INTRODUCTION TO DATABASE MANAGEMENT

DATABASE-SYSTEM APPLICATIONS

- database-management system (DBMS): collection of interrelated data & set of programs to access those data
  - ↳ database contains info relevant to an enterprise
  - ↳ primary goal of DBMS is to store/retrieve database info that's both convenient & efficient
- database systems are used to manage collections of data that are:
  - ↳ highly valuable
  - ↳ relatively large
  - ↳ accessed by multiple users & apps, often at same time
- key to managing database systems is abstraction, which allows us to use device/system w/o having to know details of how it's constructed
- 2 modes in which databases are used:
  - ↳ online transaction processing: large # users retrieve relatively small amounts of data & performing small updates
  - ↳ data analytics: process data to draw conclusions & infer rules/decision procs
    - used to drive business decisions
    - e.g. banks need to decide whether to give loan or not to applicant so data-analysis techniques attempt to create predictive models from discovering rules & patterns from data

PURPOSE OF DATABASE SYSTEMS

- typical file-processing system stores permanent records in various files & it needs diff app programs to extract records from/add records to appropriate files
- disadvantages of keeping organizational info in file-processing system:
  - ↳ data redundancy & inconsistency
  - ↳ difficulty in accessing data
    - if designers of original system didn't anticipate request, no app program to meet it
      - → use another program & manually sort data after
      - → make another app program to meet request
  - ↳ data isolation
    - data may be scattered into various differently-formatted files
  - ↳ integrity problems
    - data values stored in database must satisfy certain types of consistency constraints, but it's difficult to change programs to enforce newly added constraints
  - ↳ atomicity problems
    - e.g. money is taken from account A, but fails to be transferred to account B so funds transfer is not atomic (i.e. process happens in entirety)
    - hard to ensure atomicity
  - ↳ concurrent-access anomalies
  - ↳ security problems

VIEW OF DATA

- major purpose database system is to provide users w/ abstract view of data
  - ↳ system hides certain details on how data are stored & maintained
- data model: collection of conceptual tools for describing data, data rltnships, data semantics, & consistency constraints
  - ↳ classified into 4 categories:
    - relational model uses collection of tables (i.e. relations) to rep data & rltnships among data
      - → each table has multiple columns w/ unique name
      - → example of record-based model
      - → each table contains diff record types w/ fixed # fields/attributes (columns)

→ most common

- **entity-rltnship (E-R) model** uses collection of basic obj (i.e. entities) & rltnships among them
  → entity is obj in real world distinguishable from other objs
  → widely used in database design
- **semi-structured data model** permits spec of data where individual data items of same type can have diff sets of attributes
  → e.g. JSON, XML
- **obj-based data model** allows for objs to be stored in rltnal tables & procs to be stored in & executed by database system
  → extend rltnal model w/ notions of encapsulation, methods, & obj identity

· e.g. rltnal database

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

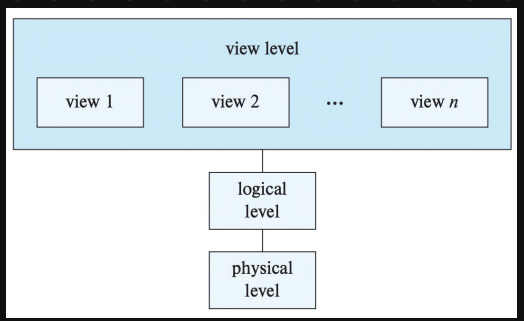
dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

· to retrieve data efficiently, complex data structures are developed to rep data in database
↳ complexity is hidden from users thru several levels of **data abstraction** (from lowest to highest):

- **physical level**: how data are actually stored
- **logical level**: what data are stored & rltnships btwn them
  → i.e. describes entire database in terms of small # simple structures
  → uses **physical data independence**, meaning user at logical lvl isn't aware of complex implementation of simple logical structures in physical level
- **view level**: describes only part of database
  → system may provide many views for same database

↳ 3 lvls of data abstraction:



- low-lvl implementation details are hidden from both users ? data-app developers

- **instance** of database is collection of info stored in it at particular moment
  schema is overall design of database
- an analogy is schema is var declarations ? type defns while instances are vals of vars at certain point in time of program
- database systems have several schemas, partitioned to lvls of abstraction
  ↳ physical
  ↳ logical
  ↳ subschemas describe diff views of database

DATABASE LANGUAGES

- database system provides **data-defn language (DDL)** to specify schema ? **data-manipulation language (DML)** to express database queries ? updates
  ↳ usually not 2 separate languages ? instead form parts of single database lang (e.g. SQL)
- **database storage ? defn lang** is special type of DDL to specify storage structure ? access methods
  ↳ define implementation details of schemas
- data values stored in database must satisfy certain consistency **constraints**
- database systems only implement integrity constraints that can be tested w/ minimal overload
  ↳ **domain constraints**: domain of possible values must be associated w/ every attribute
    - e.g. int types, char types
  ↳ **referential integrity**: ensure that val that appears in one rltn for given set of attributes also appears in a certain set of attributes for another rltn
    - e.g. *dept_name* val in *course* record must appear in *dept_name* attribute of some record of *department* rltn
      → i.e. department listed for each course must actually exist in uni
  ↳ **authorization**: differentiates access permissions to various data values btwn users
    - most common types:
      → read
      → insert (i.e. insertion of new data, but no modification of existing data)
      → update (i.e. modification, but not deletion)
      → deletion

DDL is used to specify database schema ? additional properties of data

- output of DDL is placed in **data dictionary**, which contains **metadata** (i.e. data abt data)
  ↳ data dictionary can be accessed ? updated by database system itself
  ↳ system consults dictionary before reading/modifying actual data
- **SQL** provides DDL that defines tables w/ data types ? integrity constraints
  ↳ e.g.

```
create table department
    (dept_name      char (20),
     building       char (15),
     budget         numeric (12,2));
```

DML enables users to access/manipulate data
↳ **types of access** are retrieval, insertion, deletion, ? modification

- 2 types of DML:
  ↳ **procedural**: requires user to specify what data are needed ? how to get them
  ↳ **declarative**: requires user to specify what data are needed w/o specifying how to get them
    - i.e. nonprocedural DMLs
- **query** is statement requesting retrieval of info
  ↳ although incorrect, query lang ? DML are used interchangeably
- **SQL query lang** is nonprocedural
  ↳ query takes input of one/several tables ? always returns single table

- e.g.
```
select instructor.name
from instructor
where instructor.dept_name = 'History';
```
  → query would return table w/ 1 column labeled *name* ? set of rows, each containing name of instructor whose *dept_name* is History

- e.g.
```
select instructor.ID, department.dept_name
from instructor, department
where instructor.dept_name= department.dept_name and
      department.budget > 95000;
```
  → query would return table w/ 2 colums labeled *ID* ? *name* ? set of rows, each containing ID ? name of instructor whose in a department w/ budget > 95000

- SQL doesn't support actions like user input, output to displays, or communication over network
  ↳ **app programs** are used to interact w/ databases in this fashion
    - to access database, DML stmts are sent from host to database

DATABASE DESIGN

- **initial phase** is to fully characterize data needs of prospective users
  ↳ designer interacts extensively w/ domain experts & users
  ↳ outcome is specs of user requirements
- schema that reflects the concepts of chosen data model is developed at **conceptual-design phase**
  ↳ provides detailed overview of enterprise
  ↳ focuses on describing data ? their rltnships
- conceptual-design process involves:
  ↳ what attributes we want to capture in database
  ↳ how to group attributes to form various tables, which can be done 2 ways:
    - entity-rltnship model
    - employ set of algorithms that's known as **normalization**, which takes set of attributes as input ? outputs set of tables
- fully developed conceptual schema has **spec of functional requirements**, where users describe ops / transactions performed on data
- moving from abstract data model to implementation of database has 2 final design phases.
  ↳ **logical-design phase**: maps HL schema onto implementation data model of system
  ↳ **physical-design phase**: physical features are specified
    - i.e. form of file organization ? internal storage structures

DATABASE ENGINE

- database system is partitioned into modules that deal w/ diff responsibilities of overall system
  ↳ **functional components** are storage manager, query processor, ? transaction manager
- **storage manager** is important b/c databases typically require large amounts of storage space
  ↳ main mem can't store all data in database ? is at risk of losing all data in system crash, so use disks
  ↳ movement of data is much slower than speed of CPU so it's important that database systems structure data to minimize need to move data btwn disk ? main mem
  ↳ solid-state disks (SSDs) are becoming more popular
    - faster but more costly than traditional ones
- **query processor** is important b/c it helps system simplify ? facilitate access to data
  ↳ allows users to work at view lvl ? not have to understand physical implementation
- **transaction manager** is important b/c it allows app developers to treat sequence of database accesses as single unit which entirely happens or not
  ↳ developers don't need to be concerned w/ concurrent access ? system failures

- storage manager provides interface btwn low-lvl data stored in db ? app programs ? queries submitted to system

↳ interacts w/ file manager
↳ translates DML stmts into low-lvl file system commands
↳ stores, retrieves, & updates data

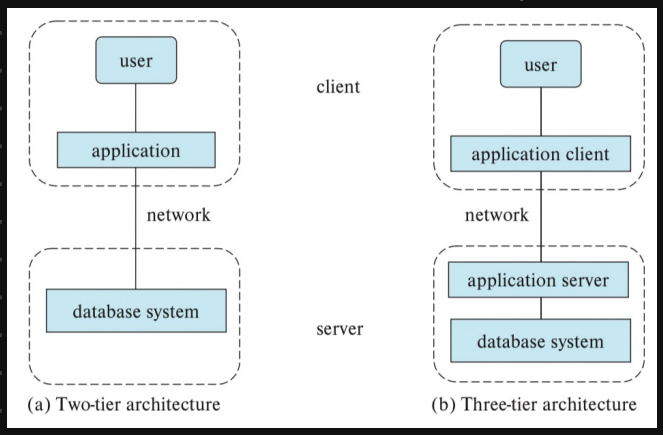
- components of storage manager:
  - ↳ **authorization & integrity manager**
  - ↳ **transaction manager**
    - ensures database remains in consistent state despite system failures
    - deals w/ concurrent transaction executions
  - ↳ **file manager**
    - manages disk space allocation & data structures
  - ↳ **buffer manager**
    - fetches data from disk storage to main mem
    - decides what to cache in main mem
- storage manager implements data structures as part of physical system
  - ↳ **data files** store db itself
  - ↳ **data dictionary** stores metadata abt schema
  - ↳ **indices** provide fast access to data items

- query processor components:
  - ↳ **DDL interpreter**
    - records defns in data dictionary
  - ↳ **DML compiler**
    - translates DML stmts in query lang into evaluation plan that has low-lvl ins that query-eval engine understants
    - can have several alt eval plans that all give same result
    - performs **query optimization**
  - ↳ **query eval engine**
    - executes low-lvl ins generated by DML compiler

- often, we need **transactions**, which are collections of ops that perform single logical fcn in db app
- transactions need to be atomic & consistent
  - ↳ **atomicity** means if any part of transaction fails, entire transaction is rolled back (i.e. failure recovery)
  - ↳ **consistency** means data integrity & validity is maintained
- **recovery manager** ensures atomicity & durability
  - ↳ **durability** means once transaction is committed & changes are saved, they're permanent & will survive later failures / restarts
- **concurrency-control manager** controls interactions among concurrent transactions to ensure consistency
- transaction manager consists of concurrency-control manager & recovery manager

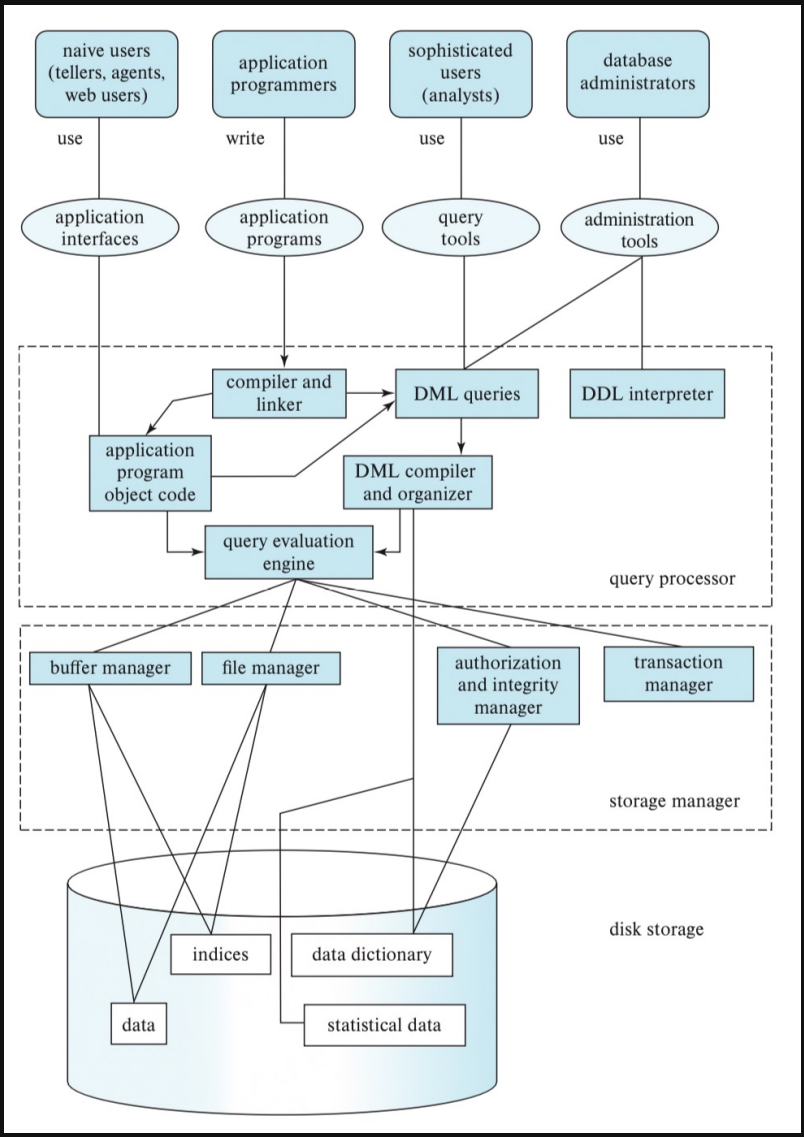
DATABASE AND APPLICATION ARCHITECTURE

- db apps use a **2-tier architecture** (earlier gen) or **3-tier architecture** (modern)
  - ↳



- in 2-tier, app resides at client machine ? invokes db system functionality at server machine thru query lang stmts
- in 3-tier, client machine is only front end ? doesn't contain any direct db calls
  → front end communicates w/ **app server** which then communicates w/ db system to access data
  → **business logic** (i.e. says what actions are to be carried out under certain conditions) is embedded in app server instead of being distributed across multiple clients

· architecture of db system that runs on centralized server machine:



↳ applicable to **shared-mem** server architectures (i.e. multiple CPUs ? exploit parallel processing, but all CPUs access common mem)

THE RELATIONAL MODEL

SIGNATURES AND THE RELATIONAL CALCULUS

- set comprehension syntax for queries: $\{\langle answer\rangle \mid \langle condition\rangle\}$
  - ↳ syntax for each $\langle answer\rangle$ is k-tuple of vars: $(x_1, \ldots, x_k)$
- ans to query is all k-tuples $(c_1, \ldots, c_k)$ of constants denoting values for each var x_i that satisfies $\langle condition\rangle$
- e.g. asking questions abt natural #s

Table PLUS

A1	A2	R
0	0	0
0	1	1
0	2	2
⋮	⋮	⋮
1	0	1
1	1	2
⋮	⋮	⋮
2	0	2
2	1	3
⋮	⋮	⋮

  - ↳ what are all pairs of natural #s that add to 5?
    - Q: $\{(x,y) \mid x+y=5\}$ or $\{(x,y) \mid PLUS(x,y,5)\}$ (rltnal form for basic conditions)
    - A: $\{(0,5), (1,4), (2,3), (3,2), (4,1), (5,0)\}$
    - look for $(x,y,5)$ in table PLUS
  - ↳ what are all pairs of #s that add to same # they subtract to (i.e $x+y = x-y$)?
    - Q: $\{(x,y) \mid \exists z \cdot PLUS(x,y,z) \wedge PLUS(z,y,x)\}$
    - A: $\{(0,0), (1,0), \ldots\}$
    - depends on content/instance of table PLUS
  - ↳ what's neutral elmt of addition?
    - Q: $\{(x) \mid PLUS(x,x,x)\}$
    - A: $\{(0)\}$
- e.g. asking qs abt employees

Table EMP

name	dept	boss
Sue	CS	Bob
Bob	CO	Bob
Fred	PM	Mark
John	PM	Mark
Jim	CS	Fred
Eve	CS	Fred
Sue	PM	Sue

  - ↳ who are all the employees + their departments who work for Bob?
    - Q: $\{(x,y) \mid EMP(x,y,Bob)\}$
    - A: $\{(Sue, CS), (Bob, CO)\}$
    - look for (x,y,Bob) in EMP
  - ↳ who are pairs of employees working for same boss?
    - Q: $\{(x_1,x_2) \mid \exists y_1, y_2, z \cdot EMP(x_1,y_1,z) \wedge EMP(x_2,y_2,z)\}$
    - A: $\{(Sue, Bob), (Fred, John), (Jim, Eve)\}$
  - ↳ who are employees who are their own bosses
    - Q: $\{(x) \mid \exists y \cdot EMP(x,y,x)\}$
    - A: $\{(Sue), (Bob)\}$
- rltnal dbs are based on first order predicate logic (FOL)
- in a rltnal db, all info is organized into finite # of rltns called tables
  - ↳ accomodates data independence
  - ↳ declarative DML based on well-formed formulas (wffs) in FOL
  - ↳ integrity constraints via wffs
  - ↳ e.g.

AUTHOR

aid	name
1	Sue
2	John

WROTE

author	publication
1	1
1	4
2	2
1	2

PUBLICATION

pubid	title
1	Mathematical Logic
3	Trans. on Databases
2	Principles of DB Systems
4	Query Languages

- components of rltnal dbs:
  - ↳ universe: set of values $\mathbb{D}$ (domain) w/equality ($\simeq$) + constants for each value
  - ↳ rltn (i.e. table):
    - intension is rltn/predicate name R + arity (#cols) k of R
      - → written as R/k
    - extension is set of k-tuples (i.e. interpretation)
      - → written as $R \subseteq \mathbb{D}^k$
  - ↳ db includes:
    - signature (i.e. metadata): finite set ρ of predicate names R_i
    - instance (i.e. data, structure): extension R_i for each R_i
- notation for rltnal db:
  - ↳ signature is $\rho = (R_1/k_1, \ldots, R_n/k_n)$

↳ instance is $DB = (D, \approx, R_1, \ldots, R_n)$

· e.g. rltnal dbs

- The integers, with addition *and multiplication*:

  (signature) $\rho = (\texttt{PLUS/3}, \texttt{TIMES/3})$
  (data) $\mathbf{DB} = (\mathbb{Z}, \approx, \mathbf{PLUS}, \mathbf{TIMES})$

- The employee database:

  (signature) $\rho = (\texttt{EMP/3})$
  (data) $\mathbf{DB} = (\mathbb{STR}, \approx, \mathbf{EMP})$

- The simple bibliography database:

  (signature) $\rho = (\texttt{AUTHOR/2}, \texttt{WROTE/2}, \texttt{PUBLICATION/2})$
  (data) $\mathbf{DB} = (\mathbb{STR} \uplus \mathbb{Z}, \approx, \mathbf{AUTHOR}, \mathbf{WROTE}, \mathbf{PUBLICATION})$

· arity is indicated by seq of attributes (i.e. identifiers)

· e.g. bibliography rltnal db ver 2:

(signature) $\rho =$ (

```
AUTHOR (aid, name),
WROTE (author, publication),
PUBLICATION (pubid, title),
BOOK (pubid, publisher, year),
JOURNAL-OR-PROCEEDINGS (pubid),
JOURNAL (pubid, volume, no, year),
PROCEEDINGS (pubid, year),
ARTICLE (pubid, appears-in, startpage, endpage)
```

)

(data) $\mathbf{DB} = (\mathbb{STR} \uplus \mathbb{Z}, \approx,$

$$
\begin{aligned}
\mathbf{AUTHOR} &= \{ (1, \text{Sue}), (2, \text{John}) \}, \\
\mathbf{WROTE} &= \{ (1,1), (1,4), (1,2), (2,2) \}, \\
\mathbf{PUBLICATION} &= \{ (1, \text{Mathematical Logic}), \\
&\quad (3, \text{Trans. on Databases}), \\
&\quad (2, \text{Principles of DB Systems}), \\
&\quad (4, \text{Query Languages}) \}, \\
\mathbf{BOOK} &= \{ (1, \text{AMS}, 1990) \}, \\
\mathbf{JOURNAL\text{-}OR\text{-}PROCEEDINGS} &= \{ (2), (3) \}, \\
\mathbf{JOURNAL} &= \{ (3, 35, 1, 1990) \}, \\
\mathbf{PROCEEDINGS} &= \{ (2, 1995) \}, \\
\mathbf{ARTICLE} &= \{ (4, 2, 30, 41) \}
\end{aligned}
$$

)

· rltnships btwn values (i.e. tuples) that present in an instance are true while absent rltnships are false

· use vars + valuations to generate query conditions

↳ e.g. AUTHOR(x,y) will be true of any valuation $\{x \mapsto v_1, y \mapsto v_2, \ldots\}$ exactly when 2-tuple of values (v_1, v_2) occur in AUTHOR

↳ valuation is fcn θ that maps var names to vals in universe

- written as $\theta : \{x_1, x_2, \ldots\} \to D$

- to denote modification to θ where x is instead mapped to value v, we write $\theta[x \mapsto v]$

- can allow for more complex conditions by using:
  - ↳ **logical connectives**:
    - conjunction (and)
      - → written as $\wedge$
    - disjunction (or)
      - → written as $\vee$
    - negation (not)
      - → written as $\neg$
  - ↳ **quantifiers**:
    - existential (there is)
      - → written as $\exists$

- given db signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, set of var names $\{x_1, x_2, \ldots\}$, ? set of constants $\{c_1, c_2, \ldots\}$, we can form **conditions**, which are wffs
  - ↳ e.g. $\varphi ::= \underbrace{R_i(x_{i1}, \ldots, x_{ik_i}) \mid x_i = x_j \mid x_i = c_j \mid \varphi_1 \wedge \varphi_2 \mid \exists x_i \cdot \varphi_1}_{\text{conjunctive formulas}} \mid \underbrace{\varphi_1 \vee \varphi_2}_{\text{positive formulas}} \mid \underbrace{\neg \varphi_1}_{\text{first-order formulas}}$
    - **conjunctive** means only conjunction (i.e. AND) is used
    - **positive** means there's no negations

- condition is **sentence** when it doesn't have free vars (i.e. all vars bounded by quantifiers)

free vars of formula φ (written $Fv(\varphi)$, are:

- ↳ $Fv(R(x_{i_1}, \ldots, x_{i_k})) = \{x_{i_1}, \ldots, x_{i_k}\}$;
- ↳ $Fv(x_i = x_j) = \{x_i, x_j\}$;
- ↳ $Fv(x_i = c_j) = \{x_i\}$;
- ↳ $Fv(\varphi \wedge \psi) = Fv(\varphi) \cup Fv(\psi)$;
- ↳ $Fv(\exists x_i \cdot \varphi) = Fv(\varphi) - \{x_i\}$;
- ↳ $Fv(\varphi \vee \psi) = Fv(\varphi) \cup Fv(\psi)$;
- ↳ $Fv(\neg \varphi) = Fv(\varphi)$;

- **truth** of condition / formula φ over signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$ is defined wrt db instance $DB = (D, \approx, R_1, \ldots, R_n)$? valuation $\theta : \{x_1, x_2, \ldots\} \to D$:

$\mathbf{DB}, \theta \models R_i(x_{i,1}, \ldots, x_{i,k_i})$	if $(\theta(x_{i,1}), \ldots, \theta(x_{i,k_i})) \in \mathbf{R}_i$;
$\mathbf{DB}, \theta \models x_i = x_j$	if $\theta(x_i) \approx \theta(x_j)$;
$\mathbf{DB}, \theta \models x_i = c_j$	if $\theta(x_i) \approx c_j$;
$\mathbf{DB}, \theta \models \varphi \wedge \psi$	if $\mathbf{DB}, \theta \models \varphi$ and $\mathbf{DB}, \theta \models \psi$;
$\mathbf{DB}, \theta \models \exists x_i . \varphi$	if $\mathbf{DB}, \theta[x_i \mapsto v] \models \varphi$, for some $v \in \mathbf{D}$;
$\mathbf{DB}, \theta \models \varphi \vee \psi$	if $\mathbf{DB}, \theta \models \varphi$ or $\mathbf{DB}, \theta \models \psi$; and
$\mathbf{DB}, \theta \models \neg \varphi$	if $\mathbf{DB}, \theta \not\models \varphi$.

