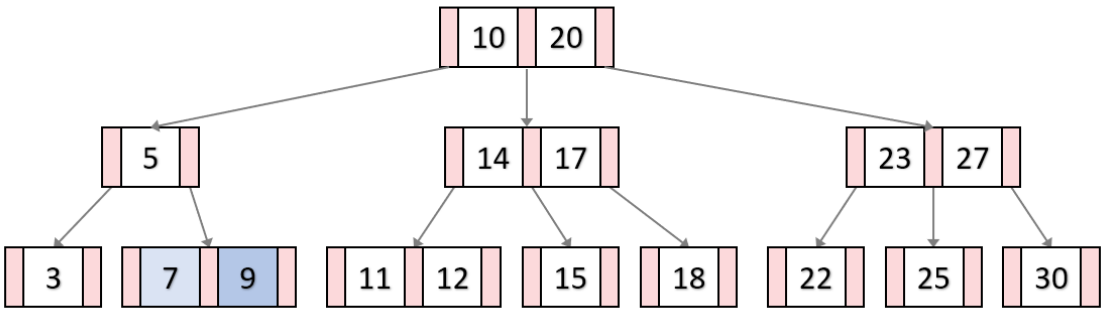
1 9 삽입 시작, 루트노드에서 key 10이 9보다 크기 때문에 가장 왼쪽 자식노드로 이동



2 9는 5보다 크기 때문에 가장 오른쪽 자식노드로 이동

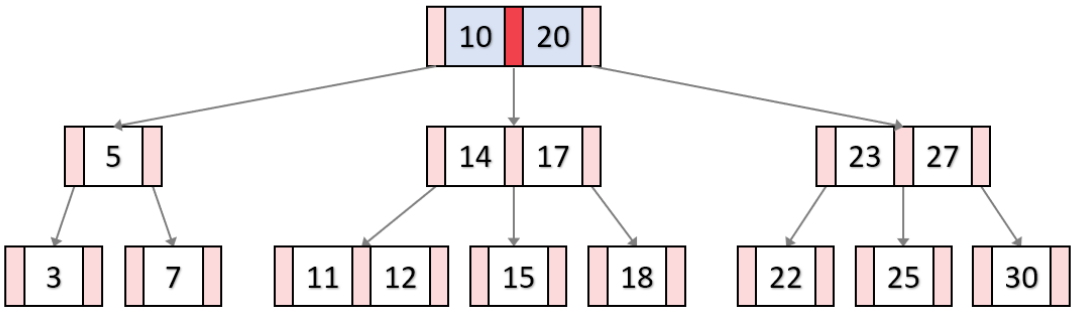


3 리프노드에 도달, 9는 7보다 크기 때문에 7 오른쪽에 9 삽입 완료

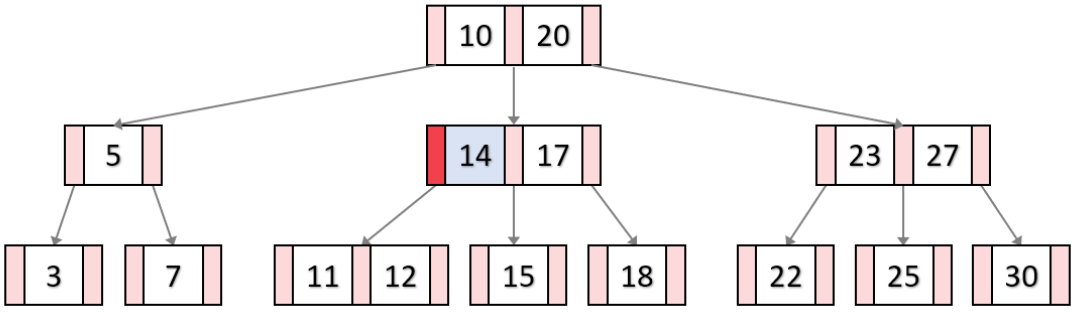


b. 분할 일어나는 경우

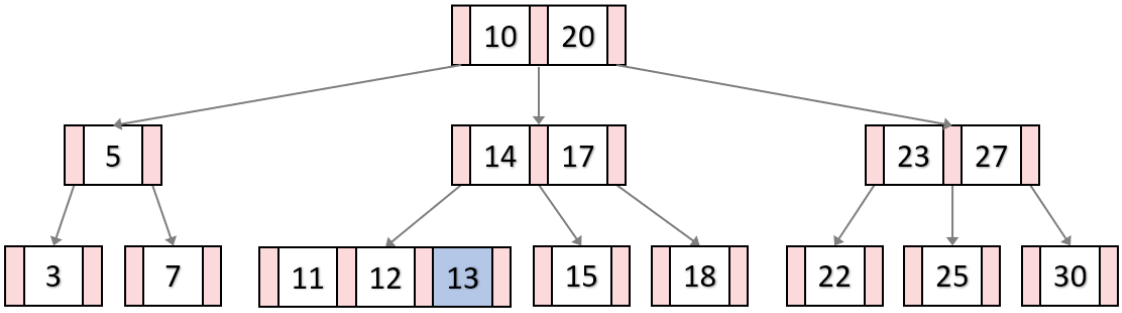
1 13 삽입 시작, 루트노드의 key 중 10보다 크고 20보다 작기 때문에 중간 자식노드로 이동



