직접 파일의 개념

레코드를 직접적으로 접근 장점: 빠른 직접 접근->대화식의 처리(목표 레코드 외에는 접근할 필요가 없음), 다른 레코드에 영향없이 검색, 삽입, 수정, 삭제 가능

상대 파일의 장점->간단, 처리 시간이 거의 걸리지 않음

단점: 물리적 저장장치에 의존적

주소 탐색 방법->사상 함수(A : 키값 -> 주소)

레코드 기록 시 : 키 값 -> 레코드가 저장될 주소

레코드 검색 시: 키 값 -> 레코드가 저장되어 있는 주소

구현방법: 직접 사상, 디렉터리 검사, 계산을 이용한 방법(해싱)

디렉터리 검사 장점-> 빠른 검색

단점: 삽입 비용이 큼, 파일과 디렉터리 재구성 필요

해싱->계산을 이용한 상대 파일 구성법, 키에서 변환된 주소에 레코드를 저장하는 과정

일반적으로 주소공간 << 키 값 공간 , 키 값들을 한정된 주소 공간으로 균등하게 분산시키는게 핵심

장점: 레코드의 주소를 구해 접근

단점: 서로 다른 키가 같은 주소로 할당될 수 있음

변환 함수 : 키->버켓주소

해싱함수 계산 시간 << 보조기억장치의 버켓 접근 시간

해싱을 이용한 파일 설계시 고려사항

적재율 -> 총 저장 용량에 대한 실제로 저장되는 레코드에 대한 비율

해싱 함수 -> 레코드 키 값으로부터 주소를 생성하는 방법

버킷을 크기 -> 하나의 주소를 가진 저장 구역에 저장할 수 잇는 레코드의 수

오버플로우 해결 방법 -> 주어진 주소 공간이 만원이 된 경우의 해결 방법

해시 함수의 특징

역함수X , 치역이 미리 정해져 있음

SHA-256(해쉬 함수로 임의의 정보를 고정된 길이의 암호문, 메시지 다이제스트 등으로 불림)

버킷: 하나의 주소를 가진 저장 구역

둘 이상의 레코드 저장 가능, 하나의 버킷 안의 레코드는 같은 버킷 주소를 가짐, 한 파일의 구성하는 버킷 수가 그 파일 주소 공간이 됨

버킷 크기: 통상적으로 한 번의 접근으로 버킷 내에 있는 레코드들을 전송할 수 있는 크기로 결정, 저장 장치의 물리적 특성과 연관

충돌: 두 개의 상이한 레코드가 똑 같은 버킷으로 해싱되는 것

동거자: 같은 주소로 해싱되어 충돌된 키 값들, 버킷이 만원일 때는 충돌의 문제가 됨-> 오버플로우

버킷 크기를 크게하면: 장점-> 오버플로우 감소 단점-> 저장 공간 효율 감소, 버킷 내 레코드 탐색 시간 증가

적제 밀도가 높으면 오버플로우 확률이 높아짐. 삽입 시 접근 수가 높아짐, 검색 시 접근 수 높아짐

적제 밀도가 낮으면 공간 효울이 떨어짐, 70%이상일 때 충돌이 너무 잦음 30%정도는 공간 필요

오버플로우(삽입할 레코드가 꽉 찬 버킷에 해싱될 때) 해결 방법

개방주소법(계산 접근법, 탐색할 버켓의 주소를 동적으로 계산)

패쇄주소법(체인법)->자료구조 접근법, 오버플로우 버켓을 체인으로 홈 버켓에 연결

선형 조사법의 단점: 집중 현상-특정한 시퀀스 상에 있는 버켓에 해싱이 집중되는 현상

기본 집중-비선형 함수를 사용하여 해결 2차 집중-이중 해싱 사용하여 해결

(2) 어떤 레코드가 파일에 없다는 것을 판단하기 위해 조사해야 할 주소의 수 -> 적제율이 클 수록 많아짐

(3)전치-한 레코드가 자기 홈 주소를 동거자가 아닌 레코드가 차지함으로 인해 다른 레코드의 홈 주소에 저장되는 것

탐색할 때 주소 수 증가 -> 삽입/검색 시간 증가를 야기 대응책->2 패스 해시 파일 생성

이중 해싱(재 해싱)

첫 번째 해싱 함수의 결과에 두 번째 해싱함수를 적용시킨다.