- some equivalences ? syntactic sugar:

Boolean Equivalences

- $\neg(\neg\varphi_1) \equiv \varphi_1$
- $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $\varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$
- $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$

First-order Equivalences

- $\forall x.\varphi \equiv \neg\exists x.\neg\varphi$

Additional Syntactic Sugar

- $R(\ldots, c, \ldots) \equiv \exists x.(R(\ldots, x, \ldots) \wedge x = c)$, where x is *fresh*
- $\exists x_1, \cdots, x_n.\varphi \equiv \exists x_1.\cdots.\exists x_n.\varphi$
- $R(\ldots, -, \ldots) \equiv \exists x.R(\ldots, x, \ldots)$, where x is *fresh*

↳ x is fresh means x is new var

- relational calculus (RC) query is set comprehension of form $\{(x_1, \ldots, x_k) \mid \varphi\}$
  ↳ $\{x_1, \ldots, x_k\} = Fv(\varphi)$
- answers to query $\{(x_1, \ldots, x_k) \mid \varphi\}$ over DB is rltn $\{(\theta(x_1), \ldots, \theta(x_k)) \mid DB, \theta \models \varphi\}$
  ↳ i.e. valuations applied to tuples of vars that make wff φ true wrt a db
- e.g. who are pairs of employees working for same boss?
  Q: $\{(x_1, x_2) \mid \exists y_1, y_2, z \cdot EMP(x_1, y_1, z) \wedge EMP(x_2, y_2, z)\}$
  A: $\{(Jim, Eve), \ldots\}$
  ↳ justification:
  1) $DB, \theta_1 (= \{x_1 \mapsto Jim, y_1 \mapsto CS, z \mapsto Fred, \ldots\}) \models EMP(x_1, y_1, z)$
  2) $DB, \theta_2 (= \{x_2 \mapsto Eve, y_2 \mapsto CS, z \mapsto Fred, \ldots\}) \models EMP(x_2, y_2, z)$
  3) $DB, \theta_3 (= \{x_1 \mapsto Jim, y_1 \mapsto CS, x_2 \mapsto Eve, y_2 \mapsto CS, z \mapsto Fred, \ldots\}) \models EMP(x_1, y_1, z) \wedge EMP(x_2, y_2, z)$
  4) $DB, \theta_4 (= \{x_1 \mapsto Jim, x_2 \mapsto Eve, \ldots\}) \models \exists y_1, y_2, z \cdot EMP(x_1, y_1, z) \wedge EMP(x_2, y_2, z)$
     → check $\{x_1, x_2\} = Fv(\exists y_1, y_2, z \cdot EMP(x_1, y_1, z) \wedge EMP(x_2, y_2, z))$
  5) $(\theta_4(x_1), \theta_4(x_2)) = (Jim, Eve)$
  ↳ $\rho = (EMP/3)$
  ↳ $DB = (STR, \approx, EMP)$

INTEGRITY CONSTRAINTS

- integrity constraints are formulas that should be true for all instances of db
  ↳ i.e. rules/conditions to ensure consistency, accuracy, & validity of data
  ↳ e.g. consider $EMP(x, y, z)$ where x is employee, y is department, & z is boss
  - "Every boss manages a unique department"
    $\neg(\exists b, d_1, d_2 \cdot EMP(-, d_1, b) \wedge EMP(-, d_2, b) \wedge (d_1 \neq d_2))$
  - "No boss has someone else as their boss"
    $\neg(\exists b_1, b_2 \; EMP(-, -, b_1) \wedge EMP(b_1, -, b_2) \wedge (b_1 \neq b_2))$
- e.g. using rltnal db for bibliography ver 2, we can write some RC queries:

(signature) $\rho = ($

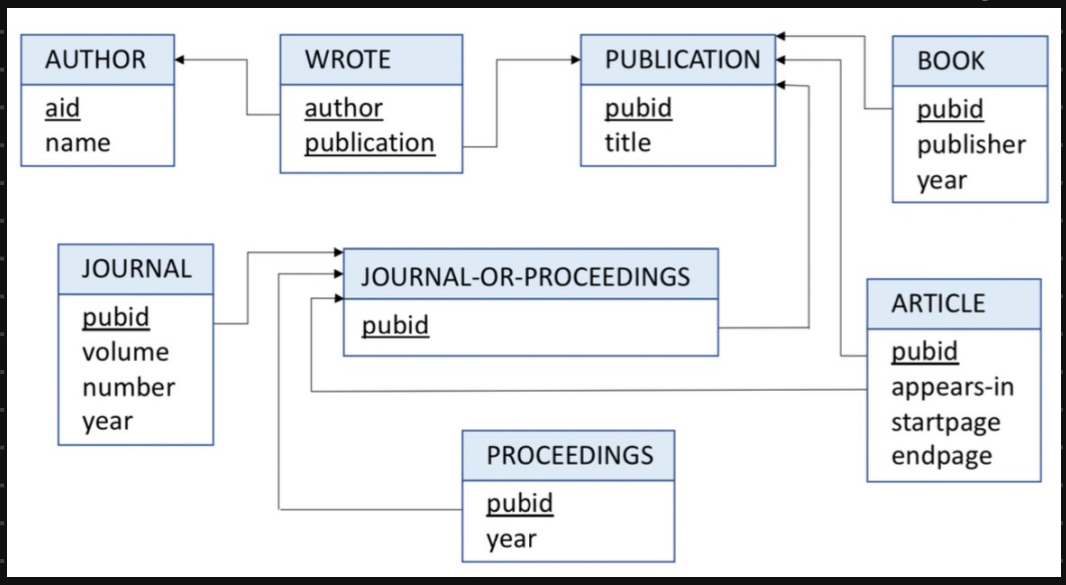
```
AUTHOR (aid, name),
WROTE (author, publication),
PUBLICATION (pubid, title),
BOOK (pubid, publisher, year),
JOURNAL-OR-PROCEEDINGS (pubid),
JOURNAL (pubid, volume, no, year),
PROCEEDINGS (pubid, year),
ARTICLE (pubid, appears-in, startpage, endpage)
```

)

↳ what are publication titles that are journals / proceedings?
$\{t \mid \exists p.\ PUBLICATION(p, t) \wedge JOURNAL\text{-}OR\text{-}PROCEEDINGS(p)\}$
↳ what are publication titles w/o authors?
$\{t \mid \exists p.\ PUBLICATION(p, t) \wedge \neg(\exists a.\ WROTE(a, p))\}$
↳ what are names of authors who've written at least 1 book?
$\{n \mid \exists a.\ AUTHOR(a, n) \wedge \exists p.\ WROTE(a, p) \wedge BOOK(p, -, -)\}$
↳ what are publication titles written by single author?

$\{t \mid \exists p\ \text{PUBLICATION}(p,t) \wedge \exists a.\ \text{WROTE}(a,p) \wedge (\forall b.\ \text{WROTE}(b,p) \rightarrow a = b)\}$

- rltnal signature only captures structure of rltns so valid dbs have to satisfy additional integrity constraints in form of sentences over signature
  - ↳ some examples of integrity constraints:
    - values of particular attribute belong to prescribed **data type**
    - values of attributes are unique among tuples in rltn (i.e. **primary keys**)
    - values appearing in 1 rltn must also appear in another rltn (i.e. **referential integrity / foreign keys**)
    - values can't appear simultaneously in certain rltns (i.e. **disjointness**)
    - values in rltn must appear in at least 1 other set of rltns (i.e. **coverage**)
  - ↳ e.g.



    - underlined attributes are primary keys
    - arrow from some attribute x in table to diff table t, x must appear as primary key in t
- **extension** of table can be determined by integrity constraint
  - ↳ e.g. $\forall p.\ \text{JOURNAL-OR-PROCEEDINGS}(p) \leftrightarrow (\text{JOURNAL}(p) \vee \text{PROCEEDINGS}(p))$
- given signature ρ, table R occurring in ρ is **view** when rltnal db schema contains 1 integrity constraint of form $\forall x_1, \ldots, x_k.\ R(x_1, \ldots, x_k) \leftrightarrow \varphi$
  - ↳ $\{x_1, \ldots, x_k\} = Fv(\varphi)$
  - ↳ φ is **view definition** of R
  - ↳ R depends on any table mentioned in φ
  - ↳ no table occurring in schema is allowed to depend on itself
- **rltnal db schema** is pair (ρ, Σ)
  - ↳ ρ is signature
  - ↳ Σ is finite set of integrity constraints that are sentences over ρ
- **rltnal db** is rltnal db schema (ρ, Σ) + instance DB of ρ
  - ↳ **consistent** iff for any integrity constraint $\varphi \in \Sigma$ + any valuation θ, $DB, \theta \models \varphi$

SAFETY AND FINITENESS

- rltnal dbs + RC queries should have following properties:
  - ↳ extension of any rltn in signature should be **finite**
  - ↳ queries should be **safe**
    - i.e. ans should be finite when db instances are finite
- RC query is **domain independent** when for any pair of instances $DB_1 = (D_1, \sim, R_1, \ldots, R_k)$ + $DB_2 = (D_2, \approx, R_1, \ldots, R_k)$ + any θ, $DB_1, \theta \models \varphi$ iff $DB_2, \theta \models \varphi$
  - ↳ safety ↔ domain independence + finite db instances
- language of dom indep RC queries is impossible so we use **range restricted RC**
- given db signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, set of var names $\{x_1, x_2, \ldots\}$ + set of constants $\{c_1, c_2, \ldots\}$, range restricted conditions are formulas defined by grammar:

$\varphi ::= R_i(x_{i1}, \ldots, x_{ik})$

$\varphi_1 \wedge (x_i = x_j)$ $\quad \{x_i, x_j\} \cap Fv(\varphi_1) \neq \emptyset$

$x_i = c_j$

$\varphi_1 \wedge \varphi_2$

$\exists x_1 \cdot \varphi_1$

$\varphi_1 \vee \varphi_2$ $\quad Fv(\varphi_1) = Fv(\varphi_2)$

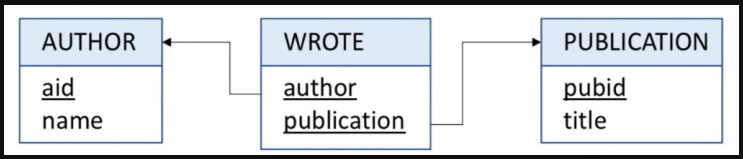
$\varphi_1 \wedge \neg \varphi_2$ $\quad Fv(\varphi_1) = Fv(\varphi_2)$

- **range restricted RC query** has form of $\{(x_1, \ldots, x_n) \mid \varphi\}$ where $\{x_1, \ldots, x_n\} = Fv(\varphi)$ & φ is range restricted condition
  - ↳ query lang is rltnally complete if lang is at least as expressive as range restricted RC
- every range restricted RC query is domain indep RC query
- every domain indep RC query has equiv formulation as range restricted RC query

introduction to SQL

RELATIONAL SCHEMATA AND CONJUNCTIVE QUERIES

- e.g.



↳ expressed as DDL requests in SQL:

```
create table AUTHOR (
      aid integer not null,
      name varchar(10) not null,
      primary key (aid) )

create table PUBLICATION (
      pubid integer not null,
      title varchar(25) not null,
      primary key (pubid) )
```

```
create table WROTE (
      author integer not null,
      publication integer not null,
      primary key (author, publication),
      foreign key (author) references AUTHOR,
      foreign key (publication) references PUBLICATION )
```

DDL requests include **integrity constraints**:

↳ **data type** constraints for each column
- mandatory for each attribute

↳ **"not null"** constraints
- strongly desired for each column

↳ **"primary key"** constraints
- unique identifiers for records

↳ **"foreign key"** constraints
- establish rltnships btwn tables by referencing primary key of another table

SQL is not case-sensitive

- basic SQL fixes universe of rlntal db instance to be union of vals given by pre-defined collection of **data types**:

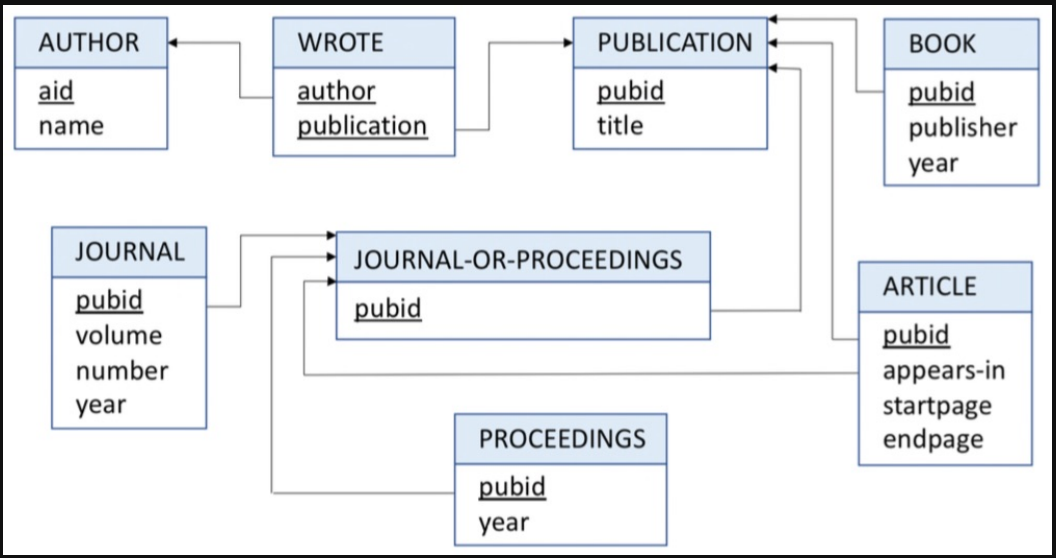
↳ integer : integer (32 bit)
↳ smallint: integer (16 bit)
↳ decimal(m,n): fixed decimal
↳ float: IEEE float (32 bit)
↳ char(n): char str (length n)
↳ varchar(n): var length str (max length n)
↳ date: yr/month/day
↳ time: hh:mm:ss.ss

- **syntax**:

```
SELECT DISTINCT  <results>
FROM             <tables>
WHERE            <condition>
```

↳ translates RC queries of form $\{ <results> \mid \exists <unused> . (\bigwedge <tables>) \wedge <condition> \}$
- conjunction of <tables> w/ condition
- <results> specifies vals in resulting tuples

- <unused> are vars not used in <results>
- SQL code assumes following schema & data:



$$
\begin{aligned}
\textbf{DB} = (\mathbb{STR} \uplus \mathbb{Z}, \approx,\\
\textbf{AUTHOR} &= \{ (1, \text{Sue}), (2, \text{John}) \},\\
\textbf{WROTE} &= \{ (1,1), (1,4), (1,2), (2,2) \},\\
\textbf{PUBLICATION} &= \{ (1, \text{Mathematical Logic}),\\
&\quad (3, \text{Trans. on Databases}),\\
&\quad (2, \text{Principles of DB Systems}),\\
&\quad (4, \text{Query Languages}) \},\\
\textbf{BOOK} &= \{ (1, \text{AMS}, 1990) \},\\
\textbf{JOURNAL-OR-PROCEEDINGS} &= \{ (2), (3) \},\\
\textbf{JOURNAL} &= \{ (3, 35, 1, 1990) \},\\
\textbf{PROCEEDINGS} &= \{ (2, 1995) \},\\
\textbf{ARTICLE} &= \{ (4, 2, 30, 41) \}\\
)
\end{aligned}
$$

- e.g. list all publications in db
  ```
  select distinct * \   ← signifies new line
  from publication
  ```
- RC uses positional notation so there's no need for attribute names
  - ↳ e.g. EMP(x,y,z) is true when x,y,z components of ans can be found as tuple in instance of EMP
  - ↳ inconvenient for rlths w/ high arity
- SQL uses correlations (i.e. tuple vars) & attributes to assign default var names to components of tuples
- e.g. list all publications w/ at least 2 authors

  $\{(p) \mid \exists a_1, a_2 .\ \text{WROTE}(a_1, p) \wedge \text{WROTE}(a_2, p) \wedge \neg a_1 \approx a_2\}$

  ```
  select distinct r1.publication \
  from wrote r1, wrote r2 \
  where r1.publication = r2.publication \
  and r1.author != r2.author
  ```
  - ↳ can't share var p in 2 WROTE tables in SQL
- e.g. list titles of all books

  $\{(t) \mid \exists p, b, y .\ \text{PUBLICATION}(p, t) \wedge \text{BOOK}(p, b, y)\}$

  ```
  select distinct title \
  ```

```
from publication, book \
where publication.pubid = book.pubid
```

↳ rlnms can serve as their own co-rlnms when unambiguous
- above, publication stands for "publication publication" (i.e. publication(publication.pubid, publication.title))
- i.e. can use same rlnm in select & from clause

- syntax for FROM clause is: from R_1 [[as] n_1], ..., R_k [[as] n_k]
  - ↳ R_i are rlnm (i.e. table) names
  - ↳ n_i are distinct identifiers
  - ↳ rep conjunction $R_1 \wedge \ldots \wedge R_k$
    - all vars of R_is are distinct
    - use co-rlnm names to solve ambiguities
  - ↳ can't appear alone
- syntax for SELECT clause: select distinct e_1 [[as] n_1], ..., e_k [[as] n_k]
  - ↳ eliminates unnecessary attributes & remaining duplicates from ans (3)
  - ↳ evals exprs e_i
  - ↳ gives name n_i to expr vals in ans
- can create vals in ans tuples using built-in fcns
  - ↳ numeric types: +, -, *, /, ... (usual arithmetic)
  - ↳ strs: || (concatenation), substr, ...
  - ↳ constants: "select 1" is valid query
  - ↳ user defined fcns (UDF)
  - ↳ note all attribute names must be present in FROM clause
- e.g.

For every article list the number of pages.

```
select distinct pubid, endpage-startpage+1 \
from article

PUBID       2
----------- -----------
          4          12

  1 record(s) selected.
```

- results of queries should look the same as instances of tables
  - ↳ query result columns should have attributes
  - ↳ as a result of a SELECT clause:
    - single attribute inherits name
    - exprs are implementation dependent
  - ↳ always ensure attribute name n is given for each select expr e, appending "as n" if necessary
  - ↳ e.g.

For every article, list the number of pages, and name the resulting attributes `id`, *and* `numberofpages`.

```
select distinct pubid as id, \
       endpage-startpage+1 as numberofpages \
from article

ID          NUMBEROFPAGES
----------- -------------
          4            12

  1 record(s) selected.
```

- syntax for WHERE clause: where <condition>
  - ↳ additional conditions on qualifying tuples
  - ↳ can be std atomic conditions:
    - equality: =, !=
      - → all types
    - order: <, <=, >, >=, <>
      - → numeric & str types
      - → <> is same as !=
  - ↳ conditions may involve exprs

↳ e.g.

