Homework 1:

3. Schema

*Customer(customer-name, street, city)*

*Branch(branch-name, city)*

*Account(customer-name, branch-name, account-number)*

**(a) Find the names of all customers who have an account in the ‘Region12’ branch.**

πcustomer−name(σbranch−name=′Region12′ (Account))

SELECT DISTINCT customer-name

FROM Account

WHERE branch-name = 'Region12'

**(b) Find the names of all customers who have an account in a branch NOT located in the same city that they live in.**

πcustomer−name(σA.city<>B.city∧A.branch−name=B.branch−name(ρB(Branch)× ρA(Customer ◃▹ Account)))

(i) Implicit cross join:

SELECT DISTINCT A.customer-name

FROM Account A, Branch B, Customer C

WHERE A.customer-name = C.customer-name AND A.branch-name = B.branch-name AND B.city <> C.city

(ii) Explicit join:

SELECT DISTINCT A.customer-name

FROM Account A JOIN Branch B ON A.branch-name = B.branch-name JOIN Customer C ON A.customer-name = C.customer-name

WHERE B.city <> C.city

**(c) Find the branches that do not have any accounts.**

πbranch−name(Branch) − πbranch−name(Account)

(i) Not in:

SELECT DISTINCT branch-name

FROM Branch

WHERE branch-name NOT IN (SELECT branch-name FROM Account)

(ii) Except:

SELECT branch-name

FROM Branch

EXCEPT

SELECT branch-name

FROM Account

**(d) Find the customer names who do not have any account in the ‘Region12’ branch.**

πcustomer−name(Customer) − πcustomer−name(σbranch−name=′Region12′ (Account))

(i) Except:

SELECT customer-name

FROM Branch

EXCEPT

SELECT customer-name

FROM Account

WHERE branch-name = 'Region12'

(ii) Not in:

SELECT DISTINCT branch-name

FROM Branch

WHERE branch-name NOT IN

(SELECT customer-name

FROM Account

WHERE branch-name = 'Region12')

**(e) Find the customer names who have accounts in all the branches located in ‘Los Angeles’.**

πcustomer−name(Customer)− πcustomer−name(πcustomer−name(Customer) × πbranch−name(σcity=′LosAngeles′ (Branch))− πcustomer−name,branch−name(Account))

(i) Except:

SELECT customer-name

FROM Customer

EXCEPT

SELECT customer-name

FROM Branch B, Customer C

WHERE B.city = 'Los Angeles' AND (C.customer-name, B.branch-name) NOT IN

(SELECT customer-name, branch-name

FROM Account)

(ii) Not In:

SELECT DISTINCT customer-name

FROM Customer

WHERE customer-name NOT IN

(SELECT customer-name

FROM Branch B, Customer C

WHERE B.city = 'Los Angeles' AND (C.customer-name, B.branch-name) NOT IN

(SELECT customer-name, branch-name

FROM Account) )

(iii) Count:

SELECT customer-name

FROM Account AS A, Branch AS B

WHERE A.branch-name=B.branch-name AND B.city = 'Los Angeles'

GROUP BY customer-name

HAVING count(DISTINCT B.branch-name) =

(SELECT count(DISTINCT branch-name)

FROM Branch

WHERE city='Los Angeles')

**(f) Find the customer names who have only one account.**

πcustomer−name(Customer)− πA.customer−name (σA.branch−name<>B.branch−name∨A.account−number<>B.account−number)∧A.customer−name=B.customer−name (ρA(Account) × ρB(Account)))

(i) Except:

SELECT customer-name

FROM Customer

EXCEPT

SELECT A.customer-name

FROM Account A, Account B

WHERE (A.branch-name <> B.branch-name OR A.account-number <> B.account-number) AND A.customer-name = B.customer-name

(ii) Not In:

SELECT DISTINCT customer-name

FROM Customer

WHERE customer-name NOT IN

(SELECT A.customer-name

FROM Account A, Account B

WHERE (A.branch-name <> B.branch-name OR A.account-number <> B.account-number) AND A.customer-name = B.customer-name)

(iii) Aggregate:

SELECT customer-name

FROM Account

GROUP BY customer-name

HAVING count(DISTINCT account-number)=1

4. Schema: *Student(sid, GPA)*

**Write a relational algebra that finds the ids of the students with the lowest GPA.**

πsid(Student) − πA.sid(σA.GP A>B.GP A∧A.sid<>B.sid(ρA(Student) × ρB(Student)))

(i) Except:

SELECT sid

FROM Student

EXCEPT

SELECT A.sid

FROM Student A, Student B

WHERE A.GPA > B.GPA AND A.sid <> B.sid

(ii) Not In:

SELECT DISTINCT sid

FROM Student

WHERE sid NOT IN

(SELECT A.sid

FROM Student A, Student B

WHERE A.GPA > B.GPA AND A.sid <> B.sid)

(iii) Aggregate:

SELECT sid

FROM Student

WHERE GPA =

(SELECT MIN(GPA)

FROM Student )

==================================================================

Homework 2:

1. Schema

*Employee(person-name, age, street, city)*

*Work(person-name, company-name, salary)*

*Company(company-name, city)*

*Manage(person-name, manager-name)*

**(a) Write a query in SQL to find the names of persons who work in one or more companies where they make a salary that is less than $22,000.**

SELECT person-name

FROM Work WHERE Work.salary

**(b) Write an SQL query to find the names of persons who work in one or more companies and make less than $22,000 in the majority (i.e., 50% or more) of the companies they work for.**

SELECT Total.person-name

FROM (SELECT person-name, count(\*) as cnt FROM Work GROUP BY person-name) as Total,