개방 주소법의 문제(레코드의 삭제), 이후의 오버플로우 검색의 실패

해결하려면 논리적인 삭제를 하거나 재해싱 하면 해결됨(주기적인 정비)

동의어 연결(synontm chaining)

별도의 오버플로우 영역 사용, 오버플로우가 발생한 레코드의 리스트 유지

버켓 번지 지정

충돌을 해결하기 위해 해시 테이블의 엔트리로 버켓 사용, 한 버켓은 여러 슬롯으로 구성됨

한 슬롯에는 한 개의 키값을 저장. 3. 해시 테이블은 디스크 유지할 때 적합

버켓 오버플로우 처리 방법

1. 연속 넘침 번지 지정(인접 버켓의 빈 슬롯을 사용)

2. 버켓 체인(충돌한 동의엉를 오버플로우 영역에 저장하고 그 오버플로우 버켓들을 체인으로 연결, 오버플로우 버켓은 보통 한 개의 레코드만 저장)

확장성 직접 파일

확장성 파일을 해싱을 위한 주소 공간이 파일과 함께 확장/축소될 수 있음

파일이 커짐에 따라 주소 공간의 크기가 커지기 때문에 오버플로우를 처리할 필요 없음

해시 값의 사용을 확장시키기 위한 모델이 트라이임

확장설 해싱에서 가장 중요한 것은 더 많은 주소 공간이 필요할 때 더 많은 비트의 해시 값을 사용

해시된 값의 또다른 비트를 사용할 때마다, 2진수를 갖는 트라이의 깊이에 또 다른 단계 추가

종류-> 동적해싱의 종류->(가상 해싱, 동적 해싱, 확장 해싱)

동적 해싱(해쉬된 주소로서 인덱스를 사용)

두단계의 조직

인덱스(주기억장치, 이진트리의 집합으로 구성)

버켓(보조 저장 장치, 실제 데이터를 저장)

두개의 해쉬 함수 사용(H->인덱스 레벨 0의 주소로 변환됨, B->비트열 함수, 1 이상의 인덱스 레벨에 대한 포인터)

인덱스(-N개의 이즌트리의 포레스트, 포레스트는 트리의 집합)

노드 구조(내부 노드-0,부모,왼쪽 자식, 오른쪽 자식, 외부노드-1, 부모, 레코드 수, 버켓 포인터)

확장성 해싱(트라이의 개념을 이용, 완전 이진트리로 변환 후 테이블로 관리) EX) 전자사전

두 단계의 조직

디렉토리 : 정수 값을 갖는 헤더와 리프에 대한 2의d승개의 포인터

리프의 집합: 헤더를 가지는 버켓

하나의 해쉬함수 사용

H: 키\_\_모조키(고정 길이의 비트 스트링)

리프의 헤더 : 버켓 레코드에 대한 공통 전위 비트의 개수

검색-모조 키의 처음 d비트를 디렉토리에 대한 인덱스로 사용하여 디렉토리에서 리프를 찾아 레코드를 접근

삽입-리프를 찾아 삽입, 리프가 가득 찼으면(새로운 리프 할당, 리프의 헤더가 T이면 모조키의 T+1번째 비트의 값에 따라 C+1개의 레코드를 분할

7장 인덱스 된 순차 파일(삽입 삭제가 매우 어려움)

순차 파일: 순차 접근 방법. 직접 파일: 직접 접근 방법 -> 장점을 취함

구조: 순차 데이터 파일->순차적으로 정렬. 인덱스->파일에 대한 포인터

인덱스 파일의 구현 방법

삽입 삭제시-> 순차 데이터 파일의 레코드 순서 유지 방법, 인덱스 갱신 방법, 2가지로 구현됨

정적 인덱스(인덱스의 구조가 기억장치외의 물리적 특성에 기반, 데이터의 삽입 삭제에도 구조가 불변)

