**Relational Algebra:**

Set: no duplicate tuples (eliminated)

Bag: allowed duplicates

SELECT – (columns\_

PROJECT – (rows)

CROSS PRODUCT -

NATURAL JOIN - (eq of shared attr)

THETA JOIN - = >

RENAME – =rename R to S

- for R(A1,A2) – rename R(A1,A2) to S(A1’, A2’)

UNION - - schemas of R&S same (use rename to be the same), no duplicates in result

DIFFERENCE - : schemas must match exactly

INTERSECT - :

DIVISION -

\* assume R(A,B) and S(B)

\* set of all a R.A such that for *every*

\* R/S =

(largest relation T such that )

Core operators:

Monotonicity ⬄ more input = more out.

“-“ is only non-monotonicity

- cannot be produced by monotonic op’s

Remember:

Data manipulation language

- relation > algebra > relation

SET vs BAG semantics

Data Modification

INSERT INTO \_\_ VALUES \_\_

DELETE FROM *R* WHERE \_\_

UPDATE *R* SET \_\_ WHERE

Expressive power of SQL

- limited expressive power

- does not support recursion and cannot compute transitive closure

**Database Integrity**

Data validity enforcement

Domain : GPA is real

Constraints: give error, abort state.

- key, referential integrity, CHECK constraint, SQL assertions

- trigger: Event-Condition-Action

Key Constraints

- one primary key per table

Referential Integrity

- means referenced val always exists

- referencing attribute or foreign key

- referenced attribute

- foreign key can be NULL, when it is, no constraint checking

- referenced must be PRIMARY or UNIQUE

- may be omitted if they are the same name with referencing attri.

- RI violation by inserting into referencing table is not allowed

- system rejects the statement

- ON DELETE [SET NULL, CASCADE, SET DEFAULT]

- ON UPDATE [^same]

CHECK

- constraint checked when tuple is updated (or inserted)

- in SQL92, can be complex

SQL Assertion

- entire set of relations or database

- CREATE ASSERTION <name> CHECK (<condition>)

Trigger

- Event-Condition-Action

CREATE TRIGGER <name>

event[AFTER INSERT]

< referencing clause > // optional

WHEN (<condition>) // optional

<action>

<event>

\* BEFORE | AFTER INSERT/ DELETE/ UPDATE [OF A1] ON R

<action>

\* Mutiple statements: BEGIN/ END

<referencing clause>

\* REFERENCING OLD | NEW TABLE | ROW AS <var>, …

\* FOR EACH ROW: row-level trigger

\* FOR EACH STATEMENT (default)

**MySQL:**

DISTINCT (to remove duplicates)

% = any string

\_ = one character

Set operators:

INTERSECT, UNION, EXCEPT

- same schema for operands

- based on set semantics

- keep duplicates with ‘ALL’

Subqueries:

- can rewrite them as long as we don’t have negation

- with negation, we need EXCEPT

Set membership (IN, NOT IN)

- a IN R is TRUE is a is in R

Set comparision (> ALL, < SOME, )

- ALL is , SOME is

- <> ALL NOT IN, = SOME IN

EXISTS and Correlated subqueries

- correlated- outer query looks 1 tuple at a time and binds the tuple to S

- for each S, we execute the inner query and check condition

- this is just interpretation

- Subqueries in FROM clause

- considered as a regular relation

- must be renamed inside

Aggregates

- SUM, AVG, COUNT, MIN, MAX

- COUNT (\*) counts tuples (NULLs)

GROUPBY- duplicates are removed

HAVING - used with aggregates

ORDER BY … [ASC/DESC]

- default ASC, used for looks

Order (by computer): FROM -> WHERE -> GROUP BY -> HAVING -> ORDER BY -> SELECT

NULL

- aggregates are computed ignoring NULL, except COUNT (\*)