(SELECT person-name, count(\*) as cnt FROM Work

WHERE Work.salary= 0.5 \* Total.cnt

**2. (a) Find the name(s) of the employee(s) whose total salary is higher than those of all employees living in Barstow.**

**SELECT person-name**

FROM Work

GROUP BY person-name

HAVING SUM(salary)>ALL

(SELECT SUM(salary) FROM Work, Employee

WHERE Work.person-name=Employee.person-name AND city=’Barstow’

GROUP BY Work.person-name)

======

SELECT person-name

FROM Employee E

WHERE NOT EXISTS

(SELECT Work.person-name FROM Work, Employee

WHERE Work.person-name=Employee.person-name AND city=’Barstow’

GROUP BY Work.person-name

HAVING SUM(salary)>=

(SELECT SUM(salary) FROM Work W

WHERE W.person-name=E.person-name))

**(b) Find the name(s) of the manager(s) whose total salary is higher than that of at least one employee that they manage**

SELECT manager-name

FROM Manage M,

(SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S1

WHERE M.manager-name=S1.person-name AND S1.total-salary > SOME

(SELECT total-salary

FROM (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S2

WHERE S2.person-name = M.person-name)

======

SELECT manager-name FROM Manage M WHERE EXISTS (SELECT \* FROM (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S1, (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S2 WHERE M.manager-name=S1.person-name AND M.person-name = S2.person-name AND S1.total-salary > S2.total-salary)

3. Schema

*MovieStar(name, address, gender)*

*MovieExec(name, address, company, netWorth)*

**(a) We want to find the names and addresses of all female movie stars (gender = ’F’ in the MovieStar relation) who are also movie executives with a net worth over $2,000,000 (netWorth > 2000000 in the MovieExec relation).**

(i) Using INTERSECT: SELECT name, address FROM MovieStar WHERE gender=’F’ INTERSECT SELECT name, address FROM MovieExec WHERE netWorth>2000000

(ii) Without INTERSECT: SELECT name, address FROM MovieStar WHERE gender=’F’ AND (name, address) in (SELECT name, address FROM MovieExec WHERE netWorth>2000000)

**(b) We want to find the movie stars who are not movie executives.**

(i) Using EXCEPT: SELECT name FROM MovieStar EXCEPT SELECT name FROM MovieExec

(ii) Without EXCEPT: SELECT name FROM MovieStar WHERE name not in (SELECT name FROM MovieExec)

4. Schema: (NOTE: Computer product is either a desktop or laptop)

*ComputerProduct(manufacturer, model, price)*

*Desktop(model, speed, ram, hdd)*

*Laptop(model, speed, ram, hdd, weight)*

**(a) Find the average speed of all desktop computers.**

SELECT AVG(speed) FROM Desktop

**(b) Find the average price of all laptops with weight below 2kg.**

SELECT AVG(price) FROM ComputerProduct CP, Laptop L WHERE CP.model=L.model AND weight<=2

**(c) Find the average price of PC’s and laptops made by “Dell.”**

SELECT AVG(price) FROM ComputerProduct WHERE manufacturer=‘DELL’

**(d) For each different CPU speed, find the average price of a laptop.**

SELECT AVG(price) FROM Laptop GROUP BY speed

**(e) Find the manufacturers that make at least three different computer models.**

SELECT manufacturer FROM ComputerProduct GROUP BY manufacturer HAVING COUNT(model)>=3

**5. (a) Using two INSERT statements, insert a desktop computer manufactured by HP, with model number 1200, price $1000, speed 1.2Ghz, 256MB RAM, and an 80GB hard drive.**

INSERT INTO ComputerProduct VALUES (‘HP’, 1200, 1000); INSERT INTO Desktop VALUES (1200, ‘1.2GHz’, ‘256MB’, ‘80GB’);

**(b) Using two DELETE statements, delete all desktops manufactured by IBM with price below $1000.**

DELETE FROM Desktop WHERE model IN (SELECT model FROM ComputerProduct WHERE manufacturer = ‘IBM’ AND price < 1000);

DELETE FROM ComputerProduct WHERE manufacturer=‘IBM’ AND price

**(c) For each laptop made by Gateway, add one kilogram to the weight. (Hint: The WHERE clause in a**

UPDATE Laptop SET weight=weight+1

WHERE model IN (SELECT model FROM ComputerProduct WHERE manufacturer=‘Gateway’);

6. Schema: *Enroll(sid, dept, cnum, sec)*

**(a) Write an SQL query to find the students who are only enrolled in the CS classes offered this quarter.**

SELECT E1.sid

FROM Enroll AS E1

WHERE E1.sid NOT IN (SELECT Sid FROM Enroll WHERE dept<> 'CS')

**(b) Write an SQL query to find the students who are enrolled in all the CS classes offered this quarter.**

SELECT E0.sid /\* an enrolled student who is not missing any CS class \*/

FROM Enroll AS E0

WHERE E0.sid NOT IN /\* a E1.sid is a student who is missing some CS class \*/

(SELECT E1.sid FROM Enroll AS E1

WHERE (E1.Stid, E1.dept, E1.cnum)

NOT IN (SELECT E1.Stid, 'CS', E2.cnum

FROM Enroll AS E2 WHERE E2.dept='CS'))

**(c) Write the previous queries using different SQL constructs. In particular can you express those queries using the count aggregate? Please explain**

SELECT E1.sid

FROM Enroll AS E1

WHERE E1.dept ='CS'

GROUP BY E1.sid

HAVING count(\*) = (SELECT count(\*) FROM Enroll WHERE dept= 'CS')

7. Schema:

*Customer (customer-name, street, city)*

*Branch (branch-name, city)*

*Account (customer-name, branch-name, account-number)*

**(a) Find the names of all customers who have an account in a branch NOT located in the same city that they live in.**

SELECT C.customer\_name FROM Customer C

WHERE EXISTS (SELECT \*

FROM Account A, Branch B

WHERE C.customer\_name = A.customer\_name

AND B.branch\_name = A.branch\_name AND C.city != B.city)

