// CS143: Database Systems

// November 5, 2017

// Midterm Review (Relational Algebra, SQL, Storage and File Structure, and Indexing/Hashing)

=========

TODO: TAKE NOTES ON THE FOLLOWING (SKIPPING FOR NOW TO SAVE TIME)

- SKIPPING VIEWS (4.2)

- SKIPPING TRANSACTIONS (4.3)

- SKIPPING INTEGRITY CONSTRAINTS (4.4)

- SKIPPING AUTHORIZATION (4.6)

- SKIPPING ALL CHAPTER 5 STUFF (5.1-5.1.1, F.1.3, 5.3,5.6)

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

Chapter 6: Relational Algebra:

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

- Types of Operations:

1. Project

2. Select

3. Rename

4. Cartesian Product

5. Natural Join

6. Theta Join

7. Set Operations

a. Union

b. Intersection

c. Difference

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

Chapter 3: SQL:

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

- Basic Types:

- char(n) => fixed-length character string with user-specified length n (NOTE: Can also use full form => character(n))

- varchar(n) => variable-length character string with user-specified “maximum” length n (NOTE: Can also use full form => character varying(n))

- int => integer (finite subset of integers that is machine-dependent) (NOTE: Can also use full form => integer)

- smallint => small integer

- numeric(p, d) => Fixed-point number with user-specified precision (Ex: numeric(3, 1) allows 44.5 to be stored exactly)

- real, double precision => Floating-point and double precision floating-point number

- float(n) => floating-point number, with precision of at least n digits

=========

Basic Schema Definition:

- CREATE TABLE => command to create table

- Provide with the each of the attributes and their domains

- Can also specify a primary key, foreign key, and any other referential integrity constraints

- NOTE: If you declare an attribute as “NOT NULL” => then creating a tuple must have a user-specified value for the given attribute

- INSERT INTO => command to insert tuples into a relation

- Syntax: INSERT INTO <table\_name> VALUES <attributes>

- DELETE FROM => command to delete tuples from a relation

- Syntax: DELETE FROM <table\_name>

- DROP TABLE => command to delete all tuples from a relation and delete the relation

- ALTER TABLE => command to add/drop attributes from a relation

- Syntax: ALTER TABLE <table\_name> ADD <attribute\_name> <domain\_type>

- Syntax: ALTER TABLE <table\_name> DROP <attribute\_name> <domain\_type>

=========

Basic Structure of SQL Queries:

1. SELECT => clause used to list the attributes desired in the result of a query

- Can contain keywords (i.e. DISTINCT, ALL)

- Can also contain arithmetic expressions (i.e. +, -, \*, /)

2. FROM => clause used to list the relations to be accessed in the evaluation of a query

- Contains groupings of tuples together (i.e. Cartesian Product, types of Joins)

- If you list relations separated by commas => Cartesian product (i.e. combines tuples of 2 relations that are unrelated to each other)

3. WHERE => clause used as a predicate involving attributes of the relation in the FROM clause

- Can contain “logical connectives” (i.e. AND, OR, NOT)

- Can also contain comparison operators (i.e. <, <=, >, >=, =, <>)

- Can be used to compare strings and arithmetic expressions

4 (Optional). ORDER BY => clause causing the tuples in the result of a query to be in some sorted order form

5 (Optional). GROUP BY => clause that is used to group sets of tuples together

6 (Optional). HAVING => clause that is used as a predicate that specifies conditions applying to groups (rather than individual tuples)

7 (Optional). WITH => clause that provides a way of defining a temporary relation whose definition is available only to the query in which the WITH clause occurs

=========

SQL Keywords:

- DISTINCT => used to force elimination of duplicates

- NOTE: In relational algebra, by default, duplicates are removed

- NOTE: In SQL, removing duplicates can be a time-consuming product; and thus, by default, duplicates are retained (unless explicitly removed with the DISTINCT keyword)

- Used in the SELECT clause

- ALL => used to explicitly retain duplicates

- NOTE: There is no need to use ALL as by default, duplicates are retained

- NATURAL JOIN => Cartesian Product that combines relations based on common values for attributes with the same names

- Considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations

- NOTE: Joins take place in the FROM clause

- JOIN … USING => feature of the join operation that allows one to natural join two relations based on specified attributes

- Syntax: FROM <relation 1> JOIN <relation 2> USING (<attribute list>)

- Way to avoid erroneously equating attributes by specifying exactly which attributes should be equated

- RENAME => way of renaming attributes of a result relation

- Syntax: <old\_name> AS <new\_name>

- NOTE: The AS clause can appear both in the SELECT and FROM clauses

- Reasons to use RENAME operation?

1. Two relations in the FROM clause may have attributes with the same name, in which case an attribute is duplicated in the result

2. If we use an arithmetic expression in the SELECT clause, then the resultant attribute doesn’t have a name

3. Even if an attribute name can be derived from the base relations, we may way to change the attribute name in the result

4. For renaming relations (i.e. to replace a long relation name with a shorter one)

5. If we wish to compare tuples in the same relation (i.e. need to take the Cartesian Product of a relation with itself, and without renaming => this would be impossible)

- “Correlation name (or table alias) (or correlation variable)” => identifier that is used to rename a relation

=========

String Operations:

- SQL specifies strings by enclosing them in single quotes

- A single quote character that is part of a string can be specified by using 2 single quote characters

- Ex: “It’s right” => “It’’s right”

- By default, equality operation on strings is case-sensitive (i.e. “comp sci” <> “COMP Sci”

- NOTE: In some database systems (e.g. MySQL and SQL Server) => they don’t distinguish uppercase from lowercase when matching strings

- String Functions:

- Concatenation => using the “||”

- Extracting substrings

- Finding the length of strings

- Converting strings to uppercase => using the UPPER(s) function

- Converting strings to lowercase => using the LOWER(s) function

- Removing spaces at the end => using the TRIM(s) function

- Pattern Matching:

- Percent => % character matches any substring

- Underscore => \_ matches any character

- Examples:

- ‘Intro%’ => matches any string beginning with “Intro”

- ‘%Comp%’ => matches any string containing “Comp” as a substring

- ‘\_\_\_’ => matches any string of exactly 3 characters

- ‘\_\_\_%’ => matches any string at least 3 characters long

- LIKE => keyword used for pattern matching in SQL queries

- ESCAPE => keyword that is used to escape characters for a LIKE comparison

- Examples:

- LIKE ‘ab\%cd%’ ESCAPE ‘\’ => matches all strings beginning with “ab%cd”

- LIKE ‘ab\\cd%’ ESCAPE ‘\’ => matches all strings beginning with “ab\cd”

- NOT => can search for mismatches instead of matches (in the LIKE operator)

- SIMILAR TO => SQL:1999 operation, which is more powerful pattern matching than the LIKE operation

=========

More SQL Keywords (Attribute Specification, Ordering, More Comparisons):

- “\*” => used in the SELECT clause to denote “all attributes”

- Ex: SELECT instructor.\* (i.e. select all attributes of the instructor relation)

- Ex: SELECT \* => indicates all attributes of the result relation in the FROM clause should be selected

- ORDER BY => clause that causes ordering of tuples

- ASC => sort by ascending order

- DESC => sort by descending order

- Performing ordering on multiple attributes:

- Can order based on the first attribute specified, and the second will do another ordering if the attributes in the first column have identical values

- Ex: SELECT \* FROM instructor ORDER BY salary DESC, name ASC;

- Thus, if there are instructors with the same salary, we order them in ascending order by name

- BETWEEN => comparison operator to do the x <= a <= y comparison

- Ex: SELECT name FROM instructor WHERE salary between 90000 AND 100000; (i.e. 90000 <= salary <= 100000)

- NOT BETWEEN => comparison operator to do the a > x AND a < y comparison

- Comparison operators on multiple tuples:

- Ex: WHERE (instructor.ID, dept\_name) = (teaches.ID, ‘Biology’) (instead of WHERE instructor.ID = teaches.ID AND dept\_name = ‘Biology’)

=========

More SQL Keywords (Set Operations):

- UNION => to find the set of all tuples satisfying relation A, relation B, or both

- INTERSECT => to find the set of all tuples satisfying relation A and relation B

- EXCEPT => to find the set difference (i.e. all tuples satisfying A that do not appear in B)

- NOTE: By default, set operations remove duplicates

- Can do UNION/INTERSECT/EXCEPT followed by the ALL keyword to retain duplicates

=========

Null Values:

- Null values can present special problems in relation operations (e.g. arithmetic, comparison, and set)

- Arithmetic Expression: (involving +, -, \*, /) is null if any of the input values is null

- Ex: r.A + 5 (where r.A is null => then the expression is null)

- Comparison Expressions:

- Can result in 3 possible logical values:

1. True, 2. False, 3. Unknown

- Results for the 3 boolean operations:

1. AND:

a. true AND unknown => unknown

b. false AND unknown => false

c. unknown AND unknown => unknown

2. OR:

a. true OR unknown => true

b. false OR unknown => unknown

c. unknown OR unknown => unknown

3. NOT:

a. NOT unknown => unknown

- If the WHERE clause predicate evaluates to either FALSE or UNKNOWN for a tuple => the tuple is not added to the result

- IS NULL => special keyword in a predicate to test for a null value

- Ex: SELECT name FROM instructor WHERE salary IS NULL (i.e. find all instructor relation where they have null values for the salary attribute)

- IS NOT NULL => the reverse of IS NULL

- IS UNKNOWN => to test if the result of a comparison is unknown rather than true or false

- When using the SELECT DISTINCT clause:

- Duplicate tuples must be eliminated

- Thus => when comparing values of corresponding attributes of 2 tuples => the values are treated as identical if either both are non-null and equal in value, or both are null

- Ex: {(‘A’, null), (‘A’, null)} => are treated as identical tuples

=========

Aggregate Functions:

- Functions that take a collection (set or multiset) of values as input and returns a single value

- SQL offers 5 built-in aggregate functions:

1. AVG => returns the average

2. MIN => returns the min

3. MAX => returns the max

4. SUM => returns the total sum of a set of input values provided

5. COUNT => returns the count of the number of elements in the set

- NOTE: The input to SUM and AVG must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types (e.g. strings)

- Ex: SELECT AVG(salary) FROM instructor WHERE dept\_name = ‘Comp. Sci’ (i.e. returns the average salary of CS instructors)

- NOTE: SQL may give an arbitrary name for the resulting relation attribute (thus, you can provide your own using the rename operator: AS)

- Duplicates:

- For taking an average => duplicates are important

- For counting the number of unique elements => duplicates is bad

- Can use DISTINCT keyword to remove them

- Ex: “Find the total number of instructors who teach a course in the Spring 2010 quarter”

SELECT COUNT (DISTINCT ID) FROM teaches WHERE semester = ‘Spring’ AND year = 2010 (i.e. if an instructor teaches multiple courses in a quarter, he will be counted once)

- Ex: COUNT(\*) => counting the number of tuples in a relation

- NOTE: Can’t use DISTINCT keyword with COUNT(\*)

- NOTE: Can use DISTINCT with MAX and MIN (even though it doesn’t change the result)

- NOTE: Can use ALL to retain duplicates explicitly

=========

Aggregation with Grouping:

- GROUP BY => when we want to apply aggregate functions to a group of set of tuples (rather than just individual tuples)

- Ex: “Find the average salary in each department”

SELECT dept\_name, AVG(salary) AS avg\_salary FROM instructor GROUP BY dept\_name

- Ex: “Find the number of instructors in each department who teach a course in Spring 2010”

SELECT dept\_name, COUNT(DISTINCT ID) AS instr\_count FROM instructor NATURAL JOIN teaches WHERE semester = ‘Spring’ AND year = 2010 GROUP BY dept\_name

- NOTE: When the GROUP BY clause is omitted => it is like taking the entire relation as one group

- RULE: When using grouping, ensure that the only attributes that appear in the SELECT statement without being aggregated are those present in the GROUP BY clause

- Any attributes not present in the GROUP BY clause must appear only inside an aggregate function if it appears in the SELECT clause (otherwise error)

- Error Example: SELECT dept\_name, ID, AVG(salary) FROM instructor GROUP BY dept\_name (i.e. ID is neither in the GROUP BY clause nor the aggregate function, so error)

=========

Having Clause:

- HAVING => when you want to state a condition that applies to groups rather than just to tuples

- Ex: “Find the dept\_name and average salary of departments that have average salary > 42000)

SELECT dept\_name, AVG(salary) AS avg\_salary FROM instructor GROUP BY dept\_name HAVING AVG(salary) > 42000

- Order of operations (for queries containing aggregation, GROUP BY, or HAVING clauses):

1. FROM clause is evaluated first to get a relation

2. If WHERE clause is present => the predicate in the WHERE clause is applied on the result relation of the FROM clause

3. Tuples satisfying the WHERE predicate are then placed into groups by the GROUP BY clause if present

- If absent, then the entire set of tuples satisfying the WHERE clause is treated as being 1 group

4. The HAVING clause if present is applied to each group; the group that do not satisfy it are removed

5. The SELECT clause uses the remaining groups to generate tuples of the result of the query (applying aggregate functions to get a single result tuple for each group)

- Ex: “For each course section offered in 2009, find the average total credits (tot\_cred) of all students enrolled in the section, if the section had at least 2 students”

SELECT course\_id, semester, year, sec\_id, AVG(tot\_cred) FROM takes NATURAL JOIN student WHERE year = 2009 GROUP BY course\_id, semester, year, sec\_id HAVING COUNT(ID) >= 2

=========

Aggregation with Null and Boolean Values:

- In general => aggregate functions treat nulls according to the following rule: aggregate functions (except COUNT(\*)) ignore null values in their input collection

=========

Nested Subqueries:

- SQL provides a way to nest subqueries

- “Subquery” => SELECT-FROM-WHERE expression that is nested within another query

- Common use => to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting in the WHERE clause

- Set membership => testing tuples for membership in a relation

- IN => connective tests for set membership, where the set is a collection produced by the SELECT clause

- NOT IN => connective testing for absence of set membership

- Ex: “Find all courses taught in both Fall 2009 and Spring 2010’

1. Using set intersection

(SELECT course\_id FROM section WHERE semester = ‘Fall’ AND year = 2009) INTERSECT (SELECT course\_id FROM section WHERE semester = ‘Spring’ AND year = 2010)

2. Set Membership

(SELECT DISTINCT course\_id FROM section WHERE semester = ‘Fall’ AND year = 2009 AND course\_id IN (SELECT course\_id FROM section WHERE semester = ‘Spring’ AND year = 2010)

- Can use NOT IN to serve as set exception (i.e. set difference)

- Ex: “Find all courses taught in Fall 2009 but not in Spring 2010”

1. Using set difference

(SELECT course\_id FROM section WHERE semester = ‘Fall’ AND year = 2009) EXCEPT (SELECT course\_id FROM section WHERE semester = ‘Spring’ AND year = 2010)

2. Set Membership

(SELECT DISTINCT course\_id FROM section WHERE semester = ‘Fall’ AND year = 2009 AND course\_id NOT IN (SELECT course\_id FROM section WHERE semester = ‘Spring’ AND year = 2010)

- Can also use IN and NOT IN on enumerated lists:

- Ex: Membership in a 1-attribute relation

“Select the name of instructors not named “Mozart” nor “Einstein””

SELECT DISTINCT name FROM instructor WHERE name NOT IN (‘Mozart’, ‘Einstein’)

- Ex: Membership in an arbitrary relation

“Find the total number of (distinct) students who have taken course sections taught by instructor with ID 110011”

SELECT COUNT(DISTINCT ID) FROM takes WHERE (course\_id, sec\_id, semester, year) IN (SELECT course\_id, sec\_id, semester year FROM teaches WHERE teaches.ID = 110011)

- Set Comparison => to compare sets using nested queries

- Ex: “Find the name of all instructors whose salary is greater than at least one instructor in the Biology department”

1. Using renaming

SELECT DISTINCT T.name FROM instructor AS t, instructor AS S WHERE T.salary > S.salary AND S.dept\_name = ‘Biology’

2. Set Comparison using the “greater than at least one” phrase => represented by > SOME

SELECT name FROM instructor WHERE salary > SOME (SELECT salary FROM instructor WHERE dept\_name = ‘Biology’)

- Types of set comparison:

1. “Greater/less than(or equal) to at least 1) => > SOME, >= SOME, < SOME, <= SOME, = SOME, <> SOME

- NOTE: = SOME is identical to IN

- NOTE: <> SOME is not the same as NOT IN

2. “Greater/less than(or equal) to all others) => > ALL, >= ALL, < ALL, <= ALL, = ALL, <> ALL

- NOTE: = ALL is not the same as IN

- NOTE: <> ALL is identical to NOT IN

- Ex: “Find the departments that have the highest average salary”

SELECT dept\_name FROM instructor GROUP BY dept\_name HAVING AVG(salary) >= ALL (SELECT AVG(salary) FROM instructor GROUP BY dept\_name)

- Start by writing a query to find the average salary of each department, and then nest it

- Testing for empty relations => testing whether a subquery has any tuples in its result

- EXISTS => construct returns true if the argument subquery is nonempty

- Ex: “Find all courses taught in both the Fall 2009 and Spring 2010 semesters”

SELECT course\_id FROM section AS S WHERE semester = ‘Fall’ AND year = 2009 AND EXISTS (SELECT \* FROM section AS T WHERE semester = ‘Spring’ AND year = 2010 AND S.course\_id = T.course\_id)

- Correlated subquery => subquery that uses a correlation name from an outer query

- Scoping rules of subqueries:

- Can only use correlation names defined in the subquery itself or in any query that contains the subquery

- If a correlation name is defined both locally and in an outer query, then the local one is applied

- NOT EXISTS => construct to test for the nonexistence of tuples in a subquery

- Can write “relation A contains relation B” as “NOT EXISTS (B EXCEPT A)”

- Ex: “Find all students who have taken all courses offered in the Biology department”

SELECT S.ID, S.name FROM student AS S WHERE NOT EXISTS ((SELECT course\_id FROM course WHERE dept\_name = ‘Biology’) EXCEPT (SELECT T.course\_id FROM takes AS T WHERE S.ID = T.ID))

- The outer SELECT takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department

- Test for the absence of duplicate tuples:

- UNIQUE => construct returns true if the argument subquery contains no duplicate tuples

- Ex: “Find all courses that were offered at most once in 2009”

1. UNIQUE construct to test for absence of duplicates

SELECT T.course\_id FROM course AS T WHERE UNIQUE (SELECT R.course\_id FROM section AS R WHERE T.course\_id = R.course\_id AND R.year = 2009)

2. Use of count comparison

SELECT T.course\_id FROM course AS T WHERE 1 >= (SELECT COUNT(R.course\_id) FROM section AS R WHERE T.course\_id = R.course\_id AND R.year = 2009)

- NOT UNIQUE => construct to test for the existence of duplicate tuples in a subquery

- Ex: “Find all courses offered at least twice in 2009”

SELECT T.course\_id FROM course AS T WHERE NOT UNIQUE (SELECT R.course\_id FROM section AS R WHERE T.course\_id = R.course\_id AND R.year = 2009)

- Formally => the UNIQUE test on a relation fails iff the relation consists of 2 tuples t1 and t2 such that t1 = t2

- Subqueries in the FROM clause:

- Key Concept: SELECT-FROM-WHERE expression returns a relation as a result; and therefore, can be inserted into another SELECT-FROM-WHERE anywhere that a relation can appear

- Ex: “Find the average instructors’ salaries of those departments where the average salary is greater than $42,000”

SELECT dept\_name, avg\_salary FROM (SELECT dept\_name, AVG(salary) AS avg\_salary FROM instructor GROUP BY dept\_name) WHERE avg\_salary > 42000

- We can give the subquery result relation a name, and rename the attributes using the AS clause

SELECT dept\_name, avg\_salary FROM (SELECT dept\_name, AVG(salary) AS avg\_salary FROM instructor GROUP BY dept\_name) AS dept\_avg(dept\_name, avg\_salary) WHERE avg\_salary > 42000

- Ex: “Find the max across all departments of the total salary at each department”

SELECT MAX(tot\_salary) FROM (SELECT dept\_name, SUM(salary) FROM instructor GROUP BY dept\_name) AS dept\_total(dept\_name, tot\_salary)

- NOTE: Nested subqueries in the FROM clause can’t use correlation variables from other relations in the FROM clause

- Can prefix a subquery in the FROM clause with the LATERAL keyword => to access attributes of preceding tables or subqueries in the FROM clause

- Ex: “Print the names of each instructor, along with their salary and average salary of their department”

SELECT name, salary, avg\_salary FROM instructor I1, LATERAL (SELECT avg(salary) as avg\_salary FROM instructor I2 WHERE I2.dept\_name = I1.dept\_name)