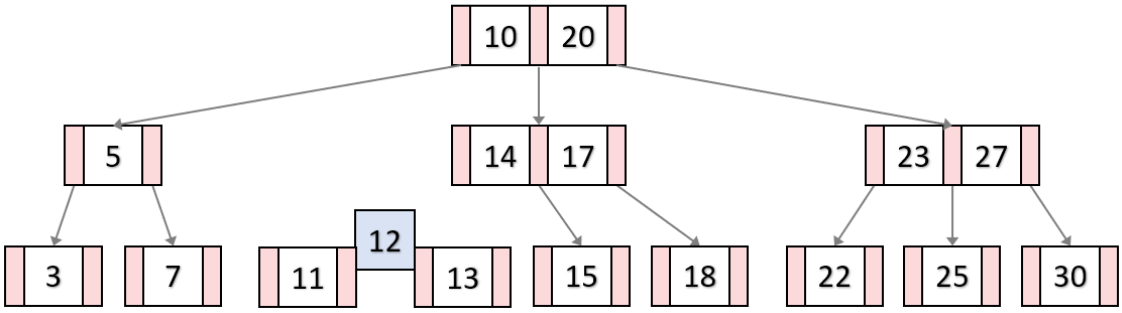
2 13은 14보다 작기 때문에, 가장 왼쪽 자식노드로 이동



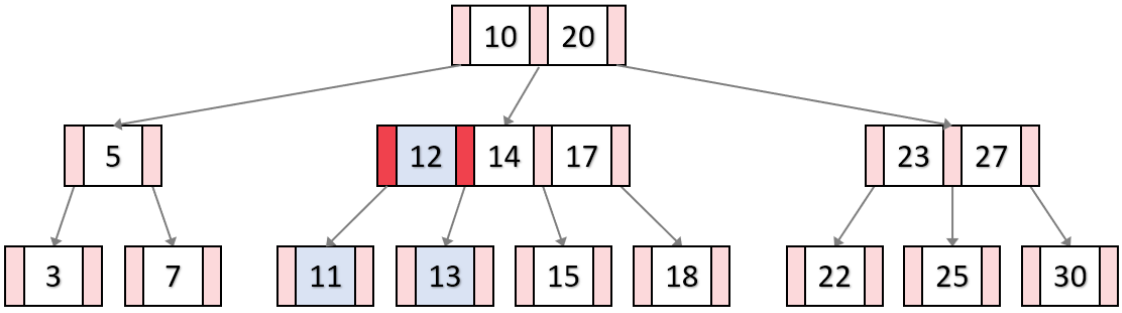
3 13은 12보다 크기 때문에 가장 마지막에 삽입. 해당 노드가 최대로 가질 수 있는 key의 개수를 초과했기 때문에 분할을 수행



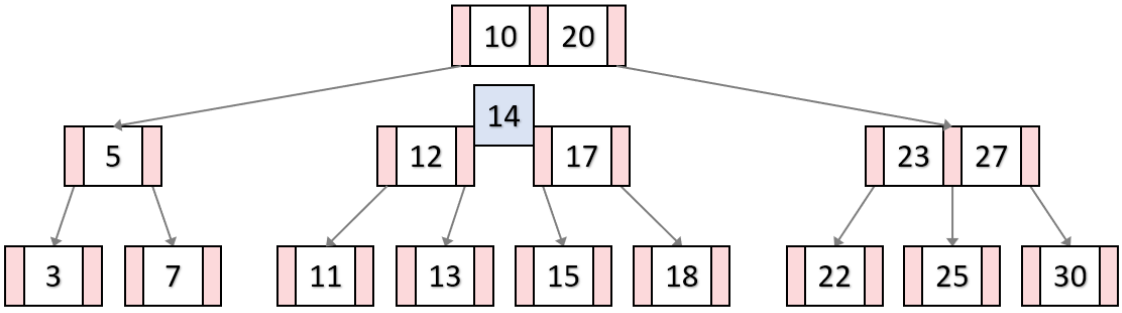
4 중앙값 12를 기준으로 분할을 수행



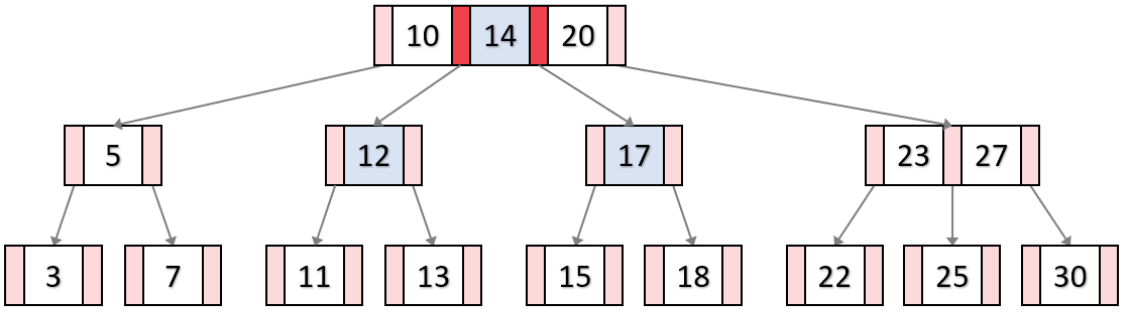
5 12는 부모노드에 오름차순으로 삽입, 11과 13은 12의 왼쪽자식, 오른쪽 자식으로 설정



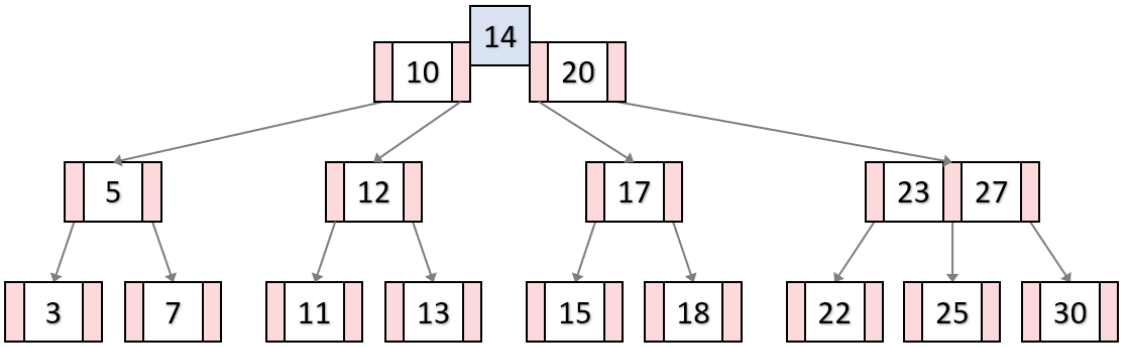
6 12가 병합된 노드가 최대로 가질 수 있는 key의 개수를 초과, 중앙값 14를 기준으로 분할을 수행