- if input to an aggregate is empty, COUNT returns 0, all others return NULL

Set operators (

- NULL is treated like other values here

- check NULL: IS NULL, IS NOT NULL

Arithmetic op’s and comp’s

- for NULL’s, UNKNOWN, only true values returned

Three-valued logic:

Truth table:

- AND: U & T = U, U & F = F, U & U = U

- OR: U | T = T, U | F = U, U | U = U

SQL and bag semantics

- everything uses bag except set oper’s

rules for bag:

- under bag, (for too)

-

OUTER Join

- preserves dangling tuples

- R LEFT/RIGHT/FULL OUTER JOIN S ON R.A = S.A

- keep dangling from: (R by padding S w/ NULL/ S by padding R w/ NULL/ both R and S)

**Relational Model:**

- relations = tables

- attributes = columns

- tuples = rows

- domain = type

SQL Types:

- char(n) – fixed length

- varchar(n) – var lenght

- integer – 32 bit

- decimal (5,2) – 999.99

- real,double – 32bit,64bit

- date – ‘2002-01-15’

- time – ’13:50:00’

- timestamp – ‘2002-01-15 13:50:00’

Tables:

- one primary key per table

- UNIQUE for other keys

- SQL92: no NULL in primary

- DEFAULT for default

Loading Data:

LOAD DATA INFILE <datafile> INTO TABLE <table>

Terms:

Data model = graph/tree model

Schema = table design/structure

Instance = data

Database construction steps:

1. domain analysis

2. database design: E/R model,

database design theory

3. table creation: DDL

4. load

5. query and update: DML

**Disks and Files**

Terms: Boom, Head, Sector, Spindle, Track, Platter, Cylinder

Each platter has: track, cylinder, sector (=block, page)

A disk:

Q: 2 platters, 2 surfaces/ platter, 500 tracks/surface, 200 sect/track, 1KB/ sector

Access Time:

= seek time + rotational delay + transfer

Seek Time

- time to move disk head between tracks

e.g. time to track ~ 1ms, average 10 ms, full ~ 20 ms, 1000 sectors:20/1000 ms

Rotational Delay

- typical: 1000-15k RPM

Q: for 6k RPM, average rotational delay

1x60sec/min/(6k RPM) = 1/100 => .01 sec/2 = 5ms

Transfer Rate

- how long to read one block

e.g. 6K RPM, 200 sec/track, 512B/sec

10ms/rot x 1/200 = 10/200 = .05ms

- transfer rate (bytes/sec)

access time: 10ms + 5ms + .05ms

Burst Transfer Rate

= (RPM/60)\*(sec/track)\*(bytes/sect)

Sequential vs Random I/O

- read 3 seq blocks = 3\*.05ms = .15ms

- 3 rand. blocks = .05ms (1st), 2\*(10ms+5ms+.05ms) = 30.15ms

Data Modification

- byte level not allowed (by blocks)

Abstraction by OS

- seq. blocks: don’t need to worry about head, cylinder, sector

- access to non-adja blocks – rand. IO

- access to adjacent blocks = seq I/O

Buffers, Buffer pool

- cache for disk blocks

**Files: Main Problem**

- how to store files in disk?

Spanned vs Unspanned:

- unspanned – extra space wasted

- spanned – “fill later”

- unspanned guarantee’s full tuple inside, worst case is waste 50% each block

**Variable-Length Tuples**

Reserved Space

- reserve max space for each tuple (problem: waste of space)

Variable Length Space

- pack tuples tightly

- end of a record by EOR (mark at end)

- or read length at the beginning

Slotted Page – array of pointers

Long Tuples – spanning or splitting tuple

Sequential File – tuples ordered by certain attribute

Sequencing Tuples – 2 options

- rearrange, linked list

- insert when full? Overflow page

**Index**

Dense, Primary Index

- dense: 1 index entry for every tuple (key, pointer) pair for every record

- primary: order of underlying table is same as index (index on the search key)

e.g. 10m records (900B/rec), 4B search key, 4B ptr, 4KB block, unspanned

How many blocks for table (how big)?

