CS2102

Integrity Constraints

Key constraints:
1. no two distinct tuples of a relation share the same values in all attributes of the key.
2. no proper subset of the key is a superkey.
3. attributes of the key cannot be null.

Foreign key constraint: each foreign key value must appear as a primary key value in the referenced relation, or be null.

Relational Algebra

Unary operators: selection (σ_c), projection (π_{a_i,a_j}), renaming ($\rho_{a_i:b_i,a_j:b_j}$).

Binary operators: set operators ($\cup, \cap, -$), cross-product ($\times$), (natural) inner join ($R \bowtie_c S$), (natural) outer joins ($R \leftarrow_c S$, $R \rightarrow_c S$, $R \leftrightarrow_c S$).
- set operators require relations to be **union compatible**, i.e., have same number of attributes + attributes have same domains (need not have same names).
- binary operators are left-associative.
- $\times$ is associative.
- useful shortcut: $R/S = \pi_A(R) - \pi_A\left((\pi_A(R) \times S) - R\right)$.

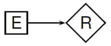
Entity-Relationship (ER) Model

Terminology

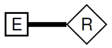
- *Entity*: real-world object distinguishable from other objects.
- *Entity set*: collection of similar entities, represented by **rectangles**.
- *Attribute*: specific information describing an entity, represented by **ovals**.
- *Key*: represented as **underlined** attributes.
- *Relationship*: association between two or more entities.
- *Relationship set*: collection of similar relationships, represented by **diamonds**.
- *Relationship roles*: shown explicitly when one entity set appears ≥ 2 times in a relationship set.
- *Relationship keys*: the primary key of a relationship set consists of the keys of its entities, as well as its underlined attributes.
- *Degree*: a relationship set with degree n (i.e., n-ary relationship set) involves n entity roles.
- *Aggregation*: can be used to model relationships between an entity set and a relationship set by representing relationships as higher level entity sets. This is indicated using a **bold square**.

ER constraints

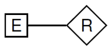
- *Key constraint*: each instance of E can participate in <u>at most one</u> instance of R.



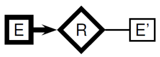
- *Total participation constraint*: each instance of E must participate in <u>at least one</u> instance of R.



- *Partial participation constraint*: each instance of E can participate in <u>0 or more instances</u> of R.

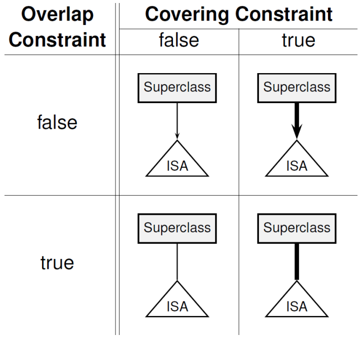


- *Weak entity set*: E does not have its own key, and requires the primary key of its *owner entity* E' to be uniquely identified. There must be a many-to-one (identifying) relationship from E to E', and a total participation on R.



ISA hierarchies

We can classify entity sets into subclasses. Every entity in a subclass entity set is an entity in its superclass entity set. ISA hierarchy constraints include:
- *Overlap constraint*: can an entity belong to multiple subclasses?
- *Covering constraint*: should an entity belong to some subclass?



SQL

Data types: boolean, integer, numeric/float, char(n), varchar(n), text, date, timestamp.

Basic syntax

- **Create table**:
  `create table [if not exists] T ( id integer primary key );`
- **Drop table**:
  `drop table if exists T [cascade];`
- **Modify table**:
```
insert into T values (X, Y);
delete from T [where id = X];
update T set id = id * X [where name = Y];
alter table T add/drop/alter column/constraint X [...];
```
  – if a value is not specified for a column when a tuple is added, the default value for the column would be assigned.
- **Conceptual evaluation of queries**:
```
select      [distinct] select-list
from        from-list
where       where-condition
group by    group-by-list
having      having-condition
order by    order-by-list
offset      offset-spec [asc (default)/desc]
limit       limit-spec
```
  1. Compute cross-product of tables in `from-list`.
  2. Select tuples evaluating to *true* for the `where-condition`.
  3. Partition selected tuples using the `group-by-list`.
  4. Select groups evaluating to *true* for the `having-condition`.
  5. Generate an output tuple for each selected group based on the `select-list`.
  6. Remove duplicate output tuples (if `distinct` is specified).
  7. Sort output tuples based on the `order-by-list`.
  8. Remove output tuples based on the `offset-spec` and `limit-spec`.