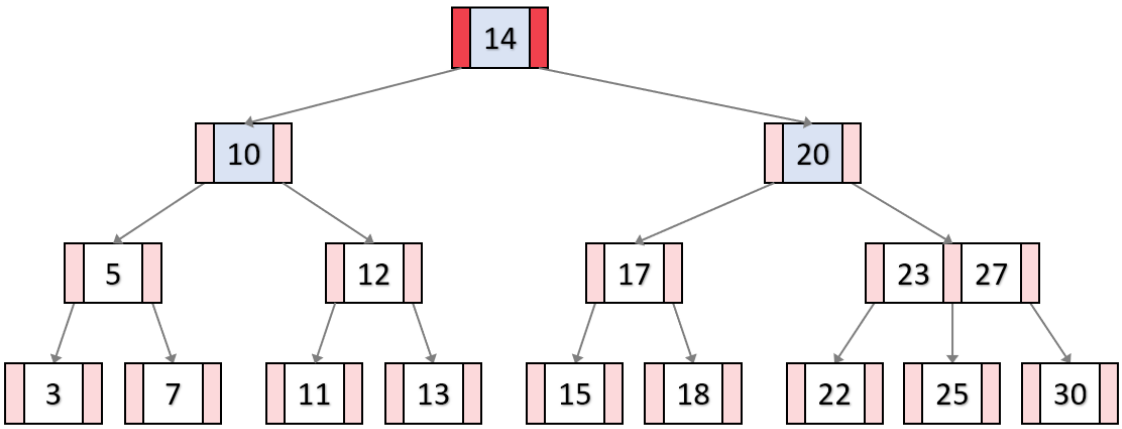
7 14는 부모노드에 오름차순으로 삽입, 12과 17은 14의 왼쪽자식, 오른쪽 자식으로 설정



8 14가 병합된 노드가 최대로 가질 수 있는 key의 개수를 초과, 중앙값 14를 기준으로 분할을 수행



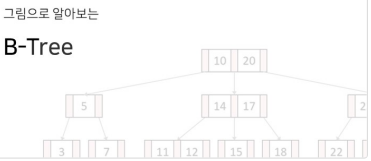
9 14가 새로운 루트노드가 되고, 10과 20은 14의 왼쪽자식, 오른쪽 자식으로 설정. 삽입과정 완료



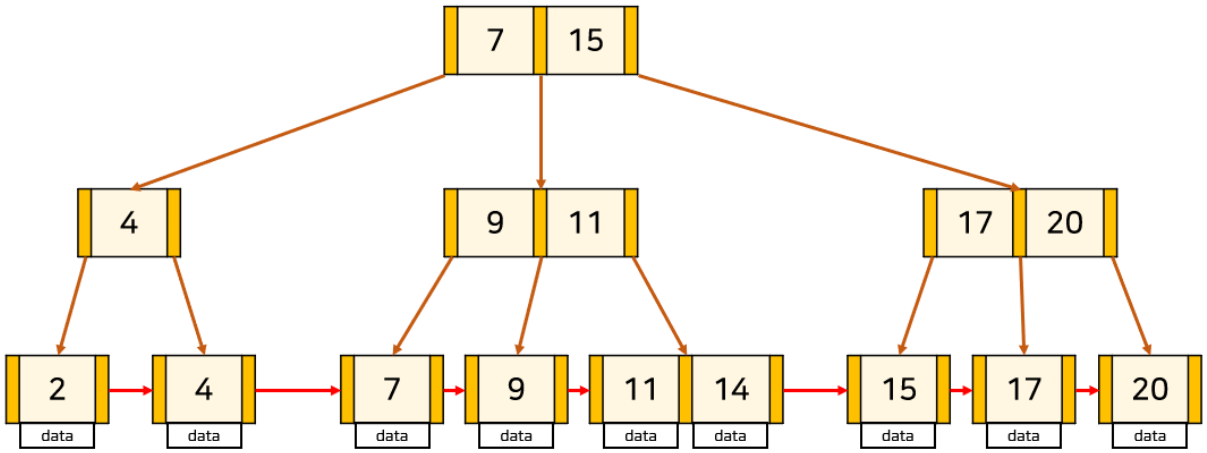
[자료구조] 그림으로 알아보는 B-Tree

B트리는 이진트리에서 발전되어 모든 리프노드들이 같은 레벨을 가질 수 있도록 자동으로 벨런스를 맞추는 트리입니다.

https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Tree



B+tree란?



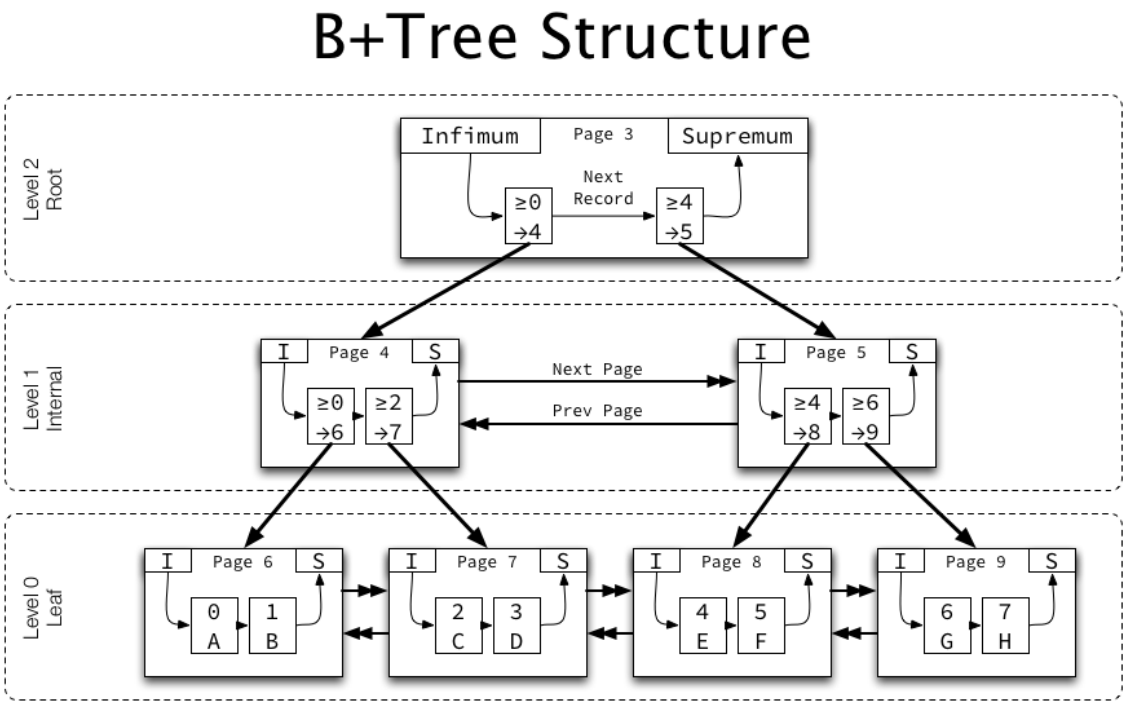
https://velog.io/@tlsdnxkr/CS-DB-인덱스-인덱스-자료구조