4096/900 = 4, 10M tuples/(4 tuples/blk) = 2.5m blocks/ (2.5blk \* 4KB/blocks) = 10 GB

e.g. how many blocks for index (how big)?

8 bytes/entry, 4096/8 = 512entr/ block, 10M/512 = 19531.25=19532, 19532\*4KB = 78MB (for index alone)

Sparse, Primary Index

- sparse: (key, ptr) pair for every block

- points to first record in block

Multi-level Index

- sparse (2nd lvl) -> 1st lvl -> sequential

Secondary (non-clustering) Index

- when tuples in table are not ordered by the index search key

- first level is always dense, sparse from the second level

Terms:

- search key primary key

**Insertion**

- overflow (a new bucket)

- redistribute

Traditional Index

- advantage: simple, sequential blocks

- disadvantage: bad for updates, ugly over time

**B+ Tree**

- advantage: suitable for updates, balanced, min space usage guarantee

- disadvantage: nonsequential index blocks

Leaf Node:

- left of # points to tuples, right pts to next leaf

Non-leaf Node

- left of # points to lower level (right side is ) (at least half pointersused)

**Insertion**: simple, leaf overflow, non-leaf overflow, new root

- non-leaf overflow: split, with new leaf node, push it up to the root

**Deletion**: simple, leaf (coalesce/redistribute with neighbor), non-leaf, (coalesce/redistribute with neigh.)

- redistribute (update the parent w/ lowest value)

***n?*** -*n* determined by size of node/search key/ index ptr

Q: 1024B node, 10B key, 8B ptr => 8*n* + 10*n* – 10 , 18*n* 1034, *n* 57.44, *n*=57

**Remember:**

- for leave node merging, we delete the mid-key from the parent

- for non-leaf node merging/redistributing, we pull down the midkey from parent

Number of Ptrs/Keys:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | MaxPtrs | MinKeys | MinPtrs | MinKeys |
| non-leaf non-root |  |  |  |  |
| leaf , non root |  |  |  |  |
| root |  |  |  |  |

**Views and Authorization**

**Views**

- CREATE VIEW <name> AS <query>

- can be created on top of other viewsb

\* tuples created on the fly using real tables

- used to save complicated tables

\* virtual database: views

\* conceptual database: tables

\* physical database: pages on disk

Modifying Views:

- possible, but must make sense

\*SQL2 : attr. Of T not projected in view allowed to be NULL or default, subqueries in WHERE must not refer to T, no aggregation

\* SELECT statement cannot have:

- a join of 2 or more tables

- an aggregate function or GROUP BY clause

- the DISTINCT keyword (or “UNIQUE”)

- the UNION keyword

\* WHERE clause must be in the SELECT

CHECK OPTION:

\* CREATE VIEW…AS…WITH CHECK OPTION

\* check INSERT/DELETE to ensure new tuple is still in view

\* reject if statement is not

DROP … [CASCADE | RESTRICT]

(dropping a view in between 2 tables)

\* CASCADE default (destroys above)

\* RESTRICT: drop statement fails if object is referenced

Materialized Views

- to be more efficient

- updates mean we have to create a new1

- refresh is costly

**Authorization**

- GRANT <privileges> ON <R> TO <users> [ WITH GRANT OPTION ]

<privileges>: SELECT, INSERT (separated by commas, or ALL priv.)

<R>: table, view

<users>: list of users/groups, or PUBLIC

Managing privileges:

- DBA (owns all)

WITH GRANT OPTION

- users can grant the same or less priv.

Authorization graph

\* nodes: users

\* edges: 1 grants perm. to 2

REVOKE <privileges> on <R> FROM <users> [ CASCADE | RESTRICT ]

- restrict is default

Views and privileges

- to create a view, user needs min. SELECT privilege on the underlying base tables of the view

- to grant a privilege on a view, user has to have GRANT OPTION privilege on the base tables of the view

- when privileges needed for the base tables are revoked, the affected views are automatically dropped