  Validity of group by/having (one of the following must hold):
  – Output column A appears in the `group by` clause.
  – A appears in an aggregated expression in the `select/having` clause.
  – the primary key of relation R appears in the `group by` clause.
- **Transactions**:
```
begin;
[set constraints CONSTR_NAME/TRIG_NAME deferred;]
...sql statements...
commit/rollback;
```
  Transactions adhere to the ACID properties:
  – *Atomicity*: either all effects are reflected or none.
  – *Consistency*: executed in isolation, preserves integrity constraints of database.
  – *Isolation*: no interference with other concurrent transactions.
  – *Durability*: effects persists even if system fails.
- **CTEs**:
```
with CTE1 as (...),
     CTE2 as (...),
     CTE3 as (...)
select ...;
```
- **Views**:
  `create or replace view V(A, B) as ...;`
- **Conditional expressions**:
```
case
  when X = Y then A
  else B
end as VAR
---
case X
  when Y then A
  else B
end as VAR
```

Other syntax

- **Schema constraints**:
  `[constraint CONSTR_NAME] CONSTRAINT [deferrable initially deferred/immediate]`
  – possible `CONSTRAINT`s:
    * `unique`
    * `not null`
    * `primary key / unique not null`
    * `foreign key (A, ...) references T [match full] [on update/delete ACTION]`
    * `check ...`
    * `default '...'`
  – <u>`match full`</u>: prevents reference to completely or partially null entries.
  – possible <u>`ACTION`</u>s:
    * `no action`: rejects delete/update if it violates constraint.
    * `restrict`: same as above, but constraint checking cannot be deferred.
    * `cascade`: propagates delete/update to referencing tuples.
    * `set default`: updates FKs of referencing tuples to some default value.
    * `set null`: updates FKs of referencing tuples to `null`.
- **Set operations**:
  `union/intersect/except [all]`
  – `all` preserves duplicate records.
- **Join operations**:
  `[natural] [left/right/full] join`
- **Comparison predicates**:
  `is [not] [null/distinct from]`
- **Subquery expressions**:
  `exists, in, any/some, all, unique`
- **Aggregate functions**:
  `count, sum, avg, min, max`
- **Coalesce**: returns the first non-null value.
  `select coalesce(a1, a2, ...) as A;`
- **NullIf**: returns null if `a1 = a2`, else returns `a1`.
  `select nullif(a1, a2) as A;`
- **Pattern matching**:
  `... where A like '_%';`
  – `_`: matches any single character.
  – `%`: matches any sequence of 0 or more characters.
- **Miscellaneous**:
  – `'a' || 'b'`: string concatenation.
  – `round(A)`: rounding off to nearest integer value.

Functions

```
create or replace function F(<param> <type>, ...)
returns <type> as $$
  ...
$$ language sql;
select F(...); / select * from F(...);
```
- **Return single tuple**:
  `function F(...) returns T as ...`
- **Return single custom tuple**:
  `function F(IN ..., OUT ...) returns <name> as ...`
- **Return $\geq$ 2 tuples**:
  `function F(...) returns setof T as ...`
  `function F(...) returns table(<param> <type>, ...) as ...`
- **Return nothing**:
  `function F(...) returns void as ...`

author: arsatis

Procedures

```
create or replace procedure P(<param> <type>, ...)
as $$ ... $$ language sql;
call P(...);
```

PL/pgSQL

Basic syntax

- **Variables**:
```
declare
  varA integer := 0;
  varB integer;
begin
  select id into varC from T;
  ...
end;
$$ language plpgsql;
```
- **Conditional statements**:
```
if c1 then ...;
elsif c2 then ...;
else ...;
end if;
```
- **Cursors**:
```
declare
  curs cursor for (select * from T where ...);
  var record;
begin
  open curs;
  loop
    fetch curs into var;
    -- assign output column names to values
    return next;
  end loop;
  close curs;
end;
fetch [prior/first/last/absolute n/relative n] from curs into var;
move [prior/first/last/absolute n/relative n] from curs;
[update/delete] <table> ... where current of curs;
```
- **Loops**:
```
loop
  exit when c3;
  ...
end loop;
---
while ... loop
  ...
end loop;
---
foreach val in <array> loop
  ...
end loop;
```

Triggers

```
create or replace function TRIG_F()
returns trigger as $$
begin ... end;
$$ language plpgsql;
---
create trigger TRIG_NAME
[after/before/instead of] [insert/delete/update] on T
for each [row/statement]
[when <condition>]                          -- trigger condition
execute function TRIG_F();
```
- *instead of* triggers can only be defined on views and on the row level.

Special values

- `NEW`: new tuple being inserted or updated. `null` for deletion.
- `OLD`: old tuple being deleted or updated. `null` for insertion.
- `CURRENT_DATE`: current date.
- `TG_OP`: operation that activates the trigger.
- `TG_TABLE_NAME`: name of table causing trigger invocation.

Return values

- `before insert`: `null` = no insertion; else output inserted.
- `before update`: `null` = no update; else tuple updated with output.
- `before delete`: `null` = no deletion; else deletion proceeds as normal.
- `after <action>`: return value does not matter.
- `instead of <action>`: `null` = ignore operations on current row; else proceed as normal.
- *statement-level triggers*: return values ignored, use `raise exception '...'` to omit subsequent operations.

Trigger condition requirements

- No `select` in `when()`.
- No `OLD` in `when()` for `insert`.
- No `NEW` in `when()` for `delete`.
- No `when()` for *instead of* triggers.

