2021058995 황인혁

세가지 조인 알고리즘의 설명

1. Nested loop join(중첩 루프 조인)

Nested loop join은 두 릴레이션 r과 s에 대하여 릴레이션 각각의 튜플들을 모두 비교하여 조인을 수행한다. 따라서 r에 있는 레코드마다 릴레이션 s를 모두 스캔한다. 만약 각각의 릴레이션에서 한 블록씩만 버퍼로 할당할 수 있다고 한다면 block transfer는 r의 레코드 수 \* s의 블록의 수 + r의 블록의 수가 된다. Seek은 r의 레코드마다 s가 스캔될 때 s에서 seek이 매번 발생하고 r의 블록 수 만큼의 seek이 r에서 발생하므로 총 seek은 r의 레코드 수 + r의 블록 수가 된다. 만약 각각의 릴레이션의 블록들을 모두 버퍼로 할당할 수 있다고 한다면 block transfer는 r의 블록 수 + s의 블록 수가 되며 seek은 2번이 된다.

Nested loop join에서 외부 릴레이션의 각 튜플이 아닌 블록마다 내부 릴레이션을 읽어 비교한다면 최악의 경우 block transfer의 횟수가 r의 블록의 수 \* s의 블록의 수 + r의 블록의 수가 되고 Seek은 2 \* r의 블록의 수가 된다.

1. Merge join(합병 조인)

Merge join은 두 정렬된 릴레이션 r과 s에 대하여 각각의 처음 레코드부터 값을 비교해가며 순차적으로 스캔한다. Block transfer는 r의 블록 수 + s의 블록 수이기 때문에 정렬된 두 릴레이션에 대하여 join을 수행한다면 최적의 block transfer를 얻는다. 하지만 정렬 되어있지 않은 두 릴레이션에 대해서는 정렬하는 cost가 별도로 들어가기 때문에 무작위로 되어있는 릴레이션들에 대해서는 좋지 않은 join 방법이다.

1. Hash join(해시 조인)

Hash join은 어떠한 두 릴레이션 r과 s가 있을 때 각각의 r과 s에 동일한 hash 함수를 적용하여 해시 값에 알맞은 버킷으로 튜플들을 이동시킨 뒤 hash index가 일치하는 probe input과 build input에 대한 튜플들을 비교하며 join하는 방법이다. Block transfer 수는 만약 모든 튜플들이 해시 함수에 맞게 probe input과 build input으로 잘 나눠져 있는 상태라면 최적의 block transfer인 r의 블록 수 + s의 블록 수가 나온다. 하지만 해시함수에 맞게 튜플들을 분배하는 과정에서의 cost가 존재하며, 해시함수에 알맞게 분배되더라도 각각의 bucket들이 uniform한지의 여부에 따라 성능이 달라진다. 또한 hash 버킷의 overflow가 발생할 수도 있으므로 상황에 매우 적합한 해시 함수를 구하는 것은 매우 까다로운 일이다. 일반적으로 Hash join은 잘 사용되지 않는 것으로 알려져 있다.

분석)

Case1에 대해서는 두 릴레이션이 name에 대해서 정렬 되어있기 때문에 merge join을 수행하면 open count가 각각의 블록 개수의 합, 즉 1000 + 1000 = 2000이 된다. 이미 정렬 되어있기 때문에 해당 상황에서 가장 좋은 알고리즘이다. Open의 횟수를 세기 위하여 open\_count라는 변수를 도입하였고 출력 결과 총 2001번의 open이 이루어졌다 (output open까지 포함). Output 파일의 튜플 수는 총 10000개 나왔다.

Case2에 대해서는 name\_age 릴레이션은 age에 대해서 정렬 되어있고 name\_salary 릴레이션은 salary에 대해서 정렬 되어있다. 따라서 조인 대상의 attribute가 아닌 attribute에 대해 정렬 되어있으므로 모두 무작위인 상황과 같고 merge join은 적합하지 않다. 해당 자료를 살펴보면 연봉과 나이는 독립적인 관계를 따른다. 하지만 각 csv파일에 속한 자료들을 살펴보면 한 csv 파일에 대하여 같은 글자로 시작하는 이름들이 많이 보인다는 사실을 알 수 있다. 따라서 이름의 첫 글자에 따라 hash bucket들을 생성한 뒤에 hash join을 실행하는 것이 맞다고 판단하였다. 해시 버킷으로 튜플들을 옮기는 과정에서 open의 횟수를 줄이기 위하여 한 csv 파일이 있을 때 만약 튜플의 첫 글자가 처음 등장하는 튜플이라면 open을 실행하고 처음 등장하는 튜플이 아니라면 이전에 open한 fstream을 활용하였다. Name의 첫 글자는 a부터 j까지 총 10종류가 있기 때문에 probe\_input 버킷과 build\_input 버킷이 총 각각 10개씩 만들어졌다. Hash bucket의 overflow는 고려하지 않는 것으로 가정하였고, 각각의 probe\_input 과 build\_input을 차례대로 hash 인덱스에 따라 nested loop join을 수행하였다. Open\_count 는 출력 결과 총 10420번 이루어졌다 (output open 포함). Output 파일의 튜플 수는 총 10000개 나왔다.

Case3에 대해서는 데이터가 모두 무작위로 저장되어 있기 때문에 nested loop join을 활용하였다. 성능을 조금이라도 향상시키기 위하여 1학기 성적을 포함하는 릴레이션과 2학기 성적을 포함하는 릴레이션의 연산 결과를 바로 학생 번호가 있는 릴레이션과 join을 수행하여 pipelining이 잘 수행될 수 있도록 하였다. 올바르게 구현을 한 것 같지만 성능이 좋지 못해 상당히 오랜 시간이 걸려 output을 결국 다 출력하지 못하였다.

Troubleshooting

Case2번째 과정에서 hash join을 위해 각 튜플들을 bucket으로 옮기는 과정에서 튜플들을 버킷에 이어쓰지 않고 새로운 파일로 만든 뒤 덮어쓰는 방식으로 진행되었다. 해결 방법을 찾아본 결과 open 함수의 인자로 ios::app을 넣어주면 파일이 없는 경우 새로 만들 것이며, 이미 있는 파일에 대해서는 덮어쓰는 것이 아닌 이어 쓰는 방식으로 진행한다는 사실을 알게 되었다. 하지만 이렇게 할 경우 이미 생성된 output 파일에 대하여 다시 실행할 경우 기존에 있던 output 파일에 이어서 똑 같은 값들을 이어 적게 된다. 따라서 실행하기 전에 output 파일과 생성된 버킷들을 모두 삭제한 뒤에 실행시켜야 올바르게 작동한다.

Case3번째 과정에서 시간이 너무 오래 걸려서 결국 output에 다 출력하지 못하였다. 기회가 된다면 나중에 성능을 향상해서 다시 해보자.