**(b) Find the branches that do not have any accounts.**

SELECT branch\_name

FROM (SELECT B.branch\_name, count(account\_number) as numAccounts

FROM Branch B LEFT JOIN Account A ON A.branch\_name = B.branch\_name

Group By B.branch\_name) as BranchAccounts

WHERE BranchAccounts.numAccounts = 0

**(c) Find the customer names who do not have any account in the ‘Region12’ branch.**

SELECT R12\_AccountCounts.customer\_name FROM

(SELECT A.customer\_name, count(\*) AS numAccounts

FROM Account A LEFT JOIN

(SELECT \* FROM Account WHERE branch\_name = "Region12") AS R12\_Accounts

ON A.customer\_name = R12\_Accounts.customer\_name

GROUP BY A.customer\_name) AS R12\_AccountCounts

WHERE R12\_AccountCounts.numAccounts = 0

**(d) Find the customer names who have accounts in all the branches located in ‘Los Angeles’. You are not allowed to use the division operator directly for this question.**

SELECT customer\_name

FROM (SELECT customer\_name, count(\*) AS numAccounts

FROM (SELECT A.customer\_name, A.branch\_name

FROM Account A, Branch B

WHERE A.branch\_name = B.branch\_name and city = "Los Angeles" /\* group by to merge cases when one has multiple accounts in a branch \*/

GROUP BY A.customer\_name, A.branch\_name) AS T GROUP BY customer\_name) AS LA

WHERE LA.numAccounts = (SELECT count(\*) FROM Branch WHERE city="Los Angeles")

==================================================================

Homework 3:

1. We want to store the table created by the following SQL statement into a disk.

CREATE TABLE Class(

dept CHAR(2), cnum INTEGER, sec INTEGER,

unit INTEGER, year INTEGER, quarter INTEGER,

title CHAR(30), instructor CHAR(20) )

We need to store tuples for 1,000 classes that have been offered so far. 10 classes are offered every year. The tuples are stored in random order (i.e., they are not sequenced by any attribute). A disk of the following parameters is used for storing the table.

• 3 platters (6 surfaces) • 10,000 cylinders

• 500 sectors per track • 1024 bytes per sector

• 6,000 RPM rotational speed • 10ms average seek time

**(a) What is the capacity of this disk?**

6 surfaces/disk \* 10,000 tracks/surface \* 500 sectors/track \* 1KB/sector = 30GB/disk

**(b) What is the average time to read a random sector from the disk?**

(seek time) + (rotational delay) + (transfer time) = 10ms + 5ms + 0.02ms = 15.02ms

**(c) Assume one disk block corresponds to one disk sector. How many disk blocks are needed to store the above table with 1,000 tuples?**

72 blocks \* floor[(1024 bytes/block)/(1 tuple/72 bytes)] = 14 tuples/block

ceil[(1000 tuples/table)/(14 tuples/block)] = 72 blocks/table.

**(d) We want to run the following query by scanning the entire table.**

SELECT \* FROM Class WHERE year = 2005

Assuming that all blocks for the table is allocated sequentially, how long will it take to run the query? Assume that the disk head is not on the same track where the first block of the table is stored.

(seek time) + (rotational delay) + (transfer time) = 10ms + 5ms + 72\*0.02ms = 16.44ms

**(e) Now assume that due to frequent updates to the table, disk blocks are allocated such that, on average, sequentiality is broken every three blocks. That is, the table is stored in 24 randomly located “clusters” of 3 consecutive blocks. Assuming that we scan the entire table to execute the above query, how long will it take?**

24 \* ((seek time) + (rotational delay) + (transfer time)) = 24 \* (10ms + 5ms + 3\*0.02ms) = 361.44ms

**(f) Now assume that we have a B+tree on the year attribute and the tree has already been loaded into main memory. None of the disk blocks containing the Class table has been cached in main memory. What is the expected time to run the above query? Is it helpful to create a B+tree to run this query?**

150.2ms. Since the tuples are not clustered by the search key, we will need to do 10 random IOs to retrieve all 10 tuples. Therefore, 10\*(10ms+5ms+0.02ms) = 150.2ms. If all blocks are allocated sequentially, using the index may actually slow down the query execution.

2. Indexes:

The table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks) contains the grades for the courses completed by UCLA students during the last 20 years. If 10,000 new students enter UCLA every year, we can assume that in taken there are 200,000 different students, each identified by a StudentID. Thus will assume there are 40,000 students enrolled each quarter, and that each student takes four classes per quarter (160,000 classes taken by students each quarter) and that there are three quarters in each year (480,000 classes taken every year). Thus we get a total of 9,600,000 tuples recording the grades of all students over the last 20 years. Also assume that the average number of students per class is 100; this implies that 4,800 classes are o↵ered each year. On this table, we have a sparse index on StNo and a dense index on the combination: (CourseID,Year,Quarter,Sec,StNo). Both indexes are implemented as B+ trees where CourseNo, Year, Quarter, Sec, and StNo take 8 bytes each, and the pointers used in the B+ trees take 10 bytes.

**(a) If the file blocks have 4096 bytes and each tuple in taken requires 100 bytes, how many blocks will be needed to store the unspanned tuples of this relation ?**

floor(4096/100) = 40 unspanned tuples per block

ceil(9,600,000/40) = 240,000 blocks total

**(b) Compute the levels and the number of blocks at each level of the B+ tree, assuming a worst-case scenario**

Key: 8 bytes per field, with 5 fields = 40B.

block >= n\*ptr + (n – 1)\*key; 4096 >= 10n + 40n – 40, n = 82

N = floor[(4096-10)/50] + 1 = 82 (worst case: why floor not ceiling?)