- WITH clause => provides a way of defining temporary relation whose definition is only available only to the query in which the WITH clause occurs

- Ex: “Find departments with the max budget”

WITH max\_budget(value) AS (SELECT MAX(budget) FROM department) SELECT budget FROM department, max\_budget WHERE department.budget = max\_budget.value

- WITH clauses defines temp relation max\_budget with one attribute value that is immediately used in the following query

- Ex: “Find all departments where the total salary is greater than the average of the total salary at all departments”

WITH dept\_total(dept\_name, value) AS (SELECT dept\_name, SUM(salary) FROM instructor GROUP BY dept\_name), dept\_total\_avg(value) AS (SELECT AVG(value) FROM dept\_total) SELECT dept\_name FROM dept\_total, dept\_total\_avg WHERE dept\_total.value > dept\_total\_avg.value

- Scalar subqueries => allow subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only 1 tuple containing a single attribute

- Ex: “List all departments along with the number of instructors in each department”

SELECT dept\_name, (SELECT COUNT(\*) FROM instructor WHERE department.dept\_name = instructor.dept\_name) AS num\_instructors FROM department

=========

Modification of the Database:

- Adding, deleting, or changing info within a database

Deletion:

- Can delete only whole tuples; we can’t delete values on only particular attributes

- Syntax: DELETE FROM <table\_name> WHERE <predicate>

- The WHERE clause can be omitted, in which case all tuples in the relation are deleted

- NOTE: DELETE command operates on only 1 relation

- If we want to delete tuples from several relations => we must use 1 DELETE command for each relation

- The WHERE clause predicate may be as complex as a SELECT command’s WHERE clause

- Ex: “Delete all tuples in the instructor relation pertaining to instructors in the Finance department”

DELETE FROM instructor WHERE dept\_name = ‘Finance’

- Ex: “Delete all instructors with a salary between $13,000 and $15,000”

DELETE FROM instructor WHERE salary BETWEEN 13000 AND 15000

- Ex: “Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building”

DELETE FROM instructor WHERE dept\_name IN (SELECT dept\_name FROM department WHERE building = ‘Watson’)

- NOTE: While we can only delete tuples from 1 relation at a time, we can reference any number of relations in a SELECT-FROM-WHERE nested in the WHERE clause of a DELETE

- NOTE: We can have a nested SELECT that references the relation from which tuples are to be deleted

- Ex: “Delete instructors with salary below the average at the university”

DELETE FROM instructor WHERE salary < (SELECT AVG(salary) FROM instructor)

- This example illustrates that order matters before performing any deletions

- If you delete any instructor tuples before taking the average salary, then the average would be incorrect

Insertion:

- To insert data into a relation:

1. Specify a tuple to be inserted

2. Write a query whose result is a set of tuples to be inserted

- NOTE: Tuples inserted must have the correct number of attributes and be of the correct domain type

- Inserting one tuple:

- Ex: INSERT INTO course VALUES (‘CS-437’, ‘DB Systems’, ‘CS’, 4);

- If you don’t remember the order of the attributes, then SQL allows the attributes to be specified as part of the INSERT statement:

- Ex: INSERT INTO course (course\_id, title, dept\_name, credits) VALUES (‘CS-437’, ‘DBMS’, ‘CS’, 4);

- Ex: INSERT INTO course (title, course\_id, credits, dept\_name) VALUES (‘DBMS’, ‘CS-437’, 4, ‘CS’);

- Inserting tuples based off of a query result:

- Ex: “Make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department with an $18,000 salary”

INSERT INTO instructor SELECT ID, name, dept\_name, 18000 FROM student WHERE dept\_name = ‘Music’ AND tot\_cred > 144

- Can insert tuples that have null value for some attributes:

- Ex: INSERT INTO student VALUES (‘3003’, ‘Green’, ‘Finance’, null)

Updates:

- Sometimes, we wish to change a value in a tuple without changing “all” values in the tuple

- Ex: “Annual salary increases where salary of all instructors is increased by 5%”

UPDATE instructor SET salary = salary \* 1.05

- Can have a where clause as well:

- Ex: UPDATE instructor SET salary = salary \* 1.05 WHERE salary < 70000

- NOTE: SQL evaluates whether to insert into the relation based on the WHERE clause before doing any inserts

- Ex: “Instructors with salaries over $100,000 receive a 3% raise, whereas all others receive a 5% raise”

1. Write two separate INSERT statements (but NOTE: that order is important; otherwise, some people might get an 8% raise)

UPDATE instructor SET salary = salary \* 1.03 WHERE salary < 100000;

UPDATE instructor SET salary = salary \* 1.05 WHERE salary <= 100000;

2. Utilizing SQL’s CASE construct (i.e. switch statement) to perform multiple updates in 1 update statement

UPDATE instructor

SET salary = CASE

WHEN salary <= 100000 THEN salary \* 1.05

ELSE salary \* 1.03

END

- UPDATE with CASE Syntax:

CASE

WHEN pred1 THEN result1

WHEN pred2 THEN result2

ELSE resultn

END

- Scalar subqueries used in the SET clause

- Ex: “Update tot\_cred for each student tuple to the sum of the credits of courses successfully completed by the student (where succeeding is getting a grade not ‘F’ and not null)

UPDATE student S

SET tot\_cred = (

SELECT CASE

WHEN SUM(credits) IS NOT NULL THEN SUM(credits)

ELSE 0

END

FROM takes NATURAL JOIN course WHERE S.ID = takes.ID AND takes.grade <> ‘F’ AND takes.grade IS NOT NULL)

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

Chapter 4: Intermediate SQL:

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

Join Expressions:

- SQL provides several forms of the join operation (i.e. natural join, explicit join predicate, etc.)

- NATURAL JOIN

- JOIN … USING => a form of the natural join that only requires values to match on specified attributes

- ON => condition that allows a general predicate over the relations being joined

- Predicate is written like a WHERE clause predicate except for the use of the keyword ON rather than WHERE

- NOTE: Like the USING keyword, the ON keyword appears at the end of the join expression

- Ex: SELECT \* FROM student JOIN takes ON student.ID = takes.ID;

- The ON condition above specifies that a tuple from student matches a tuple from takes if their ID values are equal

- In this case, this JOIN is almost identical to a natural join

- Only difference: This result keeps both the ID attributes (once for student, and once for takes) even thought they have the same value

- As we have seen before, the relation name is used to disambiguate the attribute name ID; and thus, the two occurrences can be referred to as student.ID and takes.ID

- Can express the same query that displays the ID attribute only once:

SELECT student.ID AS ID, name, dept\_name, tot\_cred, course\_id, sec\_id, semester, year, grade FROM student JOIN takes ON student.ID = takes.ID

- The ON condition can express any SQL predicate => thus, it can express a richer class of join conditions than NATURAL JOIN

- However, a query using a join expression with an ON condition can be replaced by an equivalent expression without it

- Thus, it appears that the ON condition is a redundant feature in SQL

- However, there are two good reasons for introducing the ON condition:

1. Outer join => ON condition behaves differently than a WHERE

2. SQL queries are more readable by humans if the join condition is specified in the ON clause, and the rest of the conditions are specified in the WHERE clause

Outer Joins:

- Suppose that we wish to display a list of all students, displaying their ID, name, dept\_name, and tot\_cred, along with the courses they have taken

- This might appear to work => SELECT \* FROM student NATURAL JOIN takes;

- However, in the case that a student has taken no classes, they wouldn’t appear with this natural join

- In other words, some tuples in either or both of the relations being joined may be “lost” in this way

- OUTER JOIN => operation that works in a manner similar to the join operations previously used, but it preserves those tuples that would be lost in a join, by creating tuples in the result containing null values

- For example, to ensure a student who has taken no courses appears in the resulting relation => a tuple could be added to the join result with all attributes from the student relation set to the corresponding values of the student, and all remaining attributes from the takes relation can be set to null

- Three types of outer joins:

1. LEFT OUTER JOIN => preserves tuples only in the relation before the LEFT OUTER JOIN op

2. RIGHT OUTER JOIN => preserves tuples only in the relation after the RIGHT OUT JOIN op

3. FULL OUTER JOIN => preserves tuples in both relations

- INNER JOIN => doesn’t preserve tuples (i.e. NATURAL JOIN and JOIN … USING ops)

- LEFT OUTER JOIN Procedure:

1. Compute the result of the inner join per usual

2. For every tuple t in the LHS relation that doesn’t match any tuple in the RHS relation in the inner relation, add a tuple r to the result of the join with the following attr:

a. The attributes of the tuple r that are derived from the LHS relation are filled in with the values from tuple t

b. The remaining attributes of r are filled with null values

- Ex: “Find all students who have not taken a course”

SELECT ID FROM student NATURAL LEFT OUTER JOIN takes WHERE course\_id IS null

- FULL OUTER JOIN => a combination of left and right outer-join types

- After the op computes the result of the inner join, it extends with nulls those tuples from the LHS relation that didn’t match any from the RHS relation, and adds them

- It does the same for the RHS relation

- Ex: “Display a list of all students in the CS department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the CS department has taken the course section”

SELECT \* FROM (SELECT \* FROM student WHERE dept\_name = ‘CS’) NATURAL FULL OUTER JOIN (SELECT \* FROM takes WHERE semester = ‘Spring’ AND year = 2009)

- The ON clause can be used with outer joins

- The following examples is equivalent to “student NATURAL LEFT OUTER JOIN takes”, except ID appears twice in the result

SELECT \* FROM student LEFT OUTER JOIN takes ON student.ID = takes.ID

- ON and WHERE behave differently for outer join

- The reason is that outer join adds null-padded tuples only for those tuples that don’t contribute to the result of the corresponding inner join

- The ON condition is part of the outer join specification, but a WHERE clause is not

- If we move the ON clause predicate to the WHERE clause, and set the ON condition to true, then for the student with no courses taken, it doesn’t generate the null values:

select \* FROM student LEFT OUTER JOIN takes ON true WHERE student.ID = takes.ID

Join Types and Conditions:

- INNER JOINS => normal joins (as distinguished from the OUTER JOINS)

- A join clause can thus specify INNER JOIN instead of OUTER JOIN to specify a normal join is to be used

- NOTE: INNER keyword is optional => as the default join type is inner

- Thus => SELECT \* FROM student JOIN takes USING (ID) == SELECT \* FROM student INNER JOIN takes USING (ID)

- Similarly => NATURAL JOIN == INNER NATURAL JOIN

- Types of Join:

1. INNER JOIN, 2. LEFT OUTER JOIN, 3. RIGHT OUTER JOIN, 4. FULL OUTER JOIN

- Types of Join Conditions:

1. NATURAL, 2. ON <predicate>, 3. USING (<list of attributes>)

=========

Views:

=========

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

Chapter 5: Advanced SQL:

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

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

Chapter 10: Storage and File Structure:

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

Introduction:

- In previous chapters, we have emphasized the higher-level models of a database

- For example, at the “conceptual” or “logical” levels => we viewed the database as a collection of tables (in the relational model)

- The logical model of the DB is the correct level for DB users to focus on

- Goal of the DB => simplify and facilitate access to data

- Thus, users of the system should not be burdened unnecessarily with the physical details of the implementation of the system

- Now => we probe below the higher levels

- Start off with the characteristics of the underlying storage media (e.g. disk, tape, flash, etc.)

