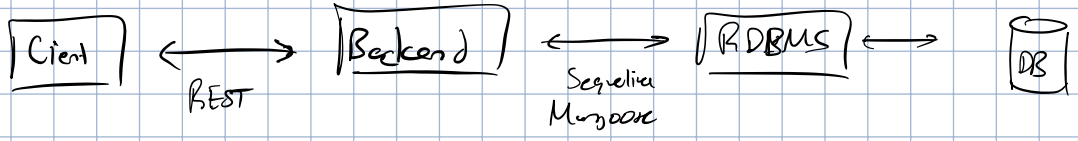
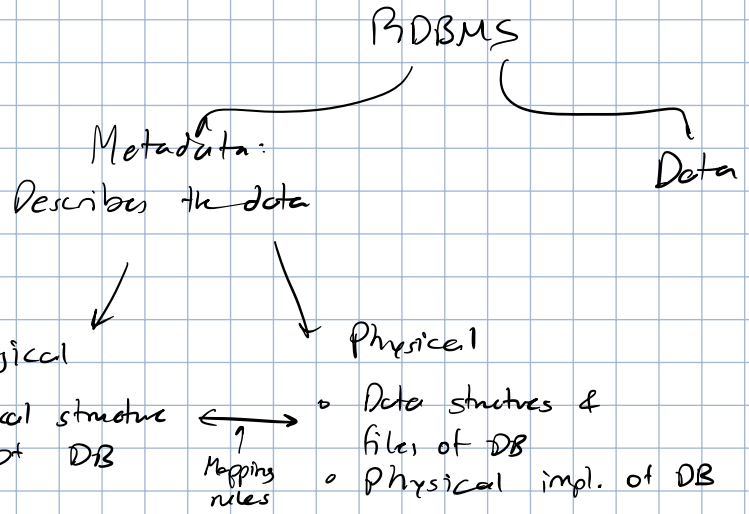
Module 1: Overview of Data Management

Components of RDBMS

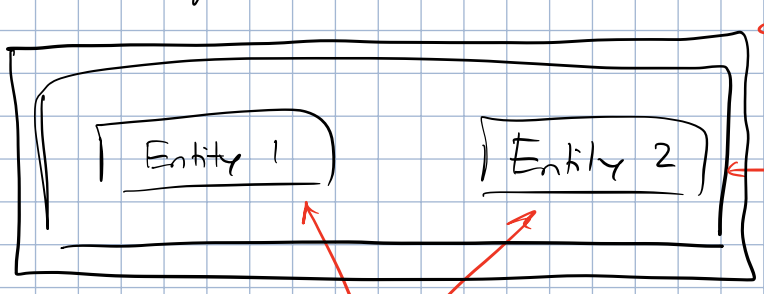
RDBMS: relational DB management system

Typically, architecture in software:





Metadata is composed of schema:



Physical schema: physical metadata.

Logical schema:
Logical metadata for all entities.
- Integrity constraints b/w tables.

External schema
Logical metadata for table.
- Primary keys, foreign keys
- Non-null values

To access DB, we use queries. Criteria for good querying:

1. Indep. of physical structure.
2. Indep. of logical metadata (Logical if violates logical schema)
3. Declarative: shouldn't tell RDBMS how to perform.

RDMS creates model of data ⇒ data model

↳ Model: notion of metadata, how do we retrieve + revise data

How to specify the model?

Data model

Data defn. lang. (DDL)
Manages metadata
(schemas, integrity constraints)
Note: indep. of physical impl.
Each schema is not sharing logic

Data manipulation lang. (DML)
Manage data
Query language (SQL)
Declarative & physically indep.

RDBMS Requirements:

#1: RDBMS should be handle concurrent transactions
- Transaction: indivisible DML requests
- RDBMS should be ACID compliant
  - A: atomic (not dividing transactions, all/nothing)
  - C: Consistnt (always in a good state)
  - I: Isolation (transactions not interfering w/ each other)
  - D: Durable (recover data)

#2: RDBMS should be under access control

#3: RDBMS should be able to maintained & monitored

To query, we used predicate logic (SQL)

MODULE 2: RELATIONAL MODEL

Signatures & Relational Calculus

Tables are defined via signatures:

NAME/(column1, column2, col3...)

Data in tables is in tuples:

A

col1	col 2	. . .	coln
x	a		α
y	b		β
z	c		ε

→ tuple

We can specify attributes & tuples via set comprehension & predicate logic.

Ex:// EMP table.

1. Pairs of employes who work for same boss

$$\{(x_1, x_2) \mid \exists y_1, y_2, z \,.\, EMP(x_1, y_1, z) \wedge EMP(x_2, y_2, z)\}$$

(Syntactic sugar.

$$\{(x_1, x_2) \mid \exists z \,.\, EMP(x_1, -, z) \wedge EMP(x_2, -, z)\}$$

Database in R.C.

Universe:
- Set of all values
- Equality
- Constants

Database

Intrinsic: Metadata, no data — $R_i / (a_1, \ldots a_k)$

Extrinsic: Metadata + data (set of tuples)

$DB = \{D, \approx, R_1, \ldots R_n\}$

$R_1 \ldots R_n$: All relations in D.B. (tables)

Signature: all relations & # of cols in database:

$$\varphi = \{R_1 / k_1, R_2 / k_2 \ldots\} \iff \{R_1(col1, col2, \ldots), R_2(col1, col2, \ldots), \vdots\}$$

(k_1: # of cols in R_1)

- Just tells the schema / relations, nothing about data

Instance: actual extension (data) in relation.

When query using predicate logic: diff. answers depending on valuation

↳ Valuation: θ: { col1 → x_1, col2 → x_3 ... } ∈ D

Valuations are diff. for diff. instances.

↳ Formally:

Query: { (col1, col2, ... colj) | φ } ← some cond. about relations.

Result: { θ(col1), θ(col2) ... | DB, $\theta \models \varphi$ }

To specify query in rel. calculus, use set comprehension.

↳ Ex:// Who are bosses that have 2 employees

$\{ b \mid \exists e_1, e_2 \cdot EMP(e_1, -, b) \wedge EMP(e_2, -, b) \wedge \neg(e_1 = e_2) \}$

Integrity Constraints

Defn: conditions that are true for all relations & instances

Specify via predicate logic.

Ex://

1. Every boss manages a unique department

$\forall b, d_1, d_2 \cdot EMP(-, d_1, b) \wedge EMP(-, d_2, b) \Rightarrow d_1 = d_2$

Why do we need integrity constraints:

↳ Keys: primary key, foreign key
↳ Disjointness
↳ Coverage

} Enforce behaviors in DB.

View: create table out of integrity constraint

$\forall x_1, x_2 \dots, x_k \cdot \underbrace{R(x_1 \dots x_k)}_{\text{View}} \iff \underbrace{\varphi}_{\text{Predicate logic query}}$

All tuples satisfying φ in R

Database

φ: schema $\quad$ Σ: integrity constraints.

Safety & Finiteness

In any DB, 2 conditions:

1. Extension is finite
2. Queries are safe: answer is finite if DB is finite

Ex:// $\{(x, y) \mid x = y\}$ ⇒ infinite

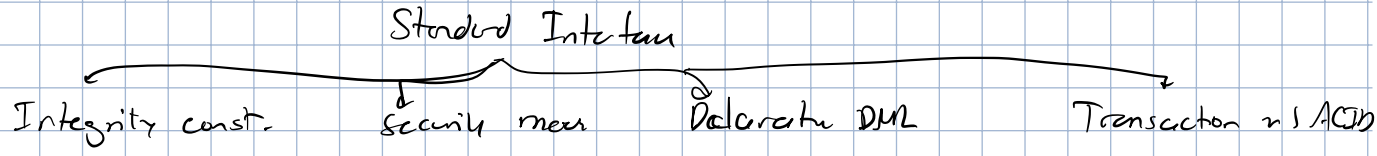
Domain independent queries: all cns. of qeis only found in extension.

↳ Way to create domain indep query: range restriction.

$\{(x_1, \ldots, x_k) \mid \varphi\}$ ← range restricted (grammar format)

MODULE 3: INTRO TO SQL

Relational Schemata & Conjunctive Queries



SQL is preferred lang. for DDL / DML requests.

① Create tables:

```
create table NAME (
    col_name  TYPE  not null,
    :
    primary key (col1, col2, ...)  ⇒ Uniquely defines tuples
    foreign key (col_k) references TABLE (col_m),
)                          ↳ col_k = col_m
```

② Query:

```
SELECT col1, col2, ..., colk
FROM TABLE
WHERE φ
```

→ Be able to Rel. query ⟺ SQL

Note: can specify multiple tables to query from ⇒ Cartesian product

↳ Implication: if 1 table is empty, Cart. product is empty
⇒ no query.

③ Aliasing.

column name of query
↓
SELECT coll as cool-name ...

Set Operations & First Order Queries

Always consider query as a table ⇒ set!