Find all journals printed since 1997.

```
select * from journal where year >= 1997

PUBID        VOLUME       NUMBER       YEAR
-----------  -----------  -----------  ----

  0 record(s) selected.
```

Find all articles with more than 4 pages.

```
select * from article \
where endpage-startpage > 4

PUBID        APPEARS_IN   STARTPAGE    ENDPAGE
-----------  -----------  -----------  -----------
           4            2           30           41

  1 record(s) selected.
```

↳ atomic conditions can be combined using **boolean connectives**: AND, OR, & NOT

- E.g.: *List all publications with at least two authors.*

```
select distinct r1.publication \
from wrote r1, wrote r2 \
where r1.publication = r2.publication \
and not r1.author = r2.author

PUBLICATION
-----------
          2

  1 record(s) selected.
```

SET OPERATIONS AND FIRST ORDER QUERIES

- SELECT block queries only cover $\exists$ & $\wedge$ queries
- for remaining connectives:
  - ↳ $\vee$ & $\neg$ expressed using **set ops**
    - easy to enforce range-restriction requirements
  - ↳ $\forall$, $\rightarrow$, & $\leftrightarrow$ are rewritten using negation & $\exists$
- since ans to SELECT blocks are rlns (i.e. sets of tuples), can apply set ops on them
  - ↳ **set union**: Q_1 UNION Q_2
    - set of tuples in Q_1 or in Q_2
  - ↳ **set diff**: Q_1 EXCEPT Q_2
    - set of tuples in Q_1 but not Q_2
  - ↳ **set intersection**: Q_1 INTERSECT Q_2
    - set of tuples in Q_1 and Q_2
  - ↳ Q_1 & Q_2 must have **union-compatible signatures**
    - i.e. same # & types of attributes
- e.g union

List all publication ids for books or journals.

```
(select distinct pubid from book) \
union \
(select distinct pubid from journal)
```

e.g. set diff

List all publication ids except those for articles.

```
(select distinct pubid from publication) \
except \
(select distinct pubid from article)
```

to use set op inside SELECT block:
↳ distributive laws: $(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$
  ◦ very cumbersome
↳ nest set op inside SELECT block

- queries denote rltns & SQL provides naming mechanism to assign names to query results
  ↳ subquery names can be used later in place of base rltns
  ↳ syntax:

```
WITH T1 [<opt-schema-1>] AS ( <query-1-goes-here> ),
     ...
     Tn [<opt-schema-n>] AS ( <query-n-goes-here> )
<query-that-uses-T1-to-Tn-as-table-names>
```

↳ e.g.

List all publication titles for books or journals.

```
with bookorjournal (pubid) as ( \
     (select distinct pubid from book) \
     union \
     (select distinct pubid from journal) ) \
select distinct title \
from publication, bookorjournal \
where publication.pubid = bookorjournal.pubid

TITLE
-------------------------
Mathematical Logic
Principles of DB Systems

  2 record(s) selected.
```

- using WITH construct is cumbersome when faced w/ many subexprs
  ↳ SQL-92 allows inlining queries in FROM clause · FROM ..., (<query-here>) <id>,..
    ◦ <id> is result of <query-here> & unlike base rltns, it's mandatory
    ◦ e.g. simplication of above query

```
select distinct title \
from publication, ( \
     (select distinct pubid from book) \
     union \
     (select distinct pubid from journal) ) as jb \
where publication.pubid = jb.pubid
```

- common mistake is using OR in WHERE clause instead of UNION operator
  ↳ e.g.

An incorrect SQL query to compute *all publication titles for journals or books*:

```
select distinct title
from publication, book, journal
where publication.pubid = book.pubid
or publication.pubid = journal.pubid
```

  ◦ doesn't work when there's no books or no journals
- so far, SQL can capture all of RC

NESTED QUERIES

- SQL allows conditions in WHERE clause to be expressed w/ subqueries

↳ same as nested quantifiers in RC conditions

- WHERE subqueries include:
  - ↳ presence/absence of single val in subquery (query is unary)
    - <attr> IN (<query>)
    - <attr> NOT IN (<query>)
  - ↳ rltnship of value to some/all vals in subquery (query is unary)
    - <attr> op SOME (<query>)
    - <attr> op ALL (<query>)
  - ↳ emptiness/non-emptiness of subquery
    - EXISTS (<query>)
    - NOT EXISTS (<query>)
- e.g. <attr> in (<query>)

Get the titles of all articles.

```
select distinct title \
from publication \
where pubid in ( select pubid from article )
```

- nesting in WHERE clause is syntactic sugar (i.e. easier for us to understand)
  - ↳ e.g.

```
select r.b                  select r.b
from r                      from r, (
where  r.a in (      ↔        select distinct b
  select b                    from s
  from s                    ) as s
)                           where  r.a = s.b
```

- e.g. <attr> not in (<query>)

All author-publication ids for all publications except books and journals.

```
select * from wrote \
where publication not in ( \
     (select pubid from book) \
     union \
     (select pubid from journal) )
```

↳ can also use boolean connectives for another formulation:

```
select * from wrote \
where publication not in ( \
      select pubid from book ) \
and publication not in ( \
      select pubid from journal )
```

- e.g. <attr> op SOME/ALL (<query>)

Find the longest articles (a way of expressing max).

```
select distinct pubid \
from article \
where endpage-startpage >= all ( \
     select endpage - startpage \
     from article )
```

- <attr> = some (<query>) ≡ <attr> in (<query>)
- <attr> <> all (<query>) ≡ <attr> not in (<query>)

- so far, subqueries have been independent of main query
  ↳ not correlated
- SQL allows **parametric (correlated) subqueries**
  ↳ form of <query> that includes $<attr>_1$, $<attr>_2$, ...
    - $<attr>_i$ is attr in main query
  ↳ e.g. EXISTS (<query>)

```
select * from wrote r \
where exists ( select * \
      from wrote s \
      where r.publication = s.publication \
      and r.author <> s.author )
```

attrs that are part of main query

  ↳ e.g. NOT EXISTS (<query>)

```
select * from wrote r \
where not exists ( select * \
      from wrote s \
      where r.publication = s.publication \
      and r.author <> s.author )
```

  ↳ e.g. <attr> IN (<query>)

```
select * from wrote r \
where publication in ( \
      select publication \
      from wrote s \
      where r.author <> s.author )
```

- WHERE subqueries are just queries so we can nest repeatedly
  ↳ every nested subquery can use attrs from enclosing queries as params
- attrs present in subqueries only can't be used to construct results
- e.g.

List all authors who always publish with someone else.

```
select distinct a1.name \
from author a1, author a2 \
where not exists ( \
      select *  \
      from publication p, wrote w1 \
      where p.pubid = w1.publication \
      and a1.aid = w1.author \
      and a2.aid not in ( \
            select author from wrote \
            where publication = p.pubid \
            and author <> a1.aid ) )
```

- WHERE subqueries enable easy formulation of queries in form "all x in R st (a part of x) doesn't appear in S"
  ↳ subqueries only stand for WHERE conditions ∴ can't be used to prod results
  ↳ input params must be bound in main query

AGGREGATE QUERIES

- **aggregation** is std & useful extension of first order logic (FOL)
- aggregate(col) fns are used to:
  ↳ count #tuples in r/tn

↳ sum vals of numeric attr over a rltn
↳ find min/max vals of numeric attr over a rltn

- can apply to groups of tuples that have equal vals for selected attrs
- usually can't be expressed as RC query except for finding min/max vals
- syntax for aggregation:

```
SELECT   x1,...,xk, agg1 [[AS] n1],..., aggj [[AS] nj]
<FROM-WHERE>
[GROUP BY x1,...,xk]
```

↳ all attrs in SELECT clause that aren't in scope of an aggregate fcn must appear in GROUP BY clause
↳ aggi are of form count(*), count(<expr>), sum(<expr>), min(<expr>), max(<expr>), or avg(<expr>)
  - <expr> is usually attr of Q ; not in GROUP BY clause

- results are aggregated as such:
  1) partition result of <FROM-WHERE> into smallest # groups w/each group having eq vals of grouping attrs
     - single group if there's no GROUP BY clause
  2) on each of partitions, apply aggregate fcns
  3) for each group, add tuple w/ grouping attr vals & results of aggregate fcns to result
  ↳ always good to name results of aggregate fcns in SELECT clause
- e.g. COUNT

For each publication, count the number of authors.

```
select publication, count(author) \
from wrote \
group by publication

PUBLICATION 2
------------ ------------
           1            1
           2            2
           4            1

  3 record(s) selected.
```

- e.g. SUM

For each author, a count of the number of article pages.

```
select author, sum(endpage-startpage+1) as pcnt \
from wrote, article \
where publication = pubid \
group by author

AUTHOR       PCNT
------------ ------------
           1           12
  2         2            0
  1 record(s) selected.
```

- WHERE clause can't impose conditions on vals of aggregates b/c WHERE conditions are applied before GROUP BY
- HAVING clause is like WHERE for aggregate vals
  ↳ aggregate fcns in HAVING clause may be diff from those in SELECT clause but grouping is commom
  ↳ e.g.

All publications with more than one author.

```
select publication, count(author) as acnt \
from wrote \
group by publication \
having count(author) > 1
```

- e.g.

For each author, the id, name and count of the number of books and articles.

```
select distinct aid, name, count(publication) as pubcnt \
from author, ( \
     ( select distinct author, publication  \
       from wrote, book  \
       where publication = pubid ) \
     union \
     ( select distinct author, publication \
       from wrote, article \
       where publication = pubid ) ) ba \
where aid = ba.author \
group by aid, name
```

TRANSACTIONS AND DATABASE UPDATE

- 3 kinds of table update:
  - ↳ **INSERT** cmd inserts single constant tuple or each tuple in result of query
  - ↳ **DELETE** cmd removes all tuples satisfying a condition
  - ↳ **UPDATE** cmd updates in place all tuples satisfying a condition
- to **insert single tuple**: INSERT INTO $T[(A_1,\ldots,A_k)]$ VALUES $(c_1,\ldots,c_k)$
  - ↳ adds tuple $(c_1,\ldots,c_k)$ to table T
  - ↳ c_i must be val in data type for attr A_i
  - ↳ e.g.

Add Martha as new author with author identification 4.

```
insert into author (aid, name) \
values (4, 'Martha')

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John
          4 Martha

  3 record(s) selected.
```

- to insert multiple tuples given by query: INSERT INTO T(Q)
  - ↳ adds each tuple computed by Q to table T
  - ↳ e.g.

Add Tim as an author, with a new unique identification.

```
insert into author ( \
     select max(aid) + 1, 'Tim' \
     from author )

DB20000I  The SQL command completed successfully.
```

```
select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John
          4 Martha
          5 Tim

  4 record(s) selected.
```

- syntax for **deletion** using condition: DELETE FROM T WHERE <condition>
  - ↳ deletes all tuples that match <condition>
  - ↳

Delete all authors who have not written anything.

```
delete from author \
where not exists ( select * from wrote \
                   where author = aid )

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John

  2 record(s) selected.
```

- syntax for **updating**: UPDATE T
  SET <assignment>
  WHERE <condition>
  - ↳ e.g.

Update anyone named Sue to be named Susan instead.

```
update author \
set name = 'Susan' \
where aid in ( \
     select aid from author \
     where name = 'Sue' )

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Susan
          2 John

  2 record(s) selected.
```

ADVANCED SQL

GENERAL INTEGRITY CONSTRAINTS AND VIEWS

- to make **general integrity constraints**, syntax is: CREATE ASSERTION <assertion-name> CHECK (<condition>)
  - ↳ <condition> must always be true to RDBMS
    - transaction fails if it makes <condition> false
  - ↳ <condition> can contain subqueries in SQL std
- e.g. uniqueness constraints

No pair of publications have the same pubid.

```
create assertion unique-pubid
check (
   not exists (
      select * from publication p1, publication p2
      where p1.pubid = p2.pubid
      and p1.title != p2.title ) )
```

  - ↳ in RC, $\forall p, t_1, t_2 \cdot (\text{PUBLICATION}(p, t_1) \wedge \text{PUBLICATION}(p, t_2) \rightarrow t_1 = t_2)$
  - ↳ **equality generating dependency**
- e.g. foreign keys

Every `author` *value in a* `WROTE` *tuple occurs as an* `aid` *value of some tuple in the* `AUTHOR` *table.*

```
create assertion author-foreign-key
check (
   not exists (
      select * from wrote w
      where not exists (
         select * from author a
         where a.aid = w.author ) ) )
```

  - ↳ in RC, $\forall a \cdot (\text{WROTE}(a, -) \rightarrow \text{AUTHOR}(a, -))$
  - ↳ **tuple generating dependency**
- to create **view** in SQL, syntax is: CREATE VIEW <view-name> [AS] (<query>)
  - ↳ view has many of same properties as table:
    - its defn is in db schema
    - access controls can be applied to it
    - other views can be defined in terms of it
    - can be queried as if it's table
  - ↳ unlike tables:
    - only some views are updatable
    - RDBMS adds updates to transaction to ensure view consistency
      - → updates to view if underlying tables / views are updated
      - → updates to underlying tables / views if view is updatable / it's updated
  - ↳ e.g.

A view called `PUBSWITHAUTHORS` *with ids, titles and author count of publications with at least one author.*

```
create view pubswithauthors as ( \
     select distinct pubid, title, count(*) as acnt \
     from publication, wrote where pubid = publication \
     group by publid, title )

DB20000I  The SQL command completed successfully.

select * from pubswithauthors

PUBID       TITLE                     ACNT
----------- ------------------------- -----------
          1 Mathematical Logic                  1
          2 Trans. on Databases                 2
          4 Query Languages                     1

  3 record(s) selected.
```

- modifications to view's instance must be propagated to tables & other views referenced by view query
- some views can't be updated unambiguously
  - ↳ e.g.

Tables:

PERSON

NAME	CITIZENSHIP
Ed	Canadian
Dave	Canadian
Wes	American

NATIONAL-PASTIME

CITIZENSHIP	PASTIME
Canadian	Hockey
Canadian	Curling
American	Hockey
American	Baseball

View: $\{(n, p) \mid \exists c.\text{PERSON}(n, c) \wedge \text{NATIONAL-PASTIME}(c, p)\}$

PERSONAL-PASTIME

NAME	PASTIME
Ed	Hockey
Ed	Curling
Dave	Hockey
Dave	Curling
Wes	Hockey
Wes	Baseball

  - if we insert (Darryl, Hockey), we don't know his nationality
  - if we delete (Dave, Curling), we don't know if we can delete (Canadian, Curling) from NATIONAL-PASTIME table
- view is only **updatable**:
  - ↳ query references 1 table
  - ↳ query outputs simple attrs (i.e. no exprs)
  - ↳ there's no grouping/aggregation / DISTINCT
  - ↳ there's no nested queries
  - ↳ there are no set ops

MULTISET SEMANTICS

- SQL has **multiset / bag semantics** that allow duplicates
  - ↳ SQL tables, views, & query results are multisets of tuples
- e.g.

EMP

NAME	DEPT
bob	cs
sue	cs
fred	pmath
barb	stats
jim	stats

$\{(y) \mid \exists x.\text{EMP}(x, y)\} \Longrightarrow$

DEPT
cs
cs
pmath
stats
stats

$\Longleftrightarrow$

DEPT	CNT
cs	2
pmath	1
stats	2

EMP

NAME	DEPT
bob	cs
sue	cs
fred	pmath
barb	stats
jim	stats

$\{(y) \mid \text{elim } \exists x.\text{EMP}(x, y)\} \Longrightarrow$

DEPT
cs
pmath
stats

$\Longleftrightarrow$

DEPT	CNT
cs	1
pmath	1
stats	1

  - ↳ each R/k ep (i.e. table w/ arity k) has hidden col that rep count of # occurrences of a tuple in any extension of R
  - ↳ multiset semantics are assumed for $\exists$ in RC query
  - ↳ additional condition "elim φ" removes duplicates
- give db signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, set of var names $\{x_1, x_2, \ldots\}$, & set of constants $\{c_1, c_2, \ldots\}$, **range restricted conditions** are formulas defined by grammar:

$\varphi ::= R_i(x_{i1}, \ldots, x_{ik_i})$
$\quad \varphi_1 \wedge (x_i = x_j)$ where $\{x_i, x_j\} \cap Fv(\varphi_1) \neq \emptyset$
$\quad x_i = c_j$
$\quad \varphi_1 \wedge \varphi_2$
$\quad \exists x_i . \varphi_1$
$\quad \varphi_1 \vee \varphi_2$ where $Fv(\varphi_1) = Fv(\varphi_2)$
$\quad \varphi_1 \wedge \neg \varphi_2$ where $Fv(\varphi_1) = Fv(\varphi_2)$
$\quad \text{elim } \varphi$

  - ↳ range restricted RC query has form $\{(x_1, \ldots, x_n) \mid \varphi\}$ where $\{x_1, \ldots, x_n\} = Fv(\varphi)$ & φ is range restricted condition

↳ note that direct access to repetition counts is not possible

- multiset semantics for range restricted RC:

A formula φ over a signature $\rho = (R_1/k_1, \ldots R_n/k_n)$ is true m *times* respect to *database instance* $\mathbf{DB} = (\mathbf{D}, \approx, \mathbf{R}_1, \ldots, \mathbf{R}_n)$ and *valuation* $\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}$, written $\mathbf{DB}, \theta, m \models \varphi$, as given by the following:

$$
\begin{array}{ll}
\mathbf{DB}, \theta, 0 \models R_i(x_{i,1}, \ldots, x_{i,k_i}) & \text{if } (\theta(x_{i,1}), \ldots, \theta(x_{i,k_i}), m) \notin \mathbf{R}, \text{ for any } m > 0 \\
\mathbf{DB}, \theta, m \models R_i(x_{i,1}, \ldots, x_{i,k_i}) & \text{if } (\theta(x_{i,1}), \ldots, \theta(x_{i,k_i}), m) \in \mathbf{R} \\
\mathbf{DB}, \theta, m \models \varphi \wedge (x_i = x_j) & \text{if } \mathbf{DB}, \theta, m \models \varphi \text{ and } \theta(x_i) \approx \theta(x_j) \\
\mathbf{DB}, \theta, 1 \models (x_i = c_j) & \text{if } \theta(x_i) \approx c_j \\
\mathbf{DB}, \theta, m_1 \cdot m_2 \models \varphi \wedge \psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \text{ and } \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, \sum_{v \in D} m_v \models \exists x.\varphi & \text{if } \mathbf{DB}, \theta[x := v], m_v \models \varphi \\
\mathbf{DB}, \theta, m_1 + m_2 \models \varphi \vee \psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \text{ and } \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, \max(0, m_1 - m_2) \models \varphi \wedge \neg\psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \text{ and } \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, 1 \models \text{elim } \varphi & \text{if } \mathbf{DB}, \theta, m \models \varphi, \text{ where } m > 0
\end{array}
$$

The *answers* to a *query* $\{(x_1, \ldots, x_n) \mid \varphi\}$ over $\mathbf{DB}$ is given as follows, when $\{x_1, \ldots, x_n\} = \text{Fv}(\varphi)$:

$$\{(\theta(x_1), \ldots, \theta(x_n), m) \mid m > 0 \wedge \mathbf{DB}, \theta, m \models \varphi\}.$$

- syntax for SELECT block becomes:

```
SELECT [DISTINCT]  <results>
FROM               <tables>
WHERE              <condition>
```

↳ duplicates are eliminated if DISTINCT is included

↳ allows formulations of RC queries of form $\{\langle\text{results}\rangle \mid [\text{elim}]\ \exists\ \langle\text{unused}\rangle.\ (\bigwedge \langle\text{tables}\rangle) \wedge \langle\text{condition}\rangle\}$

- ans to SELECT blocks are multisets of tuples so can apply multiset ops on them

↳ multiset union: Q_1 UNION ALL Q_2

- $Q_1 \vee Q_2$

↳ multiset diff: Q_1 EXCEPT ALL Q_2

- $Q_1 \wedge \neg Q_2$

↳ multiset intersection: Q_1 INTERSECT ALL Q_2

- max # tuples common to Q_1 & Q_2
- rarely used

- e.g. removing DISTINCT

For every ordered pair of authorships on the same publication, list the id of the publication.

```
select r1.publication \
from wrote r1, wrote r2 \
where r1.publication = r2.publication \
and r1.author != r2.author

PUBLICATION
-----------
          2
          2

  2 record(s) selected.
```

- e.g. UNION ALL

For every book and article, list all authors.

```
( select author \
  from wrote, book \
  where publication = pubid ) \
union all \
( select author \
  from wrote, article \
  where publication = pubid )

AUTHOR
-----------
          1
          1

  2 record(s) selected.
```

- subqueries w/ duplicates occurring in WHERE clauses won't change results computed by top-lvl query
  - ↳ if they occur in WITH / FROM clauses, they can change results

NULL VALUES

- null val can be 2 things:
  - ↳ inapplicable
  - ↳ unknown
- e.g.

Phone

Name	Office	Home
Joe	1234	3456
Sue	1235	?

  - ↳ inapplicable means Sue doesn't have home phone
  - ↳ unknown means Sue has home phone, but we don't know it
- if val is inapplicable, it's poor schema design
  - ↳ e.g. better design

OfficePhone

Name	Office
Joe	1234
Sue	1235

HomePhone

Name	Home
Joe	3456

- unknown vals can be replaced by any domain val that satisfies integrity constraints
  - ↳ many possible worlds
  - ↳ e.g.

Phone

Name	Office	Home
Joe	1234	3456
Sue	1235	?

→

Phone

Name	Office	Home
Joe	1234	3456
Sue	1235	0000

⋮

Phone

Name	Office	Home
Joe	1234	3456
Sue	1235	9999

- dealing w/ NULLs in SQL:
  - ↳ exprs: general rule is if NULL is param to op, result is NULL
    - e.g. 1 + NULL → NULL
    - e.g. 'foo' || NULL → NULL
  - ↳ predicates/comparisons: 3-valued logic
    - apx of val unknown
  - ↳ set ops: unique special val for duplicates
  - ↳ aggregate ops: do not count
- Boolean ops need to be defined for UNKNOWN so we have extended truth tables:

∧	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

∨	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

¬	
T	F
U	U
F	T

- additional syntax for WHERE clause: IS TRUE, IS FALSE, ? IS UNKNOWN
  - ↳ WHERE <cond> is shorthand for WHERE <cond> IS TRUE
  - ↳ add special comparison IS NULL
- DB2 uses "-" to denote NULL vals

- e.g.