- Then we consider alternative structures and how each is best suited for different types of data and DBs

- Choice of data structure needs to be made on the basis of the expected use of the system and of the physical characteristics of the specific machine

=========

Overview of Physical Storage Media:

- Several types of data storage classified by different attributes:

- Speed with which the data can be accessed

- The cost per unit of data to buy the medium

- The medium’s reliability

- Common medias:

- Cache:

- Fastest and most costly form of storage

- Relatively small; its use is managed by the computer system hardware

- NOTE: Don’t need to be concerned with managing cache storage of DBMS

- NOTE: However DB implementors do pay attention to the cache when designing query processing data structures and algorithms

- Main Memory:

- The storage medium used for data that are available to be operated on

- General-purpose machine instructions operate on main memory

- NOTE: Despite main memory containing several GBs of data on a PC (or even 100s of GBs of data in large server systems) => it is generally too small (or too expensive) for storing entire DBs

- The contents of main memory are usually lost if a power failure of system crash occurs => “Volatile”

- Flash Memory:

- Differs from main memory in that stored data is retained even if power is turned off (or fails) = “Non-volatile Memory” (i.e. retains stored data even if power is turned off)

- Two types of flash:

1. NAND flash => higher storage capacity for a given cost, and is widely used for data storage for devices such as cameras, music players, and cell phones

2. NOR flash

- Flash has lower cost per byte than main memory

- Being used as a replacement to magnetic disks for storing moderate amounts of data

- “Solid state disks (SSDs)” => disk-drive replacements (can get 64 GB SSD for $200 in 2009)

- Being increasingly used in server systems to improve performance by caching frequently used data, since it provides faster access to data, with larger storage capacity than main memory (for a given cost)

- Magnetic-disk storage:

- Primary medium for the long-term online storage of data

- Usually, the entire DB is stored on magnetic disk

- The system must move the data from disk to main memory so that it can be accessed

- After the system has performed its operations, the data that is modified is written back to disk

- As of 2009, the size of magnetic disks ranges from 80 GBs to 1.5 TBs (and 1 TB of disk costs $100 or so)

- Disk capacities have been growing at 50%/year

- Disk storage survives power failures and system crashes (non-volatile)

- Optical Storage:

- “Compact Disks (CDs)” => can hold about 700MBs of data and has a playtime of about 80 minutes

- “Digital Video Disks (DVDs)” => can hold 4.7 to 8.0 GBs of data per side of the disk (or up to 17GBs for 2-sided disks)

- Also known as “Digital Versatile Disks” => since DVDs can hold any digital data (not just video data)

- Data is stored optically on the disk, and read by a laser

- Higher-capacity format called “Blue-ray DVD” => can store 27GB per layer (or 54GB for 2-sided disk)

- Optical disks used in Read-Only compact disks (CD-ROM) or Read-Only DVDs (DVD-ROM) => can’t be written, but are supplied with data prerecorded

- CD-R and DVD-R and DVD+R => are “record-once” versions => called “Write-once, read-many (WORM)” disks

- CD-RW, DVD-RW, DVD+RW, DVD-RAM => “multiple write” versions

- Optical disk “jukebox” systems => contain a few drives and numerous disks that can be loaded into 1 of the drives automatically on demand

- Tape storage:

- Used primarily for backup and archival data

- Magnetic tape is cheaper than disks, but access to data is much slower (as the tape must be accessed sequentially from the beginning)

- Thus => it is known as “sequential-access” storage

- Disk is “direct-access” storage => it is possible to read data from any location on disk

- High capacity (40 to 300 GB tapes are available in 2009)

- Can be removed from the tape drive => thus they are well-suited for cheap archival storage

- Tape libraries (jukeboxes) => used to hold exceptionally large collections of data (e.g. satellites on the order of terabytes and petabytes)

- Memory Hierarchy:

- Can organize the different storage medias in a hierarchy in which the higher levels are more expensive and fast, and the lower levels are cheaper but slower

- As you move down the hierarchy => the cost per bit decreases, but the access time also decreases

- Trade-off is reasonable => if given a faster and cheaper storage then obviously the other options would be obsolete

- Previously, magnetic tapes were used to store all active data back when disks were expensive and had low storage capacity

- Now, all active data is stored on disks (except in rare cases (i.e. jukeboxes))

- Primary storage => the fastest storage media (i.e. Cache, Main Memory)

- Secondary storage (or online storage) => the next level (i.e. Flash, Magnetic Disks)

- Tertiary storage (or offline storage) => the lowest level (i.e. magnetic tape, jukeboxes)

- Attributes of the memory hierarchy:

- Cost per bit

- Access time

- Storage capacity

- Volatile storage => whether the contents is retained when the device is powered off

- Cache, main memory => volatile

- Magnetic disk, Flash, magnetic tape, jukeboxes => non-volatile

=========

Magnetic Disk and Flash Storage:

- Magnetic disks provide the bulk of secondary storage for modern computers (well actually I believe that flash is starting to take over)

- A very large DB may require 100s of disks

- In recent years, flash-memory storage sizes have grown rapidly, and flash is becoming increasingly competitive to disks (trade-offs to consider)

- Physical characteristics of disks:

- Each disk “platter” => has a flat, circular shape

- Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces

- Platters are made from rigid metal or glass

- When the disk is in use, a drive motor spins it at a constant high speed (e.g. 60, 90, 120, or even 250 revolutions per second)

- Read-write head positioned just above the surface of a platter

- Disk surface is logically divided into “tracks” => which are subdivided into “sectors”

- Sectors are the smallest unit of information that can be read/written from/to disk

- Currently available disks have sector sizes typically of 512 bytes

- There are about 50,000 to 100,000 tracks per platter

- There are about 1 to 5 platters per disk

- Inner tracks closer to the spindle are of smaller length (and thus have fewer sectors), while outer tracks are longer, and thus have more sectors

- Typically, there are around 500 to 1000 sectors per track in the inner track, and around 1000 to 2000 sectors for the outer tracks

- The “read-write head” => stores info on a sector magnetically as reversals of the direction of magnetization of the magnetic material

- Each side of a platter has a read-write head that moves across the different tracks of the platter

- “Disk arm” => all read-write heads are mounted on a single assembly, and move together

- “Head-disk assemblies” => the disk platters mounted on a spindle and the heads mounted on a disk arm are together

- All the heads on all the platters move together => thus, when the head on one platter is on the ith track, the head on all other platters is also on their respective ith track

- “Cylinder” => hence, the ith tracks of all the platters together are called the ith cylinder

- Most common platter diameter is 3(1/2) inches (as they have lower cost, and faster seek times due to smaller seek distances than do larger-diameter platters)

- The read-write head is kept as close as possible to the disk surface to increase recording density

- The head flies incredibly close to the surface, and is kept floating above the surface by the breeze generated by the disk spinning

- Head crashes can be a problem => where the head contacts the disk surface, causing scraping of the recording medium, and destruction of data

- “Disk controller” => interfaces between the computer system and the actual hardware of the disk drive (implemented in the disk drive unit)

- Accepts high-level commands to read/write a sector, and initiates actions (e.g. moving disk arm to the right track, and actually reading/writing data)

- Attach “checksums” => to each sector that is written for error-checking

- Performs “remapping of bad sectors” => if the controller detects that a sector is damaged when the disk is initially formatted (or when an attempt to write the sector is made), then it can logically map the sector to a different physical location

- Disks are connected to the computer system through a high-speed interconnection

- Common interfaces:

1. Serial ATA (SATA) or SATA II or SATA3 GB or Parallel ATA (PATA)

2. “Small-computer-system interconnect (SCSI)”

3. Serial-attached SCSI (SAS)

4. Fiber Channel Interface

5. USB interface (often used by portable external disk systems)

- Disks are usually connected directly by cables to the disk interface of the computer system

- They can also be accessed remotely and connected by a high-speed network to the disk controller

- “Storage Area Network (SAN)” => architecture where large numbers of disks are connected by high-speed network to a number of server computers

- Disks are usually arranged in “Redundant Arrays of Independent Disks (RAID)” => giving the server a view of a very large, very reliable disk

- The computer and the disks use the SCSI, SAS, or Fiber channel interface protocols to talk with each other

- “Network Attached Storage (NAS)” => alternative to SAN, where it provides a file system interface using networked file system protocols (e.g NFS or CIFS)

- Performance Measures of Disks:

- Main measures of disk quality:

1. Capacity

2. Access Time => the time from when a read or write request is issued to when data transfer begins

- To access data on a given sector, the arm must first move so that it is positioned over the correct track, and then must wait for the sector to appear under as disk rotates

- “Seek Time” => the time for the repositioning of the disk arm

- Typical seek times range from 2 to 30ms (depending on how far the track is from the initial arm position)

- Smaller disks tend to have lower seek times since the head has to travel shorter distances

- “Average seek time” => the average of seek times, measured over a sequence of random requests

- If all tracks have the same number of sectors, and we disregard positioning time => avg. seek time is 1/3 the worst-case seek time

- Avg. seek time is 1/2 of the max seek time => ranges between 4 to 10ms

- “Rotational latency” => once the arm has reached the desired track, this is the time spent waiting for the sector to be accessed under the head

- Speeds range from 5400rpm (90 rps) to 15,000rpm (250 rps) => or 4 to 11.1ms per rotation

- “Average latency time” => 1/2 of the time for a full rotation of the disk

- Access time = seek time + latency (which ranges from 8 to 20ms)

3. Data-transfer Rate => once the 1st sector of data to be accessed is under the head, the data transfer begins

- Is the rate at which data can be retrieved from or stored to disk (max transfer rates of 25 to 100MBPS)