Set operations:

1. Union: Q_1 UNION Q_2 (all tuples in Q_1 & Q_2)
2. Differnce: Q_1 EXCEPT Q_2 {all tuples in Q_1 not in Q_2)
3. Intersection: Q_1 INTERSECT Q_2 (all tuples in both Q_1 & Q_2)

} Q_1 or Q_2 must same attributes.

Name queries:

WITH query_name AS (query)

Refer to query_name as table in another query

Can inline queries:

SELECT ...
FROM (SELECT ...)

Nested Queries

SELECT ...
FROM ...
WHERE condition (SELECT ...)
↑ Q

1. <attr> in Q (query returns list of values)
2. <attr> not in Q "
3. <attr> op (>, <, =) SOME Q
4. " ALL Q (maximum, minimum extremities.
5. EXISTS Q

6. NOT EXISTS Q

Ex://
```
select distinct pubid
from article
where endpage - startpage >= all (
    select endpage - startpage
    from article
)
```
→ Maximum range.

Nested queries can use variables defined in outer query

Ex://
```
select * from wrote r
where not exist (
    select *
    from wrote s
    where s.publication = r.publication
    and s.author <> r.author
)
```

Can't use to output results. Use to check conditions.

Aggregations

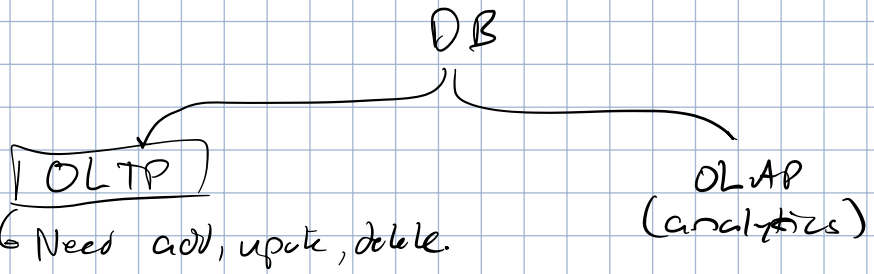
Goal: calculate attribute for groups

Syntax: ④ SELECT $x_1 \ldots x_k$ ---- agg1 (attr)
① FROM ...
② WHERE ____ → filtering
③ GROUP BY $x_1 \ldots x_k$ ⇒ Creates groups / combos
↳ Calculate agg(.) / combo

Sometimes, filter out groups based on aggregation
↳ HAVING: filters on aggregation.

```
SELECT ---, agg1(attr) as agg
FROM ____
WHERE ____
GROUP BY ____
HAVING agg condition
```

Transactions & Database Updates



Commands:

① INSERT: INSERT INTO <T> VALUES (...)

In order of attributes of T

Ex: //

T

attr1	attr2	attr3
val1	val2	val3

Add row:

... VALUES (val1, val2, val3)

VALUES (...) ⇔ query

② DELETE: DELETE FROM <T> WHERE <cond>

③ UPDATE: UPDATE <T> SET (attr1 = val1, ...)

WHERE <cond>

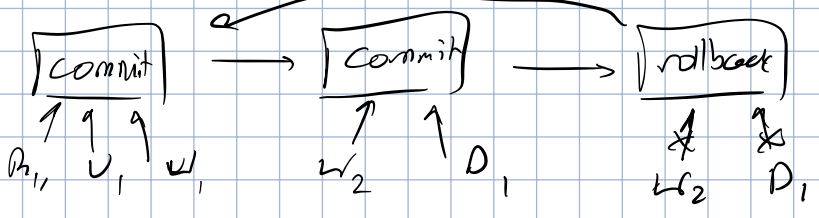
↳ Update rule

↳ Select tuples to update

Transactions: reads & write in transactions

↳ Records will not be modified @ same time

↳ Way to save changes/remove changes: commit / rollback

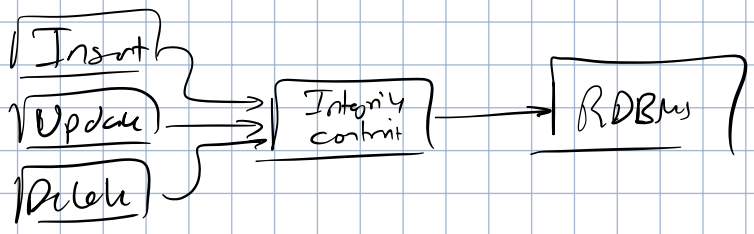


Module 4: Advanced SQL

General Integrity Constraints & Views

Integrity constraint

- Syntax: CREATE ASSERTION <name>
  CHECK (<cond>)
  ↳ SQL query



Ex:// create unique_pubid
Check (not exist (→ After going to use subquery exist.
select * from publication p1, publication p2,
where p1.pubid = p2.pubid and p1.title <> p2.title
))

Be able to convert RC integrity ⟷ SQL integrity constraints.

View: like a table defined on certain query

- Syntax: CREATE VIEW <name> AS <query> ← SQL

When is view updated:

1. View is dependent on 1 table
2. View has simple attribute (no crazy expressions for columns)
3. No aggregations / distinct
4. No nested q's
5. No set ops.

Multiset Semantics

Relation actually has a count on # of tuples

$\{ d : \exists c.\ DEPT(c, d) \}$

What we see:

CS
CS
MATH
⋮

What is actually

CS	2
MATH	5
⋮	

⇒ # of times this tuple is seen.

To get distinct tuples

RC:

$\{ val : elim\ \forall \}$

Ex: // $\{ d : elim\ \exists c.\ DEPT(c, d) \}$

↳

CS	1
MATH	1
⋮	1

SQL:

```
SELECT DISTINCT - - -
FROM ___
WHERE ___
```

Multiset operations:

Previous set ops don't allow duplicate tuples. (behaves like set)

1. Q_1 UNION ALL Q_2
2. Q_1 INTERSECT ALL Q_2
3. Q_1 EXCEPT ALL Q_2

} Same as set ops, but multiple tuples now allowed

Null Values

Null value

Inapplicable
Column doesn't apply to tuple
⇓
Remedy by adding tables

Unknown values
Tuple could have col. value but it doesn't

Soln
SQL — Default

How does SQL handle NULLS:

1. Expressions: will have value of NULL if it involves NULL

   SELECT col1, col2, col4 + col5 ⇒ If col4 == NULL / col5 == NULL ⇒ col4 + col5 = NULL

2. Comparisons: unknown if dealing w/ NULL

   SELECT ____
   FROM ____
   WHERE col1 < col2 ⇒ Unknown if col1 or col2 == NULL

   All comparisons have 3 valuations: T, F, unknown.

   If comparison involves unknown ⇒ <cond> IS UNKNOWN

   If filtering on nulls ⇒ <val> IS NULL

3. Set ops: use special values for duplicates

4. Aggregates: don't count NULLS

   ↳ COUNT(*): will count all tuples returned by query, regardless if row has NULL value.

Joins

FROM T_1 JOIN T_2 ON <cond> ← Boolean valuation between table columns

Join types: FULL, LEFT, RIGHT, INNER (default)

⇓

If LEFT: join if right col is NULL
If RIGHT: " left "

Ordering and Limits

Query results are non-deterministic ⇒ has no order

To order:

SELECT
FROM
WHERE
ORDER BY col1, col2, ... colk [DESC / ASC] ← default

To limit results:

default = 0

LIMIT c_1 [offset 0] ⟹ Will take all vals from $0 \rightarrow 0+c$

Triggers and Authorization

Trigger: SQL command that happen on event

- Syntax:

```
CREATE TRIGGER <name>
AFTER <event> ON <table/view>
[REFERENCING OLD AS <old_alias>]
[REFERENCING NEW AS <new_alias>]
FOR EACH ROW [WHEN <condition>] BEGIN ATOMIC
    <DML>
end
```

- Ex:// Create a trigger to delete all rows with author id in wrote upon delete.

```
create trigger delete_author
after delete on author
referencing old as a
for each row begin atomic
    delete from wrote where wrote.author = a.aid
end
```

- Integrity constraint checking after triggers

Foreign keys:

Bc auto/cascade deletion are super common, can create foreign reference behaviors on table creation.

- Syntax:

```
FOREIGN KEY (<from-attr>)
REFERENCES <table> [(<to-attr>)]
    [ON DELETE/UPDATE <action>]
```

→ CASCADE, SET NULL, ...

Authorization:

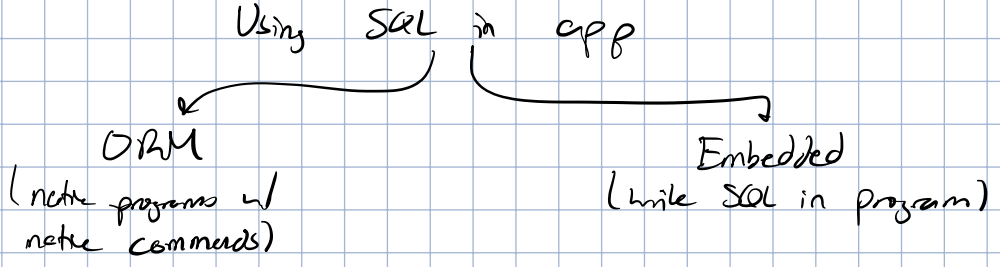
Data control lang.