동적 인덱스(블록에 기초한 구현-동적인 구현, 데이터를 블록에 저장하고 순차접근을 위한 블록을 체인으로 연결, 인덱스 구조가 데이터의 삽입 삭제에 따라 동적으로 변화)

정적 인덱스 방법(내용 변경, 구조 불변, 하드웨어의 의존적 설계->보조 기억장치의 물리적 특성(실린더 트랙)

데이터-오버플로우 구역 사용 ex) ibm의 isam

특성

정적인 인덱스 구조->기억장치의 물리적 특성에 기반한 인덱싱, 인덱스 구조 변화 없음

데이터 삽입 공간을 위해-여유공간을 가진, 오버플로우 구역

구성(엔트리->키 값, 포인터(실린더 번호, 트랙 번호)

3단계의 인덱스로 구성된(마스터 인덱스(최상위 레벨 인덱스-주기억장치에 있음, <최고키값 포인터>), 실린더 인덱스(최고키값, 실린더 번호), 트랙 인덱스(최저키값, 트랙 번호))

데이터 저장 구조(기본구역->각 실린더, 오버플로우 구역(분리된 파일->추가적인 삽입, 오버플로우 포인터<실린더, 트랙, 레코드 번호> 한 실린더 당 한 오버플로우 구역 또는 한 파일 당 한 오버플로우 구역)

Isam->특정 하드웨어의 특성에 맞도록 설계

장점-> 접근 시간 단축, 기억 공간의 효율성 단점->기억장치의 유형 변경 또는 파일의 복사시 문제

동적 인덱스 방법(b+트리, 하드웨어 독립적)

데이터->블록으로 저장, (오버플로우->분열->예비공간) 언더플로우->합병

블록에 기초한 구현(동적인 구현)

인덱스 파일-인덱스 블록의 트리구조

다중 레벨 인덱싱(인덱스의 인덱스 파일) 최고 레벨 인덱스(마스터 인덱스)는 주기억장치에 적합

인덱스 엔트리 =<키 애트리뷰트 값, 포인터>

데이터 파일

-순차적인 구조-> 데이터 블록들 블록들 사이에 자유공간이 분포(나중의 삽입을 위해)

데이터 블록들은 논리적인 순서로 연결->데이터 블록 체인

Vsam 파일의 구조(제어 구간-데이터 레코드 저장, 제어 구역-제어 구간의 모임, 순차셋-제어구역에 대한 인덱스 저장, 인덱스 셋-순차 셋의 상위 인덱스)

9장 다중 키 파일(하나의 데이터 파일에 다수의 접근 경로(키))

구현 방법(데이터 중복->각 응용에 맞는 파일을 구성) 기억 공간의 증가, 일관성, 무결성 유지 곤란

한 파일에 대한 다수의 접근 경로 구축 방법(역화일->인덱스의 개념 다중리스트 파일->레코드 간의 다중리스트 연결구조)

역화일 구조

-인덱스를 이용하는 구조

역(인덱스와 데이터 레코드 파일을 연결)

역 인덱스

-데이터 파일에 있는 키 필드 값을 인덱스 키로 모두 포함

-인덱스 엔트리 =(키 값, 레코드 포인터)

-파일은 키 필드에 대해 전도(도치)되어 있음

역의 구성(엔트리에 대한 정렬/ 비정렬 모두 가능, 역 인덱스의 키순서는 레코드 순서랑 같지 않다. 인덱스 구조(테이블, 트리)

직접 파일, 인덱스 된 순차파일 위에 구성 ex) 학번을 키로 하는 직접 파일에 주민번호로 역을 구성(두 키에 의한 접근 모두 가능)

역 인덱스가 만들어지는 수에 따라

완전 역화일(모든 필드에 대한 역 인덱스, 데이터 파일은 존재 안함, 실제 사용 예는 거의 없음)