N/2 = 41 pointers (worst case leaf and internal nodes)

Dense index: floor(9,600,000/41) = 234,146 at leaf level

floor(234,146/41) = 5,710 at first level

floor(5,710/41) = 139 at second level

floor(139/41) = 3 at third level (finally at root level)

**(c) How many blocks of B+ tree and file will the DBMS retrieve from disk to answer the following query: Find the average grade in a given class (e.g. find the average grade for: CS143, 2010, Fall, sec. 1). Assume the worst-case scenario, and that all the buffers are initially empty**

We can use the dense index because the search key is a prefix of the dense index key. 5 index blocks to the first leaf, then we need to retrieve leaf pointers for all 100 students. That’s up to three additional leaf blocks worst-case. Assuming that 100 students took that particular class, 100 more blocks will be accessed for actual data records, since this is not a a clustered index. Total: 8 B+ tree blocks, 100 file blocks.

**(d) We now want to compute the average grade over the (480,000 or so) classes taken in year 2011 (assume that they all have the same credit). Explain how the DBMS will go about searching and retrieving blocks from disk for this query, and estimate the number of blocks the system will have to fetch if those 200,000 students each took 48 classes on the average. Assume the worst-case scenario, and that all the buffers are initially empty.**

We will have to scan the whole file with the help of the sparse, and thus clustered, index on StNo.

floor[(4096-10)/18] = 227 (thus N for B+ tree is 228)

114 w.c. pointers. So, with 240,000 blocks, we see floor(240,000/114) = 2,105 blocks at leaf level. So there are 18 blocks at the next level, and then the root. The complete file is fetched by following the leftmost branch in the B+ tree and then the leaf nodes that are chained together. Thus we have to fetch 2 + 2,105 + 240,000 blocks.

3. Joins and Optimization:

In addition to the table taken whose schema and index have been described previously, we also have the table student(StNo,Level,FirstName,LastName,Major) describing our 200,000 students. This table has a primary B+ index on StNo which is the key for this relation and a foreign key for taken.There are five di↵erent levels: freshman, softmore, junior, senior, others

**(a) How many blocks will student use, if each tuple requires 100 bytes and each block contains 4096 bytes?**

200,000/40=5,000 blocks

**(b) For the query πStNo(σ=“others”(student)) ▹◃ taken estimate the size of the results (i.e., how many tuples). Also estimate the cost of implementing the query measured by the number if blocks read from disks. You can assume that the join takes advantage of the sparse index on taken.StNo (Also, for the sake of simplicity, assume that all indexes used are already in main memory). To compute these estimates use the textbook rules of relational queryoptimizers.**

Because of the FK the complete join has as many tuples as taken 9, 600,000 tuples. Others is one out of five, 9, 600,000/5= 1920,000. Computing the join using the index. 200,000/5=40,000 ”others” students. The courses taken by one student fit in two blocks. Thus we have to access 40000 ⇥ 2 blocks of taken plus the 5000 blocks of student

==================================================================

Sample Midterm:

1. Suppose that blocks can hold 100 search keys and 101 pointers. We built a dense index organized as a B+ tree on a file of 2 million records, where the records are placed in blocks that hold 10 records each. Say that the search key for the B+ tree is the candidate key for the relation. Answer in the worst-case scenario:

**(a) Compute the blocks used at each level of the B+ tree**

Candidate key: as many pointers as there are records (2 million pointers)

Bottom level: 2 million/50 = 40,000

Next level: floor(40,000/51) = 784 pointers

Next level: 784/51 = 15 blocks. Then the root

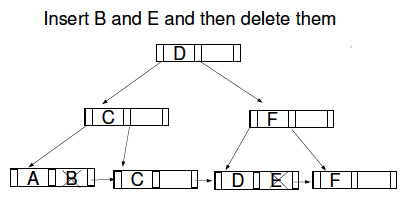
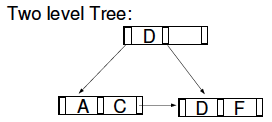
Total number of blocks = 40000 + 784 + 15 + 1

(i) If this was a sparse index, we only need 200,000 pointers. At bottom level = 4,000. Next level = 78. Then root

2. Assume that we use B+ trees of order n = 3 (where n is the max number of pointers in a node)

**(i) Draw a tree of height 2 with the following keys: A, C, D, F (lexicographically ordered)**

**(ii) Show how the insertion of two new keys, followed by their deletions, could change this 2-level tree into a 3-level tree with exactly the same keys (i.e. A, C, D, F)**



3. We store the following file in blocks having size 4096 bytes. Relation: *customer(id char(20), name char(24))*

Our relation has 2 millions of tuples, which are stored unspanned. We create sparse index on id organized as a B+ tree. Pointers take 10 bytes.

**(a) How many blocks are needed to store the whole relation?**

Since 4096/44 = 93.09 (we can store 93 unspanned tuples in each block)

Thus, ceil(2,000,000/93) = 21,506 blocks

**(b) What is the minimum number of nodes needed for the B+ tree?**

N – 1 = ceil[(4096-10)/(20+10)] = 136. Thus N = 137

Leaf Level: 137 – 1 = 136 pointers to the file. Sparse index pointing at 21,506 blocks. Since 21,506/136 = 158.13, 159 leaf nodes are used

First Level: N = 137 pointers used. Thus, ceil(159/137) = 2 blocks used

1 Root: Thus the total number of nodes in the best case is 1 + 2 + 159 = 162

**(c) What is the maximum number of nodes needed for the B+ tree, in the worst-case scenario?**

N is still 137

Leaf Level: ceil[(137+1)/2) = 69. Thus, 68 pointers to the file in each leaf node. Sparse index pointing to 21,506 blocks. Since 21,506/68 = 316.2, we conclude that 316 blocks are used at leaf level

First level: ceil(137/2) = 69 pointers in each node. Thus, floor(316/69) = 4 nodes needed

1 Root: Thus total nodes: 1 + 4 + 316 = 321

**(d) Using the worst-case B+ tree you just constructed, how many blocks must be retrieved**

**to execute the following query:**

**SELECT name FROM customer WHERE id = '7672945'**

3 blocks of the B+ tree and one from the file. Four all together.