4. Reliability => can be measured in “Mean Time to Failure (MTTF)”

- The amount of time (on average) we can expect the system to run continuously without failure

- Vendors claim the mean time to failure of disks in 2009 ranges from 500,000 to 1,200,000 hours (about 57 to 136 years)

- In practice, the claimed MTTF is computed on the probability of failure when the disk is new (on average 1 disk will fail within 1200 hours)

- Most disks have an expected life span of about 5 years, and have significantly higher rates of failure once they become more than a few years old

- Optimization of Disk-Block Access:

- Requests for Disk I/O are generated both by the file system and by the VM manager found in most OSes

- Each request specifies the address on the disk to be referenced => in the form of a “block number”

- “Block” => logical unit consisting of a fixed number of contiguous sectors (sizes range from 512 bytes to several KBs)

- Data is transferred between disk and main memory in units of blocks

- “Page” => refer to blocks (number of blocks for a given page which is used to page in/page out pages into memory)

- A sequence of requests for blocks from disk may be classified as a sequential or random access pattern

- “Sequential access pattern” => successive requests are for successive block numbers (which are on the same track, or on adjacent tracks)

- Thus, a disk seek may be required for the first block, but subsequent blocks would not need a seek, or would require short seeks

- “Random access pattern” => successive requests are for blocks that are randomly located on disk

- Each request requires a seek

- The number of random block accesses that can be satisfied by a single disk in a second depends on seek time (typically about 100 to 200 accesses per second)

- Techniques to improve speed of access to blocks:

- Buffering:

- Blocks that are read from disk are stored temporarily in an in-memory buffer to satisfy future requests

- Done by both the OS and the DBMS

- Read-ahead:

- When a disk block is accessed, consecutive blocks from the same track are read into an in-memory buffer (even if there are no pending request for the blocks)

- In the case of sequential access => such read-ahead ensures that many blocks are already in memory when they are requested (reduces wasted seek time, and rotational latency)

- Not very useful for random access

- Scheduling:

- If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting blocks in the order in which they will pass under the heads

- If desired blocks are on different cylinders => it might be advantageous to request blocks in an order minimizes disk-arm movement

- “Disk-arm scheduling” => algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed

- “Elevator algorithm” => in which arm moves from innermost to outermost track, and on its journey, it processes each track where a request exists, and then reverses

- Disk arms usually perform reordering of read requests in order to improve performance (i.e. maximize throughput by possibly sacrificing latency)

- File Organization:

- To reduce block-access time, we can organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed

- For example, if we expect a file to be accessed sequentially, then we should ideally keep all blocks of the file sequentially on adjacent cylinders

- “Extent” => guarantee by an OS to allocate a given number of contiguous blocks to a file (which allows for some amount of sequential access to a file)

- “Fragmentation” => when this occurs, the FS can choose to handle it in different ways (e.g. dynamic-handling, static-handling at low load periods, etc.)

- Nonvolatile write buffers:

- “Nonvolatile Random-Access Memory (NVRAM)” => contents aren’t lost in power failure, and are used to speed up writes

- When the DB requests that a block be written to disk, the disk controller writes the block to an NVRAM buffer, and immediately notifies the oS the write succeeded

- Then the controller writes the data to the disk when the disk doesn’t have any other requests

- On recovery from a crash, any pending buffered writes in the NVRAM are written back to disk

- Common in RAID systems

- Log Disk (“Journaling File Systems):

- Disk devoted to writing a sequential log (similar to NV write buffers)

- The data has to be written to disk at some point, but the log disk can do the writes later (without the DB having to wait for writes to complete)

- If the system crashes before writing to actual disk, then the OS can read the log when it starts up, and finish the disk writes

- Journaling FSes => can be implemented even without a separate log disk (i.e. keeping data and log on the same disk)

- Reduces monetary cost, but with lower performance

- Flash Storage:

- Two types:

1. NAND flash => requires an entire page of data (typically 512 and 4096 bytes) to be fetched from NAND flash to main memory

- Pages in NAND flash are similar to sectors in disk

- Significantly cheaper than NOR flash , and have much higher storage capacity => more widely used than NOR

2. NOR flash => allows random access to individual words of memory, and has read time comparable to main memory

- NAND flash has same block-oriented interface as disk storage

- Comparing to disk:

- Compared to disk, flash can provide much faster random access (i.e. page can be retrieved in 1 to 2ms vs. 5 to 10ms in disk)

- Has lower transfer rate than disks, but by implementing multiple flash chips in parallel (transfer rates can be much faster than disks)

- Writes:

- Typically takes a few ms

- Once written, a page can’t be directly overwritten, but must erased and rewritten

- Erase:

- Can be performed on a number of pages “erase block” at once => and takes about 1 to 2 ms

- Flash has max number of writes that can be made to a page (typically 100,000 to 1,000,000 times)

- “Translation table” => table mapping logical-to-physical pages to reduce “wear” to pages by limiting the impact of slow erase speed and updates

- Some pages that have been erased many times are assigned “cold data” => data that is rarely updated (to reduce wearing of these pages)

- Pages that have been erased very little are assigned “hot data” => data that is updated frequently

- “Wear leveling” => the idea of evenly distributing erase operations across pages

- All actions are carried out by a layer of software known as the “flash translation layer”

- Above this layer, flash looks identical to disk storage (e.g. same page/sector-oriented interface, except flash is faster)

- Hybrid disk drives exist that combine disk and flash for different usages

=========

File Organization:

- A DB is mapped into a number of different files that are maintained by the underlying OS

- These files reside permanently on disk

- “File” => organized logically as a sequence of records

- These records are mapped onto disk blocks

- Each file is logically partitioned into fixed-length storage units “blocks” => which are units of both storage allocation and data transfer

- Most DB block sizes are 4 to 8 KBs by default

- A block may contain several records => the exact set of records that a block contains is determined by the form of physical data organization being used

- Assumptions (we will make):

- No record is larger than a block (counter: images, videos)

- Each record is entirely contained in a single block (i.e. we will not split records between blocks)

- In a relational DB => tuples of distinct relations are generally of different sizes

- One approach => store records of only 1 fixed length in any given file (easier to implement)

- Another approach => accommodate multiple lengths for records

- Fixed-length records:

- Consider instructor records (i.e. ID (varchar of 5 bytes), name (varchar of 20 bytes), dept\_name (varchar of 20 bytes), salary (numeric(8, 2) will say 8 bytes)

- Thus, a record takes up 53 bytes

- Simple approach => use first 53 bytes for 1st record, second 53 bytes for 2nd record, etc.

- Problems with this approach:

1. Block sizes are usually not multiples of 53 (thus, records would cross block boundaries)

- Can avoid by allocating only as many records to a block as would fit in the block (and leaving remaining bytes as unused)

2. Difficult to delete records from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records, so that they can be ignored

- Can avoid by shifting records over to occupy the deleted record (stupid, as this is very slow process)

- Other approach: leave the deleted record space open and wait for subsequent insertion before reusing the space

- “File header” => at the beginning of the file, allocate a certain number of bytes

- Contains metadata about the file including the address of the first record whose contents were deleted

- The first record contains the address of the 2nd available record, and so on => “Free list”

- Each free record spot contains a pointer to the next free record position in the list

- On insertion, use the record pointer in the header to find the first free position, then add record there, and update header pointer

- If no space is available, add new record to the EOF

- Insertion and deletion for files of fixed-length records is simple to implement => the space made available by a deleted record is exactly the space needed to insert a record

- Variable-Length Records:

- Arise for several reasons in DBs:

1. Storage of multiple record types in a file

2. Record types that allow variable lengths for 1 or more fields

3. Record types that allow repeating fields (e.g. arrays, multisets)

- Different techniques to handle them, but all must solve two problems:

1. How to represent a single record in such a way that individual attributes can be extracted easily

2. How to store variable-length records within a block, such that records in a block can be extracted easily

- Representation of a record with variable-length attributes typically has 2 parts:

1. Initial part with fixed length attributes

2. Followed by data for variable-length attributes

- Fixed-length attributes (e.g. int, numeric, dates) => are allocated as many bytes as required to store their value

- Variable-length attributes (e.g. varchar) => are represented by an initial part of the cord by a pair (offset, length), followed by the actual data (after initial part)

- Offset => Denotes where the data for that attribute begins within the record

- Length => Denotes the length in bytes of the variable-sized attribute

- Record also might contain a “null bitmap” => indicating which attributes of the record have a null value

- Could store it right after the initial fixed-length part

- Could store it at the beginning of the record => thus, if an attribute is null, then no data is stored at all for it

- Saves storage space, but is more work to extract attributes for a record

- Handling storing of variable-length records in a block => “Slotted-page structure”

- There is a header at the beginning of each block containing:

1. The number of record entries

2. Ptr to the end of the free space in the block

3. Array whose entries contain the location and size of each record

- Actual records are allocated contiguously in the block, starting from the end of the block

- The free space is contiguous as well, between the final entry in the header, and the start of the first record

- If record is inserted, space is allocated at the end of the free space, and an entry containing its size and location is added to the header

- If record is deleted, the space is freed, and its entry is “deleted” (i.e. set size to -1), and records in the block before the deleted record are moved, so that free space is contiguous, and the end of free space ptr is updated

- The cost of moving records is cheap (as size of block is 4 to 8 KBs)

- Doesn’t require any direct pointers to records, but the use of “indirect” pointers to point to the associated entries in the header

- This level of indirection allows records to be moved to prevent fragmentation of space inside a block

- To store data that is larger than a disk block, can use special files (e.g. B+-tree file types)

=========

Organization of Records in Files:

- Ways to organize records in files:

- Heap file organization:

- Any record can be placed anywhere in the file where there is space for the record

- There is no ordering of records

- There is a single file for each relation

- Sequential file organization:

- Records are stored in sequential order, according to the value of a “search key” of each record

- “Sequential file” => designed for efficient processing of records in sorted order based on search keys

- “Search key” => attribute or set of attributes (not necessarily a key value)

- To permit fast retrieval of records in search-key order => chain together records by pointers

- Pointers point to the next record in the search-key order

- Records are stored physically in search-key order to further reduce number of block accesses

- It is difficult to maintain physical sequential order as records are inserted and deleted, since it is costly to constantly move records as a result of insertions/deletions

- Can manage deletion by using pointer chains

- Insertion rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order

2. If there is a free record within the same block as this record, the insert new record there

- Else, insert the new record in an “overflow block”