- GRANT/REVOKE <role> TO <user>
- GRANT/REVOKE <action> ON <obj.> TO <user/role>
  - <action>: CONNECT, ALTER, SELECT
  - <obj.>: Table, view

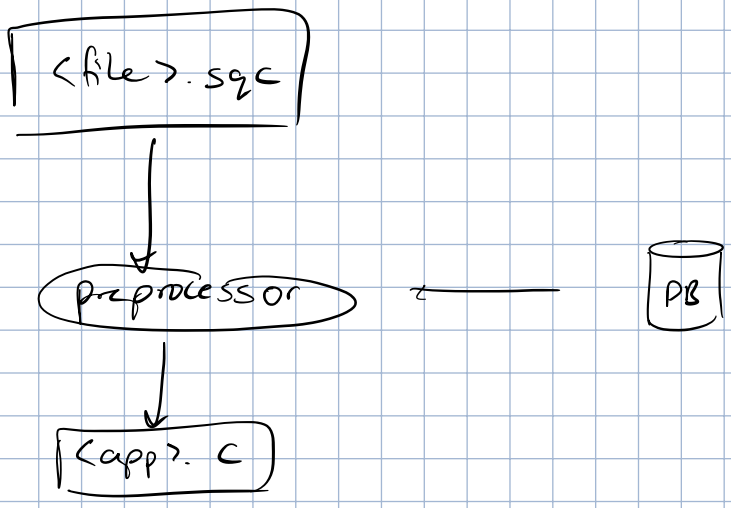
Module 5: Application Programming

Obj: how to make C-based apps to interact w/ DB.

Embedded SQL in C



General architecture:



Code structure:

```
include SQL
int main () {
    declarations;
    connect to db;
    work;
    error handling;
    commit/abort & disconnect;
}
```

① Include & declarations

1. All .sqc files start like so:

```
#include <stdio>
:
EXEC SQL INCLUDE SQLCA
```

2. Declare host variables:

Host vars: like a param. in SQL. Can hold values that can be used in SQL & in C!

```
EXEC SQL INCLUDE SQLCA
EXEC SQL BEGIN DECLARE SECTION;
    C_datatype varName;
    :
    :
EXEC SQL END DECLARE SECTION;
```

How to use host variables?

`:hostVar`

② Error handling:

```
int main () {
    EXEC SQL WHENEVER SQLERROR GO TO end;
                      NOT FOUND   ||
                      SQLWARNING  ↳ NOT FOUND is useful even if not to move to end!
    CODE

    end:
      check_error (" Error: ", &sqlca) ⇒ output error
      EXEC SQL WHENEVER SQLERROR CONTINUE. ⇒ Skip an error
      EXEC SQL ROLLBACK ⇒ Reverse
      EXEC SQL CONNECT reset; ⇒ reset DB connection.
      exit (1)
```

③ Connecting to DB:

```
int main () {
    :
    :
    EXEC SQL CONNECT TO :dB;
```

Host var that contains DB name.

④ Query:

1 row result: EXEC SQL <query> INTO :hostVar
INDICATOR :ind

Indicator < 0 ⇒ host variable is null!

What if you have multiple rows in query? Cursor!

① Declare

EXEC SQL DECLARE <name> CURSOR FOR <query>

② Error handling for cursor

EXEC SQL WHENEVER NOT FOUND GO TO <label>

③ To query:

EXEC SQL OPEN <cursorNum>

for(;;) {

EXEC SQL FETCH cursorName INTO :hostVar1,
:hostVar2

:

}

④ Close:

EXEC SQL CLOSE <cursorName>

⑤ Compile:

.sqc ⟶ `db2 prep <sqc file>' ⟶ .c file ⟶ compile ⟶ run

Stored Procedures

Functions written in RDBMS

Syntax:

CREATE FUNCTION <name> (var <sql_datatypes>)
RETURNS <sql-datatype>
LANGUAGE SQL
RETURN <query that uses var>

Usage:

SELECT <name> (some_val), ... FROM TABLE

Module 6: Dynamic App. Programming

Dynamic Embedded SQL

Q: How do we process dynamic queries?

Main tool:

```
EXEC SQL PREPARE stmt FROM :string;
```

(SQL query; passed in query)

What if we want to parameterize query?

↳ Soln:

1. Reconstruct ASCII of query w/ param ⇒ recompile
2. Specify param inclusion in query string:

" SELECT . . . FROM '?' . . . "

To execute, must have a param.

How to execute:

If query does not return tuples (DML changes, transactions)

Prepare sql query

Host vars for params.

```
EXEC SQL EXECUTE stmt USING :var1, :var2 ...
```

If query has tuples: cursors

① Declare: exact same

② Open:

```
EXEC SQL OPEN <cursorName> USING :var1, :var2, ...
```

③ Fetch & close: exact same

Helpful tool: descriptor

Stores metadata for a query.

To use:

① Include ⇒ EXEC SQL INCLUDE SQLDA

② Declare usage:

```
struct sqlda * <struct_name>
int main() {. . .}
```

③ Store info:

```
init_da (&<struct_name>, 1);
.
.
EXEC SQL DESCRIBE <query> INTO : * <struct.name>
```

④ Access info:

< struct_name > → fields → . . .

ODBC

Refer to slides, rest of my time.

MODULE 7: ENTITY RELATIONSHIP DATA MODEL

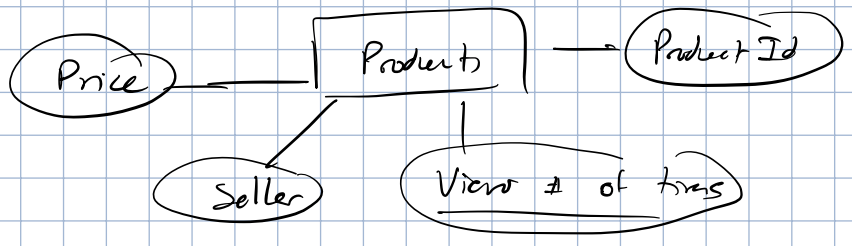
ER Modelling

Entity: set of things that are similar to each other

Teachers Employees Products

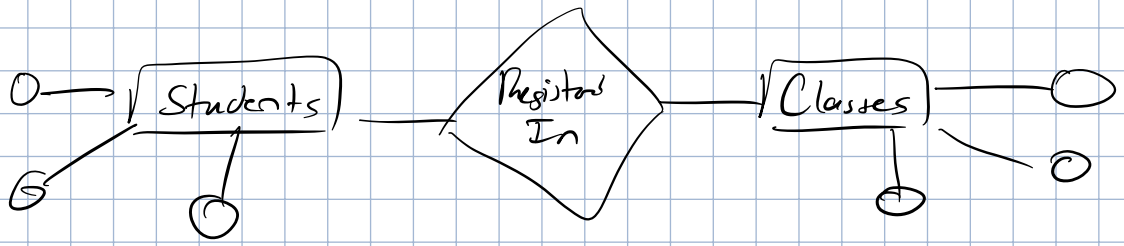
- Entity set: set of entities

Attributes: properties of entities.

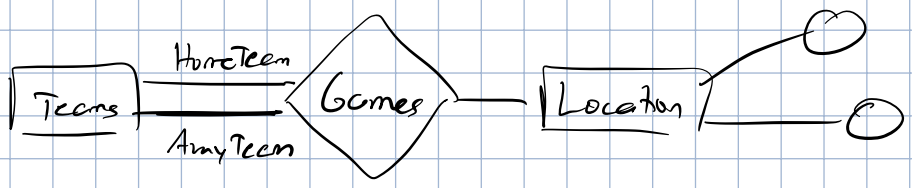


- Each attr has domain: set of all possible attr values.

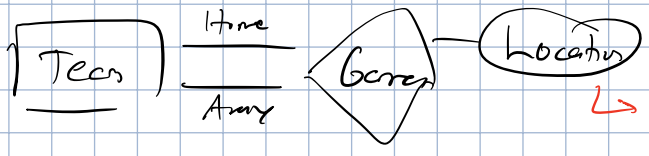
Relationships: associations b/w entities.



- Note: entities can take on diff. roles in rel.
  - ↳ Self joins



- Can add attr. to relationships but dep. of entities



↳ Dep. on HomeTeam & AwayTeam

Integrity Constraints

① Primary key

Underline attr. that is primary key

② Binary relationships:

Q: How many rel. does 1 entity have w/ another?

Many-to-many (N:N)

Entity in A → many in B
" in B → " in A

Many-to-one (N:1)

Entity in A → many in B
" B → one in A

One-to-many (1:N)

Inverse of N:1

One-to-one (1:1)

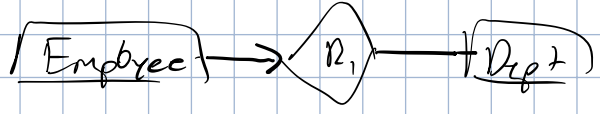
1 entity in A ⟺ 1 in B

This is putting limits on rel. #, can't have less!