3. Given the table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks):

**(a) Write a SQL query to find the students who have taken 4 or more classes and, in every class they took, got a grade that is equal to or lower than the average for that class—a class is identified by (CourseID, Year, Quarter, Sec) and Grade in taken is of type numeric.**

We are seeking students who never got a grade above the class average and took four or more classes:

select StNO from taken where

StNO not in ( select StNo

from taken as T1

where Grade > (select avg(Grade) from taken as T2

where T2.CourseID=T1.CourseID and

T2.Year= T1.Year and

T1.Quarter= T2.Quarter and

T2.Sec= T1.Sec))

group by SNO having count(\*) >= 4

5. Given the table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks):

**(a) write a relational algebra expression to compute all the students who got a grade above 3.0 in every class they took in 2014.**

πStNo(taken) − πStNo(σGrade<=3^Year=2014(taken))

6. Potpourri

**(a) Can the intersection of relations R(A,B) and S(A,B) be expressed using only natural joins?**

Yes, R ^ S = R >< S.

**(b) Can the intersection of relations R(A,B) and S(A,B) be expressed using the set difference operator?**

Yes, R ^ S = R − (R − S).

**(c) Can the intersection of relations R(A,B) and S(A,B) be expressed using the Cartesian product and projection operators?**

No

**(d) Can the intersection of relations R(A,B) and S(A,B) be expressed using the cartesian product, selection and projection operators?**

Yes, R ^ S = πR.A,R.B(σR.A=S.A^R.B=S.B(R x S)).

**(e) A relation R is indexed on its candidate key using an extendible hashing: Is this index dense, sparse or it could be either way?**

This must be a dense index. Sparse indexes only work if we can perform ordered searches as in B+ trees.

**(f) For the extendible hashing on R described in D5: does the structure of its directory and the number of the buckets it uses depend on the order in which the tuples in R have been inserted, or they only depend on the values those tuples?**

The overall structure depends only on the set of values. Different insertion orders might affect the order in which tuples are arranged in the buckets, but not the overall content of the buckets, nor the directory.

**7 (exc). Our RDBMS has compiled and optimized our relational query into a select-project-join expression consisting of 3 selections, 3 projections and 4 joins. Assume that the relations in our database are in main memory and each contains N tuples or less. Is the worst-case complexity of executing this query log(N), polynomial in N or exponential in N? Justify your answer**

The least efficient kind of join is the nested loop join that computes the four joins in time ((N x N) x N) x N = N^4. The resulting size is also less than N4 and selections are performed in linear time whereas projections by sorting are performed in time N4 x log(N^4), which is better than O(N^5). Thus in the worst case, the complexity is polynomial, and because of indexes and optimizer it will actually be better than O(N^5)

================================

Index-Sequential Access Method (ISAM):

- Static structure; not good when there are frequent updates

- Leaf pages are sequential (so fast access)

- Not suitable for updates (loses balance and becomes non-sequential due to overflow chain)

================================

B+ Trees:

- Dynamic; adjusts gracefully under insertions/deletions

- Balanced, all leaf nodes are on the same level

- Non-leaf node: Points to the nodes 1 level below

- No direct pointers to tuples

- Say, non-leaf node has 3 pointers, and values 23 and 56:

- Then, ptr1 points to nodes with keys k < 23

- Ptr2 points to nodes with keys 23 <= k < 56

- Ptr3 points to nodes with keys k >= 56

- At least half of the pointers used (i.e. ceil(n/2))

- Except root, where at least 2 pointers used

- Nodes are never too empty:

- Non-leaf node:

- Max pointers (n), min pointers (ceil(n/2))

- Max keys (n – 1), min keys ((ceil(n/2) – 1)

- Leaf node (always at least half full):

- Max pointers (n), min pointers (ceil(n+1)/2)

- Max keys (n – 1), min keys (ceil((n-1)/2))

- Root:

- Max pointers (n), min pointers (2

) - Max keys (n – 1), min keys (1)

Insertion:

- There are four cases to consider:

1. Simple case (no overflow)

2. Leaf overflow only

3. Non-leaf overflow

4. New root

- Simple Case:

1. Traverse down from the root to correct leaf node, and insert the proper search-key value, record pointer entry into an open slot in the leaf node

- Leaf overflow:

1. Traverse down to the correct leaf node, but there is no space to store the new value

2. “Leaf node splitting” => split the leaf into 2, and put the keys half and half (NOTE: Maintain search-key order)

- Ex: Leaf node to insert into contains values 50 and 60, and we want to insert 55 (but can only have 2 entries per node)

- Split the node into 2, and place 50 and 55 in 1 node, and 60 by itself in the other node

3. Copy the first key of the new node (i.e. 60) to the parent node

a. If there is no overflow in the parent node, then stop (otherwise, case 3)

- Non-leaf node overflow:

- Ex: Leaf node to insert (50, 55), parent node of that node (50, 60), and parent node to that parent node (70, null), and node to insert is 52

1. Traverse down to the leaf node (i.e. 50, 55)

2. Leaf node overflow, so split it, copy the key into the new node: left leaf node (50, 52), and right leaf node (55, null)