Deferred triggers

```
create constraint trigger TRIG_NAME
after [insert/delete/update] on T
deferrable initially [deferred/immediate]
for each row
execute function TRIG_F();
---
begin transaction;
set constraints TRIG_NAME deferred;
update ...;
commit;
```
- Deferred triggers only work with `after` and `for each row`.

Order of trigger activation

1. `before` statement-level triggers.
2. `before` row-level triggers.
3. `after` row-level triggers.
4. `after` statement-level triggers.

Within each category, triggers are activated in *alphabetical order*. Also note that if a `before` *row-level* trigger returns `null`, all subsequent triggers on the same row will be omitted.

Normal Forms

Normal forms help to reduce redundancy in relational schemas.

Functional dependencies (FDs)

$A \to B$: if two rows have the same A, then they have the same B.
- *Reflexivity*: $AB \to A$.
- *Augmentation*: if $A \to B$, then $AC \to BC$.
- *Transitivity*: if $A \to B$ and $B \to C$, then $A \to C$.
- *Decomposition*: if $A \to BC$, then $A \to B$ and $A \to C$.
- *Union*: if $A \to B$ and $A \to C$, then $A \to BC$.

Closures

$\{A\}^+$: denotes the set of attributes that can be decided by A.
To prove $A \to B$, it suffices to show that $B \in \{A\}^+$.

Algorithm for computing closures

1. Initialize the closure to $\{A\}$.
2. If there is a FD such that $A \to B$, we put B into the closure.
3. Repeat step 2 until no new attributes can be added.

Keys

- *Superkey*: set of attributes that decides all other attributes.
- *Key*: minimal superkey.
- *Prime attribute*: attribute which appears in a key.

Algorithm for deriving keys of T

1. Consider all attribute subsets in T.
2. Compute the closure of each subset.
3. Identify all superkeys based on the closures.
4. Identify all keys.

Useful heuristics:
- Check for attribute sets from smallest to largest.
- If an attribute does not appear in the R.H.S. of any FD, it must be in every key.

Non-trivial and decomposed FDs

- *Decomposed FD*: an FD whose R.H.S. has only one attribute.
- *Non-trivial FD*: an FD whose attributes in the R.H.S. do not appear in the L.H.S.

Algorithm for deriving non-trivial and decomposed FDs of T

1. Consider all attribute subsets in T.
2. Compute the closure of each subset.
3. Remove all trivial attributes from each closure.
4. Decompose all FDs (using rule of decomposition).

Boyce-Codd Normal Form (BCNF)

T is in BCNF if every non-trivial and decomposed FD has a superkey as its L.H.S. A table violates BCNF if it does not satisfy the *"more but not all"* condition, i.e., there exists a closure of size k which:
1. contains more than k attributes, but
2. does not contain all the attributes in T.

BCNF properties

(+) no update, deletion, or insertion anomalies.
(+) few redundancies.
(+) original table can be reconstructed from decomposed tables.
(+) guarantees **lossless join decomposition** (i.e., common attributes in decomposed tables always constitute a superkey of either sub-table).
(−) does not guarantee **dependency preservation** (i.e., set of FDs on the original table and set of FDs on the decomposed tables may not be equivalent).

Algorithm for BCNF decomposition

1. Find a subset X of attributes in T whose closure $\{X\}^+$ satisfies the "more but not all" condition.
2. Decompose T into two tables T_1 and T_2 such that:
   - T_1 contains all attributes in $\{X\}^+$, and
   - T_2 contains all attributes in X and all attributes not in $\{X\}^+$.
3. Check if T_1 and T_2 are in BCNF.
4. If needed, recursively decompose either T_1 or T_2.

Third Normal Form (3NF)

T is in 3NF if every non-trivial and decomposed FD either:
- has a superkey as its L.H.S., or
- has a prime attribute as its R.H.S.

Minimal basis (MB)

A *minimal basis* of a set S of FDs is a simplified version of S satisfying **all** following conditions:
1. Every FD in MB can be derived from S and vice versa.
2. Every FD in MB is a non-trivial and decomposed FD.
3. No FD in MB is **redundant** (i.e., can be derived from other FDs in the MB).
4. Every FD in MB does not have **redundant attributes** in its L.H.S. (i.e., if an attribute is removed from the L.H.S., then the FD cannot be derived from S).

Algorithm for minimal basis

1. Decompose all FDs (using rule of decomposition).
2. Remove redundant attributes on the L.H.S. of each FD.
3. Remove redundant FDs.

Algorithm for 3NF decomposition

1. Derive a MB for the set of FDs on T.
2. Combine FDs whose L.H.S. are the same in the MB.
3. Create a table for each remaining FD.
4. If none of the tables contain a key of T, create a table that contains a key of T.

Algorithm for verifying BCNF/3NF satisfiability

1. Compute the closure of all attribute subsets in T.
2. Derive the keys of T.
3. For each closure, check if:
   (a) it satisfies the "more but not all" condition, and
   (b) its additional attribute(s) is not a prime attribute. (3NF only)
4. If such a closure exists, T is not in BCNF/3NF.

Note: Alternatively, for step 3(a), we could check if the L.H.S. of each FD of T is a superkey.