- B-tree의 확장
- 다른 점
  - data는 Leaf 노드에만 저장
    - Root, Branch: (Key)
    - Leaf: (key, data)
  - Leaf 노드는 `Linked list`로 연결
  - 중간 노드의 key를 통해 leaf 노드 찾아감 → key 중복 가능

B-tree VS B+tree

구분	B-tree	B+tree
데이터 저장	모든 노드에 데이터 저장 가능	leaf 노드에만 데이터 저장 가능
키 중복	없음	있음
leaf 노드 연결	없음	leaf노드 끼리 linked list로 연결
검색 효율성	단일 탐색에 좋음	범위 탐색에 좋음 (단일 탐색 시, 데이터가 리프노드에 있어 가장 밑에까지 내려가야함)
크기	상대적으로 큼 (모든 노드에 key, data값이 존재해서)	상대적으로 작음 (리프노드를 제외하고는 key만 저장해서)
활용	파일 시스템, 데이터베이스 Index등에 사용	데이터베이스 Index에 주로 사용

▼ **[참고] MySQL에서의 InnoDB B+트리 구현**



- 같은 레벨의 노드: Double Linked List로 연결
- 자식 노드: Single Linked List

B-tree/B+tree가 DB Index에 적합한 이유

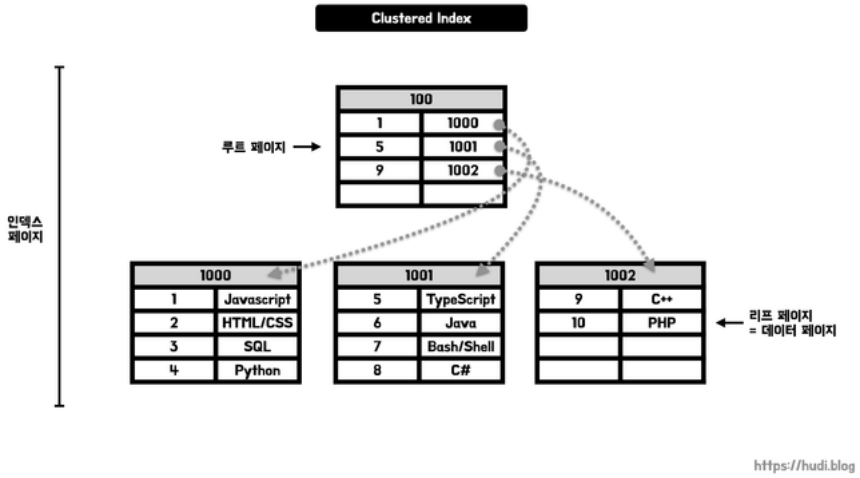
1. 항상 정렬된 상태 → 범위 연산에 유리
2. 검색, 삽입, 삭제: O(logN)

Clustering Index VS Secondary Index

> 💡 **비슷하지만 다른 용어들**
>
> - **Clustering Index VS Secondary Index (MySQL 기준)**
> - **Clustered Index vs Non-Clustered Index**
> - **Primary Index vs Secondary Index**

*** Page: MySQL의 B-Tree의 노드*

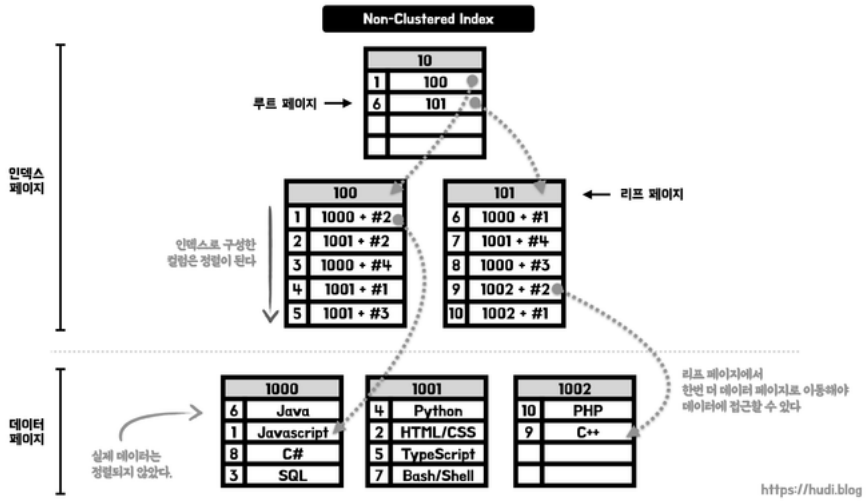
Clustering Index



https://hudi.blog/db-clustered-and-non-clustered-index/

- 영어사전!
- DB에서 Index 자동 생성해주는 경우
  - PK 선언 시
  - (PK가 없을 때에만) unique & not null 설정 시
- Index와 이어진 실제 물리적 데이터도 정렬
  (인덱스 생성시 오래 걸림)
- Index의 leaf 페이지가 실제 데이터를 가리킴
- 테이블 당 1개만 존재
- CRUD 성능
  - R: 물리 데이터에 직접 접근해서 빠름!
  - CUD: 물리 데이터의 순서까지 재정렬 해야해서 느림