- In either case, adjust the pointers to keep the chain in proper order

- If relatively few records need to be stored in overflow blocks => this approach works well

- Eventually, the correspondence between search-key order and physical order may be totally lost, and then sequential processing becomes inefficient

- Thus, during low load periods => “reorganization” can take place to make the order physically sequential again

- NOTE: If insertions rarely occur (rare case), it is possible to always keep in physical order (and there is no need to use pointers)

- Hashing file organization:

- Hash function computed on some attribute of each record

- The result specifies in which file the record should be placed

- Generally => a separate file is used to store the records of each relation

- In “Multi-table clustering file organization” => records of several different relations are stored in the same file; further, related records of different relations are stored on the same block

- I/O operations fetching related records is cheaper (while it can be more expensive for certain other queries)

- Even if multiple relations are stored in a single file, most DBs (by default) store records of only 1 relation in a given block

- This simplifies things, but for some cases, it is useful to store records of more than 1 relation in a single block (e.g. queries containing joins of tables)

- In the worst case, each record (of the join op) will reside in a different block, forcing us to do 1 block read for each record required by the query

- However, if we use multitable clustering based on specific queries, we can only have to do 1 block read for both relations

- While processing of a specific query are more efficient, others suffer in performance (trade-offs)

- Ex: SELECT \* FROM department (now our block for department contains significantly fewer department records as we have multiple relation records in the block, so it requires more block accesses)

- When using this file organization approach => must consider the types of queries that will be most frequently used

=========

Database Buffer:

- Major goal of DB system => minimize the number of block transfers between disk and main memory (to boost performance and reduce idle time)

- One way to reduce the number of disk accesses is to keep as many blocks in main memory as possible

- Goal: when a block is accessed, if it is already in main memory, then performance is significantly improved, and disk I/O is reduced

- It is not possible to keep all blocks in main memory, so we need to manage the allocation of the space available in main memory for storage of such blocks

- “Buffer” => part of main memory where we can store available copies of disk blocks

- The copy on disk could become “stale” => if we make updates to the local version, and thus we need to write back to disk with the modifications

- “Buffer manager” => system responsible for allocation of buffer space and the replacement policy of blocks

- Buffer manager:

- Programs make requests when they need a disk block

- If the block is already in the buffer, then the address of the block in main memory is given to the requester

- If the block is not in the buffer, the buffer manager must first allocate space for the block, and then throw a block out using an “eviction policy”

- The evicted block is written back to disk if local modifications were made in main memory

- Techniques used by buffer manager:

- Buffer replacement strategy:

- Policy to evict blocks when the buffer is full

- Most OSes use “Least Recently Used (LRU)” => where the block referenced the least is written back to disk and removed from the buffer

- In DB systems, LRU is often less efficient, and “Most Recently Used (MRU)” is chosen instead

- Pinned blocks:

- For the DB system to recover from crashes, it is necessary to restrict those times when a block may be written back to disk

- For example, you might pin a block to prevent writing it to disk, while updates are being made

- “Pinned” => when a block is not allowed to be written back to disk

- Forced output of blocks:

- There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed

- “Forced output” of the block

- Sometimes main memory contents are lost (e.g. crash) and as main memory is volatile memory, the data is lost

- Forced output to persist the data to disk

- Buffer replacement policies:

- Goal: minimize accesses to disk; and thus, keep the “right” blocks in the buffer to reduce disk I/O and stay in main memory as much as possible

- For general-purpose programs (e.g. OS stuff), it isn’t possible to predict accurately which blocks will be accessed in the future

- Thus, OSes use the past pattern of block references as an indicator for future references (i.e. LRU)

- In DBMSes, you can predict the pattern of future references more accurately

- A user request to the DB system involves several steps, in which the DB is able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation

- Thus, DB systems may have info regarding at least the short-term future

- Consider a query that natural joins two relations (in which the relations are stored in different blocks):

- After a tuple of one relation is processed, it is no longer needed (and thus once the block for one relation is processed, the block is no longer needed in main memory), even though it was most recently used

- This strategy is called “toss-immediate” strategy => and thus, the block that was accessed least recently is the one to be referenced next

- Therefore => the optimal replacement policy is MRU strategy => which chooses the most recently used block to be removed

- For MRU to work, the system must pin blocks currently being processed, and when unpinning them (they become the MRU block)

- NOTE: In general, buffer managers should try to not remove index blocks or dictionary blocks

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

Chapter 11: Indexing and Hashing:

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

Basic Concepts:

- Indexes in file in DB systems work much the same way as the index of a textbook

- For example, to retrieve a student record given an ID, the DB would look up an index to find on which disk block the record resides, and then fetch that block

- Keeping a sorted list of students’ ID would not work well on very large DBs with 1000s of students (since the index would be large itself, thus defeating the purpose)

- Additionally, even if it is sorted, finding students would still be time-consuming

- DB systems use sophisticated indexing methods:

1. Ordered Indices => Based on a sorted ordering of the values

2. Hash Indices => Based on a uniform distribution of values across a range of buckets (as determined by the hash function chosen)

- Each indexing technique is suited for particular databases, but all must be evaluated on the following criteria:

- Access Types => The types of access that are supported efficiently

- Can include finding records with a specified attribute value and finding records with a specified attribute value and finding records whose attribute values fall within a certain range

- Access Time => the time it takes to find a particular data item, or set of items

- Insertion Time => the time it takes to insert a new data item

- This value includes the time it takes to find the correct place to insert, as well as the time it takes to update the index structure

- Deletion Time => the time it takes to delete an item

- This value includes the time it takes to find the item to delete, as well as the time it takes to update the index structure

- Space overhead => The additional space occupied by an index structure

- Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice space to achieve improved performance

- Often times, we want to have more than 1 index for a file (e.g. we may want to search for a book by author, by subject, or by title)

- “Search key” => attribute or set of attributes used to look up records in a file

- If there are several indices on a file, then there are several search keys

=========

Ordered Indices:

- To gain fast random access to records in a file, we can use an index structure

- Each index structure is associated with a particular search key

- Ordered indices store the values of the search keys in sorted order, and associates with each search key the records that contain it

- The records themselves be stored in some sorted order in the index structure

- Files may have several indices, on different search keys

- “Clustering index” => index whose search key also defines the sequential order of the file

- If the file containing records is sequentially ordered

- Clustering indices are also called “primary indices”

- NOTE: Primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key

- The search key of a clustering index is often the primary key (although it doesn’t necessarily have to be so)

- “Non clustering indices (or secondary indices)” => indices whose search keys specify order different from the sequential order of the file

- “Index-sequential files” => files with a clustering index on the search key (i.e. files are ordered sequentially on some search key)

- They represent one of the oldest index schemes used in DBMSes

- Designed for applications that require both sequential processing of the entire file and random access to individual records

====

Dense and Sparse Indices:

- “Index entry (or index record)” => consists of a search-key value and pointers to 1 or more records with that value as their search-key value

- The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block

- Two types of ordered indices that we can use:

1. Dense Index:

- Index entries appear for every search-key value in the file

- Dense clustering index => the index record contains the search-key value and a pointer to the 1st data record with that search-key value

- The rest of the records with the same search-key value would be stored sequentially after the 1st record (as the index is clustering, and records are sorted on the same search key)

- Dense non clustering index => index must store a list of pointers to all records with the same search-key value

2. Sparse Index:

- Index entry appears for only some of the search-key values

- Used only if the relation is stored in sorted order of the search key (i.e. if the index is clustering)

- Each index entry must contain a search-key value and a pinter to the first data record with the search-key value

- To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking for

- We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record

- Consider an example => searching for student with ID “22222”:

- Dense index:

- Follow the pointer directly to the desired record

- As ID is the primary key, there is only one record with such an ID

- Sparse index:

- We don’t find an index entry “22222”, so we look at the last entry (in numerical order) before “22222”, which is “10101”

- Follow that pointer, and then read the records in sequential order (following their next pointers) until we find the desired record

- Consider a printed dictionary:

- The header of each page lists the first word alphabetically on that page

- Sparse index => the words at the top of the each page index together form a sparse index on the contents of the dictionary pages

- Consider another example in which the search-key is not the primary key => the search key is based on dept\_name:

- Thus, the dense index is non clustering

- Suppose we are looking up records for the History department:

- Using the dense index, we follow the pointer directly to the 1st History record

- We process this record, and follow the next pointer in that record to located the next record in search-key (dept\_name) order

- We continue processing records until we encounter a record for a department other than History

- Comparing dense to sparse:

- Generally, it is faster to locate a record if we have a dense index

- Sparse indices require less space and they impose less maintenance overhead for insertions and deletions

- Designer must make tradeoff decisions based on access time and space overhead

- Good design compromise => have a sparse index with 1 index entry per block:

- The dominant cost in processing a DB request is the time to bring the block from disk to main memory

- Once in main memory, the time to scan the entire block is negligible

- Using the sparse index, we locate the block containing the record we are seeking

- Thus, unless the record is in an overflow block, we minimize block accesses while keeping the size of the index (and thus space overhead) as small as possible

====

Multilevel Indices:

- Suppose we build a dense index on a relation with 1,000,000 tuples:

- Index entries are smaller than data records, so assume that 100 index entries fit on a 4KB block

- Thus, the index occupies 10,000 blocks (i.e. 100 index entries per block \* 10,000 blocks to cover 1,000,000 tuples)

- If the relation consists of 100,000,000 tuples, then the index would occupy 1,000,000 blocks (i.e. 4GB of space)

- These large indices are stored as sequential files on disk

- If an index is small enough to be kept entirely in main memory => search time to find an entry is low

- If the index is so large that not all of it can be kept in memory => index blocks must be fetched from disk when required

- NOTE: Even if an index is smaller than main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory

- The search in this case, thus requires several disk-block reads

- Binary search can be used on the index file to locate an entry, but the search is still a large cost:

- If the index occupies b blocks, binary search would take log b blocks to be read

- For a 10,000 block index, binary search requires 14 block reads

- On a disk system where a block read takes on avg. 10 ms, this would take 140 ms

- Would only allow 7 index searches a second => very slow

- NOTE: If overflow blocks are being used, binary search isn’t possible (would have to do sequential search, requiring reading b blocks, which takes even longer)

- Thus, processing of searching a large index may be costly

- To deal with this problem => can treat the index just as we would treat a sequential file:

- Construct a sparse outer index on the original index (which we now call the inner index)

- NOTE: The index entries are always in sorted order, which allows the outer index to be sparse

- To locate a record:

1. Perform binary search on the outer index to find the record for the largest search-key value less than or equal to the one we desire

2. Pointer points to a block in the inner index

3. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire

4. Pointer points to the block of the file containing the record we are looking for

- In our example, an inner index of 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks

- If we assume that the outer index is already in main memory, then this would require only 1 index block for a search using the multi-level index approach, rather than 14 blocks for just the binary search version

- Thus => we can perform 14 times as many index searches per second (or 98 as compared to 7 before)

- If the file is extremely large, even the outer index may grow too large to fit into main memory

- With 100,000,000 tuples in a relation, the inner index would occupy 1,000,000 blocks, and the outer index would occupy 10,000 blocks (or 40MBs)

- In this case (i.e. when the entire outer index can’t fit in main memory) => we could create another level of indices

- Thus, we can add as many levels as necessary

- “Multilevel” indices => indices with 2 or more levels

- Requires significantly fewer I/O ops than does searching for records using binary search

- Closely related to tree structures, such as the binary tree, used for in-memory indexing

====

Index Update:

- Regardless of the form of index used, every index must be updated whenever a record is either inserted, deleted, or updated in a file

- In the case of record updates => any index whose search-key attribute is affected by the update must be updated (i.e. changing the department of an instructor in the instructor relation, in which the index is on dept\_name)

- Updates can be modeled as deleting of the old record, followed by insertion of the new value of the record (i.e. index deletion then index insertion)

- Thus, we only need to consider index insertion and index deletion (and not handle index updates explicitly)

- Updating single-level indices:

- Insertion: First, the system performs lookup using the search-key value that appears in the record to be inserted

- The actions taken depend on whether the index is dense or sparse:

- Dense:

1. If the search-key value isn’t in the index, the system inserts an index entry with the search-key value in the appropriate position

2. Otherwise:

a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry

b. Otherwise, the index entry stores a pointer to only the first record with the search-key value

i. The system then places the record being inserted after the other records with the same search-key value

- Sparse: We assume that the index stores an entry for each block.

1. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block of the index

2. Otherwise, if the new record has the least search-key value in its block => the system updates the index entry pointing to the block (if not, the system makes no changes)

- Deletion: To delete a record, the system first looks up the record to be deleted

- The actions taken depend on whether the index is dense or sparse:

- Dense:

1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index

2. Otherwise:

a. If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry

b. Otherwise, the index entry stores a pointer to only the first record with the search-key value

i. If the deleted record was the 1st record with the search-key value, the system updates the index entry to point to the next record

- Sparse:

1. If the index doesn’t contain an index entry with the search-key value of the deleted record, nothing needs to be done

2. Otherwise:

a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order)

i. If the next search-key value already has an index entry, the entry is deleted instead of being replaced

b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value

====

Secondary Indices:

- Secondary indices must be dense, with an index entry for every search-key value and a pointer to every record in the file

- Clustering index is allowed to be sparse (storing only some of the search-key values) => since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file

- Secondary indices cannot be sparse => as records with intermediate search-key values may be anywhere in the file, and we can’t fid them without searching the entire file

- A secondary index on a candidate key looks just like a dense clustering index, except the records pointed to by successive values in the index aren’t stored sequentially

- In general, secondary indices may have a different structure from clustering indices

- If the search key of a clustering index is not a candidate key, then it suffices if the index points to the 1st record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file

- If the search key of a secondary index is not a candidate key, it is not enough to point to just the 1st record with each search-key value

- The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index

- Thus, a secondary index must contain pointers to all records

- Secondary indices can be implemented by using an extra level of indirection on search keys that aren’t candidate keys

- The pointers in the secondary index do not point directly to the file

- Instead, they point to a bucket that contains pointers to the file

- Sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order

- However, we can’t store a file physically ordered by both the search key of the clustering and secondary indices

- Because secondary-key and physical-key order differ => attempting to scan the file sequentially in secondary-key order would result in the reading of each record that is likely to require the reading of a new block from disk => which is very slow

- The procedures for insertion and deletion can be applied to secondary indices

- Secondary indices improve the performance of queries that use keys other than the search key of the clustering index

- However, they impose a significant overhead on modification of the DB

- The designer must decide which secondary indices should be included based on an estimate of the frequency of queries and modifications

====

Indices on Multiple Keys:

- Thus far, we have seen search keys only with a single attribute

- In general, search keys can have more than 1 attribute

- “Composite search key” => search key containing more than 1 attribute

- The structure of such an index is the same as for any other index, except for the fact that the search key is a list of attributes

- For example, consider an index on the takes relation, on the composite search key (course\_id, semester, year):

- Such an index would be useful to find all students who have registered for a particular course in a particular semester/year

=========

B+-Tree Index Files:

- Index Sequential Access Method (ISAM) => main disadvantage is that performance degrades as the file grows (both for index lookups and for sequential scans through the data)

- The degradation can be remedied by reorganization of the file => but frequent reorganizations incurs serious overhead, and is undesirable

- “B+-tree Index” => the most widely used of several index structures that maintains efficiency despite insertions and deletions

- Takes the form of a “balanced tree” => in which every path from the root of the tree to a leaf of the tree is of the same length

- Each non-leaf node in the tree has between ceil(n/2) and n children (where n is fixed for a particular tree)

- B+-Tree imposes performance overhead on insertion and deletion, and adds space overhead

- However, the overhead is amortized for frequently modified files => since the cost of file reorganization os avoided

- Since nodes may be as much as half empty (if they have the min number of children) => there is wasted space

- However, the space overhead is also amortized given the performance benefits of this index structure

====

Structure of a B+-Tree:

- B+-Tree index is a multilevel index (but its structure differs from multilevel index-sequential files)

- Typical node contains up to n - 1 search-key values (K1, K2, …, Kn-1) and n pointers (P1, P2, …, Pn)

- The search-key values within anode are kept in sorted order

- “Leaf nodes” => For i = 1, 2, …, n - 1, pointer Pi points to a file record with search-key value Ki

- Pointer Pn has a special purpose (i.e. points to a sibling leaf node used for sequential search-key order access)

- As there is a linear order on the leaves based on the search-key values => Pn is used to chain together the leaf nodes in search-key order (allowing for efficient sequential processing of the file)

- Nodes range from ceil((n - 1)/2) to n - 1 search-key values

- Ex: n = 4 => leaf nodes have at least 2 nodes and at most 3 nodes

- The range of values in each leaf do not overlap (save for duplicate search-key values, in which case a value may be present in more than 1 leaf)

- Specifically, if Li and Lj are leaf nodes (where i < j), then every search-key value in Li <= every search-key value in Lj

- If the B+-Tree index is used as a dense index (usually), then every search-key value must appear in some leaf node

- “Non-leaf nodes (or internal nodes)” => form a multilevel (sparse) index on the leaf nodes

- The structure is the same, except that all pointers are pointers to tree nodes (and not records in the file)

- Nodes range from ceil(n/2) pointers to a max of n pointers

- “Fanout” => the number of pointers of a node

- Consider a node containing m pointers (m <= n):

- For i = 2, 3, …, m - 1, pointer Pi points to the subtree that contains search-key values < Ki and >= Ki-1

- Pointer Pm points to the part of the subtree containing the key values >= Km-1

- Pointer P1 points to the part of the subtree containing search-key values < K1

- “Root node” => can hold fewer than ceil(n/2) pointers, but most have at least 2 pointers (unless the tree has only 1 node)

- NOTE: A B+-Tree with larger n will be shorter in height than a B+-Tree with smaller n, but will be fatter (i.e. larger in width)

- NOTE: The B in B+-Tree stands for “balanced”

- The balanced property ensures good performance for lookup, insertion, and deletion

====

Queries on B+-Trees:

- Consider how we process queries and suppose we wish to find a record with search-key value V (assume no duplicates):

- Start at the root of the tree, and traverse the tree down until it reaches a leaf node that would contain the specified value (if it exists) in the tree

- Specifically (starting at the root node), repeat the following steps until a leaf node is reached:

1. The current node is examined, looking for the smallest i such that search-key value Ki >= V:

a. If Ki = V, then the current node is set to the node pointed to by Pi+1

b. Otherwise if Ki > V, then the current node is set to the node pointed to by Pi

c. If Ki < V, then the current node is set to point to Pm

- Repeat until a leaf node is reached

2. At the leaf node, if there is a search-key value = V, let Ki be the first such value, and pointer Pi directs us to the record with search-key value Ki

- The function then returns the leaf node L and index i

a. Otherwise, if no search-key with value V is found, then no record with key value V exists in the relation, so return null (indicating failure)

- B+-Trees can be used to find all records with search key values in a specified range (L, U)

- For example, if our B+-Tree on attribute salary of instructor, we can find all instructor records with salary in a specified range (e.g. 500000, 100000) => “Range queries”

- In processing a query, we traverse a path in the tree from the root to some leaf node

- If there are N records in the file, then the path is no longer than log(base ceil(n/2))N

- In practice, only a few nodes need to be accessed

- Typically, a node is made to be the same size as a disk block (typically 4KB)

- With search-key size of 12 bytes, and disk-pointer size of 8 bytes, n is around 200

- Even with more conservative estimate of 32 bytes for search-key, n is around 100

- With n = 100, if we have 1,000,000 search-key values in the file, a lookup requires only log(base 50)(1,000,000) = 4 nodes to be accessed

- Thus, at most 4 blocks need to be read from disk for the lookup

- However, the root node of the tree is usually accessed heavily and is likely to be in the buffer (resulting in 3 or fewer blocks needed to read from disk)

- Important difference between B+-Tree structures and in-memory tree structures (e.g. binary tree):

- The size of the node => and as a result, the height of the tree

- In binary trees => each node is small, and has at most 2 pointers

- In B+-Trees => each node is large (typically a disk block), and a node can have a large number of pointers

- Thus => B+-Trees tend to be fat and short

- Thus => Binary trees tend to be skinny and tall

- In a balanced binary tree => the path for lookup can be of length log(base 2)N (with N = 1,000,000 requires 20 node accesses)

- In a B+-Tree => with N = 1,000,000, requires 4 block reads

- This difference is significant since each block read could require a disk arm seek (which together with the block read) => could take about 10ms on a typical disk

====

Updates on B+-Trees:

- When a record is inserted into, or deleted from a relation => indices on the relation must be updated

- NOTE: Updates to a record can be modeled as deletion of the old record, followed by insertion of the updated record

