* Naming and Renaming

- $\rho_{S(A_1, A_2, \ldots A_n)}(R)$
  - S: new name
  - $A_1, A_2, \ldots A_n$: attrs

* Quotient

- $T = R \div S$

$T = \{ t \mid t \in \pi_x(R) \text{ AND } \forall s \in S (t_s \in R)$

π, σ, ρ > $\times, \bowtie_\theta, \bowtie$ > $\cup, \cap, -$

(unary) (multiplicative) (addictive)

$(\cup, -, \sigma, \pi, \times, \rho)$

independent set

Precedence

Natural Join

- $R \bowtie S$
- same attrs
- join all, eliminate shared attrs

Cartesian Product

- $R \times S$
- pair all possible combination

Selection (eliminates rows)

- $\sigma_C(R)$
- C is condition

have same schemas with identical pairs

same order

Projection (eliminates columns)

- $\pi_L(R)$
- L is list of attrs of R

compability condition

R - S

Difference

Combining Operations

Theta-Join

- $R \bowtie_\theta S$
- take R×S, choose only tuples matching θ
- θ: equation with attrs

Operations

set operations

* eliminate duplicate

union

$R \cup S$

intersection

$R \cap S$

* semijoin

- $R \ltimes S$

R's tuples that agree with at least 1 in S on all shared attrs

Database

- a collection of data managed by a DBMS

Constraints

1) equal-to-the-emptyset

$R \subseteq \phi \approx R = \phi \approx$ no tuples in result of R

2) set-containment

$R \subseteq S \approx R - S = \phi$

every tuple in R must also in S

Referential Integrity Constraints

- object referenced by other object must exist

any value v in attr A of R must appear in attr B of S

$\pi_A(R) \subseteq \pi_B(S)$

Relational Algebra

Operands: relations

- relation names (variables)

Operators

= set op + special rel op

- constants (set of tuples)

Data Model

- notation to describe data seen by user
- include 1) structure of data (e.g. string) 2) operations 3) constraints } on data

Domains

- atomic values of each attr
- notation R(A1:D1, A2:D2)
- only 1 per relation
- no NULL

PRIMARY KEY

* key example

Movies(title, year, genre)

no two movies will have same value for both title and year

no 2 tuples with same value

UNIQUE

- several per relation
- allow NULL

Key

- set of attrs form a key for a relation if no 2 tuples in a relation instance have same values in all attrs of the key

attributes = column headers

Relational Model

data representation

Relation instance

- a set of tuples
- * DB maintains only current instance

Relation schema

- relation name and set of attributes
- written as: relation_name(attr_list)

tuple = row

Querying / Modifying

ADD col

DROP col

Definition: CREATE TABLE name(

column_1 type_1

);

Relation in SQL

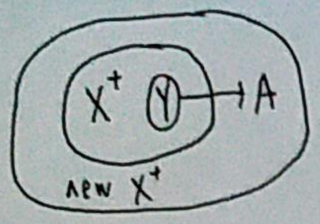
stored relations: tables

Views

- defined by computation

Temporary tables

- constructed by processor
- not stored



Design Theory for RDB

Functional Dependency

- $X \to Y$ is a FD on R if when 2 tuples of R agree on all attrs of X
  - they must agree on all attrs of Y
- Y is functionally dependent on X / X functionally determines Y

Trivial Dependency Rule

If $X \to Y \Rightarrow X \to (Y - X)$

- completely nontrivial: $\forall A_i \notin X$, ex: $A \to B$
- nontrivial: some $A_i \notin X$, ex: $A \to AB$
- trivial: $\forall A_i \in X$, ex: $A \to A$
  - holds for every instance

Notation

X is set of attrs

A_i = single attr

$X \to A_1 \dots A_n$ ⟷ $X \to A_1, \dots, X \to A_n$ (splitting / combining)

follows from

- S follows from T if every relation instance that satisfies T also satisfies S

equivalent

- follow froms each other

Keys of Relations

- set of attrs $\{A_1, \dots, A_n\}$ is a key if
  1) it FD all other attrs
  2) no proper subset of it FD all other attrs (minimal)

superkeys (superset of key)

- a set of attrs which contains a key
- all keys are superkeys
- superkeys are not minimal

Closure of Attrs

- $\{A_1, A_2, \dots, A_n\}$ = set of attrs
- S is a set of FD

closure $\{A_1, A_2, \dots A_n\}^+$ = expand $\{A_1, A_2, \dots, A_n\}$ by adding RHS of FD

1) make FD in single RHS form
2) $X^+ := X$
3) find FD $Y \to A$ in S: $Y \subseteq X^+, A \notin X^+$ ← Algorithm; $X^+ = X^+ \cup \{A\}$
4) repeat 3 until X^+ is maximal

Closure Test

a) check if $X \to A$ follows from S

1) compute X^+ under S
2) Test: $A \in X^+$
   - true: $X \to A$ follows from S

b) check if $\{A_1, \dots, A_n\}$ is key

$\{A_1, \dots, A_n\}^+$ contains all attrs of R ⟷ $\{A_1, \dots, A_n\}$ is a superkey

1) FD all other attrs
2) minimal: $\forall i, S = \{A_1, \dots, A_n\} - \{A_i\}$, S^+ does not contain all attrs of R

c) find closing sets of FD
- only keep FD with single RHS

Algo
1) compute X^+ for each subset X of attrs of R_1
* if $\{X\}^+$ includes all attrs, skip superset of X
2) add all $X \to A$ to F_1 such that $A \in (X^+ - X) \cap R_1$
3) F_1 is a basis but not minimal
a) remove FD in F_1 that follows from other FD
b) remove $XB \to A$ by $X \to A$

Algo
1) split RHS
2) try removing FD
3) try removing attr from LHS
all FD have singleton RHS

d) Find a Minimal Basis
- no proper subset is still a basis
- no attr on LHS is extra

basis: T is a basis for S if $T^+ = S^+$

Redundancy
- unnecessarily repeated info

Anomalies
- deletion anomalies
  - if delete a set of values, may lose other info
- update anomalies
  - several same info in DB
  - may change one but forgot others

e) Projecting FD
$R_1 = \pi_L(R)$
F_1 = projection of F on R_1
F_1: 1) follows from F
2) involves only attr of R_1

Design for RDB 2

Chase to FD $X \to Y$
1) start with a tableau with 2 rows agree only on X
2) apply FD
3) if final table agrees in all columns of Y, $X \to Y$ holds

* satisfies ①, ②

Boyce-Codd Normal Form BCNF
- R is in BCNF
- $\forall$ nontrivial FD $X \to Y$, X is a superkey for R

Decomposition Algo
1) find BCNF violation $X \to Y$
2) compute X^+ to augment RHS
3) decompose R into X^+ and $(R - X^+) \cup X$
4) recursively decompose

$4NF \subseteq BCNF \subseteq 3NF$

	3NF	BCNF	4NF
eliminate redundancy due to FD	M	✓	✓
due to MVD	X	X	✓
preserve FD	✓	M	M
preserve MVD	M	M	M

Properties of Decomposition
1) elimination of anomalies
2) recoverability of info (loseless join)
- can recover original R
3) Preservation of Dependencies
- can recover original FD