Add a new author URI attribute and tuple.

```
alter table author add column URI varchar (20)

DB20000I  The SQL command completed successfully.

insert into AUTHOR values (3, 'Mary', 'uwaterloo.ca')

DB20000I  The SQL command completed successfully.
```

List all author information.

```
select * from author

AID         NAME       URI
----------- ---------- --------------------
          1 Sue        -
          2 John       -
          3 Mary       uwaterloo.ca

  3 record(s) selected.
```

List all author ids and names for which we don't know a URL of their home page.

```
select aid, name from author \
where uri IS NULL

AID         NAME
----------- ----------
          1 Sue
          2 John

  2 record(s) selected.
```

- e.g. counting NULLs

```
select count(*) as Rcnt, count(url) as Vcnt \
from author

RCNT        VCNT
----------- -----------
          3           1

  1 record(s) selected.
```

↳ COUNT(URL) counts only non-NULL URLs

↳ COUNT(*) counts rows

- outer join is used to combine records from 2+ tables by including all rows from 1 table + any matching rows from other table (if there's no match, include NULL vals)

↳ 3 types:

- left outer join: all rows from left table + any matching rows from right table
- right outer join: all rows from right table + any matching rows from left table
- full outer join: all rows from both left + right tables

↳ extension of FROM clause: FROM T1 <j-type> JOIN T2 ON <cond>

- <j-type> is one of FULL, LEFT, RIGHT, or INNER

↳ e.g.

```
select aid, publication \
from author left join wrote on aid=author

AID         PUBLICATION
----------- -----------
          1           1
          1           2
          1           4
          2           2
          3           -

  5 record(s) selected.
```

↳ e.g. counting w/ outer join

Author ids and a count of the number of publications for each.

```
select aid, count(publication) as pcnt \
from author left join wrote on aid = author \
group by aid

AID          PCNT
-----------  -----------
          1            3
          2            1
          3            0

  3 record(s) selected.
```

ORDERING AND LIMITS

- no particular ordering on rows of table can be assumed when queries are written & on intermediate result in query
- can order final result of query w/ `ORDER BY` clause
  - ↳ syntax is ORDER BY e_1 [Dir_1], ..., e_k [Dir_k]
    - Dir_i is either ASC or DESC
      - → ASC is default
  - ↳ e.g.

List all authors in the database in ascending order of their name.

```
select distinct * from author \
order by name

AID          NAME       URI
-----------  ---------- --------------------
          2 John       -
          3 Mary       uwaterloo.ca
          1 Sue        -

  3 record(s) selected.
```

    - note ASC keyword is assumed
    - minor, minor minor, etc. sorts can be added
- # results in query can be limited by appending `LIMIT` clause
  - ↳ syntax is <query> LIMIT e_1 [offset e_2]
    - e_i is numeric expr
    - semantics are `non-deterministic` (i.e. multiple possible interpretations/results)
      - → as many of 1st e_1 ans to query that exist for any total order to which results of <query> can be extended
      - → if `OFFSET` is given, will be 1st e_1 ans following 1st e_2 ans for any total order
    - to make semantics deterministic, <query> has ORDER BY clause to induce `total order`
  - ↳ e.g.

List at most the first two entries of a sorted list of authors by name.

```
select distinct * from author \
order by name \
limit 2

AID          NAME       URI
-----------  ---------- --------------------
          2 John       -
          3 Mary       uwaterloo.ca

  2 record(s) selected.
```

  - ↳ e.g.

List at most the next two entries following the second entry of a sorted list of authors by name.

```
select distinct * from author \
order by name \
limit 2 offset 2

AID         NAME       URI
----------- ---------- --------------------
          1 Sue        -

  1 record(s) selected.
```

TRIGGERS AND AUTHORIZATION

- syntax for **triggers** for individual tuple events (simplified):

```
CREATE TRIGGER <trigger-name>
AFTER <event> ON <table-or-view>
     [ REFERENCING OLD AS <corelation-old> ]
     [ REFERENCING NEW AS <corelation-new> ]
     FOR EACH ROW [ WHEN <condition> ] BEGIN ATOMIC
          <DML-action-1>;
          ...
          <DML-action-n>;
END
```

  - ↳ <event> can be one of INSERT, DELETE, or UPDATE OF <attr>
- e.g.

When deleting an author, also delete related wrote tuples.

```
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
```

- **semantics** of triggers in SQL:
  - ↳ implements **event/condition/action (ECA)** rules
  - ↳ adds additional ops to ongoing transaction issuing op that triggered execution of rule
    - ↳ timing of integrity constraint checking must be carefully managed
  - ↳ makes SQL DML Turing complete
- special syntax for common rules related to **foreign key constraints**:

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

  - ↳ <action> can be:
    - RESTRICT: prod error
    - CASCADE: propagate delete
    - SET NULL: set to "unknown"
  - ↳ e.g.

```
create table WROTE (
     author integer not null,
     publication integer not null,
     primary key (author, publication),
     foreign key (author) references AUTHOR
          on delete cascade,
     foreign key (publication) references PUBLICATION )
```

- SQL DML includes **date control lang (DCL)** to manage access rights to db objs by users & user groups

↳ syntax (simplified):

```
GRANT  <role> TO <user>
GRANT  <what> ON <object> TO <user-or-role>
REVOKE <role> FROM <user>
REVOKE <what> ON <object> FROM <user-or-role>
```

- <what> ON <object> can be:
  - → db: CONNECT
  - → table / view: ALTER, REFERENCES, SELECT, INSERT, DELETE, or UPDATE
- CREATE ROLE <role> cmd creates new roles
  - → role DBA DM is superuser

↳ e.g.

Create the `PAT` *role, and grant ability to access the* `PAYROLL` *database to* `PAT`.

```
create role PAT
grant connect on PAYROLL to PAT
```

Grant ability to query table `EMPLOYEE` *to the payroll project team.*

```
grant select on EMPLOYEE to PAT
```

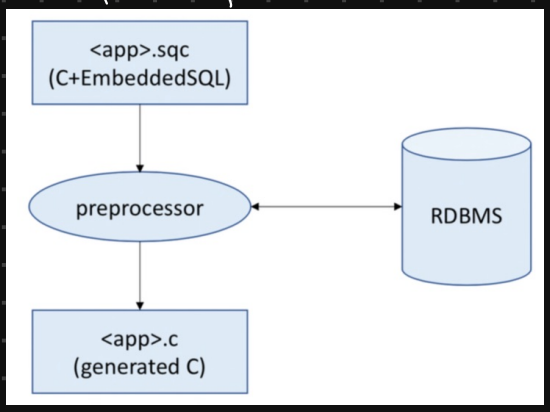
Add Jim to the payroll project team.

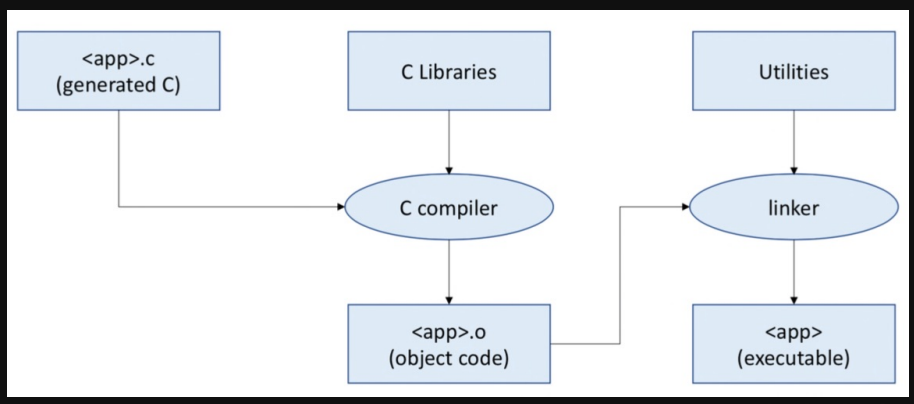
```
grant PAT to Jim
```

APPLICATIONS PROGRAMMING AND SQL

EMBEDDED SQL IN C

- SQL isn't sufficient to write general apps so we extend general-purpose PL w/ SQL
  - ↳ can do this via:
    - library calls (CLI / ODBC)
    - embedded SQL
    - usually OO PLs w/ **persistent** data types (i.e. retains val even after program termination / session end)
- at least **2-tier client-server** architecture will apply
  - ↳ SQL runs on server
  - ↳ app runs on client
- SQL can be **embedded** in host lang like C, C++, Fortran, etc.
  - ↳ app is **preprocessed** to produce pure host lang program w/ library calls
  - ↳ **advantages**:
    - preprocessing of static parts of queries is possible
    - easier to use
    - more secure
  - ↳ **disadvantages**:
    - requires pre-compiler
    - generated host lang program might be bound to particular RDBMS engine
      - → DB2 has this property
- **development process** for embedded SQL apps:





  - ↳ not all RDBMSs communicate w/ server to compile at this time
    - some products generate CLI interface (e.g. ODBC)
  - ↳ for DB2, <app> will be bound to particular server engine (i.e app is designed to work specifically w/ particular RDBMS)
- sample **Makefile** source:

```
# put the name of your application here:
#NAME= adhoc

################################################
# don't touch anything below this line
################################################
DB2PATH = /home/db2inst2/sqllib
# The following compile and link options are for the gcc
CC=gcc

CFLAGS=-I$(DB2PATH)/include
#LIBS=-L$(DB2PATH)/lib -R$(DB2PATH)/lib -ldb2
LIBS=-L$(DB2PATH)/lib -ldb2
```

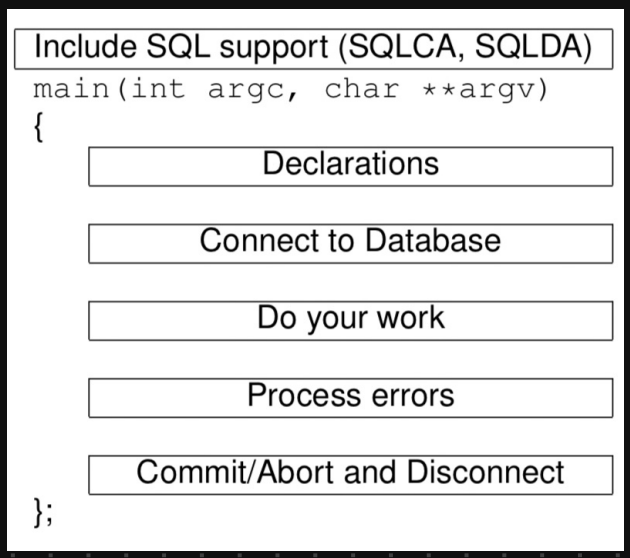
```
all: $(NAME)

$(NAME): $(NAME).sqc util.o
        db2 connect to cs348
        db2 prep $(NAME).sqc bindfile
        db2 bind $(NAME).bnd
        db2 connect reset
        $(CC) $(CFLAGS) -c $(NAME).c
        $(CC) $(CFLAGS) -o $(NAME) $(NAME).o util.o $(LIBS)

clean:
        rm -f $(NAME) $(NAME).c $(NAME).o $(NAME).bnd

util.o : util.c
        $(CC) -w -c util.c $(CFLAGS)
```

- DBMS needs to know certain info abt app.
  - ↳ PREP: precompiling (i.e. prepping SQL stmts for execution)
    - parse, validate, & optimize SQL stmts
  - ↳ BIND: binding vars / params to prepped SQL stmts
- app structure:



  - ↳ in "Include SQL support" area: EXEC SQL INCLUDE SQLCA;
    - this defines return code of SQL stmts (sqlcode) & error msgs
    - required
  - ↳ SQL stmt can be inserted by embedding following into C code: EXEC SQL <sql stmt>;
- host vars communicate single scalar vals btwn a SQL stmt & embedding lang
  - ↳ act as placeholders for dynamic vals
  - ↳ can be local/global vars
  - ↳ must be declared within SQL declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
(only declarations of host variables in host language go here)
EXEC SQL END DECLARE SECTION;
```

  - ↳ using vars that aren't host vars in EXEC SQL stmts is not supported
  - ↳ prefixed by colon (:)
- if SQL stmt fails:
  - ↳ check sqlcode != 0
  - ↳ use exception handling SQL cmds:

```
EXEC SQL WHENEVER SQLERROR GO TO <label>;
EXEC SQL WHENEVER SQLWARNING GO TO <label>;
EXEC SQL WHENEVER NOT FOUND GO TO <label>;
```

    - <label> must be in scope

- e.g. connecting to db (sample1.sqc)

```
#include <stdio.h>
#include <stdlib.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {
   char * getpass();

   EXEC SQL BEGIN DECLARE SECTION;
      char db[6] = "cs348";
   EXEC SQL END DECLARE SECTION;

   printf("Sample C program: CONNECT\n" );

   EXEC SQL WHENEVER SQLERROR  GO TO error;

   EXEC SQL CONNECT TO :db;

   printf("Connected to DB2\n");

   EXEC SQL COMMIT;
   EXEC SQL CONNECT reset;
   exit(0);

error:
   check_error("My error",&sqlca);
   EXEC SQL WHENEVER SQLERROR CONTINUE;

   EXEC SQL ROLLBACK;
   EXEC SQL CONNECT reset;
   exit(1);
}
```

- to **prep application** in DB2:
  1) write app in file <name>.sqc
  2) preprocess app: db2 prep <name>.sqc
  3) compile app: cc -c -O <name>.c
  4) link w/ DB2 libs: cc -o <name> <name>.o -L. . -l...
  5) run it: ./<name> [args]
  ↳ can use **Makefile** to do this process
    - e.g.

```
make all NAME=sample1

db2 connect to cs348

   Database Connection Information

 Database server        = DB2/LINUXX8664 11.1.4.4
 SQL authorization ID   = GWEDDELL
 Local database alias   = CS348

db2 prep sample1.sqc bindfile

LINE    MESSAGES FOR sample1.sqc
------  --------------------------------------------------------------------
        SQL0060W  The "C" precompiler is in progress.
        SQL0091W  Precompilation or binding was ended with "0"
                  errors and "0" warnings.
db2 bind sample1.bnd

LINE    MESSAGES FOR sample1.bnd
------  --------------------------------------------------------------------
        SQL0061W  The binder is in progress.
        SQL0091N  Binding was ended with "0" errors and "0" warnings.
db2 connect reset
DB20000I  The SQL command completed successfully.
gcc -I/home/db2inst2/sqllib/include  -c sample1.c
gcc -I/home/db2inst2/sqllib/include  -o sample1 sample1.o util.o -L/home/db2inst2/sq
```

→ no errors would give output:

```
./sample1

Sample C program: CONNECT
Connected to DB2
```

→ if we had wrong db name (e.g. char db[6] = "CS248",)

```
./sample1

Sample C program: CONNECT
--- error report ---
ERROR occurred : My error.
SQLCODE : 4294966283
SQL1013N  The database alias name or database name "CS248" could not be found.
SQLSTATE=42705
--- end error report ---
```

- can also add executable DML stmts
  - ↳ simple stmts:
    - constant stmts
    - stmts w/ params
    - queries returning single tuple
  - ↳ general queries
  - ↳ dynamic queries
- e.g. program that prints out title of publication for each publication id supplied as arg (sample2.sqc)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;
 // executing simple stmts
int main(int argc, char *argv[]) {
   int i;

   EXEC SQL BEGIN DECLARE SECTION;
      char db[6] = "cs348";
      char title[72];
      int pubid;
   EXEC SQL END DECLARE SECTION;

   printf("Sample C program: SAMPLE2\n" );

   EXEC SQL CONNECT TO :db;

   printf("Connected to DB2\n");

   EXEC SQL WHENEVER SQLERROR  GO TO error;

   for (i=1; i<argc; i++) {
     pubid = atoi(argv[i]); //atoi is ASCII to int

     EXEC SQL WHENEVER NOT FOUND GO TO nope;
     EXEC SQL SELECT title INTO :title
              FROM   publication
              WHERE  pubid = :pubid;

     printf("%4d: %s\n",pubid,title);
     continue;
   nope:
     printf("%4d: *** not found *** \n",pubid);
   };

   EXEC SQL COMMIT;
   EXEC SQL CONNECT reset;
   exit(0);

error:
   check_error("My error",&sqlca);
   EXEC SQL WHENEVER SQLERROR CONTINUE;

   EXEC SQL ROLLBACK;
   EXEC SQL CONNECT reset;
   exit(1);
}
```

↳ e.g. running resulting program

```
./sample2 1 7 3 5

Sample C program: SAMPLE2
Connected to DB2
1: Mathematical Logic
7: *** not found ***
3: Principles of DB Systems
5: *** not found ***
```

- at most 1 title is returned for each publication id

- if result val is NULL, it's not valid val in datatype
  ↳ embedded SQL uses indicator var that's associated w/ host var that'll retrieve val
  - e.g.

```
smallint ind; // (within a BEGIN/END DECLARE SECTION)
...
EXEC SQL SELECT firstname INTO :firstname
                          INDICATOR :ind
         FROM ... ;
```

→ if ind < 0, then firstname is NULL
↳ if indicator var isn't provided & result is NULL, we get run-time error
↳ same rules apply when using host vars in update stmts

- for SQL queries that can return 1+ rows/ans, SQL cursor is defined & used as an iterator thru each row
  ↳ declaring cursor:

```
EXEC SQL DECLARE <name> CURSOR FOR
         <query>;
```

↳ iterator protocol using cursor:

```
EXEC SQL OPEN <name>;
EXEC SQL WHENEVER NOT FOUND GO TO end;
for (;;) {
   <set up host parameters>
   EXEC SQL FETCH <name> INTO <host variables>;
   <process the fetched tuple>
 };
end:
EXEC SQL CLOSE <name>;
```

- e.g. program that lists all author names & publication titles w/ author name matching pattern given as an arg (sample3.sqc)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

// executing queries w/ many ans
EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
   char db[6] = "cs348";
   char title[72], name[20], apat[10];
   short aid;
EXEC SQL END DECLARE SECTION;
```

```
int main(int argc, char *argv[]) {
   if (argc!=2) {
      printf("Usage: sample3 <pattern>\n");
      exit(1);
   };

   printf("Sample C program: SAMPLE3\n" );

   EXEC SQL WHENEVER SQLERROR  GO TO error;

   EXEC SQL CONNECT TO :db;

   printf("Connected to DB2\n");

   strncpy(apat,argv[1],8);

   EXEC SQL DECLARE author CURSOR FOR
             SELECT name, title
             FROM author , wrote, publication
             WHERE name LIKE :apat
               AND aid=author
               AND pubid=publication;

   EXEC SQL OPEN author;
   EXEC SQL WHENEVER NOT FOUND GO TO end;
   for (;;) {
     EXEC SQL FETCH author INTO :name, :title;
     printf("%10s -> %20s: %s\n",apat,name,title);
     };
end:
   EXEC SQL CLOSE author;

   EXEC SQL COMMIT;
   EXEC SQL CONNECT RESET;
   exit(0);

error:
   check_error("My error",&sqlca);
   EXEC SQL WHENEVER SQLERROR CONTINUE;

   EXEC SQL ROLLBACK;
   EXEC SQL CONNECT reset;
   exit(1);
}
```

← %10s is format specifier used to print str s w/ min width of 10 chars (if s is shorter, it's padded w/ spaces)

↳ e.g running resulting program

```
./sample3 "%"

Sample C program: SAMPLE3
Connected to DB2
         % ->         Sue: Mathematical Logic
         % ->         Sue: Trans. on Databases
         % ->         Sue: Query Languages
         % ->        John: Trans. on Databases

./sample3 "%u_"

Sample C program: SAMPLE3
Connected to DB2
        %u_ ->         Sue: Mathematical Logic
        %u_ ->         Sue: Trans. on Databases
        %u_ ->         Sue: Query Languages
```

- % is wildcard symbol that rep 0+ of any chars
- _ is wildcard symbol that rep 1 char
- %u_ matches w/ up, consul, & about, but not clutch & putt

STORED PROCEDURES

stored proc executes app logic directly inside RDBMS server

↳ possible implementations:

- ability to invoke **externally-compiled app**, meaning stored proc is wrapper/interface that executes external program
  → stored proc passes input params to external program, receives output/return vals, & perform necessary processing before/after invoking program
- support for **SQL / PSM**, which is a std SQL lang extension that provides

procedural programming capabilities within db

- reasons for stored procs:
  - ↳ reduces volume of data transfer btwn client & server
  - ↳ centralizes app code at server
  - ↳ conceptual schema enhancement
- e.g. atomic-valued fcn

```
CREATE FUNCTION sumSalaries(dept CHAR(3))
    RETURNS DECIMAL(9,2)
LANGUAGE SQL
RETURN
    SELECT sum(salary)
    FROM employee
    WHERE workdept = dept
```

```
db2 => SELECT deptno, sumSalaries(deptno) AS sal \
    => FROM department

DEPTNO SAL
------ -----------
A00      128500.00
B01       41250.00
C01       90470.00
D01              -
D11      222100.00
D21      150920.00
E01       40175.00
E11      104990.00
E21       95310.00

  9 record(s) selected.
```

- e.g. table-valued fcn

```
CREATE FUNCTION deptSalariesF(dept CHAR(3))
    RETURNS TABLE(salary DECIMAL(9,2))
    LANGUAGE SQL
RETURN
    SELECT salary
    FROM employee
    WHERE workdept = dept
```

```
db2 => SELECT * FROM TABLE \
    => (deptSalariesF(CAST('A00' AS CHAR(3)))) AS s

SALARY
-----------
   52750.00
   46500.00
   29250.00

  3 record(s) selected.
```

- e.g. branching

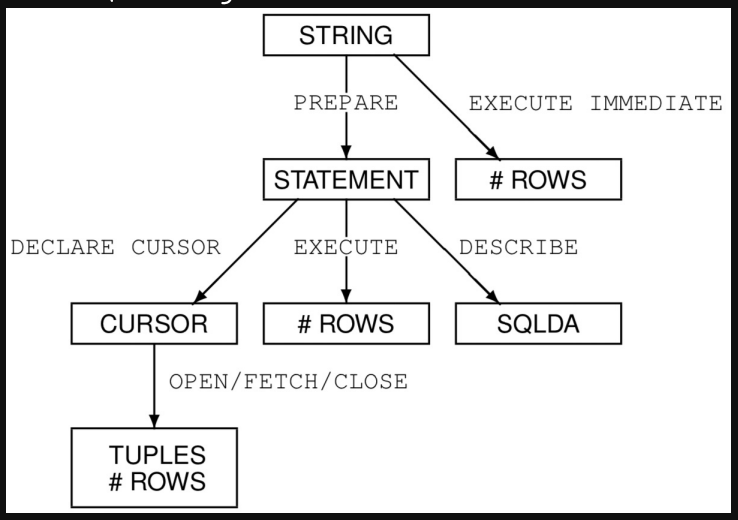
```
CREATE PROCEDURE UPDATE_SALARY_IF
      (IN employee_number CHAR(6), INOUT rating SMALLINT)
   LANGUAGE SQL
```

```
BEGIN
   DECLARE not_found CONDITION FOR SQLSTATE '02000';
   DECLARE EXIT HANDLER FOR not_found
      SET rating = -1;
   IF rating = 1 THEN
      UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
   ELSEIF rating = 2 THEN
      UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
   ELSE
      UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
   END IF;
END
```

APPLICATIONS PROGRAMMING WITH DYNAMIC SQL

DYNAMIC EMBEDDED SQL IN C

- protocol for using SQL within another lang is **dynamic** if it enables arbitrary SQL source code to be constructed & executed at run-time
- problems:
  - ↳ string is valid stmt?
    - parsing & compilation at run-time
  - ↳ how to execute valid stmts?
    - supply params & retrieve results
  - ↳ extreme case is that nothing is known abt str at run-time
- we'll develop **adhoc app** that accepts SQL stmt as argument, executes it, & prints out ans if stmt is valid query
- **roadmap** for dynamic embedded SQL:



- to execute **non-parametric** stmt, use cmd **EXEC SQL EXECUTE IMMEDIATE :string;**
  - ↳ :string is host var containing ASCII rep of cmd
    - can't be cmd that returns ans (e.g. query)
  - ↳ should only be used for cmds that'll be executed only once by app
  - ↳ :string is compiled each time control flow reaches cmd
- to **compile** SQL cmds & **provide handle** (i.e. name of fcn) to generated code at run-time, use cmd **EXEC SQL PREPARE stmt FROM :string;**
  - ↳ code generated for :string can be invoked via stmt handle in other SQL cmds
    - avoids recompilation of :string
  - ↳ :string can contain params & be query that returns ans
  - ↳ stmt handle is not host var & **not reentrant** (i.e. can't safely be called by multiple threads/processes simultaneously)
    - don't use stmt handles in recursion
- 2 ways that app can communicate param values to stmt handles:
  - ↳ reconstruct ASCII str encoding cmd
    - needs recompilation
  - ↳ allow **param markers** in an ASCII str encoding a cmd
    - **? chars** in locations of param vals that'll be sub when compiled cmds are executed
    - e.g.
      :string = "SELECT * FROM employees WHERE dept = ? AND salary > ?"
- to execute prepped SQL cmd w/param markers via a stmt handle, use cmd

```
EXEC SQL EXECUTE stmt
         USING   :var1 [,...,:vark];
```

↳ used for cmds that don't return tuples
- e.g. db modification (INSERT, ...), transaction management (COMMIT), data defn (CREATE TABLE, ...)

↳ vals of host vars :var1 to :vark are sub for param markers in order of appearance
- mismatch causes SQL run-time error
- e.g.