To show:

↳ If —:1: use an arrowhead.

Ex:// Employees ↔ Dept. is a 1:N rel.

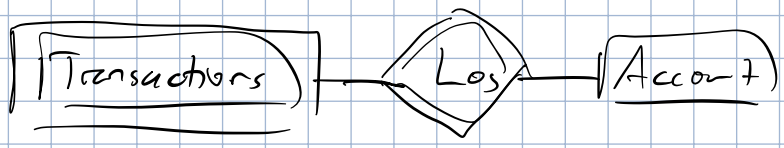


③ Existence dependencies

x is existent dependent on y ⇒ x cannot exist w/out y

↑ subord. ent. ↑ dominant entity

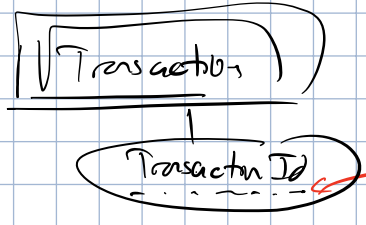
Notation: double bord on subord. & rel.



Trans. is exist dep. on Account

Week entity set: subord entity:

↳ Discriminatr: w/out dominat entity, what is the PK attr.?



Note: not unique

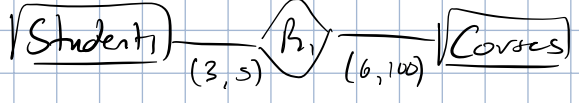
↳ Primary key: discriminatr + PK of dominant

Recursive! Dominant could also be weak.

④ General cardinality constraints.

Specific limit on # of rel. Give lower & upper bound.

Ex.// Students can take 3-5 courses. Courses must have 6-100 students.



To put this into RC:

$$\varphi = \{\ \text{entity\_name} / \text{self},$$
$$\vdots$$
$$\text{attribute\_name} / (\text{self}, \text{attribute\_name}),$$
$$\vdots$$
$$\text{relationship\_name} / (e1, e2, \ldots)$$
$$\}$$

ref. to outside table

Each attr is relative

⇒ Addrs on attr. val on table instance.

Use RC queys to write integ. constraints

Ex:// 1. AID is PK of AUTHOR:

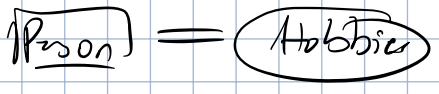
$$\forall e_1, e_2, v \cdot \text{AUTHOR}(e_1) \wedge \text{AID}(e_1, v) \wedge \text{AID}(e_2, v) \wedge \text{AUTHOR}(e_2) \Rightarrow e_1 = e_2$$

Extended ER modeling

① Structured attr

- Composite attr: attributes w/ other attributes:

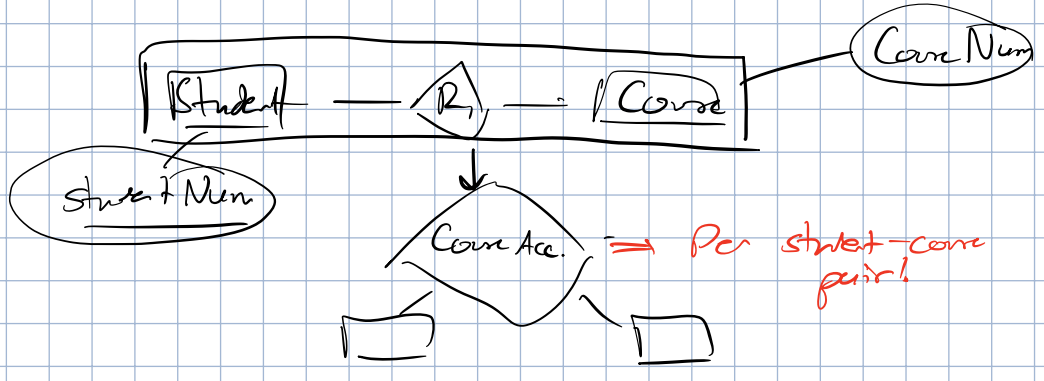
- Multivalued attr: 1 instance = several instance of attr.



② Aggregation:

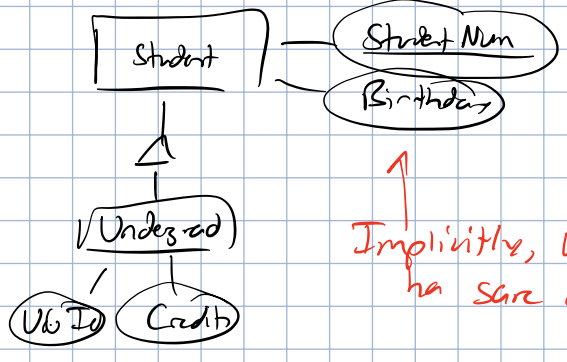
Reify (group) entity & relationships ⇒ super entity

Make rel. according to super entity:

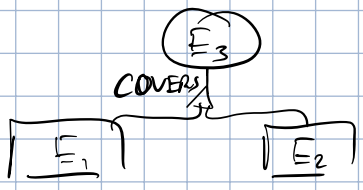


③ Specialization

Polymorphism: entity of 1 set is an entity of another specialized set



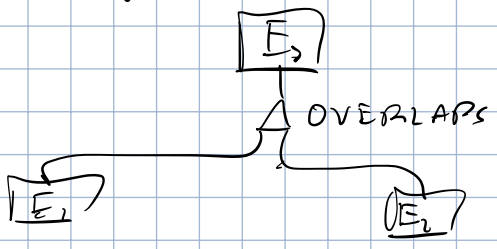
④ Generalization : opposite of specialization.



V. similar to specialization

By default, E_1 is disjoint from E_2

To not have disjointness



How to use in RC?

Specialization:

$$\forall e.\ \text{specialization}(e) \Rightarrow \text{generalization}(e)$$

Generalization:

$$\forall e.\ s1(e) \Rightarrow \text{generalization}(e)$$
$$\forall e.\ s2(e) \Rightarrow g(e)$$
$$\forall e.\ g(e) \Rightarrow s1(e) \vee s2(e)$$

Disjoint: $\forall e.\ s1(e) \Rightarrow \neg s2(e)$

Aggregation:

1. Add new relations:

rel_ent(self)
e1_ent(self, e1)
⋮
eN_ent(self, eN)

2. Add integrity constraints + table @ slides

3. Make rel. a rich.

MODULE 8: LOGICAL MAPPING

Extended ER ⟶ DB schema

Basic ER Diagram Mappings

Entity → table. Attr → cols. Relationship → table/col

① Entity:

Entity set w attr. $a_1 \dots a_k$ ⇒ table with cols $a_1 \dots a_k$

∴ 1 entity → 1 row in table.

Additional integrity constraints:

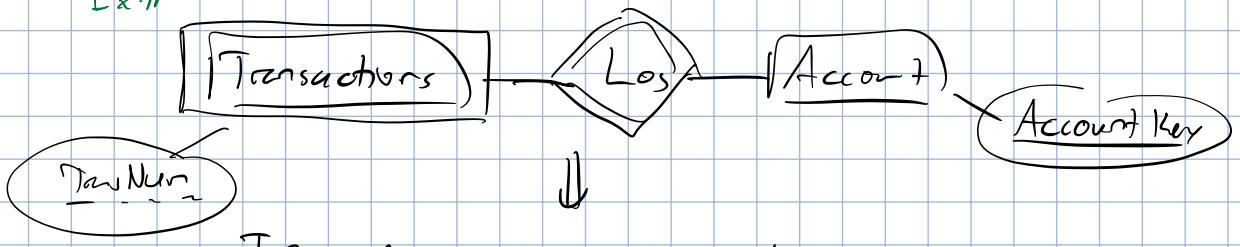
① Primary key: primary attr ⇒ primary key.

② Existence dep:

Together ⇒ PK of "weak" table

Weak entity set ⇒ table w/ discriminator, dominant PK., attr.

Ex:



TRANSACTIONS

PK. { Trans Num, Account Num }

attr 1

attr k

ACCOUNT

Account Key

⋮

② Relationships:

Case #1: Relation. for weak entity.

↳ Do nothing.