The Chase Test
1) draw a tableau, every row (a unsubscripted if R_i contains a)
2) decompose FD so RHS is single
3) apply FD on table: unsub → sub
sub → unsub
4) if a row is all unsub, loseless join

Loseless Join
- projection of tuples can be joined again by natural join to form original tuples

* MVD rule

if MVD $X \to\to Y$

and any FD with RHS $= Z \subseteq Y$

↓

$X \to Z$

ex: $A \to\to BC$

$D \to C$

↓

$A \to C$

1) start with tableau of $X \to\to Y$ and find target with all unsub

2) apply FD

3) apply MVD to create new rows by swapping

Chase to MVD

$X \to\to Y$

- If $X \to\to Y$ and $Z \subseteq XUY$ then $X \to\to Z$

Complementation Rule

- $X \to\to Y$ if
  - whenever 2 tuples of R agree on X
  - we can swap their Y and get 2 new tuples in R

$X \to\to Y$ if $Y \subseteq X$

Multivalued Dependencies MVD

FD promotion

- if $X \to Y$, $X \to\to Y$

Reasoning

no splitting /combining

transitive

Trivial MVD

$X \to\to Y$ if $R = XUY$

Design for RDB 3

- ∀ nontrivial MVD $X \to\to Y$ ↔ - X is a superkey
- R is in 4NF

Fourth Normal Form 4NF

↓

Third Normal Form 3NF

* satisfies ②, ③
- R is in 3NF

↕

- ∀ nontrivial FD $X \to Y$ either a) X is a superkey or b) $A \in Y - X$ is prime

* prime : it is a member of some key

Algo

1) find 4NF violation $X \to\to Y$
2) decompose R → R1(X,Y)
   → R2(X,Z) $Z = R - (XVY)$
3) find FD/MVD on R1, R2
4) recursively decompose R1, R2

Projecting MVD

$X \to Y$, same col → Y

produce unsub row for all attrs of R_i

- If $R \to R_i$, test every FD/MVD for R_i
- not exhaustive : a) don't check FD/MVD with LHS not given
  b) don't check trivial FD/MVD
  c) only check FD with single RHS

→ Algo

1) find minimal basis G
2) for each FD $X \to A$ in G, use XA as schema of decomposition
3) if none of the set of relations from step 2 is a superkey for R add another relation whose schema is a key for R

High-level Database Models

- The Entity/Relationship Model

E/R

3 principal element types

- entity sets ES
  - collection of entity
  - " abstract object
- attributes
  - properties of entities
- connection between ≥2 ES
  - relationships

Design Principles

- reflect reality
- Faithfulness
- avoid redundancy
- pick right kind of elements
- simplicity
  - avoid unnecessary element
- choose the right R
  - avoid deduced R

condition to replace ES

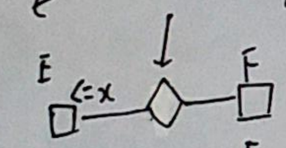
1) all R in ES have arrows enter ES
2) if ES has >1 attrs, no attr depends on other
3) no R involves ES more than once

how to replace ES

a) if many-one R from F→E
   - remove R, attr of E → attr of F
b) if multiway R $\xrightarrow{\text{arrow}}$ E
   - remove R→E, attr of E → attr of R

Constraints

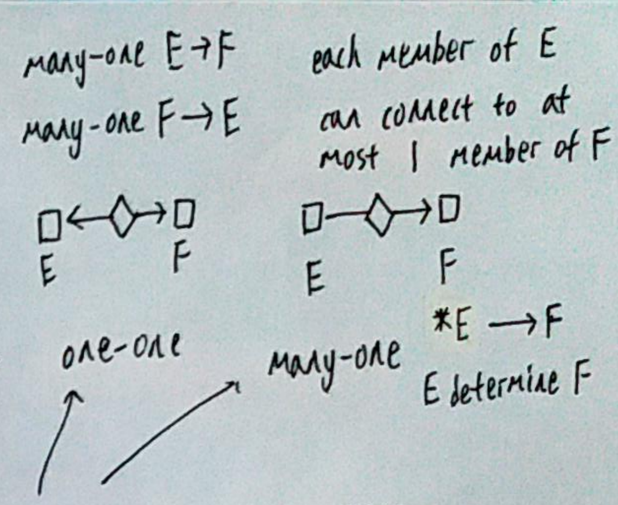
- Degree Constraints
  - limits on no. of E connected
  - arrow ⟹ <=1
  - ⟹ =1



F can have <=x E

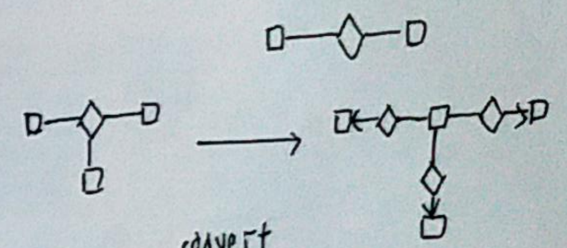
- Referential Integrity
  - E → F
  - if E exists, F must exists
  - in isa
    - root has all attr of key
    - subclass inherit key of root
- Key (underline)
  - set of attrs such that no 2 E agree on the set
  - if >1 key, choose primary key, underline
  - every ES must have a key

Multiplicity of Binary E/R



Multiway Relationships

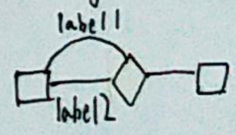
- ≥3 ES connected by 1 R



1) create new connecting ES as tuples of R
2) create many-one R from connecting ES to original ES

Roles

- multiple lines between same ES and R
- label edges with roles



Attributes on R

- attr functionally determined by the relationship set

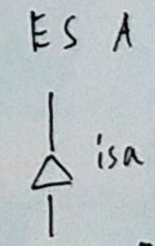
or

create a new ES

- attach to original R and attach the attr

Subclass (one-one)

- special case with ↓ entities, ↑ attr



* B is a A

- B has all attr of A + its own attr

Comparison of issues

1) queries involves several relations
   NULL ✓ , other 2 depends on query type

2) no. of relation
   null < E/R < OO

3) minimize space
   - attr = full key of WES

OO : 1 tuple per entity, with min attr

E/R : several tuples per entity, only key repeated

NULL : 1 tuple per entity, but all attr

b) 1 relation for each relationships that WES participates

* no relation for SR, redundant

Weak ES → relation

a) 1 relation for WES
- attr = a) attrs of WES
  b) key attrs of SES

From E/R to Relational Design

Combining Relations
- many-one R from E → F

combine R + E
- attr = a) all attrs of E
  b) key attrs of F
  c) attrs of R

ES → relation
- ES attr → relation attr

Relationship → relation
- attr of relation
  ||
  1) keys of connected ES
  2) attr of relationship

a) Binary R / Multiway R
- 1 separate relation
- attr = key A + key B + R attr

b) Multiple Roles for an ES A
- multiple use of key A
- rename attr

c) NULL
↓
put NULL in attr that don't belong to them

only 1 relation for isa hierarchy

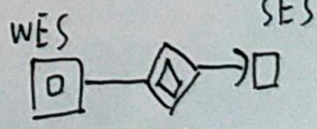
subclass → relation

E/R Model 2

→ Weak Entity Set
- ES's key comes from linked ES

notation
WES ▣
- WES ▣
- SR ◇

Key of WES
||
≥ 0 own attrs
+
attrs from supporting ES



a) E/R style
- → 1 relation for 1 ES
- → 1 relation for 1 R

no relation for isa

→ causes → a) isa hierarchy

b) Object Oriented Style

eliminate same relation

1 relation for 1 possible subtree

if ES ≠ root, include key attr of root

→ b) connecting ES
- from conversion of multiway to binary
* key attr come from connected ES