```
/* Update column in table using DYNAMIC SQL*/
strcpy(hostVarStmtDyn, "UPDATE staff SET salary = salary + 1000  WHERE dept = ?");
EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
EXEC SQL EXECUTE StmtDyn USING :dept;
```

to use SQL cursors, use USING clause w/ OPEN cmd & follow cursor iterator protocol:

```
EXEC SQL DECLARE cname CURSOR FOR stmt;

EXEC SQL OPEN  cname
         USING :var1 [,...,:vark];

EXEC SQL FETCH cname
         INTO  :out1 [,...,:outn];

EXEC SQL CLOSE cname;
```

↳ :var1 to :vark supply vals for param markers
↳ :out1 to :outn store vals for retrieved tuple
↳ #retrieved tuples is defined in sqlca.sqlerrd[2]

SQL descriptor is used to communicate info abt params & results of prepped SQL cmd
↳ cmds for descriptor use:

- ALLOCATE DESCRIPTOR descr
- GET DESCRIPTOR descr <what>
  SET DESCRIPTOR descr <what>

  where <what> indicates
  1. GET/SET a value for COUNT, or
  2. GET/SET properties for the *i*-th attribute: VALUE :i <prop>
     where <prop> can be DATA, TYPE, INDICATOR, ...
- DESCRIBE [INPUT | OUTPUT] stmt INTO descr

↳ in practice, we use explicit sqlda data structure in host lang
- e.g. struct sqlda *slct;
  → name of descriptor is *slct

sqlda data struct is SQL description area that defines what attrs are params & query ans look like (i.e. where data is located)
↳ how RDBMS communicates w/ app
↳ main things data struct contains:
- str 'SQLDA' for id
- #allocated entries for attrs
- #acc attrs
  → 0 if none
- for each attr:
  → numeric code of its type
  → length of storage for its val
  → pointer to a data var
  → pointer to an indicator var
  → name (str & length)

↳ sqlda in DB2:

```
struct  sqlname         /* AttributeName                */
{
  short      length;    /* Name length [1..30]          */
  char       data[30];  /* Variable or Column name      */
};

struct  sqlvar          /* Attribute Descriptor         */
{
  short          sqltype;   /* Variable data type       */
  short          sqllen;    /* Variable data length     */
  char          *SQL_POINTER sqldata; /* data buffer     */
  short         *SQL_POINTER sqlind;  /* null indiciator */
  struct sqlname sqlname;   /* Variable name            */
};

struct  sqlda        /* Main SQLDA                      */
{
  char   sqldaid[8]; /* Eye catcher = 'SQLDA   '        */
  long   sqldabc;    /* SQLDA size in bytes=16+44*SQLN  */
  short  sqln;       /* Number of SQLVAR elements       */
  short  sqld;       /* Number of used SQLVAR elements  */
  struct sqlvar  sqlvar[1];  /* first SQLVAR element    */
};
```

↳ sqlda in Oracle6:

```
struct SQLDA {
   long     N; /* Descriptor size in number of entries      */
   char  *V[]; /* Arr of addresses of main variables (data) */
   long   L[]; /* Arr of lengths of data buffers            */
   short  T[]; /* Arr of types of buffers                   */
   short *I[]; /* Arr of addresses of indicator vars        */
   long     F; /* Number of variables found by DESCRIBE     */
   char  *S[]; /* Arr of variable name pointers             */
   short  M[]; /* Arr of max lengths of attribute names     */
   short  C[]; /* Arr of current lengths of attribute names */
   char  *X[]; /* Arr of indicator name pointers            */
   short  Y[]; /* Arr of max lengths of ind. names          */
   short  Z[]; /* Arr of cur lengths of ind. names          */
};
```

- prepped stmt can be described via a sqlda data struct w/cmd `EXEC SQL DESCRIBE stmt INTO :sqlda`
  - ↳ result is:
    - # result attrs
      - → 0 when not a query
    - for every attr in ans tuple when stmt is query:
      - → name? length
      - → type

can use SQLDA descriptor to supply params (i.e. descriptors can sub host vars as arg of SQL USING clause):

```
EXEC SQL EXECUTE stmt
         USING DESCRIPTOR :sqlda;

EXEC SQL OPEN cname
         USING DESCRIPTOR :sqlda;

EXEC SQL FETCH cname
         USING DESCRIPTOR :sqlda;
```

- e.g. adhoc.sqc is app that executes a SQL stmt provided as its arg on cmd line

// declarations

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

EXEC SQL BEGIN DECLARE SECTION;
   char  db[6] = "cs348";
   char  sqlstmt[1000];
EXEC SQL END DECLARE SECTION;

struct sqlda *slct;
```

// start up & prep stmt

```
int main(int argc, char *argv[]) {
   int i, isnull; short type;

   printf("Sample C program : ADHOC interactive SQL\n");

   /* bail out on error */
   EXEC SQL WHENEVER SQLERROR  GO TO error;

   /* connect to the database */
   EXEC SQL CONNECT TO :db;
   printf("Connected to DB2\n");

   strncpy(sqlstmt,argv[1],1000);
   printf("Processing <%s>\n",sqlstmt);

   /* compile the sql statement */
   EXEC SQL PREPARE stmt FROM :sqlstmt;

   init_da(&slct,1);

   /* now we find out what it is */
   EXEC SQL DESCRIBE stmt INTO :*slct;
```

// stmt is a query

```
   i= slct->sqld;
   if (i>0) {
     printf("      ... looks like a query\n");

     /* new SQLDA to hold enough descriptors for answer */
     init_da(&slct,i);

     /* get the names, types, etc... */
     EXEC SQL DESCRIBE stmt INTO :*slct;

     printf("Number of slct variables <%d>\n",slct->sqld);
     for (i=0; i<slct->sqld; i++ ) {
       printf("  variable %d <%.*s (%d%s [%d])>\n",
                  i,
                  slct->sqlvar[i].sqlname.length,
                  slct->sqlvar[i].sqlname.data,
                  slct->sqlvar[i].sqltype,
                  ( (slct->sqlvar[i].sqltype&1)==1 ? "": " not null"),
                  slct->sqlvar[i].sqllen);
     }
     printf("\n");
```

```
    /* allocate buffers for the returned tuples */
    for (i=0; i<slct->sqld; i++ ) {
      slct->sqlvar[i].sqldata = malloc(slct->sqlvar[i].sqllen);
      slct->sqlvar[i].sqlind = malloc(sizeof(short));
      *slct->sqlvar[i].sqlind = 0;
    }

    /* and now process the query */
    EXEC SQL DECLARE cstmt CURSOR FOR stmt;
    EXEC SQL OPEN cstmt;
    EXEC SQL WHENEVER NOT FOUND GO TO end;

    /* print the header */
    for (i=0; i<slct->sqld; i++ )
      printf("%-*.*s ",slct->sqlvar[i].sqllen,
                       slct->sqlvar[i].sqlname.length,
                       slct->sqlvar[i].sqlname.data);
    printf("\n");
    //fetch & print ans
    for (;;) {
      /* fetch next tuple into the prepared buffers */
      EXEC SQL FETCH cstmt USING DESCRIPTOR :*slct;
      for (i=0; i<slct->sqld; i++ )
        if ( *(slct->sqlvar[i].sqlind) < 0 )
          print_var("NULL",
                    slct->sqlvar[i].sqltype,
                    slct->sqlvar[i].sqlname.length,
                    slct->sqlvar[i].sqllen);
        else
          print_var(slct->sqlvar[i].sqldata,
                    slct->sqlvar[i].sqltype,
                    slct->sqlvar[i].sqlname.length,
                    slct->sqlvar[i].sqllen);
      printf("\n");
      };
  end:
    printf("\n");
  } else {
    printf("      ... looks like an update\n");
    //otherwise, it's a simple stmt so just execute it
    EXEC SQL EXECUTE stmt;
  };
  //printf("Rows processed: %d\n",sqlca.sqlerrd[2]);

  /* and get out of here */
  EXEC SQL COMMIT;
  EXEC SQL CONNECT reset;
  exit(0);

error:
  check_error("My error",&sqlca);
  EXEC SQL WHENEVER SQLERROR CONTINUE;

  EXEC SQL ROLLBACK;
  EXEC SQL CONNECT reset;
  exit(1);
}
```

↳ e.g. running adhoc.sqc

```
./adhoc "select * from author"

Sample C program : ADHOC interactive SQL
Connected to DB2
Processing <select * from author>
      ... looks like a query
Number of select variables <3>
  variable 0 <AID (496 not null [4])>
  variable 1 <NAME (453 [22])>
  variable 2 <URL (453 [42])>
```

```
AID NAME           URL
  1 Toman, David   http://db.uwaterloo.ca/~david
  2 Chomicki, Jan  http://cs.buffalo.edu/~chomick
  3 Saake, Gunter  NULL
```

- given a **str**:
  - ↳ if nothing is known, use DESCRIBE
  - ↳ if it's simple stmt used once, use EXECUTE IMMEDIATE
  - ↳ otherwise, use PREPARE
- given **stmt handle** obtained via PREPARE:
  - ↳ if simple stmt used once, use EXECUTE
  - ↳ for query otherwise, use DECLARE CURSOR & process as ordinary cursor

ODBC

- **ODBC** stands for open database connectivity
  - ↳ provides a way for apps to communicate w/ dbs, regardless of specific DBMS being used
- SQL **CLI (call lvl interface)** is an interface defined by a C lib of fcn calls (defined by Microsoft)
  - ↳ allows interaction w/ db using cmd line env
  - ↳ for CLI, apps are developed entirely in host lang
    - no pre-compilation so no type checking required at compile time
    - less transparent than static embedded SQL
- ODBC is Microsoft spec for an API that constitutes implementation of stds for SQL CLI
  - ↳ i.e. SQL CLI std incorporates ODBC (& X/Open) stds
- 3 kinds of **handles** in ODBC program:
  - ↳ **environment**
    - exactly 1 per app thread
  - ↳ db engine **connection**
  - ↳ SQL **stmt**
- note dynamic SQL has stmt handles only
  - ↳ executing thread has at most 1 ongoing transaction
- e.g. connect & disconnect in ODBC

```
int main()
{
   SQLHENV   henv;
   SQLHDBC   hdbc;
   SQLRETURN rc;
   SQLCHAR   server[SQL_MAX_DSN_LENGTH + 1] = "DBCLASS";
   SQLCHAR   uid[19] = "<your uid>";
   SQLCHAR   pwd[31] = "<your password>";

   SQLAllocEnv(&henv);
   SQLAllocConnect(henv, &hdbc);

   rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
   if (rc != SQL_SUCCESS) {
          printf("Error connecting to %s\n", server); exit(1);
   } else printf("Connected to %s\n", server);

/* DO SOMETHING HERE */

   SQLDisconnect(hdbc);
   SQLFreeConnect(hdbc);
   SQLFreeEnv(henv);
}
```

- each SQLxxx fcn returns an **error code**
  - ↳ should be checked after each call
  - ↳ return codes:
    - SQL_SUCCESS
    - SQL_ERROR

↳ use SQLError fcn to obtain more details

- SQL stmts in ODBC:
  - SQLAllocStmt (allocates a statement handle)
  - SQLExecDirect (execute)
  - SQLPrepare (compile statement)
  - SQLExecute (execute compiled statement)
  - SQLSetParam (initialize a procedure parameter)
  - SQLNumResultCols (number of result columns)
  - SQLBindCol ("host variables" in ODBC)
  - SQLGetData (obtaining values of result columns)
  - SQLFetch (cursor access in ODBC)
  - SQLError (obtains diagnostics)
  - SQLRowCount (number of affected rows)
  - ...
  - SQLFreeStmt (frees a statement handle)

- param markers are specified by '?' in txt of query
  ↳ SQLNumParams retrieves # param markers in prepped SQL stmt
  ↳ SQLBindParameter binds acc val to param marker
- results of queries are specified by # resulting cols
  ↳ SQLNumResultCols
  ↳ SQLDescribeCol
  ↳ SQLBindCol
    - binds buffer to specific col in result set
    - when we call SQLFetch / SQLFetchScroll to retrieve row, data for bound cols is automatically fetched & stored in buffer
  ↳ SQLGetData
    - retrieves data from specific col in result set w/o binding a buffer
      → specify col index
- can get # affected tuples (i.e. updates) w/ fcn SQLRowCount
- params are bound to buffers by SQLBindParameter:

```
SQLBindParameter(stmt-handle,
                 param-nr,
                 inp/out,
                 c-type,
                 db-type,
                 db-prec,
                 db-scale,
                 val-ptr,val-len,
                 val-NULL-ptr)
```

  ↳ subs val-ptr (w/ length val-len, indicator var val-NULL-ptr, & C data type c-type) for param-nr th param of stmt-handle (using db-type)
- executing stmt w/o binding all params can be accomplished by setting last arg of SQLBindParameter to point to constant SQL_DATA_AT_EXEC
  ↳ results in SQL-NEED_DATA error code
    - param vals can be supplied when this error code is detected w/ fcns:
      → SQLParamData to determine which arg

→ SQLPutData to supply data

- late binding of params allows us to update several rows using 1 update stmt by supplying diff vals for each individual row to be updated
- e.g.

```
SQLCHAR  stmt[] = "UPDATE author SET url = ? WHERE aid = ?";

SQLINTEGER aid;
SQLCHAR    s[70];
SQLINTEGER ind;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLPrepare(hstmt, stmt, SQL_NTS);

printf"(Enter Author ID:  "); scanf("%ld",&aid);
printf"(Enter Author URL: "); scanf("%s", s);

rc = SQLBindParameter(hstmt, 1,
              SQL_PARAM_INPUT, SQL_C_CHAR,
              SQL_CHAR, 0, 0, s, 70, &ind);

rc = SQLBindParameter(hstmt, 2,
              SQL_PARAM_INPUT, SQL_C_SLONG,
              SQL_INTEGER, 0, 0, &aid, 0, NULL);

rc = SQLExecute(hstmt);
```

- several fcns can obtain output vals computed by SQL stmt:
  - ↳ # affected rows:
    - SQLRowCount
  - ↳ ans to queries:
    - SQLBindCol
      - → binds vars before execution
    - SQLGetData
      - → get vals after execution
  - ↳ obtain next tuple of query eval:
    - SQLFetch
      - → result is just a result code
- e.g. query w/ SQLBindCol:

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER   rows;
struct { SQLINTEGER ind;
         SQLCHAR    s[70];
       } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
                (SQLPOINTER)pubid.s, 8, &pubid.ind);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
                (SQLPOINTER)title.s, 70, &title.ind);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
   printf("%-8.8s %-70.70s\n", pubid.s, title.s);

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

- e.g. query w/ SQLGetData

```
SQLCHAR  sqlstmt[] = "SELECT pubid, title FROM publication";

SQLINTEGER   rows;
struct { SQLINTEGER ind;
         SQLCHAR    s[70];
       } pubid, title;

rc = SQLAllocStmt(hdbc, &hstmt);

rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);

while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS) {
   rc = SQLGetData(hstmt, 1, SQL_C_CHAR,
            (SQLPOINTER) pubid.s, 8, &(pubid.ind));
   rc = SQLGetData(hstmt, 2, SQL_C_CHAR,
            (SQLPOINTER) title.s, 70, &(title.ind));
   printf("%-8.8s %-70.70s \n", pubid.s, title.s);
}

rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);

rc = SQLFreeStmt(hstmt, SQL_DROP);
```

- to determine # result cols for queries + col name + type info for each, use these fcns:

```
SQLNumResultCols(hstmt, &num)
SQLDescribeCol(hstmt,ColNo,
    ColNamebuf, sizeof(ColNamebuf),
    NULL,
    &sqltype,&sqlprec,&sqlscale,
    &ifNullable );
```

  - ↳ e.g.

```
SQLINTEGER sqlprec;
SQLSMALLINT i, num, sqltype, sqlscale, nullable;
SQLCHAR name[32];

rc = SQLNumResultCols(hstmt, &num);

for (i=0; i<num; i++) {
   rc = SQLDescribeCol(hstmt,i+1,name,32,NULL,
                  &sqltype,&sqlprec,&sqlscale,&nullable);
   printf("attribute %d is %s (%d,%ld,%d,%d)\n"
               i,name,sqltype,sqlprec,sqlscale,nullable);
}
```

- start of **new transaction** happens implicitly w/ executable SQL cmds (e.g. SQLPrepare, SQLExecute, SQLExecDirect, etc.)
- to signal end of transaction, use fcn **SQLTransact(henv, hdbc, <what>)**
  - ↳ <what> = SQL_COMMIT or
  - ↳ <what> = SQL_ROLLBACK
- ODBC is at least as expressive as dynamic embedded SQL
  - ↳ not bound to particular RDBMS engine
  - ↳ single threads can access multiple engines
- all stmts in ODBC are dynamic
  - ↳ no pre-compilation
  - ↳ requires explicit binding of params
    - user responsible for ensuring types match

THE ENTITY RELATIONSHIP DATA MODEL

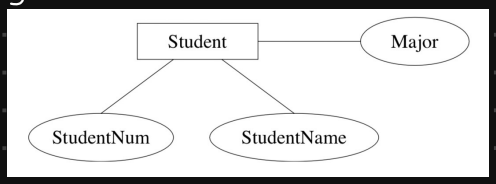
ENTITY RELATIONSHIP (ER) MODELLING

- **ER data model (ERM)** is conceptual data model, which is first step in formally capturing metadata for info systems
- **metadata** is captured in terms of:
  - ↳ entities
  - ↳ attributes
  - ↳ r/tnships

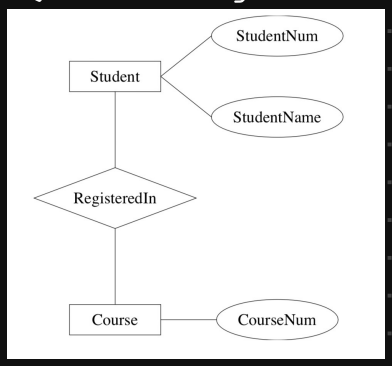
syntax for specifying ER model is graphic visualization

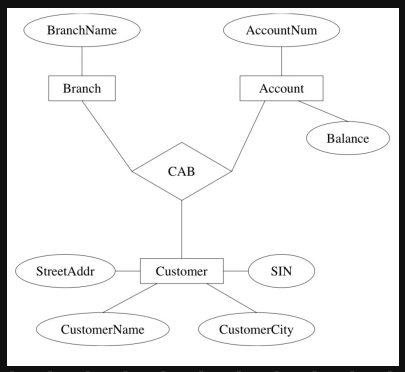
- an **entity** is a distinguishable thing
  - ↳ **entity set** is set of entities of same variety
    - e.g. students currently at UW, flights offered by Air Canada, burglaries in Ontario during 1994
      - → graphic visualization:

- **attribute** captures concrete fact abt entity
  - ↳ **domain** is set of possible vals for an attr
  - ↳ e.g. attrs for entities that are students: student #, name, major
    - graphic visualization:

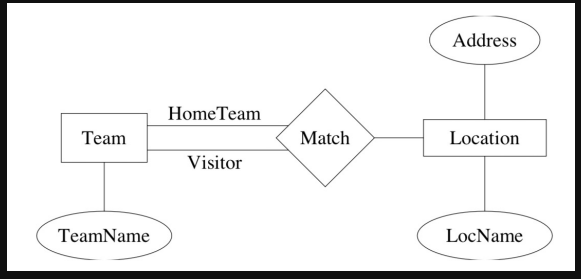


- **r/tnship** captures existence of an association btwn 2+ entities
  - ↳ **r/tnship set** is set of r/tnships of same variety
    - e.g. students registered in courses & bank branches, customers, & their accounts