Case #2: (1,1) cardinality constraint

1. Add relationship attributes to components
2. Add PK of component to other components.

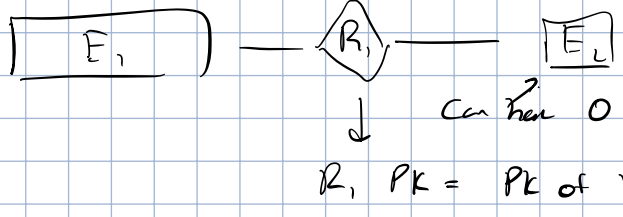
Case #3: Else:

1. Create new table
2. PK of table = PKs of components

3. Attr. of rel. entity ⇒ cols of table

Exception: (0, 1) cardinality

↳ PK of table = PK of table with potentially 0 of other records



↓ Can have 0 records of E_1

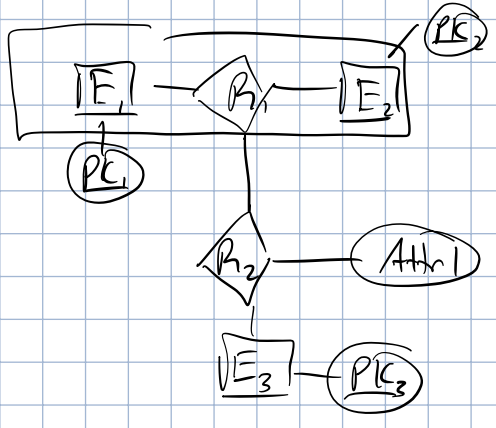
R_1 PK = PK of E_2 (not PK of E_1 + E_2)

Extended ER Mapping

① Aggregation:

Any relation that uses aggregation ⇒ new table

↳ PK: can include PK of the component table in aggregation.



⇒

Table for:

E_1, E_2, E_3, R_1, R_2

PKs:

R_2: → cuz do PK of E_3

↳ PK = PK_1 + PK_2 → PK of aggregation.

FKs:

R_1: FKs to E_1 and E_2

R_2: FKs to R_1 not E_1 / E_2

↳ Looking at E_1, E_2, R as circle.

Bottom line ⇒ same as before, outside relation reference agg. relations, not entities.

② Specialization

Specialization is "weak entity" of generalized table

↳ Create table w/ PK being PK of generalized table

③ Generalization

Multiple specialization. Take PK of generalized table

If we have disjoint tables ⇒ add integrity constraints.

Table qualified for views if no FK reference it
↳ All attributes must be defined on child tables instead.

④ Multivalued attributes:

⇒

Table E_1
$\underline{PK_1}$, A_PK

Table A_1:
$\underline{A\_PK}$, $\underline{PK_1}$

Triple Store

View that contains all data in DB.

Triple store:

| subject | property | object |

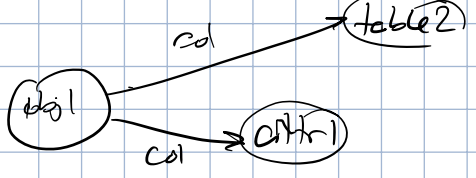
To change DB to accomodate TS:
↳ Add OID to PK ⇒ URIs for each table (OID = unique id)

Triple store:

(select OID as subject, 'in' as prop., 'table-name' as obj. from table)
UNION
(
(select OID as subject, 'attr1' as prop., A1 as obj. from table)
union
: } All tables & attributes.

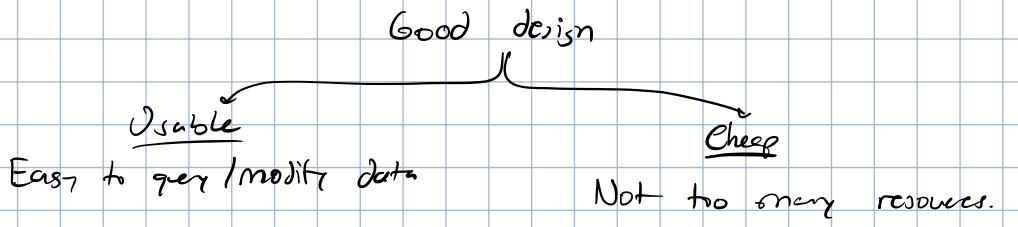
Query on regular DB ⇒ Query on triple store

Useful for graph DBs



Module 9: Data Dependencies & Normal Form

Objective: Create good design for DB



Usually: need to decompose tables for ease of query & modification

Data Dependencies

$R/(A_1 \dots A_k)$. $X \subseteq$ attr., $Y \subseteq$ attr. If you can determine values of Y from values of X: $X \longrightarrow Y$ (X functionally det. Y)

Ex: // EmpProj/(SIN, PNum, Hours, EName, PName, PLoc, Allowance)

Examples of functional dep:

1. SIN $\longrightarrow$ EmployeeName
2. PNum $\longrightarrow$ PName, PLoc
3. PLoc, Hours $\longrightarrow$ Allowance
   (How both)

If wish to identify dependencies in table given data:

① See when col. values match
↳ If col 1 always col 2 match $\rightarrow$ col 1 $\longrightarrow$ col 2

Functional dependency closure:

F is functional dep. F^+: list of <u>all</u> dependencies implied by F.

How to derive more functional dep from existing dep? Armstrong Axioms!

Armstrong Axioms:

1. Reflexivity: $Y \subseteq X \Rightarrow X \longrightarrow Y$
2. Augmentation: $X \longrightarrow Y \Rightarrow XZ \longrightarrow YZ$
3. Transitivity: $X \longrightarrow Y$, $Y \longrightarrow Z \Rightarrow X \longrightarrow Z$

4. Union: $X \to Y, X \to Z \Rightarrow X \to YZ$

5. Decomposition: $X \to YZ \Rightarrow X \to Y$

} Need to prove to us.

Ex:11 $F = \{ SIN \to EName;\ PNum \to PName, PLoc;\ PLoc, Hours \to Allowance,$
$SIN, PName \to Hours. \}$

Determine $SIN, PNum \to Allowance.$

Use SE 2 12 principles

1. $SIN, PNum \to Hours \quad (\in F)$
2. $PNum \to PName, PLoc \quad (\in F)$
3. $PLoc, Hours \to Allowance \quad (\in F)$
4. $SIN, PNum \to PNum \quad (\in F)$
5. $SIN, PNum \to PName, PLoc$ (transitivity on 2, 4)
6. $SIN, PNum \to PLoc$ (decomp. on 5)
7. $SIN, PNum \to PLoc, Hours$ (union on 1, 6)
8. $SIN, PNum \to Allowance$ (transitivity on 3, 7)

Keys:

1. Superkey: set of cols that determine <u>entire</u> relation.
2. Candidate key: minimal (least amount of keys) superkey.
   ↳ Primary key: selection for DB

How to know if you have superkey? Want to compute <u>closure</u> of either.

Attribute Closure: list of all columns that can be det. from X & F

↳ Compute $X^+ (X, F)$

① $X^+ := X$

② While exists $Y \to Z \in F \wedge Y \subseteq X^+ \wedge Z \notin X^+$ (Y: Already in X^+; Z: Not in X^+)
   ↳ Add $X^+ := X^+ \cup Z$

③ Return X^+

This also correctly creates our attribute closure! Runs in $O(n)$ time.

If X is superkey $\iff$ Compute $X^+(X, F)$ = all attr.

Table Decomposition.

Goals:

1. Lossless
2. Dependency preserving
3. Normal form

① Lossless:

Defn: able to reconstruct original table from decomposed without additional tuples (spurious)

How to determine if decomp. is lossless?

Decomposition of $\{R_1, R_2\}$ from R. This is lossless iff

$$R_1 \cap R_1 \to R_1 \quad \text{or} \quad R_1 \cap R_2 \to R_2$$

↳ Common attr. are superkeys of either table in decomp.

Ex:// $R = \{$ Student, Assignment, Group, Mark$\}$

$F = \{$ Student, Assignment $\to$ Group, Mark $\}$

$R_1 = \{$Student, Group, Mark$\}$

$R_2 = \{$ Assignment, Mark $\}$

Is this lossless?

$R_1 \cap R_2 = \{$Mark$\}$ $\Rightarrow$ This is not a superkey of R_1 or R_2 $\Rightarrow$ not lossless.

② Dependency Preservation

All original F should hold without inter-relational dependency checks

Ex:// R (Proj, Dept, Div)

$F = \{$ Proj $\to$ Dept ; Dept $\to$ Div ; Proj $\to$ Div $\}$

Decompositions:

$D_1 = \{ R_1 / (Proj, Dept),\ R_2 / (Dept, Div) \}$

Is this good decomp?

① Lossless? Yes ($R_1 \cap R_2$ is superkey of R_2)

② Dependency preserving?

a) Look at each dep. of original

b) Check if dep. can hold if table in isolation

c) If dep. not held in isolation, check if dependencies remains can be inferred

R_1 has $Proj \to Dept$ which leads to $Proj \to Div$ since $Dept \to Div$. Good

$D_2 = \{ R_1 / (Proj, Dept);\ R_2 / (Proj, Div) \}$

① Lossless? Yes

② Dep. preserving? No

↳ $Dept \to Div$ is missing.

③ Normal form

Normalization ⇒ reducing as much redundancy as possible

↳ If redundant ⇒ change anomalies (multiple records of same type, update change not cascading to others)

Normal forms

- BCNF
- 3NF

④ BCNF:

Defn: R is in BCNF wrt F iff

$\forall (X \to Y) \in F^+$ and $XY \in R$, then

1. $X \to Y$ is trivial ($Y \subseteq X$)

OR

2. X is superkey of R.

Ex:// 1. $R(A, B, C)$. $F = \{AB \to C, C \to B\}$.