Secondary Index



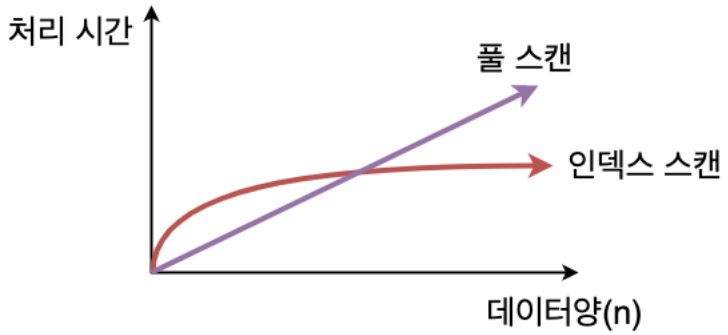
https://hudi.blog/db-clustered-and-non-clustered-index/

- 교과서 뒤의 단어 index!
- 자동 생성
  - unique 키 선언 시
  - MySQL에서 Foriegn Key 선언 시
    (다른 DB는 생성 안해줄 수도 있음)
- 수동 생성: 사용자가 직접 index 생성

```
CREATE INDEX [인덱스이름] ON [테이블이름]([column1],[column2]...)
```

- Index와 이어진 실제 **물리적 데이터은 그대로** 있음
- Index의 leaf 페이지가 데이터의 주소를 가리킴
  (논리적 / 그래서
  `페이지 번호 + 오프셋`을 저장함)
- 테이블 당 여러개 존재 가능
- CRUD 성능
  - R: 데이터 주소를 통해 다시 찾아가야해서 느림
  - CUD: 인덱스 구조만 재조정하면 되서 빠름!

Full Table Scan와 Index Scan



https://zorba91.tistory.com/293

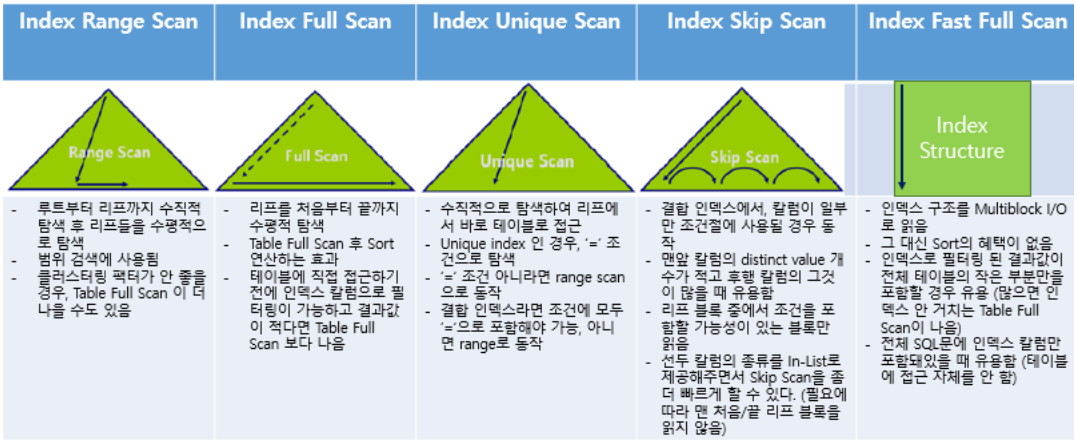
Full Table Scan

- 인덱스 사용하지 않은 경우
- 테이블 내부적으로 들어오는 순서대로 데이터 저장
  ⇒ 테이블에 있는 모든 데이터 읽어서 탐색

Index Scan

- Index를 이용하여 스캔

  (Unique > Range > full > fast full > skip)



https://velog.io/@jkjan/인덱스-기본원리

Index 효과적으로 사용하는 방법

💡 **Cardinality & Selectivity**

*** 두 집합의 특성을 고려한 절대적인 수치가 아닌 상대적인 수치!*

- Cardinality(카디널리티):
  - 특정 Column 데이터의 종류(중복 정도/distinct)
    ex) 성별: 남/여 ⇒ 2
  - 중복이 적을수록 카디널리티 높음
    중복이 많을수록 카디널리티 낮음
  - ex)
    high: 주민등록번호, 이메일 주소
    mid: 이름, 우편번호
    low: 성별, 상태
  - 중복이 적어야 효과적으로 index를 필터링하기 좋음
- Selectivity(선택도):
  - 특정 값을 얼마나 잘 선택할 수 있는지

```
Selectivity = Cardinality / 레코드수
```

학생 테이블 예시

ID	Name	Gender	Age
1	John	Male	25
2	Alice	Female	30
3	Bob	Male	28
4	Mary	Female	22
5	David	Male	35
6	Emily	Female	28
7	Alex	Male	30
8	Sarah	Female	32
9	Jake	Male	26
10	Emma	Female	29

Column	**Cardinality**	**레코드수**	**Selectivity 계산**
ID	10	10	10 / 10 = 1.0 (100%)
Name	10	10	10 / 10 = 1.0 (100%)
Gender	2	10	2 / 10 = 0.2 (20%)
Age	8	10	8 / 10 = 0.8 (80%)

- Column 선정 기준
  - Where 절에서 **자주 조회 /**

    Join 조건으로 자주 사용
  - 데이터 **수정 빈도 낮음**
  - **Cardinality 높음**

    - **Selectivity 낮음**
- 데이터 양이 많은 경우
- 한 테이블 당 4~5개 정도로만 index 생성
  (수정 오버헤드 때문에)

Full Scan이 더 효과적인 경우

- table 데이터가 조금 있을 때 (몇 십, 몇 백건)
- 조회하려는 데이터가 테이블의 많은 부분을 차지할 때

▼ *[틈새 퀴즈] Index 성능 좋아지니까 모든 컬럼에 인덱스 사용해도 되나?!*

NO!