3. Copy the first key in the new node (i.e. 55) up the parent node: non-leaf node overflow (50, 55, 60)

4. Parent leaf overflow, so split it, and move up the key in the middle: Left parent node (50, null), right parent node (60, null)

a. Middle key (i.e. 55), moves up to that node’s parent node (i.e. (70, null)): so now (55, 70) (no overflow, so done)

- New root:

- Ex: Leaf node to insert into (20, 30), parent node/root node (50, 60), and key to insert is 25

1. Traverse to leaf node (i.e. (20, 30)), and insert 25 there, leading to node overflow: (20, 25, 30)

2. Split leaf node: left leaf node (20, 25), right left node (30, null)

3. Copy first key in the new node (i.e. 30) up to the parent node, leading to parent node overflow (30, 50, 60)

4. Split the parent node: left parent node (30, null), right parent node (60, null), and move up the middle key (i.e. 50)

5. As there is no node above, add new node, and place 50 in that node (i.e new root node)

- Summary:

- Leaf node overflow => the first key of the new node is copied to the parent

- Non-leaf node overflow => the middle key is moved up to the parent

====

Deletion:

- 5 Cases to consider:

1. Simple case (no overflow)

2. Leaf node, coalesce with neighbor

3. Leaf node, redistribute with neighbor

4. Non-leaf node, coalesce with neighbor

5. Non-leaf node, redistribute with neighbor

- Simple case (no overflow):

- Ex: Parent node a(20, 40, 60), children nodes b, c(20, 25, 30), d(40, 50, null), and e, and we want to delete 25

1. Delete the value in the leaf node, and check to make sure that we have a valid number of pointers (i.e. at least ceil(n/2) pointers)

- Originally: Started with 4 pointers (1. c1 pointing to the record of 20, 2. c2 pointing to the record of 25, 3. c3 pointing to record of 30, 4. cn pointing to sibling node)

- After deletion: Contains 3 pointers (and min is 3 pointers), so done

- Leaf node, coalesce with neighbor:

- Ex: parent node a(20, 40, 60), children nodes b, c(20, 30, null), d(40, 50, null), e, and we want to delete 50

1. Delete 50, and check number of pointers => now we have 2 pointers (3 min), so underflow

2. Try merging with a sibling, merge c and d (i.e. move everything from d into c) => c(20, 30, 40), and update pointers (i.e. cn pointer points to e now, and not d)

3. Once everything is moved, delete d

4. After leaf node merge, from the parent node, delete the pointer and key to the deleted node (i.e. delete 40), and check for underflow at a => 3 pointers (min 3), so done

- Leaf node, redistribute with neighbor:

- Ex: parent node a(20, 40, 60), children nodes b, c(20, 25, 30), d(40, 50, null), e(60, 70, 80), and we want to delete 50

1. Delete 50, and check for underflow => 2 pointers (min 3), so underflow

2. Check if d can be merged with siblings c or e, and if not, then redistribute the keys in d with a sibling

a. Say c (for example) => so redistribute c and d, so that nodes c and d are roughly “half full”

- Move 30 and its tuple pointer to d (i.e. c(20, 25, null) and d(30, 40, null))

3. Update the parent node (as the lowest value in d is now 30 (and not 40), replace 40 with no in the parent node) => parent node a(20, 30, 60)

a. Check for underflow => 4 pointers (min 3), so done

- Non-leaf node, coalesce with neighbor:

- Ex: a(50, 90, null), a’s children are b(30, null, null) and c(70, null, null), b’s children are d(10, 20, null), and e(30, 40, null), c’s children are f(50, 60, null), and g(70, 80, 90), and we want to delete 20

1. Delete 20, and check for underflow => d(10, null, null) => 2 pointers (min 3) => so underflow

a. Merge d with e (if you can) => you can, so move everything in e to d => e is empty, d(10, 30, 40)

2. From parent node, delete pointer and key to the deleted node (i.e. delete 30) => check for underflow => 2 pointers (min 2) => so underflow

a. Try to merge with its sibling => b(null, null, null) and c(70, null, null)

- IMPORTANT NOTE: When merging non-leaf nodes => always pull down the mid-key in the parent and place it in the merged node

- So, move pointers from c to b, pull down 50, and bring over 70 from c => a(90, null, null), b(50, 70, null), and delete c

- Non-leaf node, redistribute with neighbor:

- Ex: a(50, 99, null), a’s children are b(30, null, null) and c(70, 90, 97), b’s children are d(10, 20, null) and e(30, 40, null), and c’s children are f(50, 60, null), and g(70, 100, 110), and we want to delete 20

1. Delete 20, and check for underflow => d(10, null, null) => underflow

2. Merge d with e (i.e. bring stuff in e to d) => d(10, 30, 40)

a. Remove key and pointer to deleted node from parent (i.e. remove 30) => b(null, null, null)

3. Try to merge b with c => b has 1 pointer, c has 4 pointers = 5 pointers (max 4) => can’t merge, so try to redistribute

a. Redistribute b and c:

i. Temporarily, make left node b “overflow” by pulling down mid-key and moving everything from c to b => b(50, 70, 90, 97), c(null,…), and a(99, null, null)

ii. Apply overflow handling algorithm:

a. pick the mid-key (say 90) in the node and move it to the parent => b(50, 70, 97) and a(90, 99, null)

b. Move everything to the right of 90 to the empty node => b(50, 70, null) and c(97, null, null)

- Summary of deleting:

- For leaf node merging => delete the mid-key from the parent

- For non-leaf node merging/redistributing => pull down mid-key from their parent

- In practice, coalescing is often not implemented (as it is too difficult and not worth it)

===============================

Static Hashing:

- Disadvantage of sequential file organization => we must access an index structure to locate data (or must use binary search) => that results in more I/O ops