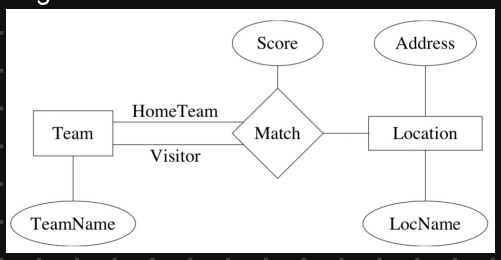




  - ↳ uniquely determined by participating entities
- **role** is purpose served by particular entity in r/tnship
  - ↳ **role name** is id indicative of this purpose
    - e.g. match takes place btwn home team & visitor



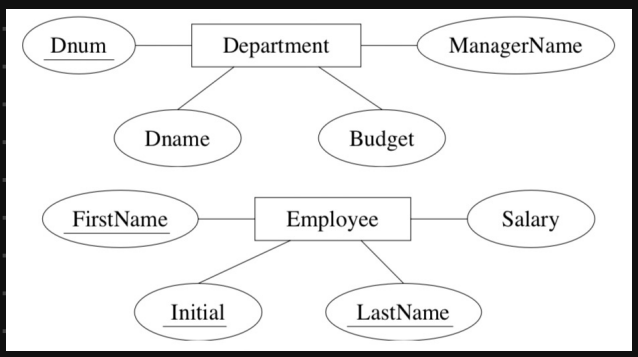
- explicit role names needed whenever component entity set of rltnship set serves more than 1 purpose in its rltnships
  - ↳ may also have attrs
    - e.g. match has score



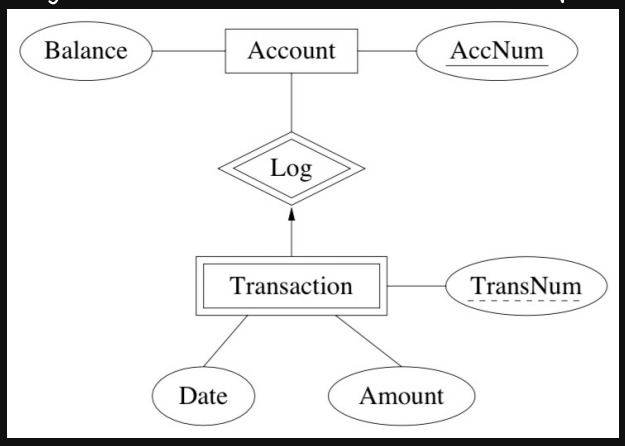
- still uniquely determined by participating entities ∴ not differing attrs

INTEGRITY CONSTRAINTS

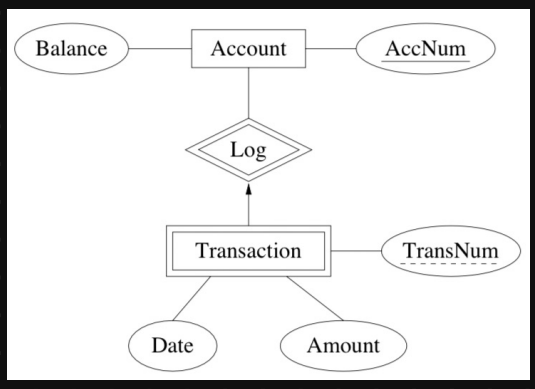
- 4 varieties of integrity constraints in ER models:
  - ↳ primary keys
  - ↳ bin rltnship types
  - ↳ existence dependencies
  - ↳ general cardinality constraints
- general integrity constraints can be captured as sentences in RC by defining mapping of ER diagrams to rltnal signatures
  - ↳ yields formal semantics ∴ ER query lang
- primary key is selection of attrs for entity set for which facts serve as means of ref to its entities
  - ↳ e.g. departments id by department # ∴ employees id by first name, last name, ∴ middle initial



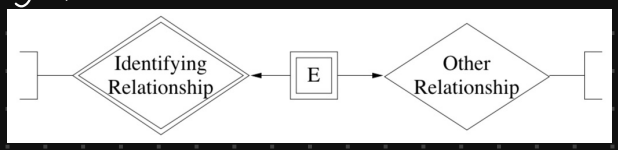
- if x is existent dependent on y, then y is dominant entity ∴ x is subordinate entity
  - ↳ e.g. transactions are existence dependent on accounts



- weak entity set is a set containing subordinate entities
  - ↳ discriminator is selection of attrs for weak entity set for which facts serve as means of distinguishing subordinate entities for any given dominant entity
    - e.g. each transaction for given account has unique #

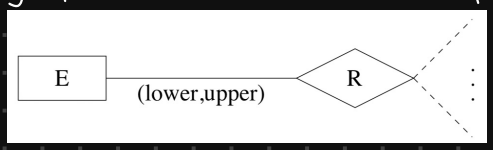


↳ must be (N:1) rltnship w/at least 1 distinct entity set
↳ graphical annotation via **double boundaries**:



↳ **primary key for weak entity set** is combo of facts for its discriminator ∧ facts for each attr of primary keys of dominant entity sets
- defn is recursive b/c dom entity may itself be sub to another entity

- no cycles of bin rltnship sets that are all identifying are allowed
- **general cardinality constraint** determines lower ∧ upper bounds on # rltnships of given rltnship set in which component entity must participate
  ↳ graphical annotation via **component edge labelling**:



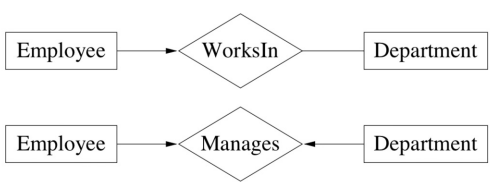
↳ e.g. students must takes btwn 3-5 courses ∧ courses must have btwn 6-100 students taking them

↳ if upper bound is N, none exist

- bin rltnship types:
  ↳ **many-to-many (N:N)**: entity in 1 entity set can be related to any # entines in other ∧ converse holds
  ↳ **many-to-one (N:1)**: each entity in 1 entity set can be related to at most 1 entity in other entity set, but no limit exists for converse
  ↳ **one-to-many (1:N)**: inverse of many-to-one
  ↳ **one-to-one (1:1)**: each entity in 1 entity set can be related to at most 1 entity in other entity set ∧ same holds for converse
  ↳ none of these rltnship types imply mandatory participation of entities
  ↳ e.g.

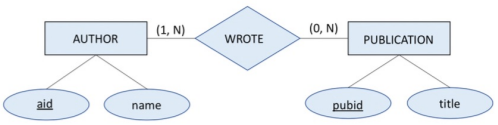
1. *Employees work in at most one department.*
2. *Employees manage at most one department, and departments are managed by at most one employee.*

Graphical annotation (via *arrowheads*):



- general integrity constraints for ER diagram can be expressed in RC via mapping to rltnal signature ρ satisfying:

↳ <E>(self) ∈ ρ for each entity set <E>
↳ <A>(self, <a>) ∈ ρ for each attr <a>
↳ <R>(c₁, ..., cₖ) ∈ ρ for each rltnship set for k-ary associations <R>
  • c_i is either entity set / role name
↳ e.g.



$\rho = (\ \texttt{AUTHOR}(\texttt{self}), \texttt{AID}(\texttt{self}, \texttt{aid}), \texttt{NAME}(\texttt{self}, \texttt{name}),$
$\texttt{PUBLICATION}(\texttt{self}), \texttt{PUBID}(\texttt{self}, \texttt{pubid}), \texttt{TITLE}(\texttt{self}, \texttt{title}),$
$\texttt{WROTE}(\texttt{author}, \texttt{publication})\ )$

• some integrity constraints induced by bibliography ER diagram:

- *Attributes* `aid` *and* `name` *are single-valued.*
  $\forall e, v_1, v_2.(\texttt{AID}(e, v_1) \wedge \texttt{AID}(e, v_2) \rightarrow v_1 = v_2)$
  $\forall e, v_1, v_2.(\texttt{NAME}(e, v_1) \wedge \texttt{NAME}(e, v_2) \rightarrow v_1 = v_2)$
- *Entities of a given entity set have a given attribute value.*
  $\forall e.(\texttt{AUTHOR}(e) \rightarrow \texttt{AID}(e, -))$
- *Participating entities of a* `WROTE` *relationship must be an author and a publication.*
  $\forall e.(\texttt{WROTE}(e, -) \rightarrow \texttt{AUTHOR}(e))$
  $\forall e.(\texttt{WROTE}(-, e) \rightarrow \texttt{PUBLICATION}(e)).$
- *Values for attribute* `aid` *are used to identity authors.*
  $\forall e_1, e_2, v.(\texttt{AUTHOR}(e_1) \wedge \texttt{AID}(e_1, v) \wedge \texttt{AUTHOR}(e_2) \wedge \texttt{AID}(e_2, v) \rightarrow e_1 = e_2)$
- *An author must have written at least one publication.*
  $\forall e.(\texttt{AUTHOR}(e) \rightarrow \texttt{WROTE}(e, -))$

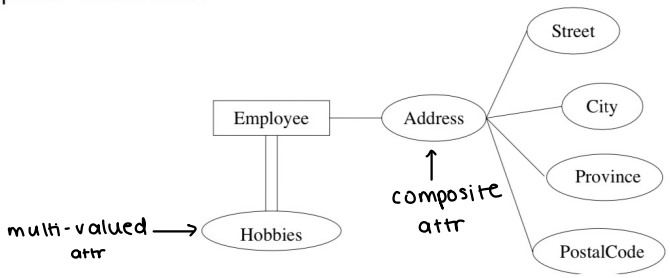
EXTENDED ENTITY RELATIONSHIP (EER) MODELLING

- composite attr denotes fixed collection of other attr facts
- multi-valued attr denotes finite set of similar facts

e.g.

Example: `Hobbies` *are a set of facts about leisure activities; an* `Address` *consists of a* `Street`, `City`, `Province` *and* `PostalCode`.

Graphical visualization:

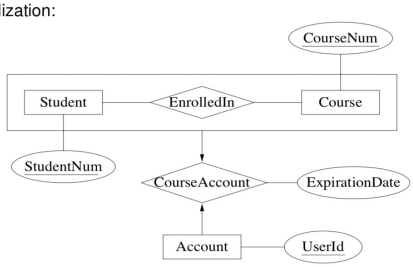


- composite attrs may be multi-valued ? can be collection of facts abt attrs for which some are also composite
- rltnship set can be aggregated to enable its rltnships to be higher-lvl entities that can in turn participate in other rltnships

↳ e g

Example: *Accounts are assigned to a given student enrollment.*

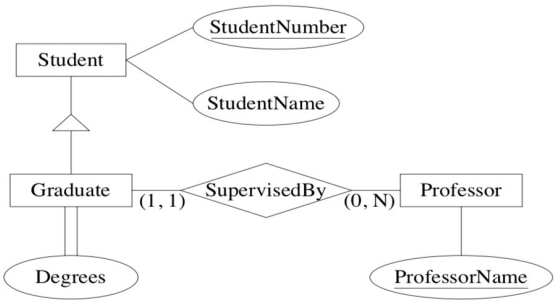
Graphical visualization:



- specialization is integrity constraint asserting that entities of 1 entity set are also entities of another entity set
  - ↳ e.g.

Example: *Graduate students are students who have a supervisor and a number of degrees.*

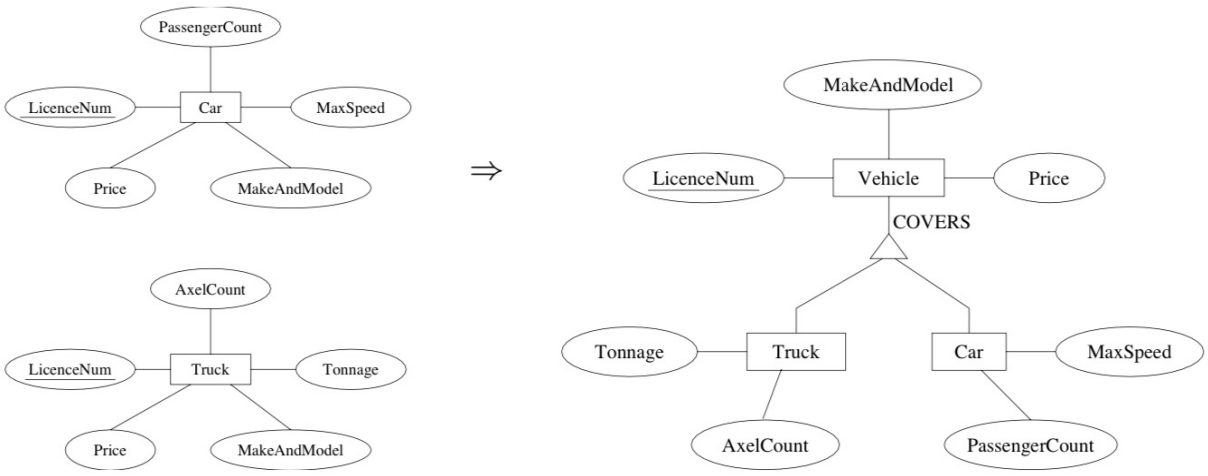
Graphical visualization:



  - ↳ enables top down authoring of ER diagram
    - i.e. overall structured is defined first, then we fill in specific details
- generalization is integrity constraint asserting that entities of 1 entity set are also entities of at least 1 of 2+ other entity sets
  - ↳ e.g.

Example: *A vehicle is also either a car or a truck.*

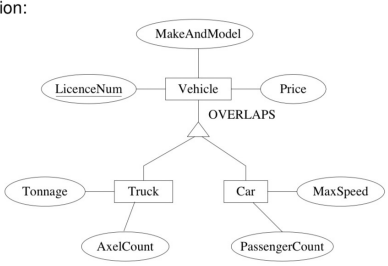
Graphical visualization:



    - "COVERS" annotation is optional ? always assumed
  - ↳ enables bottom up authoring of ER diagrams
- 2 entity sets participating in generalization are assumed to be disjoint by default
  - ↳ can be overridden w/ "OVERLAPS" annotation
    - e.g.

Example: *There are entities that can be both a car and a truck, such as a utility vehicle.*

Graphical annotation:

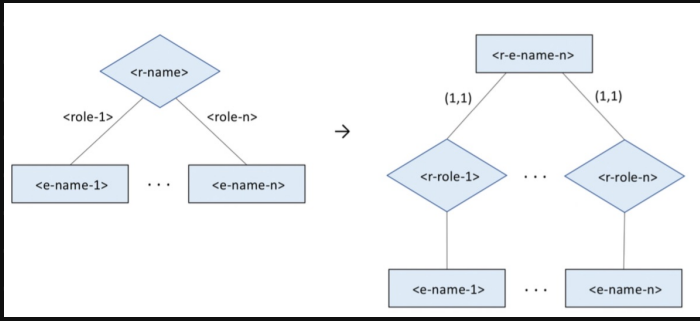


- e.g. additional predicates induced by extended features:

- (*specialization*) *Entity set* GRADUATE *is a specialization of entity set* STUDENT.
  $\forall e.(\texttt{GRADUATE}(e) \rightarrow \texttt{STUDENT}(e))$
- (*generalization and disjunction*) *Entity set* VEHICLE *is a generalization of entity sets* TRUCK *and* CAR.
  $\forall e.(\texttt{TRUCK}(e) \rightarrow \texttt{VEHICLE}(e))$
  $\forall e.(\texttt{CAR}(e) \rightarrow \texttt{VEHICLE}(e))$
  $\forall e.(\texttt{VEHICLE}(e) \rightarrow (\texttt{TRUCK}(e) \vee \texttt{CAR}(e)))$

  If generalization is not annotated with "OVERLAPS", then the entity sets TRUCK and CAR are disjoint.
  $\forall e.(\texttt{TRUCK}(e) \rightarrow \neg\texttt{CAR}(e))$

- reification replaces rltnship on n entity sets w/a new entity set w/n bin rltnships on the entity sets:



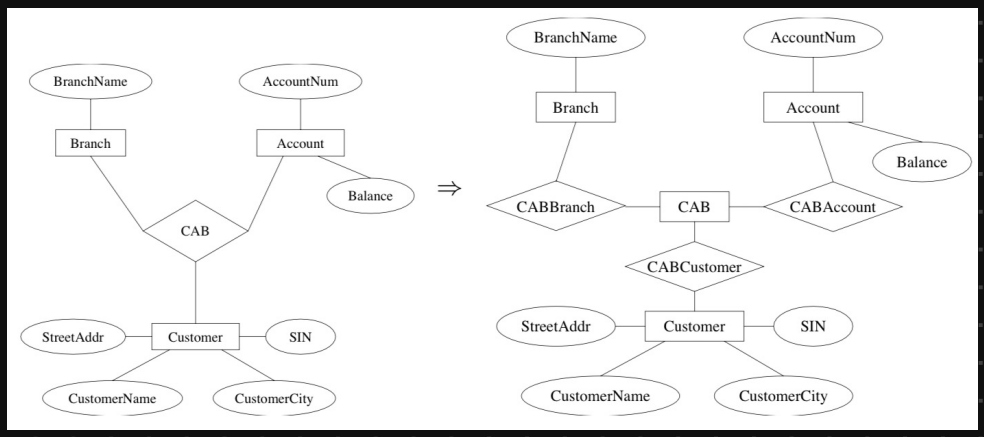
↳ e.g.

(*aggregation*) EnrolledIn *relationships can themselves participate as component entities of* CourseAccount *relationships.*

1. add to ρ the new predicates
   ENROLLEDIN-ENT(self),
   STUDENT-COMP(self,student) and
   COURSE-COMP(self,course)
2. add the integrity constraints
   $\forall e.(\texttt{ENROLLEDIN-ENT}(e) \rightarrow \texttt{STUDENT-COMP}(e,-))$,
   $\forall e.(\texttt{ENROLLEDIN-ENT}(e) \rightarrow \texttt{COURSE-COMP}(e,-))$,
   $\forall e_1.(\texttt{STUDENT-COMP}(-,e_1) \rightarrow \texttt{STUDENT}(e_1))$,
   $\forall e, e_1, e_2.(\texttt{STUDENT-COMP}(e,e_1) \wedge \texttt{STUDENT-COMP}(e,e_2) \rightarrow e_1 = e_2)$,
   $\forall e_2.(\texttt{COURSE-COM}(-,e_2) \rightarrow \texttt{COURSE}(e_2))$ and
   $\forall e, e_1, e_2.(\texttt{COURSE-COMP}(e,e_1) \wedge \texttt{COURSE-COMP}(e,e_2) \rightarrow e_1 = e_2)$;
3. and make ENROLLEDIN a view with the integrity constraint
   $\forall e_1, e_2.\texttt{ENROLLEDIN}(e_1,e_2)$
   $\leftrightarrow \exists e.(\texttt{ENROLLEDIN-ENT}(e) \wedge \texttt{STUDENT-COMP}(e,e_1) \wedge \texttt{COURSE-COMP}(e,e_2))$.

DESIGN METHODOLOGY

- ER diagram for info system is usually obtained from 2 sources:
  ↳ from parts of ER diagrams for existing info systems
  ↳ from informal requirements for info system obtained by requirements elicitation
- rule of thumb when modelling smth by attr vs entity: it's an entity if there's affirmative ans to any of following:
  ↳ is it a separate obj?
  ↳ do we maintain info abt it?
  ↳ can several of its kind belong to a single entity?
  ↳ does it make sense to delete such an obj?
  ↳ can it be missing from some of entity set's entities
  ↳ can it be shared by diff entities?
- e.g.

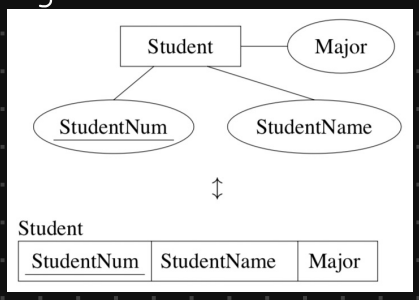


- simple methodology:
  1) recognize entity sets
  2) recognize rltnship sets ? participating entity sets
  3) recognize attrs of entity ? rltnship sets
  4) define rltnship types ? existence dependencies
  5) define general cardinality constraints, keys, ? discriminators
  ↳ for each step, update ER diagram ? maintain log of assumptions ? restrictions

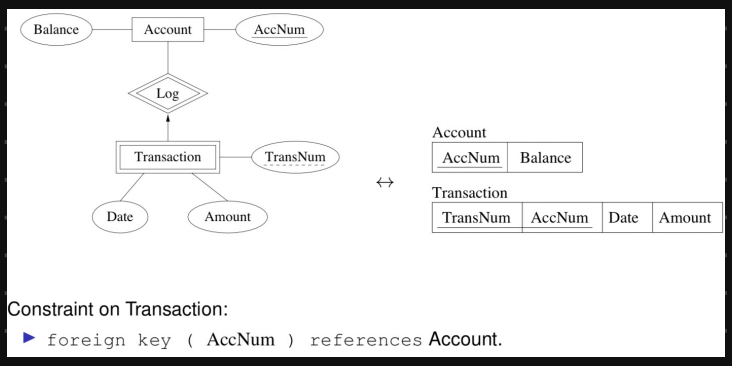
LOGICAL MAPPING

LOGICAL MAPPING OF BASIC ER DIAGRAMS

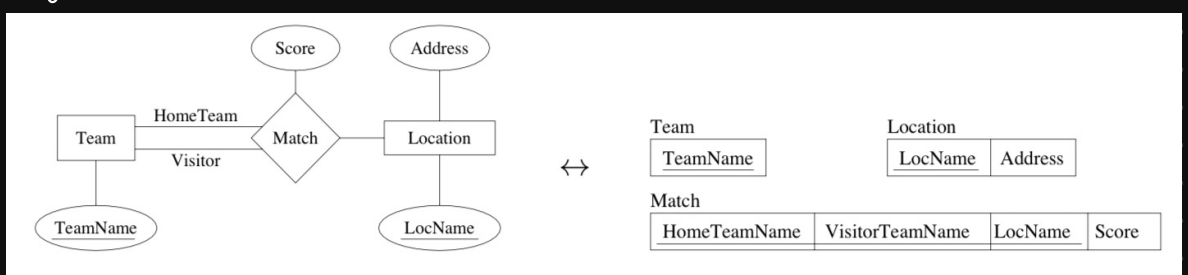
- **logical mapping** is to obtain logical design for db given conceptual design
  - ↳ each entity set maps to new table
  - ↳ each attr maps to new table col
  - ↳ each rltnship set maps to either new table col or new table
- **entity set** E w/ attrs $a_1, \ldots, a_n$ maps to new table E w/ attr cols $a_1, \ldots, a_n$
  - ↳ entity of type E is row in table E
  - ↳ e.g.