부분 역화일-몇 개의 필드에 대해서만 역 인덱스 구성

역 인댁스 구성 방법과 대안

역 인덱스의 키값을 데이터 레코드에서 제외 – 처리가 어려워짐

대안-> 간접 주소 기법 이용 인덱스 엔트리=(보조키, 기본키)->주소 값 대신 기본키 사용

질의 처리

오름 차순으로 유지(신속한 레코드 검색, 갱신 시의 추가 부담)

역 리스트 파일(텍스트를 위한 파일)

문서 정보 검색에 쓰이는 파일

대상-> 텍스트로만 구성된 문서, 텍스트 필드를 포함한 레코드

목표 레코드 식별->필드 값 하나가 아닌 여러 개가 목표 문서 식별

키워드 검색의 활용 ex)포털 사이트의 검색시스템, 도서관의 도서 검색시스템, 신문기사 검색 시스템

역 리스트 파일 구조

구성-> 역 인덱스 파일 + 포스팅 파일 + 데이터 파일

- 인덱스 파일 또는 사전(키워드 + 레코드 수 + 포스팅에 대한 포인터)

- 포스팅 파일->키워드를 포함한 데이터 레코드에 대한 포인터의 리스트

역 리스트 파일의 탐색 방법

질의- 특정 키워드를 포함하는 문서를 검색, 불리언 질의와 랭킹 질의로 구분

불리언 질의-> 탐색 키워드 간의 논리적 관계를 논리 연산자($ | !)를 이용하여 표현 ex) 데이터 베이스 & 질의어

랭킹 질의

-질의 키워드와 문서간의 유사도를 평가해서 유사도가 높은 순서로 K개를 결과로 생성함

- 유사도 평가는 대개 휴리스틱 알고리즘을 사용

- 초기 인터넷 검색엔진에 많이 사용

- 최근에는 불리언 질의 방식이 주로 사용됨

장점(구현 용이, 속도가 빠름, 동의어 자원 용이->인덱스, 즉 역화일에 잇는 동의어끼리 스레드(체인)

단점(많은 기억 공간 요구->키워드가 많아질수록 역에 대한 부담 증가, 역의 갱신 및 재구성에는 많은 유지비용 발생->키워드의 삽입, 삭제가 동적으로 이루어지는 경우)

시그니처 파일

구현원리(시그니처라는 개략적인 필터를 사용, 문서의 내용을 부호화하여 공간을 줄임, 자격이 안되는 데이터를 먼저 걸러낸 후 남은 데이터 중에서 목표데이터를 가려 냄, 파일 전체를 검색하는 것보다 훨씬 빠름)

접근 방법

1. 문서들을 텍스트 파일에 순차적으로 저장

2. 이 문서들에 대한 문서 시그니처를 만들어 시그니처 파일에 저장->시그니처는 해시 코드된 비트 패턴으로 구성됨

3. 질의문 처리시 이 시그니처 파일을 먼저 검사해서 부적격 문서를 걸러 냄

4. 나머지 문서들을 검사해서 결과를 새성

시그니처 생성(일반적으로 중접 코딩 방법을 이용하여 생성, 키워드 시그니처->블록 시그니처-> 문서 시그니처)

중첩 코딩 생성 방법

1. 각 문서를 일정수의 상이한 키워드를 포함하는 논리 블록으로 분할

2. 각 치코드에 대해 F개의 비트로 구성된 키워드 시그니처를 작성

3. 한 논리 블록에 속한 D개의 키워드 시그니처들에 OR연산을 수행하여 하나의 블록 시그니처를 작성\

4. 문서에 속한 블록 시그니처들을 모두 접합하여 문서 시그니처를 작성

시그니처 파일의 장점-> 전문검색보다 2배 빠름, 10~15%의 추가 공간만 필요(역 파일의 인덱스는 50%~300%의 추가 공간을 필요), 추가적인 삽입만을 허용하므로 삽입 연산이 간단)

단점->대규모 DB에서는 속도 저하, 개략적 필터(서로 다른 키워드가 동일한 해시 결과를 가져올 수 있다. 블록 시그니처를 만들 때 OR연산을 하는데 다른 키워드와 우연히 일치할 수 있음)

허위통과 또는 허위 적중->시그니처 구성에 포함되지 않은 키워드가 매치되는 것

시그니처 파일을 이용한 검색 절차

필터링 단계 – 시그니처를 사용하여 유사 가능성이 높은 후보 문서를 구한다

후처리 단계 – 해당 후보 문서가 실제로 유사한지 검사한다.