Requirements of WES

1) R is binary, many-one

2) R has referential integrity

3) SES supplies its key attr to WES
* SES could be WES, supplied by superclass

4) if different R from WES to SES, each is used to supply a copy of key to WES

Representing R in relation
- 1 relation for 1 pair R
- many-one R (combine class relation on "many side)

Multiway Relationships (only binary in ODL)
- simulate multiway R by connecting class
- ex: class X, Y, Z, relationship R
1) devise class C, object: (x, y, z)
2) 3 many-one R from (x, y, z) to x, y, z

structured types
- multiple structure/ collection Type of attr
class names
Types
Basis
primitive types
- int, float, char, bool, enumerations

- key type T
- range types
- no repeat keys
Dictionary <T,S>
- # no. of elements
- ordered
Array <T, i>
List <T>
- ordered
- can repeat
Collection types (Type T)
Bag <T>
- can repeat
- unordered
Set <T>
- finite set of elements
* no repeat
* unordered

Type of R
- class type/ single use of collection

Object Description Language
- text-based language for specifying the structure of database in OO terms

- add position (represent order)
- same as List
- or 1 attr per element
List
Array
- set of struct with key and range attr
Dictionary
Type constructors
Bag
- add count (count no. of objects)

ODL design → Relational design

- D in C
- set<C> in D
C → D
many-one
- C in D
- D in C
one-one
many-many
- set<D> in C
- set<C> in D
Multiplicity

b) one relation for each set
set-valued attrs
Class
a) one tuple for each value in set
- redundancy
- BCNF/4NF violations

struct of primitives
- expand the struct
- if field name conflict → rename
primitive
<name>(attr1, ..., attrn)

Subclasses
- class C extends D
C is subclass of D
- inherit all properties D
* multiple inheritance
class C extends D, E;
Subclass → relation
- 1 relation for each subclass

Class Declaration
class <classname> {
<list of properties> → Methods
}
attributes
Relationships
→ - Notation: relationship <type> <name>
a) many-one
- class C
one-many / many-many
- collection of class C

- Notation: attribute <type> <name>
- can be struct / collections
Notation: attribute Struct <name>{attr1 n1, attrn, nn} <object name>

Declare keys (optional)
- Notation: class C (key (attr))
- can declare multiple keys

Inverse Relationships
- class A { rel set<B> a inverse B::b }
- class B { rel set<A> b inverse A::a }

Extensional EDB vs Intensional IDB

- relations stored in DB
- only in body

- computed by applying ≥1 Datalog Rules
- can be in body / head / both

meaning of Datalog Rules

a) variable assignment (an assignment make all subgoals TRUE)

b) tuple assignment (consistent tuples make subgoals satisfied)

result is finite and subgoal makes sense ← such that
- every variable that appears anywhere in the rule must appear in some nonnegated, relational subgoals

Safety condition

Query ← collection of ≥1 Rules

result

a) only 1 relation in head = answer

b) ≥1 in rule head, * answer + assist

Rules ← Datalog ← Algebraic and Logical Query Languages → Extended Operators of Relational Algebra → Duplicate elimination $\delta(R)$

Head ← Body

1 relational atom

if

≥1 atoms (subgoals) connected by AND

*infinite and uncharging

$\theta(exp1, exp2)$

Arithmetic Atoms

Atom R(t)
- predicate followed by arguments

conversion from RA to Datalog → Relation: Predicate → schema R(x) .. atom R(x)

attr: arguments

tuple $t \in R$ .. R(t) is real

Predicate R
- take fixed no. of arguments
- return boolean value

*R(a) is true if R contains tuple a

Relational Operations on Bags

-why?: - efficient implementation (no comparison for duplicates elimination)
- some queries use bags

Union
- R ∪ S
no. of t
=
n + m

Intersection
- R ∩ S
no. of t
=
min(n, m)

Difference
- R - S
no. of t
=
max(0, n-m)

Others
- do not eliminate duplicates

steps → 1) R ⋈ S

2) add dangling tuples (tail to pair in 1))

3) pad dangling tuples with null ⊥

$R \overset{\circ}{\bowtie}_L S$

$R \overset{\circ}{\bowtie}_R S$ ← variants → $R \overset{\circ}{\bowtie}_B S$

Outerjoins $R \overset{\circ}{\bowtie} S$

expression involves attrs of R / constants / ari op

new name of results

E → z

- rename x as y
x → y

L consists of → single attr of R

Projection Operator $\pi_L(R)$

Sorting Operator

$\tau_L(R)$: sort R based on L

result = list

Aggregation Operators
- used to summarize values in 1 column

COUNT

MIN/MAX

AVG

SUM

Grouping $\gamma_L(R)$

1) partition R into groups with (a) ← Compute

2) for each group, produce tuple with:
(i) grouping attr
(ii) aggregations result

- group tuples and aggregate within each group
- L: list of elements
$\theta(A) \rightarrow$ new name ← op

a) grouping attr - attr of R which γ applies

b) aggregated attr $\theta(A)$ → attr

- expressed by collection of Datalog rules
  * use IDB predicates in bodies of rules
  Recursion in Datalog

Datalog don't support aggregation and grouping

non-recusive datalog = RA

Datalog x support bag op

recursive datalog > RA

Datalog vs RA

none is Turing complete

- create IDB predicates for intermediate relations
Simulating Multiple Operations with Datalog

Answer(x) ← R(x) AND C2

translate to DNF $\sigma_{C1 \text{ or } C2}(R) \leftrightarrow$ Answer(x) ← R(x) AND C1

* using DeMorgan's law
Complex

Answer(x) ← R(x) AND C1 AND C2

$\sigma_{C1 \text{ and } C2}(R)$

Datalog ≥

Selection

Simple condition $\sigma_C(R) \leftrightarrow$ Answer(x) ← R(x) AND C

Natural Join
R ⋈ S

Answer(x, y, z) ← R(x, y) AND S(y, z) ↔

RA and Datalog

Theta-join
- product followed by selection

Product
R × S
↕
Answer(x, y) ← R(x) AND S(y)

Boolean operations
- R, S with same no. of attrs ∧

Projection

$\pi_A(R) \leftrightarrow$ Answer(a) ← R(a, b)

R - S
↕
Answer(x) ← R(x) AND NOT S(x)

R ∪ S
↕
Answer(x) ← R(x)
Answer(x) ← S(x)

R ∩ S
↕
Answer(x) ← R(x) AND S(x)

*scoping rules

1) an attr belongs to tuple variables in FROM if tuple variable's relation has that attr

2) else, look at surrounding subquery

3) use T.A to make attr A belong to tuple variable T

R CROSS JOIN S (R×S)

JOIN Expressions ← (R ⋈ S) → R NATURAL JOIN S

R JOIN S on θ (R ⋈θ S)