- **weak entity set** E also maps to new table E
  - ↳ cols should include:
    - attrs of weak entity set
    - attrs of identifying rltnship set
    - primary key attrs of dominating entity set
  - ↳ e.g



- **rltnship set** R may either map to new table cols for existing tables or entirely new table
  - ↳ if it's alr identifying rltnship set for weak action set, no action needed
  - ↳ if general cardinality constraint is (1,1) for component entity set E, add these cols to E:
    - attrs of rltnship set
    - primary key attrs of remaining component entity sets
  - ↳ otherwise, make new table R
- cols of **table R** should include:
  - ↳ attrs of rltnship set
  - ↳ primary key attrs of each component entity set
- **primary key of R** is the same as component entity set E if general cardinality constraint is (0,1)
  - ↳ otherwise, choose primary key attrs of each component entity set
- e.g.



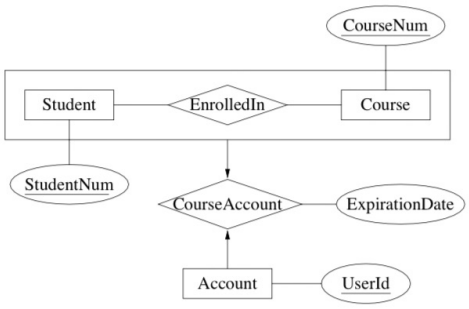
Role names and component entity set names are combined to form more descriptive attribute names.

Constraints on table Match:

- `foreign key ( HomeTeamName ) references` Team.
- `foreign key ( VisitorTeamName ) references` Team.
- `foreign key ( LocName ) references` Location.

MAPPING FOR EXTENDED FEATURES

- when repping **aggregation** of R, treat it as an entity set whose primary key is same as R's table
  - ↳ always map rltnship set R that's aggregated to new table R
  - ↳ e.g.



↔

Student

StudentNum

Course

CourseNum

Account

UserId

EnrolledIn

StudentNum	CourseNum

CourseAccount

UserId	StudentNum	CourseNum	ExpirationDate

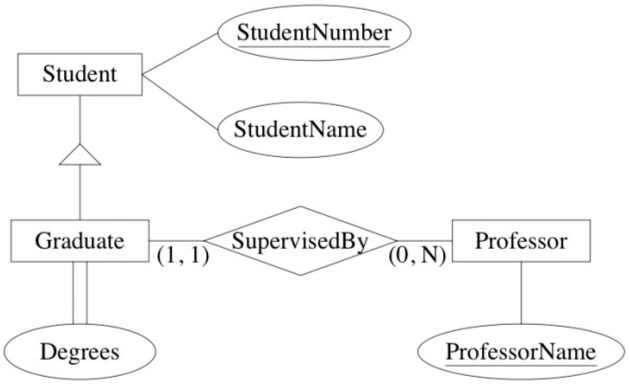
Constraints on table EnrolledIn:

- `foreign key ( StudentNum ) references` Student.
- `foreign key ( CourseNum ) references` Course.

Constraints on table CourseAccount:

- `foreign key ( UserId ) references` Account.
- `foreign key ( StudentNum, CourseNum ) references` EnrolledIn.

- when repping **specialization** of entity set, treat it as weak entity set w/ empty discriminator set ? is existence dependent on each of its entity sets that it's a specialization of
  - ↳ e.g.



↔

Student

StudentNumber	StudentName

Graduate

StudentNumber	ProfessorName

Degree

StudentNumber	Degree

Professor

ProfessorName

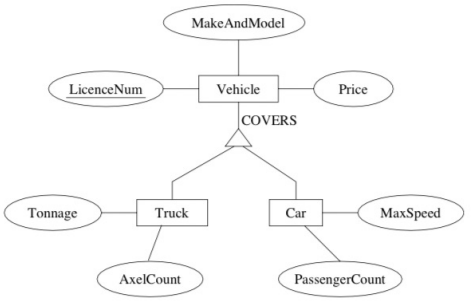
Contraints:

- (Graduate) `foreign key ( StudentNumber ) references` Student.
- (Graduate) `foreign key ( ProfessorName ) references` Professor.
- (Degree) `foreign key ( StudentNumber ) references` Graduate.

  - also shows how to map multi-valued attrs (in this example, Degree)

- when repping **generalization** of n entity sets, treat them as n specializations ? add constraints for coverage ? disjointness

↳ e.g.



↔

Vehicle

LicenceNum	MakeAndModel	Price

Truck

LicenceNum	Tonnage	AxelCount

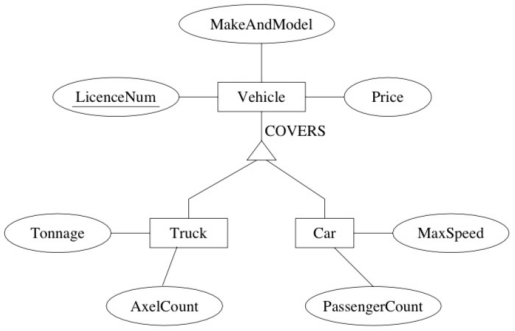
Car

LicenceNum	MaxSpeed	PassengerCount

Contraints (expressing the last two in SQL is an exercise):

- (Truck) `foreign key ( LicenceNum ) references` Vehicle.
- (Car) `foreign key ( LicenceNum ) references` Vehicle.
- An assertion that vehicle licence numbers are also truck or car licence numbers.
- An assertion that truck licence numbers are disjoint from car licence numbers (unless `OVERLAPS` annotates the generalization).

- entity set can be mapped to **view** instead of table
  - ↳ when entity set E is **specialization** of 1 parent set
    - in addition:
      - → only typing attrs are declared on E
      - → no foreign key constraints ref E
      - → all entity sets that are specializations of E are also mapped to views
    - add **2-valued typing attr** is-E tgt w/ E's existing typing attrs to parent entity set so we can enable view defn for E's mapping
    - multiple typing attrs can be replaced w/ single one when underlying entity sets are disjoint
  - ↳ when entity set E is **generalization** of 2+ child entity sets & has no foreign key constraints that refs it
    - e.g.



↕

Truck

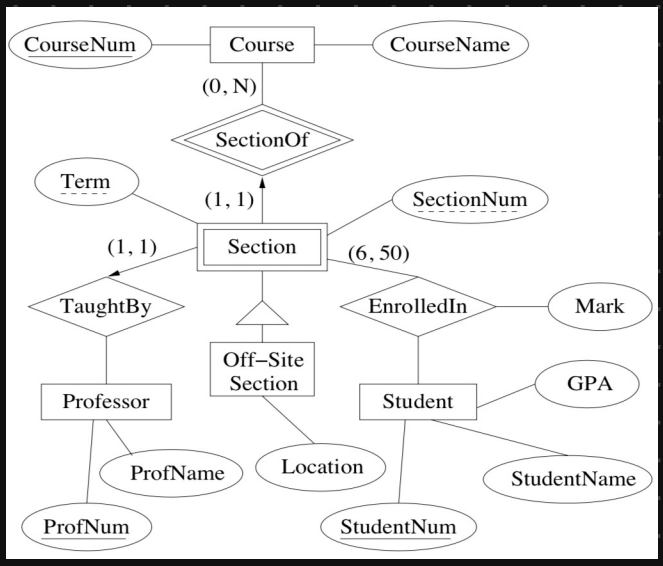
LicenceNum	MakeAndModel	Price	Tonnage	AxelCount

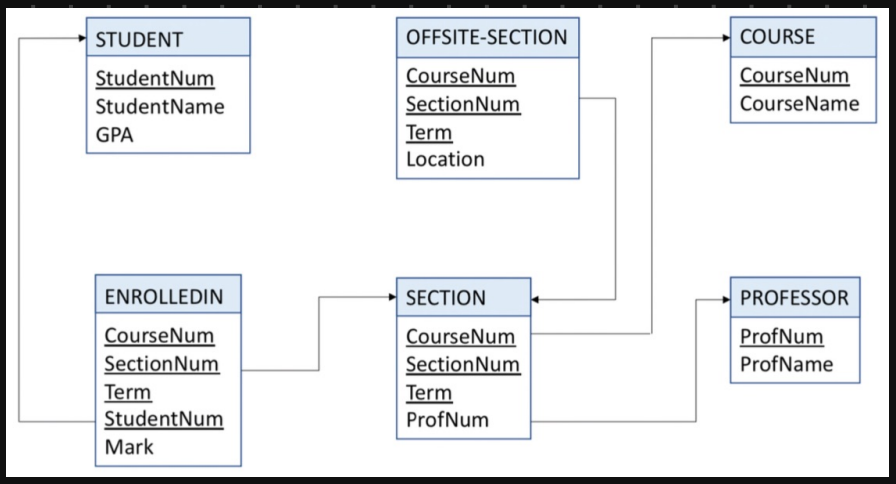
Car

LicenceNum	MakeAndModel	Price	MaxSpeed	PassengerCount

```
create view Vehicle as (
   ( select LicenceNum, MakeAndModel, Price from Truck ) union
   ( select LicenceNum, MakeAndModel, Price from Car ) )
```

- e.g. translating ER diagram to logical design
  1) map entity sets
  2) map specializations
  3) map rlmship sets





TRIPLE STORE

- all triple store ritnal schemas have additional view that presents data in form of a simple sentence
  - ↳ create view TRIPLE as (<query>)
  - ↳ schema is TRIPLE / (subject, property, object)
- modify ER diagram by:
  - ↳ making all weak entity sets into regular ones
  - ↳ adding OID attr & making it primary key for each entity set

assuming Ti has schema Ti / (OID, A1, ..., An), query QTi is:

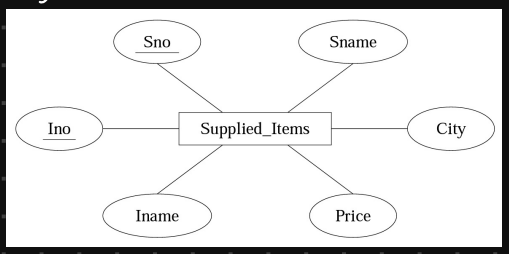
```
(select OID as subject, 'in' as property, 'Ti' as object from Ti)
  union
(select OID as subject, 'A1' as property, A1 as object from Ti)
  union
   :
  union
(select OID as subject, 'An' as property, An as object from Ti)
```

- data in TRIPLE replicates data in all other tables
  - ↳ i.e. all queries over triple store schema can be done using only table TRIPLE

DEPENDENCIES AND NORMAL FORMS

GOOD DESIGN OF RELATIONAL SCHEMA

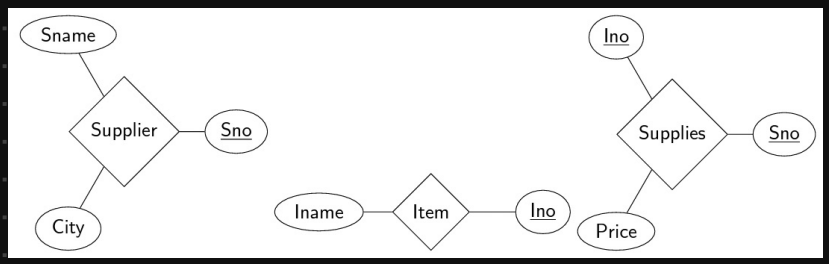
- criteria to choose btwn rltnal schema designs
  - ↳ **usability** : how design affects productivity
    - easily expressing queries
    - easily expressing data revision
    - metadata transparency
  - ↳ **resource requirements**: allowed schema instances
- e.g. ER diagram for a supplied item entity



Supplied_Items

Sno	Sname	City	Ino	Iname	Price
S1	Magna	Ajax	I1	Bolt	0.50
S1	Magna	Ajax	I2	Nut	0.25
S1	Magna	Ajax	I3	Screw	0.30
S2	Budd	Hull	I3	Screw	0.40

- ↳ problems w/ expressing data revision:
  - updating existing data
    - → e.g. change name of supplier S1 to "ACME"
  - adding new data
    - → e.g. add item T4 w/ name "Washer"
  - deleting existing data
    - → e.g. supplier named "Budd" no longer supplies screws
- ↳ better alt is rltnal schema design that **decomposes** table into 3 separate ones:



Supplier

Sno	Sname	City
S1	Magna	Ajax
S2	Budd	Hull

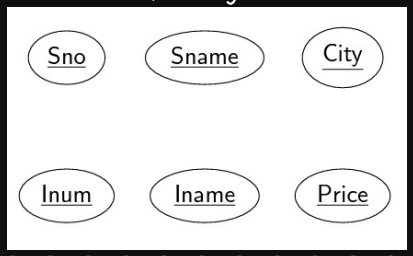
Item

Ino	Iname
I1	Bolt
I2	Nut
I3	Screw

Supplies

Sno	Ino	Price
S1	I1	0.50
S1	I2	0.25
S1	I3	0.30
S2	I3	0.40

- ↳ however, splitting table too much results in loss of info abt rltnships:



- to diagnose & repair anomalies:
  - ↳ certain families of **integrity constraints** can help detect regularities / patterns in schema instance that may lead to anomalies
  - ↳ possible repair is table decomposition
- assume rltnal schema is given by set of attrs $\{A_1, \ldots, A_k\}$, let R be rltnal schema, & $\{R_1, \ldots, R_n\}$ be set of n rltnal schema
  - ↳ $\{R_1, \ldots, R_n\}$ is **decomposition** of R if $R = \bigcup_{1 \le i \le n} R_i$
    - decomp doesn't lose info

FUNCTIONAL DEPENDENCIES

- instance that satisfies all real constraints is **legal instance** of rltn

- let R be k-ary rltn $R/(A_1, \ldots, A_k)$ where $\ell(i) = A_i$ for $1 \le i \le k$; let X ? Y be subsets of $\{A_1, \ldots, A_k\}$ (attrs of R)
  - ↳ **functional dependency** over $\{A_1, \ldots, A_k\}$ is written $X \to Y$
    - shorthand for constraint: $\forall v_1, \ldots, v_k, w_1, \ldots, w_k \; . \; R(v_1, \ldots, v_k) \wedge R(w_1, \ldots, w_k) \wedge \left(\bigwedge_{A \in X} v_{\ell^{-1}(A)} = w_{\ell^{-1}(A)}\right) \to \left(\bigwedge_{A \in Y} v_{\ell^{-1}(A)} = w_{\ell^{-1}(A)}\right)$ (i.e. $\forall t_1, t_2 \; . \; t_1[X] = t_2[X] \to t_1[Y] = t_2[Y]$)
    - X functionally determines Y in R
    - i.e for a given X, there's a unique association w/ Y
    - e.g.

EmpProj

SIN	PNum	Hours	EName	PName	PLoc	Allowance

Examples of functional dependencies over EmpProj:†

- *Employee SIN numbers functionally determine employee names.*
  SIN → EName
- *Project numbers functionally determine project names and locations.*
  PNum → PName, PLoc
- *Project locations and the number of hours functionally determine allowances.*
  PLoc, Hours → Allowance

- given rltn $R/(A_1, \ldots, A_k)$? set $F \cup \{X \to Y\}$ of FDs over R, we say **F logically implies $X \to Y$** when $X \to Y$ holds in all instances of R that satisfies each FD in F
  - ↳ **closure F^+** of F is set of all FDs logically implied by F
    - $F \subseteq F^+$
    - e.g. if $F = \{A \to B, B \to C\}$, then $\{A \to B, B \to C, A \to C\} \subseteq F^+$
- **Armstrong's axioms** for inferring FDs from FDs:
  - ↳ reflexivity: if $Y \subseteq X$, then $X \to Y$
  - ↳ augmentation: if $X \to Y$, then $XZ \to YZ$
    - XZ ? YZ are shorthands for $X \cup Z$? $Y \cup Z$
  - ↳ transivity: if $X \to Y$? $Y \to Z$, then $X \to Z$
  - ↳ union: if $X \to Y$? $X \to Z$, then $X \to YZ$
  - ↳ decomposition: if $X \to YZ$, then $X \to Y$
- axioms are sound (i.e. anything derived from F is in F^+) ? complete (i.e. anything in F^+ can be derived)
- e.g.

Example: *Let F be the following FDs over EmpProj*:

$F = \{$ SIN → EName
PNum → PName, PLoc
PLoc, Hours → Allowance
SIN, PNum → Hours† $\}$

The FD SIN, PNum → Allowance can be derived from *F* as follows:

1.	SIN, PNum → Hours	($\in F$)
2.	PNum → PName, PLoc	($\in F$)
3.	PLoc, Hours → Allowance	($\in F$)
4.	SIN, PNum → PNum	(reflexivity)
5.	SIN, PNum → PName, PLoc	(4, 2 and transitivity)
6.	SIN, PNum → PLoc	(5 and decomposition)
7.	SIN, PNum → PLoc, Hours	(6, 1 and union)
8.	SIN, PNum → Allowance	(7, 3 and transitivity)

- given rltn R w/attrs $\{A_1, \ldots, A_k\}$? subset K of R, K is **superkey** of R if FD $K \to R$ holds on any instance of R
- given rltn R w/attrs $\{A_1, \ldots, A_k\}$? subset K of R, K is **candidate key** of R if FD $K \to R$ holds on any instance of R ? no strict subset of K is a superkey
- given rltn R w/attrs $\{A_1, \ldots, A_k\}$, **primary key** of R is candidate key chosen by DBA group
- assume R is set of attrs, X is subset of R, ? F is set of FDs over R
  - ↳ **Compute X^+** algo returns subset of R
    - can be implemented in linear time
    - $(X \to Y) \in F^+$ iff $Y \subseteq$ Compute $X^+(X, F)$

- ```
  function ComputeX+(X, F)
  begin
      X+ := X;
      while true do
          if there exists (Y → Z) ∈ F such that
              (1) Y ⊆ X+, and
              (2) Z ⊈ X+
          then X+ := X+ ∪ Z
          else exit;
      return X+;
  end
  ```

- superkey thm for FDs: X is superkey of rltn w/ schema R iff R ⊆ ComputeX+(X, F)

FORMAL PROPERTIES OF GOOD DESIGN

- if given rltnal db schema ⟨ρ, Σ⟩ has alr undergone repair by decomp ? Σ consists of set of FDs F, it should satisfy:
  - ↳ lossless-join: no loss of info in comparison to og schema before decomp
  - ↳ dependency preserving: no added complications in checking for violation of FDs in F
  - ↳ normal form: no / fewer change anomalies in expressing data revision
- lossless-join decomp means we should be able to construct instance of any og table from instances of tables in decomp
  - ↳ e.g.

Example: *Consider where the table*

Marks

Student	Assignment	Group	Mark
Ann	A1	G1	80
Ann	A2	G3	60
Bob	A1	G2	60

has been replaced by the following two tables in the decomposition.

SGM

Student	Group	Mark
Ann	G1	80
Ann	G3	60
Bob	G2	60

AM

Assignment	Mark
A1	80
A2	60
A1	60

- query in SQL expresses natural join of SGM ? AM:

```
select distinct Student, Assignment, Group, x.Mark as Mark
form SGM x, AM y
where x.Mark = y.Mark
```

Computing the natural join of SGM and AM produces the following table.