시그니처 파일의 문제점(N개의 블록 시그니처를 나타내는 N \* F 이진행렬, 대량의 데이터베이스 시스템의 경우 N,F값이 매우 크다, 질의 시그니쳐와 파일시그니쳐 매칭 시키는 연사이 느려짐)

다음과 같은 3가지 기법 사용(압축, 행렬의 수직분할, 행령의 수평분할)

다중 리스트 파일

역 인덱스(키 값, 동일 키의 모든 레코드에 대한 포인터), 다중리스트 인덱스(키 값, 하나의 레코드에 대한 포인터)

나머지 레코드(데이터 파일에서 연결리스트 구조로 유지)

각 리스트의 헤드만 유지(디렉토리 역할)

역 인덱스와 다중 리스트 인덱스의 차이점

-> 역 인덱스는 질의 처리 능력이 우월하고 다중 리스트 인덱스는 인덱스 관리가 용이하다 또한 역 인덱스는 프로그래머에 투명하며 다중 리스트 인덱스는 논리적 관계에 따른 접근 경로를 제공한다.

파일의 압축

파일 압축의 사용(자주 사용하지 않는 파일들을 장기 보관할 목적으로 저장하는 데 사용, 자료의 양을 줄임으로써 전송시간의 단축과 전송비용의 절감을 위해 사용)

파일 압축의 원리(반복 문자를 반복횟수로, 자주 나타나는 문자는 짧은 부호어로 드물게 나타나는 문자는 긴 부호어로 냄) -> 허프만 코딩

무손실 압축(텍스트 문서나 실행형 파일처럼 완전히 원래의 같은 데이터로 복원할 필요가 있는 경우)

손실 압축(음성 신호 화상 및 그래픽 데이터처럼 대략 비슷한 데이터로 복원해도 좋을 경우)

파일 압축 기법

숫자 자료에 대한 압축(숫자 0의 압축, 델타 코딩 기법)

문자 자료에 대한 압축기법(RLE 가변 길이 인코딩(허프만 코딩)

MPEG의 압축원리

VKI 방식(대개 일정한 간격으로 키프레임을 삽입, 동작 변화가 적은 부분에서는 용량 낭비, 동작 변화가 심한 곳에서는 화질 저하, 키프레임의 삽입 간격을 가변적으로 함) RLE

문자 자료의 압축(VLE)->허프만 코딩

가변-길이 인코딩(자주 나타나는 문자에는 적은 비트를 사용하고 드물게 나타나는 문자에는 많은 비트를 사용하여 인코드 하는 압축 기법)

트라이-하나의 문자 코드가 다른 문자 코드의 접두부가 되지 않는 것을 보장하는 자료구조,

이진 트라이로 비트 스트링을 유일하게 디코드 할 수 없음

허프만 인코딩->여러 트라이중 가장 좋은 트라이를 결정하는 일반적인 기법

우선 순위 큐를 사용하여 빈도수가 가장 작은 문자부터 차례로 트라이를 만듬

허프만 압축법->1. 압축할 파일을 읽어 문자들의 출연 빈도 구함

2. 빈도수가 가작 작은 문자끼리 연결하여 이진 트라이 생성

3. 과정 2를 계속 진행. 모든 문자에 대한 이진 트라이를 완성한다.

4. 완성된 이진 트라이 에서 각 문자를 대표하는 값 부여,

5. 파일 내의 문자들을 대표값으로 변환하여 압축파일을 만듬