R OUTER JOIN S

1) NATURAL OUTER JOIN

2) ON θ after JOIN

3) LEFT/RIGHT/FULL before OUTER

pad dangling tuples of R ← default, pad both

UNKNOWN

T = 1, F = 0, U = 1/2

1) x AND y = min(x,y)

2) x OR y = max(x,y)

3) NEGATION of v = 1 - v

* pitfall

x <= 18 OR x > 18

if x = NULL, no output

true if s ∈ R

2) s IN R

3) s > ALL R
true if s > ∀v in R

4) s > ANY R
true if s > ∃v in R

true if R is not empty

1) EXISTS R

Relation R operations

Scalar-valued
- used as constant
- produce

Subqueries

1) use in WHERE

2) use in FROM
↓ produce
tuple-valued
(v1, v2)
* no. of components must match

subqueries in FROM
- FROM R, (s-f-w) tuple var

ORDER BY <L>
* at last
can be expr

NULL values
* not constant, cannot explicitly used as operand

operate rules
1) arith op with NULL = NULL
2) compare NULL = UNKNOWN

meaning
1) value unknown
2) value inapplicable
3) value withheld

Structured Query Language

SQL

Simple query
- Principal Form:

SELECT <attr list> → projection
* can do arithmatic operation / constant
↓ renaming
SELECT L as <new name>

FROM <table name>

WHERE <condition>
↓
Selection
* can do ar op

String op
- compare in lexicographic order
- str1 || str2 → concatenation
↓
Pattern matching
string LIKE pattern
= true iff same

match any sequence ≥0 char
↑
string with % or _ → match 1 char
* ESCAPE clause
('x%%x%' ESCAPE 'x'
x% = %

Dates and Times

DATE 'YYYY-MM-DD'

TIME 'HH:MM:SS'
or
TIME 'HH:MM:SS +/- HH:MM' *timezone
↓
ahead/behind

TIMESTAMP 'YYYY-MM-DD HH:MM:SS'

operations: a) <: earlier
b) -: difference

EXCEPT (difference)

INTERSECT

set operations

UNION

1) R1 X ... X Rm
2) σC ( )
3) π A1,...An ( )

3) conversion to RA

Multi-Relation Queries
* FROM R, S, ...
* if same name attr → R.name
↓
Tuple variables = alias
- ≥2 tuples from same relation

FROM R R1, R R2

Interpretion
SELECT A1,..., An
FROM R1,..., Rm
WHERE C

1) nested loops
for each tuple in R1 do
for each tuple in R2 do
:
for each tuple in Rm
if C is true for
t1 ... tm
then evaluate
A1,..., An
↓
tuple

1) assign tuples to tuple variables in arbitray order (parallel)
2) test C
3) if true, → A1,..., An

2) Parallel assignments

N: SET TRANSACTION RW/RO
ISOLATION LEVEL <IL>

- system crash : implicit
- in code explicit
- end successfully commit

ROLLBACK

READ WRITE
- default

READ ONLY

access level

- default : READ ONLY
READ UNCOMMITTED
- can read dirty data

Transaction modes

isolation level

T1 write, no commit
T2 read T1
risk: T2 commit, but T1 roll back

SERIALIZABLE
- default
- repeatable read
- no phantom

READ COMMITTED
- no dirty read
- non-repeatable read
* 2 same query may have different results

REPEATABLE READ
- phantom
* result of insertions

end of transaction

Transactions
- collection of ≥1 op which is 1) atomic 2) executed in serializable manners

T1 read → x, y
T2 insert
T1 read → x, y, z
↓
phantom

- how: commit
- both done / neither done
Atomicity: logical unit of work

step 1 → step 2
↓
fail
- undesirable state

Problems

Serializability
- control sequence of SQL statements

→ concurrent
- efficient but dangerous

→ serial
- safe but inefficient

serializable
- results look like serial
- concurrency control: locking

SQL 2

Full-Relation Operations
- op that act on relation as whole
* SQL = bags (have duplicates)

Aggregation Operators
- SUM, AVG, MIN, MAX, COUNT

COUNT(DISTINCT x)
- counts no. of distinct values in col. x
- used to apply to scalar expr in SELECT

COUNT(*)
- count all tuples in relation

GROUP BY ... HAVING
- selection on groups
- often have agg

rules: 1) agg in HAVING only apply to tuples of the group
2) any attr in FROM may in HAVING, but only grouping attr can be magg in HAVING
* same for SELECT

1) $\theta_C(R \times S \times \dots)$
2) group by A
3) Compute aggregation for each group
4) produce 1 tuple in result for each group

Grouping

GROUP BY <grouping attr>
- such that aggregation applied to each group

Steps
SELECT Aggregation
FROM R, S, ...
WHERE C
GROUP BY A

Database Modifications

INSERT INTO R(A1, ..., An) values (v1, ..., vn)
- create a tuple using v_i for A_i $i \in [1, n]$
* 1) missing attr get default values
2) if providing all attr values, can omit ($A_1, \dots, A_n$)
3) can insert by subquery

DELETE FROM R WHERE C
- if no C, delete all

UPDATE R SET <new-value assignments>
WHERE C
attr = expr / constant

SELECT DISTINCT name
- eliminate duplicates
* expensive

∪, ∩ default: eliminate duplicates
- to prevent: UNION ALL

* if aggregation is used, each element in SELECT must be
a) aggregated
b) appear in GROUP BY

* Order of Clauses
1) SELECT *
2) FROM *
3) WHERE
4) GROUP BY
5) HAVING
6) ORDER BY

* must have

*NULL value
1) ignored in any aggregation
2) treated as ordinary value when forming groups
3) if only NULL in col., most agg = NULL
* 0 for COUNT

- default
- check immediately

NOT DEFERRABLE

- check after each statement

INITIALLY IMMEDIATE

DEFERRABLE
- wait until transaction finish before checking

INITIALLY DEFERRED
- check before commit

Deferring Checking (*circular constraints)

no: a) insert with no values
b) update to NULL
c) set-NULL policy

not-null constr
- name type NOT NULL

attr-based CHECK constr
- attr CHECK(c)
- check when attr get new value

tuple-based CHECK constr
- attrs CHECK(c)
- invisible to other relations

Constraints on attr and tuples

- delete update K→FK=null
3) set-Null (3,4)

* ON DELETE SET NULL
ON UPDATE CASCADE
- delete/update K → FK set default
4) set-default (3,4)

enforcing RI
→ 2) cascade (3,4) - delete/update K→FK
1) default (1,2,3,4) - reject

4) update S(K) on referenced K values

3) delete from S(K) referenced K value

S(K) referenced by R(FK) violations

2) update R(FK) with nonexistent FK value

1) insert into R(FK) with nonexistent FK value

Referential Integrity
- data must make sense across DB, source: Relationship

Foreign Key
- if attr A of R references (PK or unique) key K of another relation S
A = foreign key of R
- any non-NULL in FK of R must appear in referenced K of S

Declaring FK

2) FK>1
FOREIGN KEY (<attr>)
REFERENCES <table> (<attr>)

1) FK is single attr
type name REFERENCES <table> (<attr>)

Types of IC
1) table constraint
a) unique b) referential
c) table check
2) domain
3) assertions