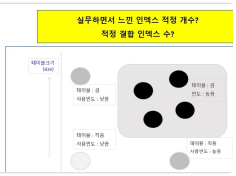
- 데이터 수정 시 모든 index 업데이트, 정렬 해야해서 성능 저하
- 저장 공간 차지

▼ 실무에서의 Index 고려 예시

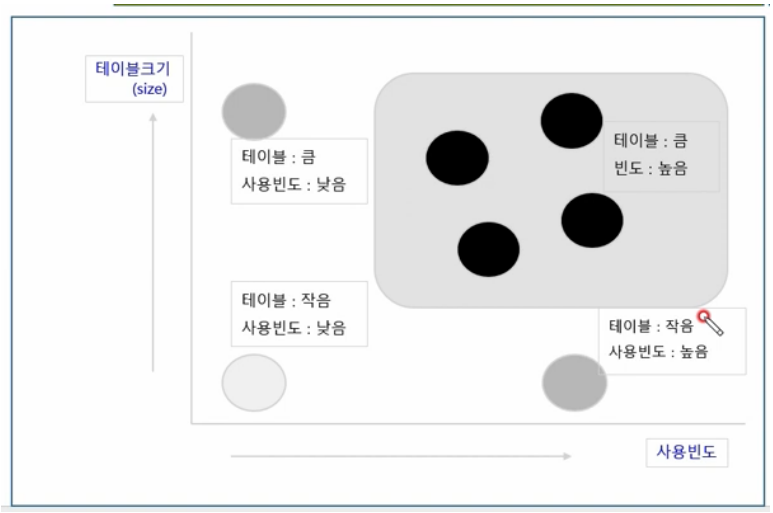
#7. Oracle Index 현업 실무자의 Index 이야기 | 적정인덱스 개수는 몇개인가? 결합 인덱스 컬럼은 몇개까지 만드는게 좋을까?

현업에서 오라클 DB를 운영하면서 Index에 느낀 부분을 개인적으로 정리해 보았습니다.
DBA도 아니고 전문 튜닝을 하는 것오 아니라서 조금 어설프지만,
저와 같은 업무를 하는 분들에게 도움이 되기를 바라면서 영상 올립니다.

https://www.youtube.com/watch?v=IrinK1_YUS4



- 현장에서 성능의 필요에 따라서 만들기
- 보통은 테이블 크고, 빈도 높을 때 만들면 좋음



- 예시

Table : Library

컬럼 (전체 Rows = 1,000,000)		카디널리티 (distinct)	선택도 (selectivity) (Cardinality / Total Rows)
Key	SEQ	1,000,000	100.00%
책제목	TITLE	1,000,000	100.00%
저자	WRITER	100,001	10.00%
저자 이메일	EMAIL	100,001	10.00%
출판사	PUBLISHER	100,001	10.00%
국내/국외	DOMESTIC	2	0.00%
전자책 엽	EBOOK	2	0.00%
장르	KIND	1,001	0.10%
등록일자	REG_DAY	45	0.00%

- 이것도 역시 자주 사용하는지, select하는 구간이 정말 많은지 확인
  (선택도가 높더라도 자주 사용하지 않으면 만들필요가 없다)
- 해당 카디널리티 내의 데이터값의 분포에 따라 다를 수 있다
- 선택도가 10%로 낮아보여도, 테이블의 크기를 보았을 때 데이터 자체는 높은 값일 수 있다
- 다중 컬럼 인덱스에 관하여

```
select *
from library
where 1=1
and domestic='Y'
and kind='KFRKYLZGHF-장르'
and writer='HDWNPGYMEM';
```

  - domestic, kind, writer 을 다중 컬럼 인덱스로 했을 경우

1	PK_LIBRARY	UNIQUE	VALID	NORMAL	N	NO	(null)	NO	SEQ
2	IDX_LIBRARY_01	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	DOMESTIC, KIND, WRITER

    - domestic만 탐색 ⇒ FULL
    - kind/writer만, kind+writer 탐색 ⇒ skip scan
    - 3가지 다 ⇒ range scan
  - writer만 인덱스로 했을 경우

4	IDX_LIBRARY_03	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	WRITER

    - 3가지 다 ⇒ range scan

- index에서 가져온 후(access) 필터링(filter)

⇒ 인덱스로 만들기 좋은 것을 대표적으로 만든 후 가져오는게 나을 수 있지 않을까

⇒ 결론: 현업간의 협의를 하자!!

참고

MySQL vs MongoDB

(둘다 공식문서로는 B-tree라하고, 사람들은 B+tree라고 함)

MySQL의 InnoDB

- 구현: B-tree

> MySQL :: MySQL 8.0 Reference Manual :: 17.6.2.2 The Physical Structure of an InnoDB Index
>
> With the exception of spatial indexes, InnoDB
> indexes are B-tree data
> structures. Spatial indexes use
>
> https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html

- Clustering Index, Secondary Index

MongoDB

- 구현: B-tree

> Indexes
>
> .leafygreen-ui-zzkys8{font-size:16px;line-height:28px;font-family:'Euclid Circular A','Helvetica Neue',Helvetica,Arial,sans-serif;display:-webkit-inline-box;display:-webkit-inline-flex;display:-ms-inline-flexbox;display:inline-flex;-webkit-align-items:center;-webkit-box-align:center;-ms-flex-align:center;align-
>
> https://www.mongodb.com/docs/manual/indexes/#b-tree