- Thus, we only consider insertions and deletions

- Insertions and deletions are more complicated than lookup in B+-Trees, since it may be necessary to “split” a node that becomes too large as a result of an insertion, or to “coalesce” nodes (i.e. combine nodes) that become too small (i.e. fewer than ceil(n/2) pointers)

- Additionally, when nodes are split/coalesced, we must make sure that balance is preserved (and may have to make adjustments to parent nodes and higher tree levels)

- Assuming that nodes never become too small or too large:

- Insertion:

- Use the same technique for lookup, and once we find the leaf node in which the search-key value would appear, we insert an entry (i.e. search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order

- Deletion:

- Use the same technique for lookup, and once we find the leaf node containing the entry to be deleted; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted

- We then remove the entry from the leaf node

- All entries in the leaf node that are to the right of the deleted entry are shifted left by 1 position, so that there are no gaps in the entries

====

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)

====

Non-unique Search Keys:

- If a relation can have more than 1 record containing the same search key-value (i.e. 2 or more records can have the same values for their indexed attributes), then the search key is said to be “non-unique search key”

- Problem with non-unique search keys:

- Low efficiency of record deletion

- Suppose a particular search-key value occurs a large number of times, and 1 of the records with that search key is to be deleted:

- Then, the deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record to delete

- Simple solution => make search keys unique by creating a composite search key containing the original search key and another attribute => which together are unique across all records

- The extra attribute can be a record-id, which is a pointer to the record, or any other attribute whose value is unique among all records with the same key value

- “Uniquifier” => the extra attribute

- When a record is to be deleted, the composite search-key value is computed for the record, and then used to look up the index

- Since the value is unique, the corresponding leaf-level entry can be found with a single traversal from root to leaf (thus record deletion is efficient)

- Search with the original search-key attribute ignores the uniquifier attribute value

- With non-unique search keys, the B+-Tree structure stores each key value as many times as there are record containing the value

- An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle non-unique search keys

====

Complexity of B+-Tree Updates:

- Although insertion and deletion are complicated for B+-Trees => they have relatively few I/O operations, which is important operation since I/O ops are expensive

- The number of I/O operations in the worst-case for an insertion and deletion => log(base(ceil(n/2))N, where n = max number of pointers per node, N = number of records in file being indexed

- The cost of insertion/deletion operations in terms of I/O ops is proportional to the height of the B+-Tree, and is therefore low

- The speed of ops on B+-Trees that makes them frequently used index structure in DB implementations

- In practice, ops on B+-Trees result in fewer I/O ops than the worst-case bounds

- Although B+-Trees only guarantee that nodes will be at least half full, if entries are inserted in random order => nodes can be expected to be more than 2/3 full on avg

- If entries are inserted in sorted order => nodes will be only 1/2 full

=========

B+-Tree Extensions:

====

B+-Tree File Organization:

- Main drawback of index-sequential file organization => degradation of performance as the file grows

- With growth => an increasing percentage of index entries and actual records become out of order, and are stored in overflow blocks

- Solve degradation of index lookups by using B+-Tree indices on the file

- Solve degradation of storing actual records by using the leaf-level of the B+-Tree to organize the blocks containing the actual records

- Use the B+-Tree structure not only as an index, but also as an organizer for records in a file

- In “B+-Tree File Organization” => the leaf nodes of the tree store records, instead of storing pointers to records

- Since records are usually larger than pointers => the max number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node

- Leaf nodes are still required to be at least half full

- Insertion/deletion of records are handled in the same way as for a B+-Tree Index

- B+-Tree file organizations can be used to store large objects (e.g. SQL clobs and blobs) => which may be larger than a disk block, and as large as multiples of GBs

- They can be stored by splitting them into sequences of smaller records

- The records can be sequentially numbered, or numbered by the byte offset of the record within the large object, and the record number can be used as the search-key

====

Secondary Indices and Record Relocation:

- Some file organizations (e.g. B+-Tree File Organization) => may change the location of records when the records have not been updated

- Ex: Redistributing records between sibling nodes when insertion node overflow occurs

- In this case, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed

- Each leaf node may contain a fairly large number of records, and each of them may be in different locations on each secondary index

- Thus, a leaf-node split may require 10s or 100s of I/O ops to update all affected secondary indices (very expensive)

- Widely used solution:

- In secondary indices, in place of pointers to the indexed records => store the values of the primary-index search-key attributes

- For example, primary index of ID on instructor relation, and secondary index of dept\_name:

- Store with each department name a list of instructor’s ID values of the corresponding records, instead of storing pointers to the records

- Relocation of records because of leaf-node splits then doesn’t require any update on any such secondary indices

- However, locating a record using the secondary index now requires 2 steps:

1. Use the secondary index to find the primary index search-key values

2. Use the primary index to find the corresponding records

====

Indexing Strings:

- Creating a B+-Tree index on string-valued attributes creates two problems:

1. Strings can be of variable length

- With variable length search-keys => different nodes could have different fan-outs even if they are full

- A node must then be split if its full

2. Strings can be long => leading to a low fanout and a correspondingly increased tree height

- Also, nodes can be merged or entries redistributed depending on what fraction of the space in the nodes is used, instead of being based on the max number of entries the node holds

- “Prefix Compression” => technique to increase the fanout of nodes

- Don’t store the entire search-key value at non-leaf nodes

- Only store a prefix of each search-key value that is sufficient to distinguish between the key values in the subtrees that it separates

- Ex: If we index on names, then the key value at a non-leaf node could be a prefix of a name => (may suffice to store “Silb” instead of the full “Silberschatz”)

====

Bulk Loading of B+-Tree Indices:

- Insertion of a record in a B+-Tree requires a number of I/O ops that in the worst case is proportional the the height of the tree (which is usually fairly small)

- Consider the case where a B+-Tree is built on a large relation:

- Suppose the relation is significantly larger than main memory, and we are constructing a non-clustering index on the relation such that the index is also larger than main memory

- Thus, as we scan the relation and add entries to the B+-Tree, it is quite likely that each leaf node accessed is not in the DB buffer when it is accessed, since there is no particular ordering of the entries

- With randomly ordered accesses to blocks => each time an entry is added to the leaf, a disk seek will be required to fetch the block containing the leaf node (costly)

- “Bulk loading” => insertion of a large number of entries at a time into an index

- Efficient way to perform bulk loading of an index:

1. Create a temporary file containing index entries for the relation

2. Sort the file on the search key of the index being constructed

2. Scan the sorted file and insert the entries into the index

- Advantage to sorting before inserting into B+-Tree:

- When the entries are inserted in sorted order, all entries that go to a particular leaf node will appear consecutively, and the leaf needs to be written out only once

- Nodes will never have to be read from disk during bulk load, if the B+-Tree was empty to begin with

- Each leaf node thus incurs only 1 I/O op even though many entries may be inserted into the node

- If each leaf node contains 100 entries, the leaf level will contain 1,000,000 nodes, resulting in 1 million I/O ops for creating the leaf level

- With current disks, 1ms per block is a reasonable estimate for mostly sequential I/O ops, and 10ms per block for random I/O ops

- Thus, an index that would have taken 1,000,000 seconds to be built, can be constructed in 1000 seconds by sorting entries before inserting them

- If the B+-Tree is initially empty, it can be constructed faster by building it bottom-up from the leave level => “Bottom-up B+-Tree Construction”

- After sorting the entries, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block

- The resulting blocks form the leaf level of the B+-Tree

====

B-Tree Index Files:

- Primary distinction between B-Tree and B+-Tree => B-tree eliminates redundant storage of search-key values

- In B+-Trees, every search-key value appears in some leaf node; several appear in non-leaf nodes (repeated)

- B-tree allows search-key values to appear only once

- Since search keys are not repeated, we may be able to store the index in fewer tree nodes than in the B+-Tree index

- However, since search keys that appear in non-leaf nodes, don’t anywhere else => we are forced to include an additional pointer field for each search key in a non-leaf node

- These additional pointers point to either file records or buckets for the associated search key

- As we have fewer search keys in non-leaf nodes => the fanout is smaller, thus => depth is greater in B-Trees

- This may be more efficient for some search keys (i.e. in non-leaf nodes), and slower for others (i.e. go down farther to get to the leaf nodes)

- Deletion is more complicated (as deletion doesn’t just occur in leaf nodes, but also for non-leaf nodes now too)

- Space advantages of B-Trees are marginal for large indices, and thus most DB systems use B+-Trees (and not B-Trees)

====

Flash Memory:

- Thus far, we have assumed that the data is resident on disks

=========

Multiple-Key Access:

- Thus far, we have assumed implicitly that only 1 index on 1 attribute is used to process a query on a relation

- However, for certain types of queries => it is advantageous to use multiple indices if they exist, or to use an index built on a multi-attribute search key

====

Using Multiple Single-Key Indices:

- Assume the instructor file has 2 indices: (1) dept\_name, (2) salary

- Consider the query: “Find all instructors in the Finance department with salary equal to $80,000”:

SELECT ID FROM instructor WHERE dept\_name = ‘Finance’ AND salary = 80000

- Three strategies for processing this query:

1. Use the dept\_name index to find all records pertaining to the Finance department

- Then examine such records to see whether salary = 80000

2. Use the salary index to find all records pertaining to instructors with salary = 80000

- Then examine such records to see whether department = ‘Finance’

3. Use index on dept\_name to find pointers to all records pertaining to the Finance department

- Also, use index on salary to find pointers to all records pertaining to instructors with salary = 80000

- Those pointers that are in the intersection point to records pertaining to both

- The last strategy makes use of multiple indices, but still may be a more choice if:

- There are many records pertaining to the Finance department

- There are many records with salary of 80000

- There are only a few records that satisfy our query

- If these conditions hold, then we must scan a large number of pointers to produce a small result

- A “Bitmap index” => could greatly speed up the intersection operation used in the this strategy

====

Indices on multiple keys:

- Alternative strategy => create and use an index on a composite search key (dept\_name, salary)

- This search key would consist of the department name concatenated with the instructor salary

- Can use an ordered B+-Tree index on the above composite search key to answer efficiently queries of the previously mentioned form

- Queries consisting of an equality condition on the first attribute, and a range on the second attribute (e.g. <) can also be handled efficiently (as it corresponds to a range query)

- Queries only consisting of 1 of the composite search-key can also be handled efficiently

- However, if the query consists of a comparison operation on the first attribute (i.e. <), and not a range query, then the result would be inefficient with our ordered index

- As department name is ordered alphabetically, each record is likely to be located on a different block => leading to many I/O ops

=========

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