- File organizations based on “hashing” => allows us to avoid accessing an index structure

- Hashing provides a way of constructing indices

- “Bucket” => unit of storage that can store 1 or more records

- Bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block

- Let K denote the set of all search-key values, and B denote the set of all bucket addresses

- “Hash function” h => is a function from K to B

- To insert a record with search key K, compute h(Ki) => which gives us the address of the bucket for that record

- Then the record is stored in the bucket

- To perform a lookup on a search-key value Ki, compute h(Ki) => then search the bucket with that address

- To delete => if the search-key value of the record to delete is Ki, compute h(Ki), search bucket for the record, and delete it from the bucket

- Suppose that 2 search keys, K5 and K7, have the same hash value (i.e. h(K5) = h(K7))

- If we perform a lookup on K5, the bucket h(K5) contains records with search-key values K5 and records with search-key values K7

- This would require additionally looking at the search-key value of every record in the bucket to verify it is the record we want

- Hashing can be used for 2 different purposes:

1. “Hash file organization” => obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record

2. “Hash index organization” => organize the search keys, with their associated pointers, into a hash file structure

====

Hash Functions:

- Worst possible hash function maps all search-key values to the same bucket

- Such a function is undesirable because all records have to be kept in the same bucket

- Lookup has to examine every such record to find the desired record

- Ideal hash function distributes the stored keys uniformly across all buckets, so that every bucket has the same number of records

- As we don’t know at design time which search-key values will be stored => we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has the following qualities:

- Distribution is “uniform” => hash function assigns each bucket the same number of search-key values from the set of “all” possible search-key values

- Distribution is “random” => in the avg. case => each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key vals

- The hash function will not be correlated to any externally visible ordering on the search-key values (e.g. alphabetical or ordering based on length)

- Hashing appears random

- Assume that we decide to have 26 buckets, and we define a “naive” hash function that maps names beginning with the ith letter to the ith numbered bucket

- This hash function performs poorly, as we are likely to have more names starting with letters like B and R, than Q or X

- Assume that we want a hash function on the attribute salary, and that the min salary is 30,000 and max is 130,000, and our hash function divides the values into 10 ranges

- This hash function is uniform, but is not random (as we know that if a salary falls within a range, it will be within a certain bucket)

- As it is not random, it is also not uniform (as there are more common salaries than others)

- Typical hash functions perform computation on the internal binary machine representation of characters in the search key

- Simple hash function of this type first computes the sum of the binary representation of the key’s characters, and then returns the sum modulo the number of buckets

- Hash functions require careful design

- Bad hash functions may result in slow performance (due to longer lookup times, etc.)

====

Handling of Bucket Overflows:

- Thus far, we have assumed that when a record is inserted, the bucket to which it is mapped has space to store the record

- “Bucket overflow” => if the bucket doesn’t have enough space, and can occur for several reasons:

- Insufficient buckets => the number of buckets nB must be chosen such that nB > (nR / fR) [where nR is total number of records, and fR is the number of records per bucket]

- Skew => some buckets are assigned more records than others, so a bucket may overflow even when others still have space, and can occur for 2 reasons:

1. Multiple records may have the same search key

2. The chosen hash function may result in a nonuniform distribution of search keys

- To reduce the probability of bucket overflow:

- The number of buckets should be (nR / fR) \* (1 + d), where d is a “fudge factor” (typically around 0.2)

- Some space is wasted (about 20% of the space in the buckets will be empty), but the benefits is that bucket overflow is reduced

- However, bucket overflow can still occur

- Handle bucket overflow by using “overflow buckets”

- If a record must be inserted into a bucket b, and b is full, then the system provides an overflow bucket for b,and inserts it there

- If the overflow bucket is full, then another is made, and so on

- All of the overflow buckets are chained together via a linked list => “overflow chaining”

- The lookup algorithm must be changed to handle overflow chaining

- As before, the system uses the hash function on the search key to identify a bucket b

- The system must examine all records in bucket b; in addition to overflow buckets (if they are present for a given bucket)

- “Closed addressing (closed hashing)” => records that map to a given bucket only are placed in that bucket (or overflow buckets corresponding to that bucket)

- “Open addressing (open hashing) (linear probing)” => each bucket location has a fixed amount of record spots available, and if they fill up, then records mapped to a given bucket can be placed in other buckets by linear probing down the buckets

- Closed hashing is preferable for DB systems => as deletion under open hashing is troublesome

- But in DB systems, it is important to be able to handle deletion efficiently

- Disadvantages of closed hashing:

- The hash function must be chosen when we implement the system

- The size cannot be easily changed after deciding on one (i.e. fixed, closed number of buckets)

- As function h maps search-key values to a fixed set B of bucket addresses => we waste space if B is too large, and we are limited (and overflow occurs frequently) if B is small

- As the file grows, performance suffers as the buckets become filled up, and lookups and insertions have to traverse over more records to get the desired location

====

Hash Indices:

- Hashing can be used not only for file organization, but also for index-structure creation

- “Hash index” => organizes search keys, with their associated pointers, into a hash file structure

- Construct a hash index as follows:

- Apply hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or overflow bucket)

- We use the term “hash index” => to denote hash file structures as well as secnodary hash indices

- Hash indices are only secondary index structures

- Hash indexes are never needed as a clustering index structure => since, if a file is organized by hashing, there is no need for a separate hash index structure on it

=========

Dynamic Hashing:

- Most DBMSes grow larger over time

- If we are using static hashing for such a DB, there are 3 options:

1. Choose a hash function based on the current file size

- Will result in performance degradation as the DB grows

2. Choose hash function based on anticipated size of the file at some point in the future

- Performance degradation is avoided, but a significant amount of space may be wasted internally