Intergrity constraints IC
- define valid states of DB by constraining values in base tables
a) restrictions on allowable data values
b) structure and type restrictions by schema def

Why?
- catch data entry error
- correctness criteria during updates
- enforce consistency on data

How?
1) DB changes
2) check IC
3a) violated
- use statement stored to change DB
3b) no violated
- nothing

Constraints and Triggers

Modification of constr

Assertions
- bool valued SQL expr that must be true

Triggers

ECA rules
1) event: changes in DB
2) condition: test for trigger
3) action: ≥1 SQL statements

declaration
- CREATE TRIGGER name
{BEFORE|AFTER|INSTEAD OF} event
REFERENCING {OLD ROW|OLD TABLE} AS oldvar
{NEW ROW|NEW TABLE} AS newvar
FOR EACH {ROW} → omitted default = statement level
WHEN (condition) *optional

trigger run for each modified tuple

DROP TRIGGER name

Naming

Altering tables
- name type CONSTRAINT constrname (subquery def)

ALTER TABLE Frame
DROP CONSTRAINT name
ADD CONSTRAINT name (def)
- can only add a tuple-based constr
- must hold for current instances

CREATE ASSERTION (name) CHECK (c)
- condition must be true when created
- checked when mentioned R changes

DROP ASSERTION (name)

Rewriting Queries to use MV

Given: MV V SELECT C_V FROM R_V WHERE C_V
query Q SELECT L_Q From R_Q WHERE C_Q
to replace part of Q by V
conditions:
1) relation in $R_V \subseteq R_Q$
2) $C_Q = (C_V$ AND $C)$
3) If C mentions $A \in R_V$, $A \subseteq L_V$
4) $(L_Q \cap R_V) \subseteq L_V$

if all met
∴ SELECT L_Q FROM V, $R_Q - R_V$ WHERE C

Automatic creation of MV
1) establish workload
2) advisor can generate candidate views:
a) have a list of relations in FROM that is a subset of FROM of at least 1 query of workload
b) have a WHERE that is the AND of conditions that each appear in at least 1 query of workload
c) have a attr list in SELECT that is sufficient to be used in at least 1 query of workload
3) optimizer estimate time with/out MV
4) if benefit of MV positive, create

←

Materialized Views MV
- if view is frequently used, materialize it (store in base table)
- efficiency ↑

CREATE MATERIALIZED VIEW name
AS (subquery)

↓

Maintaining MV
- each time base table changes, need to recompute
- all change is incremental (no need to reconstruct)

↓

periodic maintenance

1) establish query workload
2) specify constraints
3) generate and evaluate candidate indexes
4) index set which ↓ cost is suggested

prob(Q_1) = P_1, prob(Q_2) = P_2
- calculate cost of no index, index A, B, or both
- depends on value of P

Automatic Selection ← Calculating best index

↓

Greedy approach
1) evaluate each I, choose positive
2) reevaluate remaining, max(positive)
3) repeat 2 until no positive

View
- a query that defines a relation without physically creating it
- why: a) hide some data from users
b) make some queries easier to express
c) modularity
* only def of views are stored

↓

create (* can create views from views)

CREATE VIEW name(A_1, ..., A_n) AS query

↓

renaming attrs

CREATE VIEW R(a1, a2) AS query

Indexes
- index on R.A is a data structure to efficiently find tuples of R with value A
- A: index key
* can be any attr/ set of attrs

Useful Indexes → 1) Index on key of relation
- common queries involve key
- useful in join
- at most 1 page I/O

- many tuples but few pages I/O
3) index on attr which tuples are clustered

2) Index on attr which is almost a key (few tuples given value for that attr)

→ Declaring Indexes

CREATE INDEX name ON R(a);

dropping: PROP INDEX name;

→ Motivation (con: mod op ↓ need modify indexes)
- speed up queries (ex: R.V = value / R.A <= value)
- speed up joins (use index to find matching tuples)

→ Modifying View
- updatable view (for simple view, translate mod to table)

↓

requirement
1) SELECT some attrs (* not SELECT DISTINCT)
- enough attrs for insertion into underlying R
(ex: cannot select not null)
2) FROM only 1 R
- R can be updatable view
3) WHERE must not involve R in a subquery

→ view removal

DROP VIEW name
- delete view only
- can work on non-updatable

* Instead-Of Triggers on View
- action of trigger is done instead of the event itself

Cursor usage:

1) DECLARE name CURSOR FOR query
2) OPEN name
3) FETCH FROM name INTO shared_variables
4) CLOSE name * can be open again

modifications
- DELETE FROM ...
  WHERE CURRENT of name

cursor options
DECLARE ... INSENSITIVE
CURSOR FOR
- concurrent change between OPEN...CLOSE is invisible for the cursor
FOR READ ONLY
- can run simultaneously with insensitive cursor

fetch tuples into shared variables one at a time
- use cursor
* a handle assoc with a relation that ranges over all tuples

set-valued SELECT
SELECT A1, A2, ...
INTO :v1, :v2, ...
FROM ... WHERE...

Queries

single-row SELECT
- store single tuple into single shared variable

DECLARE section
EXEC SQL BEGIN DECLARE SECTION;
variable declarations
...
EXEC SQL END DECLARE SECTION;

SQL/HL Interface
- to distinguish SQL with HL
EXEC SQL sql statement

data exchange
- shared variables
v in HL ↔ :v in SQL
*prefixed colon

SQLSTATE
- return code of SQL execution system for HL
ex: '00000' = success; '02000' = no data

- compiled host-language program = module

- a collection of stored procedures/functions

True modules

Embedded SQL

} types → Generic SQL Interface
- each query = module

Modules
- an application program
- SQL agent = an execution of module

The SQL/Host Language Interface

Host language+embedded SQL

DBMS preprocessor

host language + function calls

host language compiler ← SQL library

object code

an authorized user

a current schema

a current catalog

consists of

- coextensive with conn
- SQL op performed while a connection is active

Sessions

DISCONNECT conn_name SET CONNECTION conn_name

drop connection set active

CONNECT TO server name
AS conn_name
AUTHORIZATION (name and password)

Connections → only 1 can be active

manages SQL sessions execute SQL statements manage SQL data

- processor that ≥1 SQL server

implementation
- a processor that executes SQL-statements

1 SQL client
- processor that establish SQL connections with servers

SQL in a server environment

SQL environment
- the framework under which data may exist and SQL op or data may be executed

DBS architecture
- small DB: standalone → 2-tier
- large DB: client/server
  - client: SQL
  - server: query processing

3 tier
1) front end: application clients
   - browser + web servers
   - connect clients to DBS
2) middle-tier: application server
   - execute business logic
   - data → response to user
3) backend: database servers
   - execute queries/mods on DB
   - connection pool: keep connections to DB open

organization structures

schemas
- collection of persistent objects
  "
  (base tables, views, constraints, assertions, triggers)
- basic unit of organization
- CREATE SCHEMA name (element declaration)

mod: DROP ALTER
current: SET SCHEMA name

catalogs
- collection of schemas,
- basic unit for supporting unique, accessible terminology
* name of schemas within catalog is unique
  "
  {INFORMATION_SCHEMA, S1, S2, ...}

current: SET CATALOG name

clusters
- collection of catalogs
- max scope of query
- DB seen by users

Usage of PSM proc
anywhere SQL statements can appear
- embedded SQL
- from PSM code itself
- from SQL issued to the generic interface