- 기본키는 ObjectID Document 생성 시 자동으로 ObjectID형성

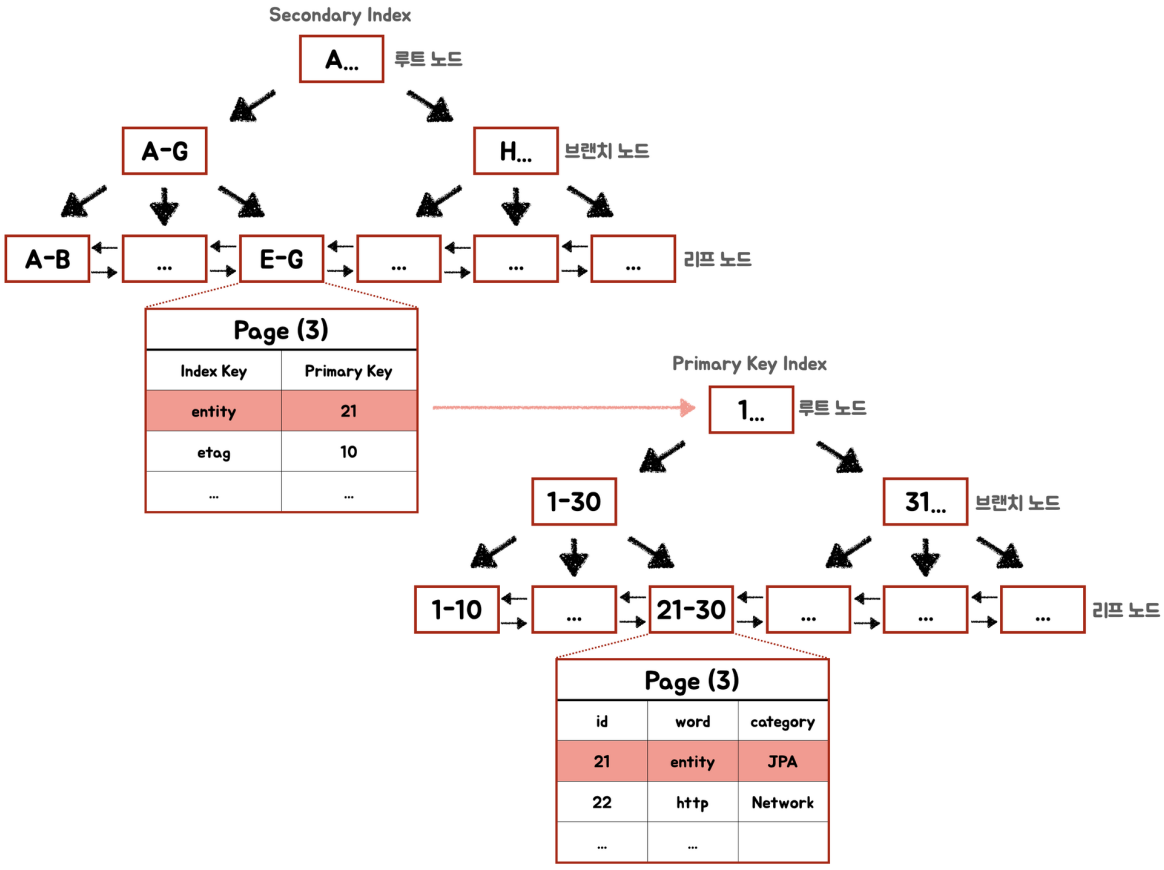
MySQL InnoDB란?

- MySQL의 스토리지 엔진 중 하나

    - 스토리지 엔진: DBMS가 데이터 베이스에 대해 CRUD를 할 때 사용하는 기본 SW 컴포넌트

  ** 서버 엔진(DBMS)가 팀장, 스토리지 엔진(InnoDB)가 사원 느낌!

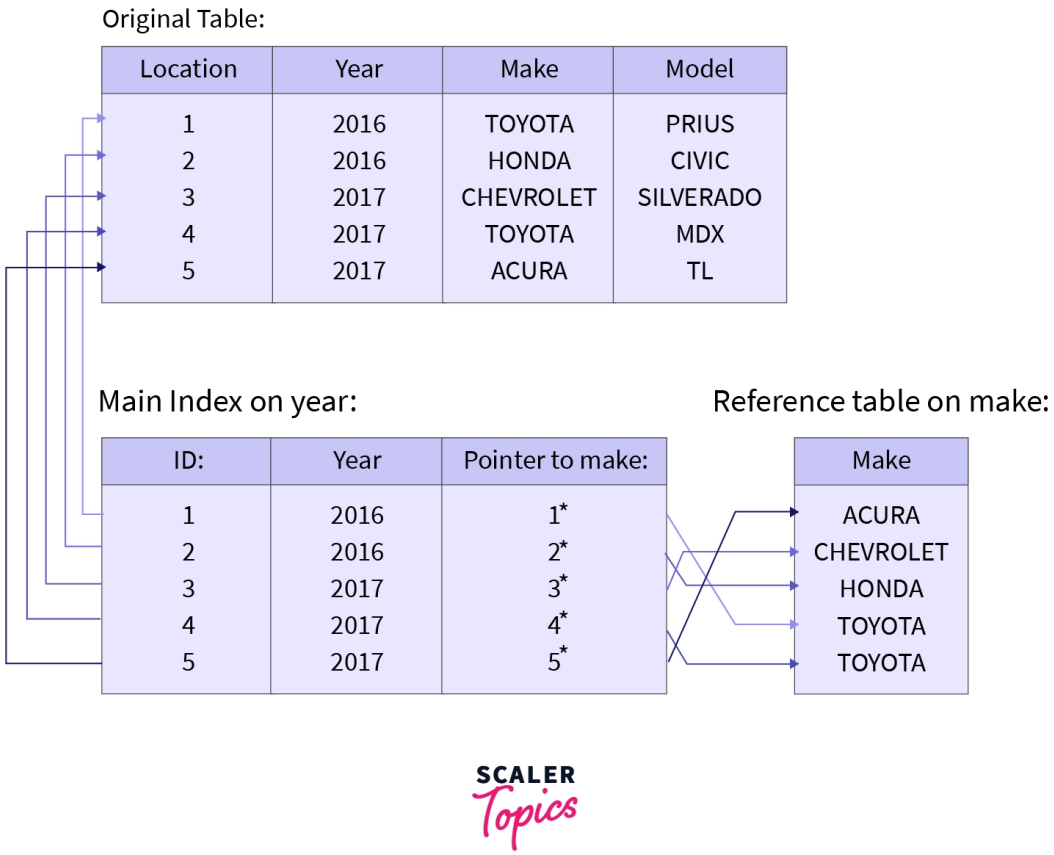
- 기능: 트랜젝션 제공, 외래키 제공 등
- Secondary Index가 Primary Key Index에 대한 값을 담고 있음



https://sihyung92.oopy.io/database/mysql-index

다중 컬럼 인덱스는 어떤 식의 구조를 가지는가?