Is R in BCNF?

a) $AB \to C$: AB is superkey of R, so it's good.

b) $C \to B$: not trivial, $C \not\in$ superkey $\Rightarrow$ <u>not</u> in BCNF

How to change rel. into BCNF?

↳ Compute BCNF (R, F):

1. Result := $\{R\}$
2. As long as $\exists R_i \in$ Result and some dep. $X \to Y \in F^+$ that violate BCNF
   Replace R_i w/ $R_i - (Y - X)$
   Add $X \cup Y$ to Result
3. Return result.

This also returns BCNF, lossless decomp. of R.

Ex:// Decompose $R(A, B, C, D, E)$, $F = \{E \to CD, BC \to A, D \to B, A \to E\}$. into BCNF:

Use Compute BCNF algo:

① $E \to CD$, E is not superkey of R.

Result = $\{R_1(A, B, E), R_2(E, C, D)\}$

② 1. $E \to CD$, $ECD \not\subseteq R_1$, so no check
$ECD \subseteq R_2$, E is superkey

2. $BC \to A$, $BCA \not\subseteq R_1, R_2$

3. $D \to B$, $DB \not\subseteq R_1, R_2$

4. $A \to E$: $AE \subseteq R_1$. Is A a superkey?
Yes! Prove via Armstrong's Axioms $(A \to BE)$

③ Return:
$\{R_1(A, B, E), R_2(E, C, D)\}$

Problems:

1. Non-determinism: depends on order of dep. in algo.

2. Not always dep.-preserving.

(B) 3NF

Defn: R is in 3NF wrt $F \iff \forall X \to Y \in F^+ \wedge XY \subseteq R$, then:

1. $X \to Y$ is trivial ($Y \subseteq X$)
2. X is superkey
3. Each attr. of $Y - X$ is candidate key of R.

Why better than BCNF?

Both lossless & dependency-preserving decomp.

To create 3NF, we need a minimal cover

↳ Minimal cover: least amount of FD necessary.

F is minimal if:

1. Every RHS of D.P is singular ($A \to BC$, $A \to B$)
2. For no $X \to A$ is $F - \{X \to A\}$ equiv. → to F
3. For no $X \to A$ & $Z \subset X$ is $F - \{X \to A\} \cup \{Z \to A\}$ equiv.

How to compute: repeat algo until no updates

1. Replace $X \to YZ$: $X \to Y$, $X \to Z$
2. Remove $X \to A$ if $A \in$ Compute X^+ $(X, F - \{X \to A\})$
   ↳ Can reach A w/out $X \to A$!
3. Remove A from X in $X \to B$ if $B \in$ Compute X^+ $(X - \{A\}, F)$
   ↳ $AB \to C$. If C can be reached w/ only B, no need to have A.

To create 3NF decomp, follow this algo:

Compute 3NF (R, F):

1. Result := $\emptyset$, G := minimal cover of F
2. $\forall X \to Y \in G$:
   Result := Result $\cup$ (XY) } Updating w/ concat.
3. If no $R_i \in$ Result s.t. R_i contains candidate key for R
   a) Compute candidate key K for R
   b) Result := Result $\cup$ (K)
4. Return Result.

Ex:// $R(A, B, C, D, E, F)$. $F = \{E \to CDE,\ DE \to BC,\ E \to C,\ C \to F\}$
Construct 3NF decomp. of R.

① Minimal cover of F:

1. First step: $\{E \to C,\ E \to D,\ E \to E,\ DE \to B,\ DE \to C,\ E \to C,\ C \to F\}$
2. 2nd step: $\{E \to D,\ E \to E,\ DE \to B,\ DE \to C,\ E \to C,\ C \to F\}$
3. 3rd step: $\{E \to D,\ E \to E,\ E \to B,\ E \to C,\ C \to F\}$

② Update result initial:

Result: $\{FD,\ FE,\ EB,\ EC,\ CF\}$

④ Check for candidate keys:

① Check candidate key (look back at minimal cover work)

∴ AF is candidate key.

② Add candidate key in Result if not included.

Result: $\{FD,\ FE,\ EB,\ EC,\ CF,\ AF\}$

Additional Dependencies & Normal Forms

① Multivalued dependency:

Multivalued attributes: can take multiple for one entity ⇒ multiple rows.

Ex:// CAR:

Model Num	Brand	Color
A	Toyota	Black
A	Toyota	Blue
A	Toyota	Green

Multivalued dep: 2 attributes are independent of each, but both dep. on common column.

- $A \to B,\ A \to C \Rightarrow A \twoheadrightarrow BC$

specific axioms of MVD

Dependency basis

Lossless decomp. or MVD: $\{R_1, R_2\}$ is lossless:

$(R_1 \cap R_2) \to (R_1 - R_2)$ or $(R_1 \cap R_2) \twoheadrightarrow R_2 - R_1$

② 4NF:

R is 4NF $\iff$ $X \twoheadrightarrow Y \in F^+ \wedge XY \subseteq R$, then either:

1. $X \twoheadrightarrow Y$ is trivial ($Y \subseteq X$, or $XY = R$)
2. X is superkey of R

Exactly same as BCNF, so we the also to construct 4NF

③ Join dependency:

$$\bowtie [R_1 \ldots R_k]$$

(decomp.)

2 constraints:

1. $\forall x_i \cdot R_i(x_i) \iff \exists y \cdot B(x_i, y) \quad \forall i, 1 \le i \le k$ → No new cols. in $\bowtie[\ldots]$

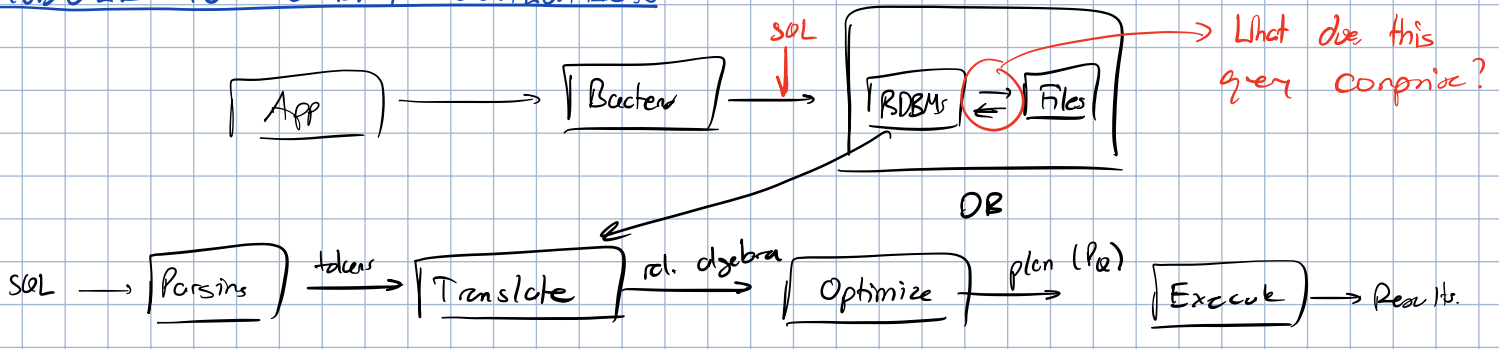
(R_i = decomp, B = original rel.)

2. $\forall x \cdot B(x) \iff R_1(x_1) \wedge \ldots \wedge R_k(x_k)$

Meaning: we can reconstruct original relation w/ decomp.

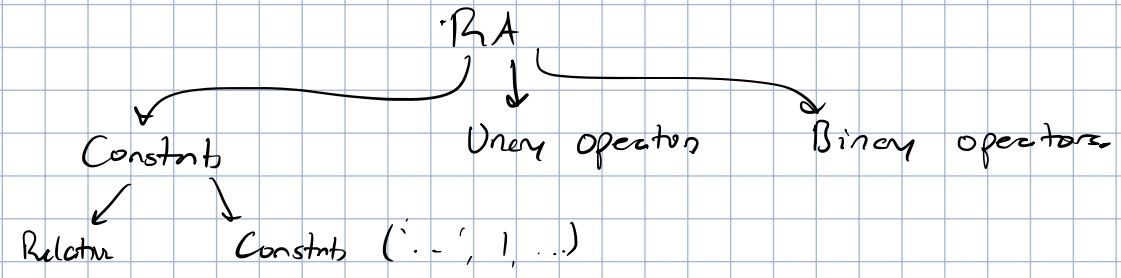
This is generalization of MVD

Module 10: Query Evaluation



Note: rel. algebra ≠ rel. calculus.

Relational Algebra



Unary operators:

① Selection operator:

$$\sigma_{condition}(relation)$$

Ex: // Account/(anum, type, balance, bank, bnum, cnt)

Q: Find all accounts of type A

$$\sigma_{\#2 = 'A'}(ACCOUNT)$$

Column to search (1-indexed)

Q: Accounts w/ balance > $1000

$$\sigma_{\#3 > 1000}(ACCOUNT)$$

Condition can be any Boolean condition

↳ Column-to-col. comparison is used a lot!