DECLARE <name> CONDITION
FOR SQLSTATE <value>

↳ in addition, PSM function can be used as part of an expr

- leave BEGIN... END block, execute after the block

- same as EXIT, undo any changes to DB/local var in the block

UNDO

EXIT

where to go next

CONTINUE
- execute next statement

Exceptions
- Exception handler (code invoked when error appears)

REPEAT loops

```
REPEAT
  (statements)
UNTIL (condition)
END REPEAT;
```

WHILE loop

```
WHILE (condition) DO
  (statements)
END WHILE;
```

DECLARE (where to go next) HANDLER
FOR (condition list)
statements

declared conditions or SQLSTATE form expr

FOR Loops
- only used to iterate a cursor
- handles all details of cursor usage
- attr of query are treated as local

```
FOR (loop name) AS (cursorname)
  CURSOR FOR
    (query)
  DO
    (statements)
END FOR;
```

```
LOOP
  statements
END LOOP
```

* if loop is labelled, can break out the loop
LEAVE looplabel

- usually involves cursor fetching tuples

Loops in PSM

Cursors are used as in embedded SQL

* with INTO local var/para a single-row SELECT is legal

Queries in PSM

→ queries that return single value can be used in assignments

subqueries can be used anywhere as in SQL

PSM
Persistent Stored Modules
- write procedures/function and store them in DB as part of DB schema
- can be used in SQL queries or other statements

create
- Modules: collection of func/proc def, temporary relation declarations

```
CREATE PROCEDURE (name) ((parameters))
    (local declarations)
    (procedure body);
```

mode-name-type
IN(d), OUT, INOUT

```
CREATE FUNCTION (name) ((parameters))
RETURNS (type)
    (local declarations)
    (function body);
```

only IN mode, omit

Simple Statements Forms

1) call statement
CALL (name) ((arg list));
* only procedure can made from → generic SQL Interface

host-language program
EXEC SQL CALL name(...);

another PSM func/proc

2) return statement
* only in function
RETURN (exp);
* do not terminate function

3) declaration of local variables
- value not preserved
- must precede executable statements

DECLARE (name) (type)

4) SET (variable) = (expr)
assignment statement

can be NULL/query
* return single value

5) Statement groups

```
BEGIN
  local declarations
  statement list
END;
```

* treated as single statement
- can appear anywhere as if single statement

6) Statement Label
label: statement
* for loop LEAVE

7) Branching Statement

```
IF c1 THEN
  statements
ELSEIF c2 THEN
  statements
ELSEIF...
ELSE
  statements
END IF;
```

optional

* c1, c2 like in WHERE
- statements no need BEGIN.. END

create environment handle

```
SQLAllocHandle(
SQL_HANDLE_ENV,
SQL_NULL_HANDLE,
&myEnv);
```

→ create connection handle

```
SQLAllocHandle(
SQL_HANDLE_DBC,
myEnv, &myCon);
```

→ connect to DB server

```
SQLConnect(myCon,
sqlserver, SQL_NTS,
userid, SQL_NTS,
pword, SQL_NTS);
```

→ create statements handle

```
SQLAllocHandle(
SQL_HANDLE_STMT,
myCon, &myStmt);
```

→ prepare SQL statement

```
SQLPrepare(myStmt, st, sl);
```

→ Execute

```
SQLExecute(myStmt)
```

```
SQLBindParameter(handle, position of
para, SQL_PARAMETER_MODE_IN,
SQL_INT, SQL_INT, 4, 0, &v, 4, &ind_v)
                                  ↑ indicator
```

Call-Level Interface

CLI

API: use HL program to access DB

- SQL statements as string para of functions in API
- system independence

↓

SQL/CLI * import sqlcli.h

4 records:

1) Environments: created by app for preparing ≥1 connections
2) Connections: connect app to DB server
3) Statements: pass SQL statements to the DB server
4) Descriptions: info of tuples/paras

→ records represented by handle (pointer to record)

rt = SQLAllocHandle(hType, hIn, hOut)

1) hType = type of Handle
   SQL_HANDLE_ENV, DBC, STMT
   ↕ create
   handle of type SQLHENV, SQLHDBC, SQLHSTMT
2) hIn = input handle = higher-level element
   SQL_NULL_HANDLE > env > DB[> STMT
3) hOut = address of handle
4) rt = return value of type SQLRETURN → =0 ≈ no error, !=0 ≈ error

Passing Parameters to Queries

steps

1) use SQLPrepare() to prepare statement with parameters with some placeholder(?)
2) use SQLBindParameter() to bind values to parameters
3) use SQLExecute()
   * if change values of para, needs to call again

processing statements

a) SQLPrepare(sh, st, sl);
   - prepare statements
   statement handle, pointer to string, length
   * can use SQL_NTS to set as NULL

b) execute

```
SQLExecute(sh);
```

combined a+b
- without generating a query plan

```
SQLExecuteDirect(sh, st, sl);
```

Fetching Result Data

Bind variables to tuple components

```
SQLBindCol(sh, colNo,
colType, pVar, varSize, varInfo)
```

- colNo: no. of comp in tuple
- colType: type of variable
- pVar: pointer to var
- varSize: length(B) of value of pVar
- varInfo: pointer to an int used by SQLBindCol to provide info

SQLFetch(sh)

Fetch data

* SQLGetData - only copy data once

- return a value of SQLRETURN
  - 0 = success
  - !0 = error
  - SQL_NO_DATA
    - no tuples were left in the query result
    - * used to end loop

$pQuery = $myCon->prepare("...(?,?)");
$args = array('007','bond');
$result = $myCon->execute($pQuery, $args);
*execute by using args as value of (?)

$myCon = DB::connect("<vendor>://<user name>:<password>@<host name>/<database name>");

$myCon->disconnect();

$result = $myCon->query(str)
$tuple = $result->fetchRow()

making a connection

PEAR DB library
include(DB.php);
→ invoke with DB::connect(...);

Dynamic SQL in PHP
- allow placeholder and binding

<?php PHP code ?>
PHP

→ arrays
- associative (mappings)
  - indexes (keys) by strings
  - key ↑ x => y → value
  - a[x] = y
- numeric
  - indexed 0, 1, ...