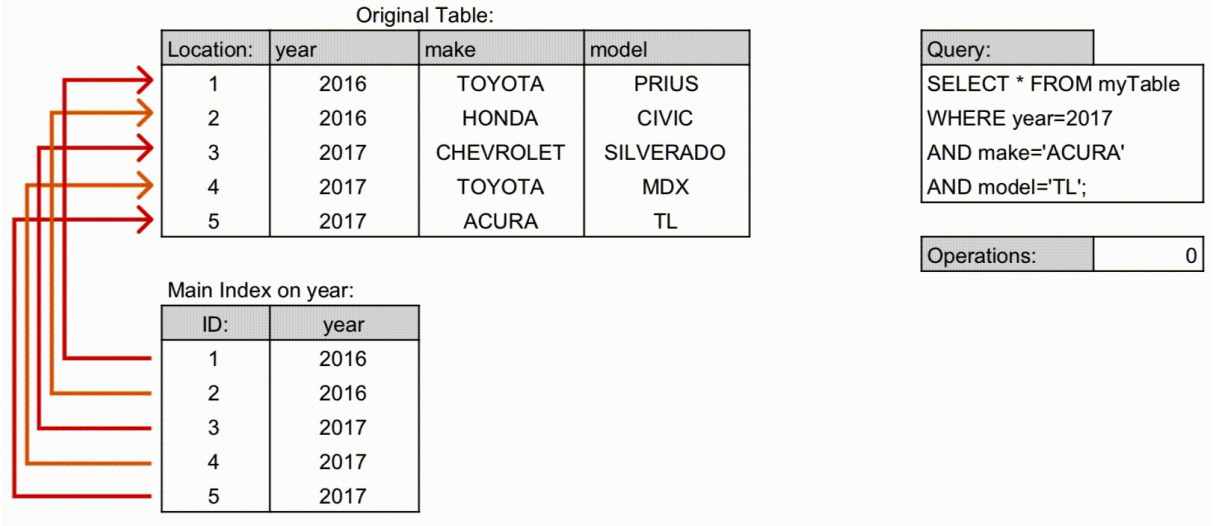
(year, make)를 다중 컬럼 index로 둔 경우



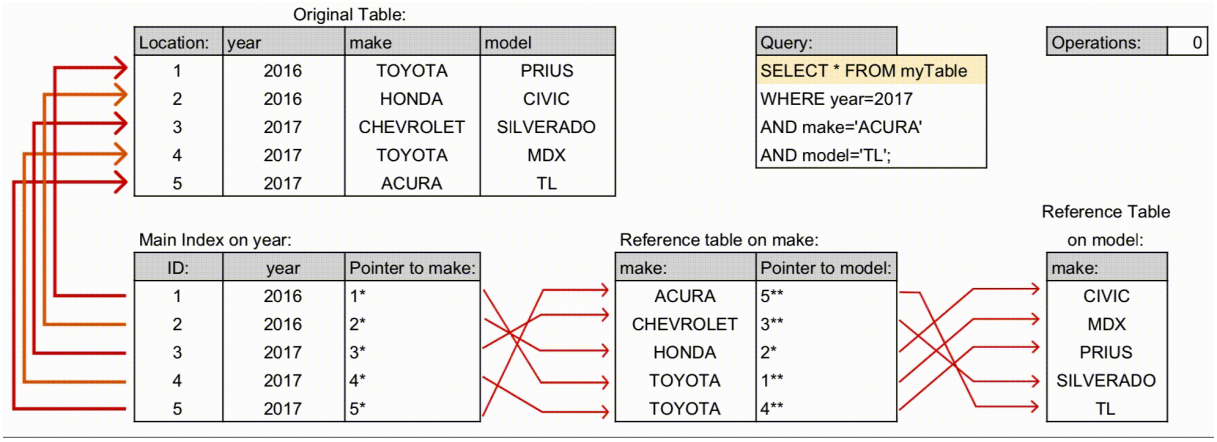
https://www.scaler.com/topics/dbms/indexing-in-dbms/

총 Operations 차이

- 기존 인덱스: (year)을 index로 둔 경우



- 다중 컬럼 인덱스: (year, make, model)을 다중 컬럼 index로 둔 경우



추가 질문

왜 DB 쿼리의 where 절에서 1=1를 사용하는가?

```
SELECT *
FROM USER
WHERE 1=1
AND name = 'Jack'
AND gender = 'M'
AND age > 20
```

- 1=1: true
- 일관성을 통해 쿼리 가독성, 디버깅, 작성에 편리함을 준다

1. 쿼리 디버깅 시 주석처리

```
# 1=1을 사용하지 않는 경우
SELECT *
FROM USER
WHERE -- name = 'Jack'
-- AND gender = 'M'
AND age > 20

# 1=1을 사용하는 경우
SELECT *
FROM USER
WHERE 1=1
-- AND name = 'Jack'
-- AND gender = 'M'
AND age > 20
```

2. 동적 쿼리를 처리 (복잡함이 줄어듦)

```
// 1=1을 사용하지 않는 경우
String query = "SELECT * FROM USER";

if(userName == null) {
    query += " WHERE NAME = '" + userName + "'";
}
if(userGender == null) {
    query += (userName == null) ? " WHERE" : " AND";
    query += " GENDER = '" + userGender + "'";
}

// 1=1을 사용하는 경우
String query = "SELECT * FROM USER WHERE 1=1";

if(userName == null) {
    query += " AND NAME = '" +  userName+ "'";
}
if(userGender == null) {
    query += " AND GENDER = '" + userGender + "'";
}
```

- 주의 사항
  - select 절에만 사용할 것 (Update, Delete에는 의도치 않게 변경, 삭제될 위험이 있음)
  - 쿼리 성능 저하 가능성