② Projection:

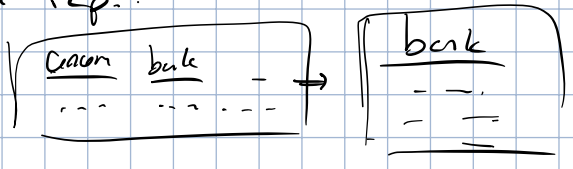
$$\Pi_{\#i, \#j \ldots}(relation)$$

Basically 'select' columns.

Q: Want bank num for all accounts w/ balance > $1000

$$\Pi_{\#4}(\sigma_{\#3 > 1000}(ACCOUNT))$$

Actual rep.:



③ Duplication elimination:

elim (relation) ⇒ removes all duplicate tuples.

Q: Want distinct bank nums //

$$elim\ \Pi_{\#4}(\sigma_{\#3 > 1000}(ACCOUNT))$$

Binary operators:

① Cross product:

$$R_1 \times R_2$$

Take 1 tuple in R_1 ⇒ pair w/ all tuples in R_2 & repeat for all tuples in R_1

Adding on more columns.

Ex:// Accounts w/ balances > $1000

$$\sigma_{\#3=\#7}(\text{ACCOUNTS} \times 1000)$$

Added 1 column to ACCOUNTS w/ value 1000

② Union:

Exact same as set union

③ Difference:

// set difference.

To evaluate rel. algebra:

Eval (IQ , DB)

↳ instance to query over.

Codd's Theorem:

Query in RC ⇒ Query in RA.

Chaining multiple operators ⇒ pipelining! Efficient

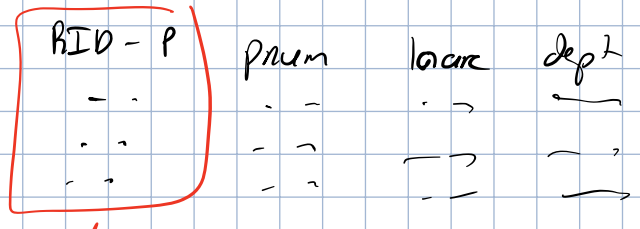
↳ No need to store intermediate values.

To make RA more efficient, use index scans:

↳ Indices on table are usually in Btree format

Ex:// PROF (lpnum, lname, dept)

Primary index of PROF is a Btree



↳ Unique id / PK in table ⇒ Really fast for searching.

Why is this important?

↳ Use index scan rather than rel. scan.

Ex: // 1. Get last nam & dept for prof # 14.

Unoptimized:

$$\pi_{\#2,\#3}\left(\sigma_{\#1=14}(PROF)\right)$$

Optimized:

$$\pi_{\#2,\#3}\left(\sigma_{\#2=14}(PROF\text{-}PRIMARY)\right)$$

↳ Why? PROF-PRIMARY / rid, pnum, lname ...

Often times, eg. put secondary indices to make query faster.

Nested join:

$$R_1 \times \sigma_{\#i=\#jl}(R_2)$$

Joining R_1 on R_2 s.t. Col j of R_1 = col i on R_2

Query Optimization

Idea: construct all plans ⇒ calculate cost ⇒ select cheapest plan

Difficult & unsatisfiable, but reasonable approx. can be done.

To estimate costs, we look @ # of pages need to be queried from Btree.

Formulas:

① Search on clustered primary index:

$$\text{Cost} = 2 + \frac{\#\text{ of tuples to search}}{\text{blocking factor of index}}$$

② Search on clusted secondary index:

$$\text{Cost} = 2 + \#\text{ of tuples}$$

③ Entire index scan:

$$\text{Cost} = \frac{|\text{relation}|}{\text{blocking factor}}$$

Note: try to reduce amount of int. results & subexpr. costs

Module 11: Transaction And Recovery Management

Concurrency Control

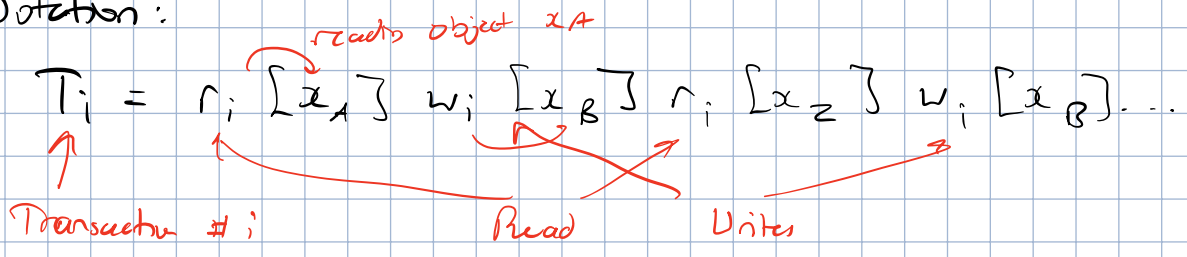
Problem: multiple users reading & writing records @ same time.

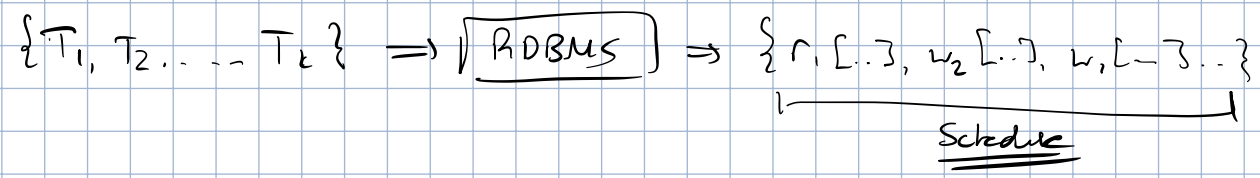
Recall transactions!

↳ Indivisible DMS requests: reads & writes.

Schedule ops. w/ transactions to prevent integrity issues.

Notation:





Transactions look as if sequential, but can actually mix & match.

Gold standard for schedules: serializable schedule

↳ End result is same as sequentially executing on data!

Ex: // $S_a = w_1[x]\ r_2[x]\ w_1[y], c_1, r_2[y], c_2$

commit (1 is done)

Equiv: $S_a = \underbrace{w_1[x]\ w_1[y]\ c_1}_{T1}\ \underbrace{r_2[x]\ r_2[y], c_2}_{T_2}$

Is this equiv? Yes!

Scheduling conflict:

2 ops conflict if:

1. Ops belong to diff. transaction.
2. Access same obj
3. One of them is write.

Not necessarily bad ⇒ only bad if happens in non-serializable schedule.

Conflict serializable sched.: have conflits, but actually serializable.

Q: How to tell if schedule is conflict serializable?

Ans: Use topological sorting

① Create a graph w/ directed edges.

Node: transaction

Edges: $T_1 T_2$ exists if T_1 & T_2 conflict on some op & T_1's op scheduled 1st

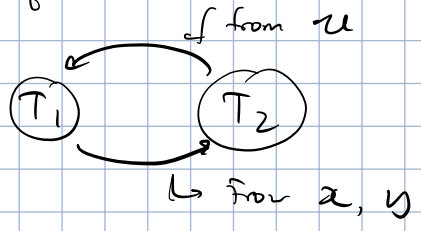
② Check if cycles ⇒ if so, not serializable

③ If no cycle ⇒ use topological sorting to get serialized schedule.

Ex:// 1. $S_1 = w_1[x], w_1[y], r_2[u], w_1[z], w_2[x], r_2[y], w_1[u]$

Is this conflict serializable?

① Create graph:

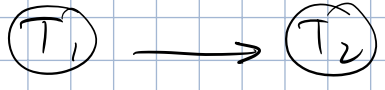


② Serializable?

No! Cycle.

2. $S_2 = w_1[x], w_1[y], r_2[u], w_2[x], r_2[y], w_2[y], w_1[z]$

① Create graph



② Serializable?

Yes! No cycle

∴ Do T_1, then T_2

Unique situations:

① Recoverable sched.

Problem: T_i writes to x → T_j reads x → T_j commits → T_i aborts.

Soln: commits in order of read order

⑦ Cascadeless sched.

Problem: T_i writes to $x \rightarrow T_j$ reads $x \rightarrow T_i$ aborts

↳ T_i abort → T_j is forced to abort → cascade of aborts.