- function 'fetchRow' applies to the result object and returns the next tuple, or 0 if no next tuple
- each tuple is a numeric array

Cursor in PHP
- result object of the query is a list of tuples returned

variables
- start with $
- ok to x declare type
- object variable: create from class
  $obj = new myClass();
- call variables / functions
  $obj->myVar
  $obj->myFunc()

String values
() single quotes = string
(( )) double quotes
" " replace variable by value

Executing SQL
$rs = $myCon->query(str)
- function 'query' applies to connection obj
- take a string arg, return a result

Parameter from web Page
- when user fill into
- get into from associative array
$_POST

Parameter Passing
- string argument of preparedStatement() can contain dynamic (?) placeholder
- placeholder must be given value before execution, using setXXX()

myPstmt.setString(i,v)
↓
replace ith "?" with v

Cursor
- getXXX(i)
  - return ith component of tuple pointed by cursor
- next()
  - fetch next row
  *return false if no more tuple

API ↓

Java Database Connectivity
JDBC (API for Java)
1) import java.sql.*;
2) load a driver
   Class.forName("<driver name>");
3) establish a connection to DB
   Connection myCon;
   myCon = DriverManager.getConnection(url, uid, pwd);
   myCon.close()

app → driver manager → driver → driver → DBMS

4) Execute a prepared mod/DDL statement
myPstmt.executeUpdate();

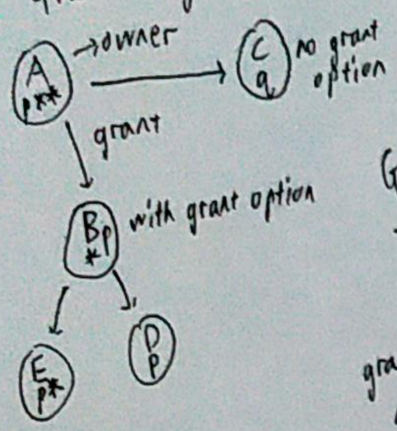
3) execute a prepared query
ResultSet rs = myPstmt.executeQuery();

→ 4 methods to execute SQL → 2) prepare and execute mod or DDL statement U
mystmt.executeUpdate(U);
*return no of rows affected

4) prepare and execute Q
ResultSet rs = mystmt.executeQuery(Q);

creating statement
1) Statement mystmt = myCon.createStatement();
2) PreparedStatement myPstmt = myCon.createStatement(Q);
↓ string

Grant Diagrams q∈P



GRANT privileges ON DB-element
TO users [WITH GRANT OPTION]

- privileges: q types or ALL PRIVILEGES
- DB-element: relation or type or assertion
- optional

granted with grant option
can grant ⊆ privileges
to others

Granting privileges

Revoking privileges

GRANT OPTION
or
REVOKE privileges ON
DB-element FROM users
[CASCADE|RESTRICT]

- CASCADE: revoke all privileges granted by removed; delete any node not accessible from owner
- RESTRICT: cannot execute if result in cascading

Recursion

define in SQL

WITH R AS <def of R>
<query involving R>

WITH
[RECURSIVE] R1 AS query1
...
[RECURSIVE] Rn AS queryn
<query involving R1,...,Rn>

ss
1) compute R1,...,Rn
2) evaluate <query involving R1,...,Rn>
3) Destroy R1,...,Rn

Recursive rules in Datalog

1) start by assume all IDB are empty
2) repeat evaluate rules using EDB and previous IDB to get new IDB
3) end when no change to IDB

* legal SQL Recursion
2 rules (see example)

Security and User Authorization in SQL

2 aspects: a) user only see allowed data
b) guard against malicious user

how?: a) authorization ID
b) privileges

AU ID in session
- session provides AU ID a context to execute SQL
- a session is associated with a user ID or a role name
- on session initialization session uid determined by: a) explicit CONNECT TO... USER usr;
b) implementation-defined manner

SESSION_USER : session user ID
CURRENT_USER : current user ID → SET SESSION AUTHORIZATION...
CURRENT_ROLE : current rolename → SET ROLE...

Privilege Checking

- if agent A operates on DB element, privilege of A derive from current AU ID which is → a) AU ID of module if has module
or
b) session AU ID

principles

1) pri always available if data owned by U and U = current AU ID
2) pri available if current AU ID U has been granted by owner, or priv granted to PUBLIC
3) execute module owned by owner of data, or data owner grant→ to module owner
4) execute a publicly available module during session whose AU ID is user with pri

Creating privileges
- owner can grant privileges

1) when a schema is created
CREATE SCHEMA ...
AUTHORIZATION usr
↓ owner

if define in a module, module = owner
otherwise owner = session uid

2) when initiating a session
CONNECT TO svr
AS conn
AUTHORIZATION usr;

3) when creating a module
MODULE modname...
AUTHORIZATION usr

Privileges (associated with AU ID)

q types
- applied to a relation: SELECT, INSERT, DELETE, UPDATE
- the right to refer to relation in IC: REFERENCES
- the right to use schema elements (except relations/assertions) in declaration: USAGE
- the right to create subtypes of a given type(UDT): UNDER
- the right to define trigger: TRIGGER
- the right to execute codes: EXECUTE

generated unique by DBMS ← - used to reference the table

use primary key of table ← Reference Column ↖

Declaration of OID for tables ← A REF(T) SCOPE R ← if not specified, can applied to any table

∴ attr A references to R, R is table with type T

CREATE TABLE (table name) OF (type name) REF IS (attr-name) (how-generated)

if T is UDT

REF(T) is type of reference to type T

SYSTEM GENERATED or DERIVED

x is value of type REF(T)

a) x → a (attr a of t)

b) PEREF(x) (=t)

Following References ←

t.x(v)
- set v to attr x of tuple t
Mutator

References ↖

observer method x() for attr x ←

- t.x() is value of x in tuple t

*() is omitted in practice

Accessing Components of Types ←

Operations on OR data ↙

generator()
- T() return empty object of type T

Ordering Relationship on UDT ↓

Tuple ID = identifiers of tuples ↙

a) strict object equality
- equal iff same object

b) method-defined equality
- function(x1, x2) return T/F

c) method-defined mapping
function(x1) → real number
compare numbers

* if 'ORD FULL' → 'EQ ONLY'
F(x1,x2) = 0 → x1 = x2
≠0 → x1 ≠ x2

CREATE ORDERING FOR T ORDERING FULL BY RELATIVE WITH F;
- apply F to compare
F(x1, x2) < 0 ≃ x1 < x2

CREATE ORDERING FOR T EQUALS ONLY BY STATE;
- 2 object of UDT T is equal if all components are equal

subtable vs supertable
TB1/2 has associated structured type TP1/2
TP1 is subtype of TP2
- TB1 is subtable of TB2
- TB2 is supertable of TB1

Typed Table ←
- table declared to be based on structured type
- have 1 self-referencing col, with type (ref type to structured type)

CREATE TABLE (name) of (UDT-name) (list of ele); ← key, foreign keys, tuple-based constraints

Declaring Relations with UDT ↖

The Object Relational Model

OR

↓

From Relation to OR → 3) Methods

↓

2) References
A(*R)
↓
A = attr reference to R
A({*R})
A's type = set of references to tuples of R

1) structured types for attrs
- value of components of tuple can be relations
=
nested relations
R(A, B, C(C1, C2, C3))

→ User-Defined Types UDT

2 ways
a) the type of a table
b) the type of an attr belonging to some tables

↘ Define type

CREATE TYPE T AS (primitive type)
- rename primitive type

CREATE TYPE (name) AS (attr-declarations)
[method declarations] - class like UDT

OR vs OO

Type Systems, Methods similar

References and OID	sometimes allow user to see	hide from users
Backward compatibility	can	cannot

CREATE METHOD (method name) (arg1 type1, ...)
RETURNS (return-type) FOR (UDT name)
BEGIN ... END

Definition ↑

Methods in UDT
- need 1 decl + 1 def

Declaration
* inside UDT decl
CREATE TYPE name AS ...
METHOD (name) (arg1 type1, ...)
RETURNS (return-type)

a) key declarations are part of table def, not type def
b) relationship ≠ properties, but distinctions ↑ by separate relation or references