3. Periodically reorganize the hash structure in response to file growth

- Such reorganization involveschoosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments

- Massive, time-consuming operation

- “Dynamic hashing” => techniques that allow the hash function to be modified dynamically to accommodate the growth/shrinkage of the DB

====

Data Structure:

- “Extendable hashing” => copes with changes in DB size by splitting and coalescing buckets as the DB grows and shrinks

- Thus, space efficiency is retained

- In addition, as the reorganization is performed only on 1 bucket at a time, the resulting performance overhead is acceptably low

- With extendable hashing, choose a hash function h with the desirable properties of uniformness and randomness

- But this hash function generates values over a relatively large range (b-bit binary integers) (typical b value is 32)

- We don’t create a bucket for each hash value (as 2^32 > 4 billion, and this many buckets is unreasonable for all but the largest DBs)

- Instead, create buckets on demand, as records are inserted into the file

- We don’t use the entire b bits of the hash value initially

- At any point, we use i bits, where 0 <= i <= b

- The value if i grows and shrinks with the size of the DB

- Although i bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket

- Thus, these entries will have a common hash prefix, but the length of the prefix may be less than i

- Thus, we associate with each bucket an integer giving the length of the common hash prefix

- The number of bucket-address-table entries that piont to bucket j is 2^(i - ij)

====

Queries and Updates:

- We now observe how to perform lookup, insertion, and deletion on an extendable hash structure

- Lookup:

- To locate a bucket containing search-key value Ka, the system takes the first i high-order bits of h(Ka), looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry

- Insert:

- The system follows the same procedure as lookup as before, ending in some bucket (say j)

- If there is room in the bucket, then the system inserts the record in the bucket

- Otherwise, it must split the bucket and redistribute the current records, plus the new one

- To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses

- If i = ij:

- Only 1 entry in the bucket address table points to bucket j

- Thus, the system needs to increase the size of the bucket address, so that it can include pointers to the 2 buckets that result from splitting bucket j

- It does so by considering an additional bit of the hash value, increments value of i by 1 => which doubles the size of the bucket address table

- It replaces each entry by 2 entries, both of which contain the same pointer as the original entry

- Now 2 entries in the bucket address table point to bucket j

- The system allocates a new bucket (call it z), and sets the 2nd entry to point to the new bucket (it sets ij and iz to i)

- Then, it rehashes each record in bucket j, and depending on the first i bits => either keeps it in j, or allocates it to the new bucket z

- If i > ij:

- More than 1 entry in the bucket address table points to bucket j

- Thus, the system can split bucket j without increasing the size of the bucket address table

- NOTE: That all the entries that point to bucket j correspond to hash prefixes that have the same value on the leftmost ij bits

- System allocates a new bucket (call it z), and sets ij and iz to the result of adding 1 to the original ij value

- The system needs to adjust the entries in the bucket address table that previously pointed to j

- The system leaves the first half of entries as they were (pointing to bucket j), and sets all remaining entries to point to the newly created bucket z

- Next, the system rehashes each record in bucket j, and allocates it either to bucket j or bucket z

- The system then reattempts the insert

- Deletion:

- The system follows the same lookup procedure to find the bucket (say j)

- It removes both the search key from the bucket and the record from the file

- The bucket is removed too (if it is empty)

- NOTE: At this point, several buckets can be coalesced, and the size of the bucket address table can be cut in half

====

Static Hashing vs. Dynamic Hashing:

- Extendable hashing:

- Advantages:

- Main advantage => performance doesn’t degrade as the file grows

- There is minimal space overhead

- Buckets can be allocated dynamically (and thus, no buckets need to be reserved for future growth)

- Disadvantages:

- Lookup involves an additional level of indirection (since the system must access the bucket address table before accessing the bucket itself)

=========

Comparison of Ordered Indexing and Hashing:

- We can organize files of records as ordered files by using index-sequential organization (or B+-Tree organizations)

- We can organize files by using hashing

- Alternatively, we can store them as heap files (where the records are not ordered in any particular way)

- Most DB systems support B+-Trees and may additionally support some form of hash file organization or hash indice

- DB designer must consider the following when deciding which implementation to go with:

- Is the cost of periodic organization of the index or hash organization acceptable?

- What is the relative frequency of insertion/deletion?

- Is it desirable to optimize average access time at the expense of increasing worst-case access time?

- What types of queries are users likely to pose?

- If queries are of the form: SELECT A1, A2, …, An FROM r WHERE Ai = c;

- The system will perform a lookup on an ordered index or hash structure for attribute Ai = c

- Thus => hashing is preferable

- Ordered index lookup requires time proportional to the log of the number of values in r for Ai

- In hash structure => the average lookup time is constant independent of the size of the DB

- Only disadvantage is that the worst-case lookup time is N (as there could be N items in a bucket) vs. log N for ordered index

- If queries are of the form: SELCET A1, A2, …, An FROM r WHERE Ai <= c2 AND Ai >= c1

- Ordered-index techniques are preferable to hashing

=========

Index Definition in SQL:

- The SQL standard doesn’t provide any way for the DB user/admin to control what indices are created and maintained by the DB

- Indices aren’t required for correctness, since they are redundant data structures

- They are important for performance

- Most SQL implementations provide the programmer contorl over creation and removal of indices via DDL commands

- CREATE INDEX => command to create an index

- Syntax: CREATE INDEX <index-name> ON <relation-name> (<attribute-list>);

- The attribute-list is the list of attributes of the relations that form the search key for the index

- Ex: CREATE INDEX dept\_index ON instructor (dept\_name) (i.e. dept\_name is the search key on index called dept\_index for relation instructor)

- CREATE UNIQUE INDEX => to create a search key that is a candidate key

- DROP INDEX <index-name> => to drop an index