Soln: Don't read uncommitted objects

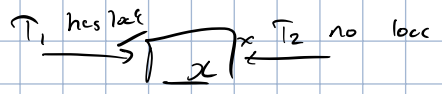
Two Phase Locking

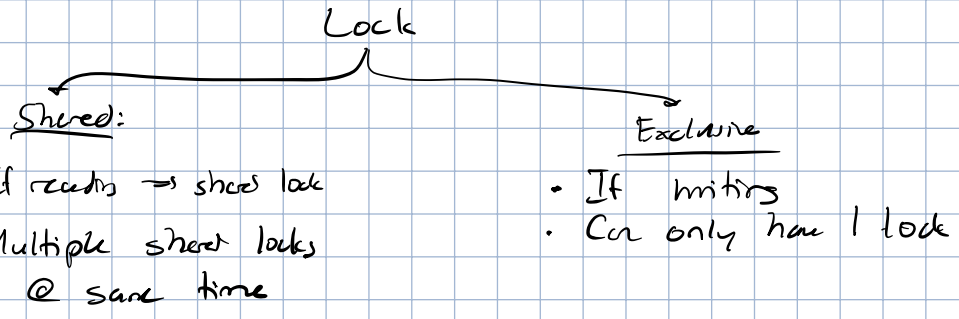
{ops..} → [scheduler] → sched'd

↳ Can delay, reject, ignore

Lock-based scheduler:

Every transaction access must have object lock before accessing.





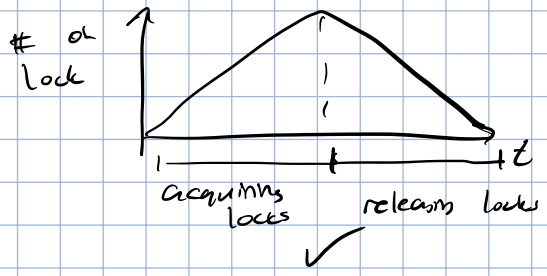
Analogy: blackboard:

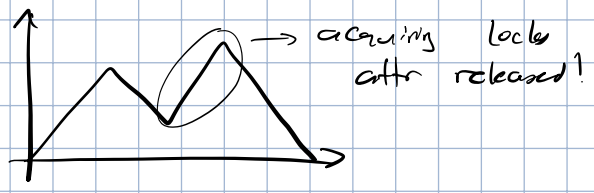
↳ Everyone can read blackboard @ same time (shared lock)

↳ Teacher/student write, cannot read bc blocking view (exclusive lock)

2 phase locking protocol:

If transaction releases locks, cannot acquire locks again





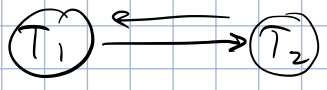
If schedule is serializable ⇒ 2PL schedule possible.

To prevent cascade aborts, use strict 2PL protocol:

- Cannot release lock until transaction commits/aborts.

Deadlock: T_{i+1} depends on T_i, but T_n depends on T_1

Ex:// $r_1[x], r_2[y], w_2[x], w_1[y]$



Both T_1 & T_2 come first ... deadlock!

How to prevent deadlock?

① Use 2PL protocol: locks granted in order of schedule

② Detect deadlocks via graphs

Other types of locks:

1. Multi-granularity lock: locks depend on size of obj
2. Predicate lock: lock object based on predicate
3. Tree locking: Btree.

Phantom tuple problem:

Issue: get diff # of tuples depending on read & insert & delete order

Ex:// $r_1[x]\ w_2[x]$
vs.
$w_2[x]\ r_1[x]$

Soln: lock entire index / predicate locking.

2PL is too strict ⇒ diff. level of concurrency control

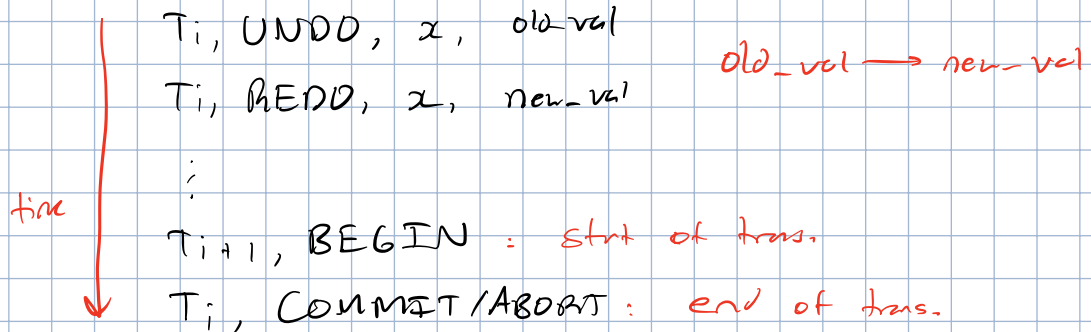
Failure Recovery

Module that does recovery must:

1. Atomicity: follow commit/aborts
2. Consistent

Schedule → [Recovery module] → log (append-only)

Log structure:



To recover from log:

1. Note down all operations from start of failure
2. Note which transactions complete by end.
3. All remaining transactions are aborted ⇒ restore to UNDO state.

MODULE 12: DB TUNING

Goal: change phys. design & parameters ⇒ perf. ↑

↳ Might also require logical design changes & DML changes.

To optimize, we need description of workload + bottlenecks.

Workload

Queries:
- Frequency
- Rel. & attr accessed.
- Selection
- Joins

Updates
- Type of update
- Relations effected

If we want to optimize, then 2 goals:

① Reduce latency

② Increase throughput

Physical DB Design

Most powerful design tool ⇒ index!

To create indices:

Materialized view:

create view INDEX_NAME as (query)
materialized as DATA_STRUCT. with search key (col)

What is stored in index?

DATA_STRUCT: Btree, heap file, hash file....

col: index column.

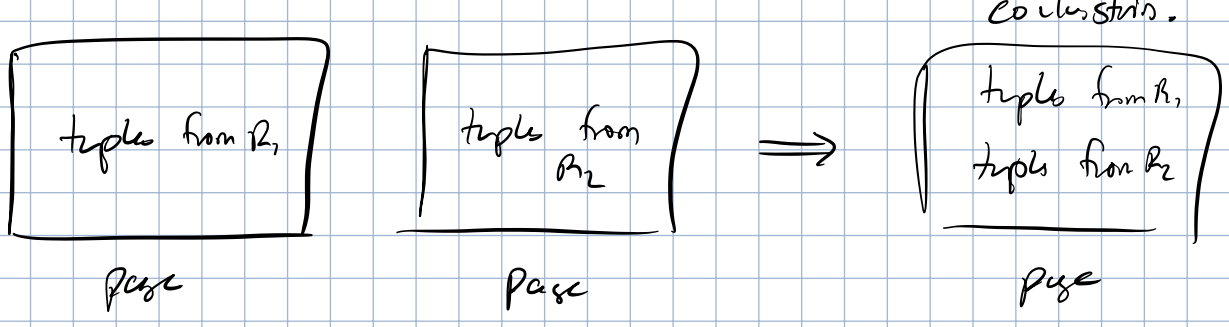
Ex:// 1. Primary index on pnum

create view PROF_PRIMARY as (select * from prof)
materialized as BTREE with search key (pnum)

2. Secondary index on lname

create view PROF_SEC. as (select lname, rid from PROF_PRIMARY)
materialized as BTREE with search key (lname)

To improve perf, we can do co-clustering.



- When to use?
  1. Lots of join b/w R_1 & R_2
  2. (1, N) rel.
- Pros: speeds up joins.
- Cons: sequential reads slower.

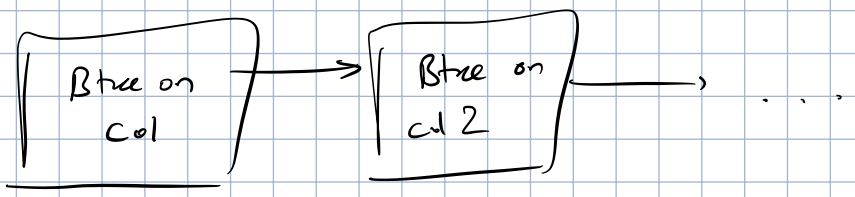
Multi-attribute search keys:

Useful in following scenario:

```
SELECT ...
FROM ...
WHERE col1 = ..., col2 = ..., ...
```

Order remains same across many queries

Physical:



Pro: very fast to do seeks on specified order

Con: Not useful if only filters on non-specified order

Join index: just do a materialized view on RIDs of table to join!

Guidelines:

- Only do indices if benefit > cost of impl.
- Try to use columns that are used for filtering for indices.
- Try to choose common columns.

Logical Schema & Query Tuning

Goal: avoid joins as much as possible & retrieve data w/ few ops.

Techniques to avoid joins:

① Renormalization: Use a diff. BCNF decomp. that might be better for workload.

② Denormalization: merge tables → redundancy

Why is this good? No need joins!

③ Partioning: split up data across servers ⇒ throughput ↑

Partitions

Vertical:
- Split up columns
- Useful if certain columns accessed a lot

Horizontal
- Split up rows
- More common

Techniques to optimize queries:

① Avoid sorting, grouping, joins

② Replace subqueries w/ joins.

3. Make sure plan cost estimation is updated

4. Reduce # of columns & rows returned (WHERE ... useful, SELECT ... as few cols as possible)

5. Reorganize transactions & updates to avoid hotspots.