SELECT ...
FROM ...
WHERE ...
GROUP BY ... WITH CUBE
or
GROUP BY ... WITH ROLLUP
in SQL

to store the cube
CREATE MATERIALIZED VIEW
<myCube> AS
(cube generator)

The Cube Operator
- fact table F, augmented table CUBE(F)
  - add a value * to each dim
  - " aggr along the dim
  - have the sum of value for each dependent attr in all tuples, usually replace key

example: ↗ key
Sale (serialNo, price)
↓
Sale (model, color, price)
Sales (*, red, 1000) = sales of all model of red and 1000 price

Multidimensional OLAP
↑ -uses formal data cube

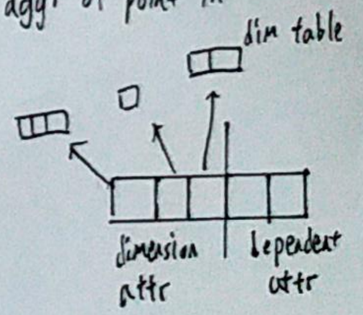
On-Line Analytic Processing
OLAP
- examination of data for patterns
- involve decision-support queries (complex)
- involve large amount of data
*app: identify sales trend
VS
On-Line Transcation Processing
OLTP
- involve little data
- daily transaction

↓ OLAP store in

data warehouse
(separate copy of master)
- not updated frequently
- organized & centralized data to support OLAP
- OLTP on master DB, OLAP on this

Multidimensional view of OLAP data
- fact table (central relation of data): a) dimension attr
  b) dependent attr
  ss
  raw data cube
- formal data cube (precomputes all possible aggr)
  a) includes aggr of data in all subset of dimensions
  b) point in fdc = initial aggr of point in raw-data cube

↓
Relational OLAP
- uses star schema
  center = fact table
  points = dimension tables
* fact table links to points by foreign keys



WHERE
FROM
↑ Slicing and Dicing
- partition into smaller cubes

SELECT <grouping attr and aggr>
FROM <fact table + dim tables>
WHERE <certain attrs are constant>
GROUP BY <grouping attrs>;

↓
a) roll-up
- aggr along ≥1 dim
- fine → coarse

b) drill-down
- break aggr
- add dim

ID type <Student SNo = "007"> </Student>

<Course CNo = "CS123" takenBy = "007" >

a) Documents(docID, strXML)

b) DocRoot(docID, rootElementID)
SubElement(parent, child, position)
ElementAttr(elementID, name, value)
ElementValue(elementID, value)

c) since SQL:2003 xml type

1) store XML data in parsed form
provide library to navigate
2) represent docs and eles as relations, use RDBMS

IDREF type

attrs that connect eles

<element><name>value</name></element>
=
<ele name=value/>

<myns:myTag xmlns: myns = "URI">
Namespace

Attrs

XML in DB

Extensible Markup Language XML

<?xml version = "1.0" standalone="yes"?>
<tags> ... </tags>

well-formed XML
min req: a) doc begin with decl
b) have a root ele = entire body of doc

a) <tags> ele </tags>
b) <tags/> : no ele nested

modes
a) well-formed XML
- invent own tags, no predefined schema
b) valid XML
- DTD (specify tags + grammar)
- inter between strict schema and schemaless

SemiStructured-Data Model

- schema is implied by the data

Info Integration via SD
Problem: how to merge DBs into 1
legacy DB problem: once DB exists, app hard to disentangle
solution: a) build new DB (expensive)
b) intergration

user
↓
interface(support SD)
↙ legacy DB
↘ legacy DB

- allow an attr contains multiple refs IDREFS

IDREF
- refer to eles by ID

Identifiers and References

ID
- unique id to element

Document Type Definitions DTD
- grammar set of rules
a) what tags
b) how nesting

SemiStructured Data
- self describing (data itself carry info of schema)
- adv: flexibility

vs structured data
- separate schema to describe structure
- adv: efficient storage + query process

- Label L from node N to M could be
a) N = object, M = attr, L = name of attr
b) N, M = objects, L = name of relationship from N to M

attrs lists
(!ATTLIST element-name attr-name type default-value>
→ "def-value"

#IMPLIED - optional
#REQUIRED - attr must present
#FIXED

using DTD
1) set standalone = "no"
2) a) include DTD as preamble
or
b) follow xml tag by DOCTYPE decl with root tag + SYSTEM + a file to find DTD

<!DOCTYPE root-tag[
(!ELEMENT name(components))
more eles
]>
root tag surrounds doc to conforms to this DTD

connect list of option by '|' *only 1 option appears

multiplicity of subeles
? 0 or 1
+ ≥1
* ≥0

element decl
(!ELEMENT name (components))
a) list of subelements
b) (#PCDATA) parsed char data → a value of text + no ele nested

Graph Representation
- node = leaf or interior → have ≥ 1 arcs out
↓
associated data of atomic type
- arcs: have a label (how head relates to tail)
- 1 interior node = root (no arcs enter, represent the DB)
*every node reachable from root

XQuery Aggregations
- const, sum, max (take any seq as arg)
* can be applied to result of any expr

XQuery Branching
if (expr1) then expr2 else expr3
↓
use () if no else

every $var in expr1 satisfies expr2
some $var in expr1 satisfies expr2
1) evaluate seq produced by expr1
2) check every item of seq (expr1), evaluate expr2
3) return true if expr2 is true for
↓
Quantification
i) all $var → every
ii) ≥1 $var → some

eliminate duplicates
- distinct-values(seq)
* subtlely: strip tags away

comparison operators: ge, le, eq, ne, lt, gt

Joins in XQuery → but need to use data(element)
- can join ≥2 XQuery docs as SQL

extract value of element

curly brackets
<A>{$x}</A> → $x ('$' considered as string)
<A>{{$x}}</A> → correct

X[T]
- true for X having subeles with tag T

X[A]
- true for X having a value for attr A

X[i]
- true for ith child

conditions

tag[cond]
- only tag-ele that satisfy cond are included in result seq

wildcard
- * = any tag
or @* = any attr

Axes
- modes of navigation
- prefix tag or attr by axis::
↓
child (default)

descendant-or-self (//)
skip from anywhere directly to X

parent (..)

attr (@)

Path Expressions
/T1/T2/.../Tn
- get seq of elements of Tn
↓
Relative Path
- not starting with /, depends on current node

Query Language for XML
↓
XPath
↓
Data Model
- XML doc = tree
- XPath = describe path in XML doc
↓
result = seq of items → a value of primitive type
↓
a node → Documents: file containing XML doc

attrs
- content within opening tag

elements
- XML elements (tag + content)

doc(filename)

XQuery
- XPath ⊂ XQuery
- functional language: any expr can be used in place expecting expr
- Item seq: XQ sometimes form seq of seq, all seqs are flattened

- append expr to seq of items
- may be executed many times
- do not end query

FLWR expr
- start with ≥0 for and let (any order)
optional where
optional order by
return

return expr

- expr evaluated for each assign to vars

order by expr (ascending/des)

where cond
- if cond = true
↓
return applied to current value of any vars

let $var := expr
or
let var1 := expr1, var2 := expr2
- expr evaluated and assigned to var

for $var in expr
- assign each item in result of seq of expr
- execute follows for once for each value of var