Student	Assignment	Group	Mark
Ann	A1	G1	80
Ann	A2	G3	60
Ann	A1	G3	60 !
Bob	A2	G2	60 !
Bob	A1	G2	60

→ tables loses info by computing spurious tuples (i.e. extra tuples)

- decomp {R1, R2} of R is lossless-join decomp if for any instance of R, natural join of R1 ? R2 has no spurious tuples
  - ↳ decomp {R1, R2} of R is lossless-join decomp iff common attrs of R1 ? R2 form superkey for either schema
    - i.e. $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$ is logical consequence of integrity constraints
    - e.g.
```

Example: *With table* Marks *we have*

R = {Student, Assignment, Group, Mark}
F = {Student, Assignment → Group, Mark}

R_1 = SGM = {Student, Group, Mark}
R_2 = AM = {Assignment, Mark}

Decomposition $\{R_1, R_2\}$ *is lossy because* {Mark} $(= R_1 \cap R_2)$ *is not a superkey of either* SGM *or* AM *according to integrity constraints* F.

- e.g.

Example: *Assume an original table and integrity constraints over the table are given as follows:*

R / (Proj, Dept, Div)

F = { FD1: Proj → Dept, FD2: Dept → Div, and FD3: Proj → Div }

and consider two possible decompositions:

D_1 = { R1 / (Proj, Dept), R2 / (Dept, Div)}

D_2 = { R1 / (Proj, Dept), R3 / (Proj, Div)}

↳ both are lossless
↳ D_1 is easier to check than D_2 that all constraints in F continue to hold after revision to 1+ of its tables
- D_1: test FD1 on R1, test FD2 on R2, ? if they're both satisfied, FD3 is automatically satisfied
- D_2: test FD1 on R1, test FD3 on R3, ? test FD2 on natural join of R1 ? R3 b/c it's an inter-rltnal constraint

- decomp $D = \{R_1, \ldots, R_n\}$ of R is dependency preserving if there's equiv set G of FDs that aren't inter-rltnal in D
- rule of thumb is to record facts abt diff entities in separate tables
↳ each rltn schema should consist of 2 sets of attrs:
- 1st set has primary keys to record facts that identify entity
- 2nd set records indep facts abt entity
↳ decomp schema to normal form

- schema R is in Boyce-Codd Normal Form (BCNF) wrt set of FDs F iff whenever $(X \to Y) \in F^+$ where $XY \subseteq R$, then $(X \to Y)$ is trivial (i.e. $Y \subseteq X$) or X is superkey of R
↳ db schema $\langle \rho = \{R_1, \ldots, R_n\}, F \rangle$ is in BCNF if each R_i is in BNCF wrt F
- e.g. BCNF avoids redundancy

Example: *For the schema* Supplied_Items *we had the FD:*

Sno → Sname, City

Consequently, the supplier name Magna *and the city* Ajax *must be repeated for each item supplied by supplier* S1.

Assume the above FD holds over a schema R that is in BCNF.

Then:

1. Sno *is a superkey for R (by definition of BCNF).*
2. *Therefore* S1 *appears on one row only.*
3. *And therefore* Magna *and the city* Ajax *are not repeatedly recorded in R for supplier* S1.

↳ i.e. decomp st Sno is primary key of table w/ Sname ? City as other cols

- to compute BCNF decomp:

```
function ComputeBCNF(R, F)
begin
    Result := {R};
    while some R_i ∈ Result and (X → Y) ∈ F+
            violate the BCNF condition do begin
        Replace R_i by R_i − (Y − X);
        Add X ∪ Y to Result;
    end;
    return Result;
end
```

↳ returns lossless-join decomp of R for which each table in decomp is BCNF wrt F
↳ no efficient procedure exists

- possible that no lossless-join dependency preserving BCNF decomp exists
  ↳ e.g.

Example: *Consider* $R = \{A, B, C\}$ *and* $F = \{AB \rightarrow C, C \rightarrow B\}$.
$\Rightarrow$ R is not in BCNF with respect to F
$\Rightarrow$ $AB \rightarrow C$ will be an inter-relational FD in any decomposition of R

THIRD NORMAL FORM

- schema R is in **Third Normal Form (3NF)** wrt set of FDs F iff whenever $(X \rightarrow Y) \in F^+$ & $XY \subseteq R$, then $(X \rightarrow Y)$ is trivial, X is superkey of R, or each attr of $Y - X$ is contained in candidate key of R
  ↳ BCNF implies 3NF, but not reverse
    - 3NF allows more redundancy
    - e.g. $R = \{A, B, C\}$ is in 3NF ✓
  ↳ lossless-join & dependency-preserving decomp into 3NF rltn schemata always exists
- 2 sets of FDs F & G are **equiv** iff $F^+ = G^+$
- **minimal FD sets**:
  ↳ every RHS of dependency in F is single attr
  ↳ for no $X \rightarrow A$ is $F - \{X \rightarrow A\}$ equiv to F
  ↳ for no $X \rightarrow A$ & Z is proper subset of X is $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ equiv to F
- for every F, there's an equiv minimal set of FDs, called **minimal cover**
  ↳ to compute:
    1) replace $X \rightarrow YZ$ w/ $X \rightarrow Y$ & $X \rightarrow Z$
    2) remove $X \rightarrow A$ from F if $A \in \text{Compute}X^+(X, F - \{X \rightarrow A\})$
    3) remove A from LHS of $X \rightarrow B$ in F if B is in $\text{Compute}X^+(X - \{A\}, F)$
    4) replace $X \rightarrow Y$ & $X \rightarrow Z$ in F by $X \rightarrow YZ$
- to compute 3NF decomp:

```
function Compute3NF(R, F)
begin
    Result := ∅;
    G := a minimal cover for F;
    for each (X → Y) ∈ G do
        Result := Result ∪ {XY};
    if there is no R_i ∈ Result such that
        R_i contains a candidate key for R then begin
        compute a candidate key K for R;
        Result := Result ∪ {K};
    end;
    return Result;
end
```

ADDITIONAL DEPENDENCIES AND NORMAL FORMS

- there's update anomalies & data redundancies that can't be captured by FDs
  ↳ e.g.

Example: *Consider the following relation schema and sample valid data:*

Course	**T**eacher	**B**ook
Math	Smith	Algebra
Math	Smith	Calculus
Math	Jones	Algebra
Math	Jones	Calculus
Advanced Math	Smith	Calculus
Physics	Black	Mechanics
Physics	Black	Optics

There are no non-trivial FDs that hold on this instance.

The scheme (**C**ourse, **T**eacher, **B**ook) *or* CTB *for short, is therefore in BCNF.*

- redundant b/c whenever (c, t_1, b_1) & (c, t_2, b_2) occurs in CTB, then (c, t_1, b_2) also occurs in CTB
- multivalued dependency (MVD) holds on CTB
  - → $C \twoheadrightarrow T$ (i.e. given 1 course, set of teachers are uniquely determined & indep of remaining attrs)
  - → $C \twoheadrightarrow B$

- e.g.

Course	**T**eacher	**H**our	**R**oom	**S**tudent	**G**rade
CS101	Jones	M-9	2222	Smith	A
CS101	Jones	W-9	3333	Smith	A
CS101	Jones	F-9	2222	Smith	A
CS101	Jones	M-9	2222	Black	B
CS101	Jones	W-9	3333	Black	B
CS101	Jones	F-9	2222	Black	B

- *FDs that hold on any instance:*

  $C \to T$, $CS \to G$, $HR \to C$, $HT \to R$, $HS \to R$

- *An MVD that also holds on any instance:*

  $C \twoheadrightarrow HR$

- inference axioms:

The following axioms infer both FDs and MVDs from FDs and MVDs:

1. (*reflexivity*) $Y \subset X \Rightarrow X \twoheadrightarrow Y$
2. (*complementation*) $X \twoheadrightarrow Y \Rightarrow X \twoheadrightarrow (R - Y)$
3. (*augmentation*) $X \twoheadrightarrow Y \Rightarrow XZ \twoheadrightarrow YZ$
4. (*transitivity*) $X \twoheadrightarrow Y, Y \twoheadrightarrow Z \Rightarrow X \twoheadrightarrow (Z - Y)$
5. (*conversion*) $X \to Y \Rightarrow X \twoheadrightarrow Y$
6. (*interaction*) $X \twoheadrightarrow Y, XY \to Z \Rightarrow X \to (Z - Y)$

- given rltn schema R & $X \subseteq R$, dependency basis for X wrt set of FDs & MVDs F is partition of $R - X$ to subsets $\{Y_1, \ldots, Y_k\}$ st F logically implies $X \twoheadrightarrow Z$ iff $Z - X$ is union over some subset of $\{Y_1, \ldots, Y_k\}$
- not possible to split RHS of MVDs to single attrs to make minimal covers
- decomp $\{R_1, R_2\}$ of R is lossless-join decomp iff $F \models (R_1 \cap R_2) \twoheadrightarrow R_1 - R_2$ & $F \models (R_1 \cap R_2) \twoheadrightarrow R_2 - R_1$
- schema R is Fourth Normal Form (FNF) wrt set of FDs & MVDs F iff whenever $(X \twoheadrightarrow Y) \in F^+$ & $XY \subseteq R$, then $(X \twoheadrightarrow Y)$ is trivial (i.e. $Y \subseteq X$ or $XY = R$) or X is superkey of R
  - ↳ e.g.

Example: *The* CTB *relation schema can be decomposed to 4NF by appeal to the MVD* $C \twoheadrightarrow T$ *to the following pair of tables:*

Course	Teacher
Math	Smith
Math	Jones
Physics	Black
Advanced Math	Smith

Course	Book
Math	Algebra
Math	Calculus
Physics	Mechanics
Physics	Optics
Advanced Math	Calculus

- **join dependency** on rltn schema R is expressed as $\bowtie[R_1, \ldots, R_k]$
  - ↳ $\{R_1, \ldots, R_k\}$ is decomp of R & holds over instance of R if:
    - $\forall x_i \,.\, R_i(x_i) \leftrightarrow \exists y \,.\, R(x_i, y)$
      - → i.e each R_i is projection of R
    - $\forall x \,.\, R(x) \leftrightarrow R_1(x_1) \wedge \ldots \wedge R_k(x_k)$
      - → i.e. natural join of R_i is R
  - ↳ generalizes MVDs: $X \twoheadrightarrow Y$ is same as $\bowtie[XY, X(R-Y)]$
- **inclusion dependency** on rltn schemata R & S is expressed as $R[X] \subseteq S[Y]$
  - ↳ X & Y are respective subsets of R & S
  - ↳ **projection** of R on X is subset of projection of S on Y
  - ↳ generalizes foreign-key constraints

QUERY EVALUATION

RELATIONAL ALGEBRA

- relnal algebra (RA) consists of set of ops on universe $\mathcal{U}$ of finite rltns over underlying universe of vals D of db instance DB
  - ↳ notation: $(\mathcal{U}; R_0, \ldots, R_k, \times, \sigma, \pi, \cup, -, \text{elim}, c_1, c_2, \ldots)$
  - ↳ constants:
    - R_i is rltn name
    - c_i is constant
  - ↳ unary operators:
    - σ is selection
      - → removes rows
    - π is duplicate preserving projection
      - → removes cols
    - elim is duplicate elimination
  - ↳ binary operators:
    - $\times$ is cross product
    - $\cup$ is multiset union
    - $-$ is multiset diff

given db signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$ & set of constants $\{c_1, c_2, \ldots\}$, RA query is expr E:

$$
\begin{array}{rcl}
E & ::= & R_i \\
& | & c_j \\
& | & \sigma_{\#i=\#j}(E_1) \\
& | & \pi_{\#i_1,\ldots,\#i_m}(E_1) \\
& | & \text{elim } E_1 \\
& | & E_1 \times E_2 \\
& | & E_1 \cup E_2 \\
& | & E_1 - E_2
\end{array}
$$

- ↳ Eval(E, DB) denotes table computed by evaluating RA query E
  - Cnum(E) denotes arity

semantics of Eval(E, DB) are same as range-restricted RC w/ multiset semantics

- ↳ Answers(Q, DB) denotes ans to RC query Q over DB

- assume:
  - ↳ S_i is table w/ arity $n = \text{Cnum}(E_i)$ & extension $\text{Eval}(E_i, DB)$
    - E_i is RA subquery
  - ↳ $\bar{x}$ is short for vars $x_1, \ldots, x_n$
- semantics:
  - ↳ rltn name for $R_i/n \in \rho$ is $\text{Eval}(R_i, DB) = \text{Answers}(\{(\bar{x}) \mid R_i(\bar{x})\}, DB)$
    - $\text{Cnum}(R_i) = n$
  - ↳ constant is $\text{Eval}(c) = \text{Answers}(\{(x) \mid x = c\}, DB)$
    - $\text{Cnum}(R_i) = 1$
  - ↳ selection is $\text{Eval}(\sigma_{\#i=\#j}(E_1), DB) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge (x_i = x_j)\}, DB)$
    - $\text{Cnum}(\sigma_{\#i=\#j}(E_1)) = \text{Cnum}(E_1)$
  - ↳ projection is $\text{Eval}(\pi_{\#i_1,\ldots,\#i_m}(E_1), DB) = \text{Answers}(\{(x_{i_1}, \ldots, x_{i_m}) \mid \exists x_{i_{m+1}}, \ldots, x_{i_n}. S_1(\bar{x})\}, DB)$
    - $\text{Cnum}(\pi_{\#i_1,\ldots,\#i_m}(E_1)) = m$
    - $(i_1, \ldots, i_n)$ is permutation of $(1, \ldots, n)$
  - ↳ duplicate elim is $\text{Eval}(\text{elim } E_1, DB) = \text{Answers}(\{(\bar{x}) \mid \text{elim } S_1(\bar{x})\}, DB)$
    - $\text{Cnum}(\text{elim } E_1) = \text{Cnum}(E_1)$
  - ↳ cross product is $\text{Eval}(E_1 \times E_2, DB) = \text{Answers}(\{(\bar{x}, \bar{y}) \mid S_1(\bar{x}) \wedge S_2(\bar{y})\}, DB)$
    - $\text{Cnum}(E_1 \times E_2) = \text{Cnum}(E_1) + \text{Cnum}(E_2)$
  - ↳ multiset union is $\text{Eval}(E_1 \cup E_2, DB) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \vee S_2(\bar{x})\}, DB)$

- $\text{Cnum}(E_1 \cup E_2) = \text{Cnum}(E_1)$
  - ↳ multiset diff is $\text{Eval}(E_1 - E_2, DB) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge \neg S_2(\bar{x})\}, DB)$
  - $\text{Cnum}(E_1 - E_2) = \text{Cnum}(E_1)$
  - using EXCEPT ALL
  - alt is NOT EXISTS semantics: $\text{Eval}(E_1 - E_2, DB) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge \neg(S_1(\bar{x}) \wedge S_2(\bar{x}))\}, DB)$
    - → $\text{Cnum}(E_1 - E_2) = \text{Cnum}(E_1)$

- e.g.

(signature) $\rho = ($
ACCOUNT / (anum, type, balance, bank, bnum),
BANK / (name, address))

(data) **DB** $= (\mathbb{STR} \uplus \mathbb{Z}, \approx,$

ACCOUNT =

anum	type	balance	bank	bnum	cnt
1234	CHK	$1000	TD	1	1
1235	SAV	$20000	TD	2	1
1236	CHK	$2500	CIBC	1	1
1237	CHK	$2500	Royal	5	1
2000	BUS	$10000	Royal	5	1
2001	BUS	$10000	TD	3	1

,

BANK =

name	address	cnt
TD	TD Centre	1
CIBC	CIBC Tower	1

)

duplicate preserving projection

Example: *The type and balance of all accounts.*

$\text{Eval}(\pi_{\#2,\#3}(\text{ACCOUNT}), \textbf{DB}) =$

type	balance	cnt
CHK	$1000	1
SAV	$20000	1
CHK	$2500	2
BUS	$10000	2

duplicate elimination

Example: *All account types.*

$\text{Eval}(\text{elim } \pi_{\#2}(\text{ACCOUNT}), \textbf{DB}) =$

type	cnt
CHK	1
SAV	1
BUS	1

relation name

Example: *All account information.*

$\text{Eval}(\text{ACCOUNT}, \textbf{DB}) =$

anum	type	balance	bank	bnum	cnt
1234	CHK	$1000	TD	1	1
1235	SAV	$20000	TD	2	1
1236	CHK	$2500	CIBC	1	1
1237	CHK	$2500	Royal	5	1
2000	BUS	$10000	Royal	5	1
2001	BUS	$10000	TD	3	1

constant

Example: *$10000.*

$\text{Eval}(\$10000, \textbf{DB}) =$

	cnt
$10000	1

cross product

Example: *Every pair of accounts and banks.*

$\text{Eval}(\text{ACCOUNT} \times \text{BANK}, \textbf{DB})$

=

anum	type	balance	bank	bnum	name	address	cnt
1234	CHK	$1000	TD	1	TD	TD Centre	1
1234	CHK	$1000	TD	1	CIBC	CIBC Tower	1
1235	SAV	$20000	TD	2	TD	TD Centre	1
1235	SAV	$20000	TD	2	CIBC	CIBC Tower	1
1236	CHK	$2500	CIBC	1	TD	TD Centre	1
1236	CHK	$2500	CIBC	1	CIBC	CIBC Tower	1
1237	CHK	$2500	Royal	5	TD	TD Centre	1
1237	CHK	$2500	Royal	5	CIBC	CIBC Tower	1
2000	BUS	$10000	Royal	5	TD	TD Centre	1
2000	BUS	$10000	Royal	5	CIBC	CIBC Tower	1
2001	BUS	$10000	TD	3	TD	TD Centre	1
2001	BUS	$10000	TD	3	CIBC	CIBC Tower	1

selection

Example: *All account information, including bank addresses.*

$\text{Eval}(\sigma_{\#4=\#6}(\text{ACCOUNT} \times \text{BANK}), \textbf{DB})$

=

anum	type	balance	bank	bnum	name	address	cnt
1234	CHK	$1000	TD	1	TD	TD Centre	1
1235	SAV	$20000	TD	2	TD	TD Centre	1
1236	CHK	$2500	CIBC	1	CIBC	CIBC Tower	1
2001	BUS	$10000	TD	3	TD	TD Centre	1

Example: *All account information for accounts with a $10000 balance.*

$\text{Eval}(\sigma_{\#3=\#6}(\text{ACCOUNT} \times \$10000), \textbf{DB})$

=

anum	type	balance	bank	bnum		cnt
2000	BUS	$10000	Royal	5	$10000	1
2001	BUS	$10000	TD	3	$10000	1

multiset union

Example: *The type and balance of all checking and savings accounts.*

$\text{Eval}(\sigma_{\#1=\#3}(\pi_{\#2,\#3}(\text{ACCOUNT}) \times \text{CHK}) \cup \sigma_{\#1=\#3}(\pi_{\#2,\#3}(\text{ACCOUNT}) \times \text{SAV}), \textbf{DB})$

=

type	balance		cnt
CHK	$1000	CHK	1
SAV	$20000	SAV	1
CHK	$2500	CHK	2

multiset difference

Example: *Banks that do not have addresses.*

$\text{Eval}((\text{elim } \pi_{\#4}(\text{ACCOUNT})) - \pi_{\#1}(\text{BANK}), \textbf{DB}) =$

bank	cnt
Royal	1

- range restricted RC queries w/ multiset semantics have at least expressiveness of RA queries
- thm (Codd) states that for every domain indep RC query, there's equiv RA expr
  - ↳ RA is rltnally complete query lang
- to translate RC queries to RA queries:

$$
\begin{array}{lcl}
\text{RCmap}(R_i(x_1,\ldots,x_k)) & = & R_i \\
\text{RCmap}(\varphi \wedge x_i = x_j) & = & \sigma_{\text{Vmap}(x_i)=\text{Vmap}(x_j)}(\text{RCmap}(\varphi)) \\
\text{RCmap}(x_i = c_j) & = & c_j \\
\text{RCmap}(\exists x_i.\varphi) & = & \pi_{\text{Vmap}(\text{Fv}(\varphi)-\{x_i\})}(\text{RCmap}(\varphi)) \\
\text{RCmap}(\varphi_1 \wedge \varphi_2) & = & \text{RCmap}(\varphi_1) \times \text{RCmap}(\varphi_2) \\
\text{RCmap}(\varphi_1 \vee \varphi_2) & = & \text{RCmap}(\varphi_1) \cup \text{RCmap}(\varphi_2) \\
\text{RCmap}(\varphi_1 \wedge \neg\varphi_2) & = & \text{RCmap}(\varphi_1) - \text{RCmap}(\varphi_2)
\end{array}
$$

↳ $\text{Fv}(Q_1) \cap \text{Fv}(Q_2) = \emptyset$ for $\wedge$? $\vee$ cases
↳ must define Vmap to map vars to col pos
↳ add top-lvl elim ? projection to RA expr

- implementation for RA operator provides cursor OPEN/FETCH/CLOSE interface
  ↳ mostly avoids need to store intermediate results
  ↳ implements cursor interface to prod ans ? uses same one to get ans from children
- must provide at least 1 physical implementation for cursor protocol for each operator
  ↳ e.g.

```
// select_{#i=#j}(Child)
  OPERATOR  child;
  int i,j;

public:
  OPERATOR  selection(OPERATOR c, int i0, int j0)
                { child = c; i = i0; j = j0; };
  void open()   { child.open(); };
  tuple fetch() { tuple t = child.fetch();
                  if (t==NULL || t.attr(i) = t.attr(j))
                       return t;
                  return this.fetch();
                };
  void close()  { child.close(); }
```

  - fully pipelined b/c it requires constant space overhead

other fully pipelined implementations:
↳ constant: 1st fetch returns constant ? next fetch fails
↳ cross product: nested loops algo
↳ duplicate preserving projection: eliminate unwanted attrs from each tuple
↳ multiset union: simple concatenation
↳ multiset diff: nested loops algo that checks for tuple in inner loop
  - only works w/ alt NOT EXISTS semantics
↳ rltn name: file scan of primary index for rltn
↳ duplication elim: remember tuples that have been returned

- to improve efficiency:
  ↳ use concrete (usually disk-based) data structs for efficient searching
  ↳ use better algos to implement operators based on sorting / hashing
  ↳ rewrite RA expr to equiv expr enabling more efficient implementation
- std physical design for rltnal schema defines for each rltn name R:
  ↳ primary index materializing its extension as concrete data struct
    - materialization of rltn adds record identifier (RID) attr to rltn
  ↳ 0+ secondary indices materializing projections of primary index for R as concrete data structs
    - include RID attr of primary index
    - provides alt way to access data based vals in indexed col ? speeds up row retrieval
- index scan op in RA is $\sigma_\varphi(\langle\text{index}\rangle)$
  ↳ φ is cond supplying search vals for underlying data struct
  ↳ $\sigma_\varphi(\langle\text{index}\rangle) = \pi_{\#1,\ldots,\#k}(\sigma_{\#i=\#k+1}(\langle\text{index}\rangle \times c))$
    - $k = \text{Cnum}(\langle\text{index}\rangle)$
    - φ is cond $\#i = c$

e.g

Example: *Assume relation name* `PROF/(pnum,lname,dept)` *has the following physical design:*

1. a Btree primary index on `pnum` called `PROF-PRIMARY` (see next slide), and
2. a Btree secondary index on `lname` called `PROF-SECONDARY` (see slide following next slide).

A visualization of the indices as relations is as follows:

`PROF-PRIMARY`

RID-P	pnum	lname	dept
@1.1	10	Davis	CS
@1.2	14	Smith	C&O
@2.1	17	Taylor	CS
⋮	⋮	⋮	⋮

`PROF-SECONDARY`

RID-S	lname	RID-P
@12.1	Ashton	@5.1
@12.2	Davis	@1.1
@12.3	Dawson	@2.3
⋮	⋮	⋮

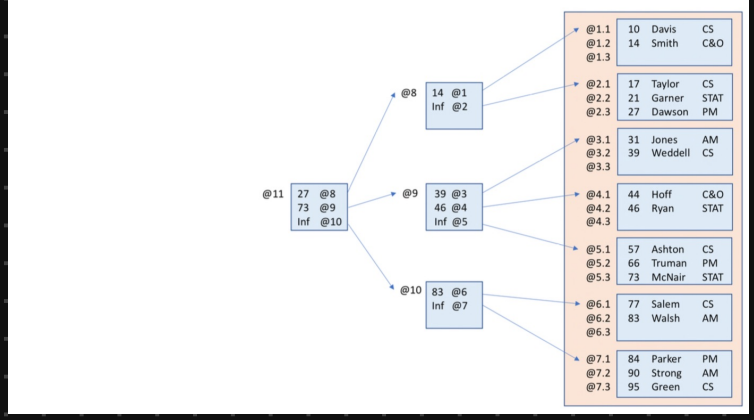
Example (cont'd): *The last name and department of professor number 14:*

$$\pi_{\#3,\#4}(\sigma_{\#2=14}(\texttt{PROF-PRIMARY}))$$

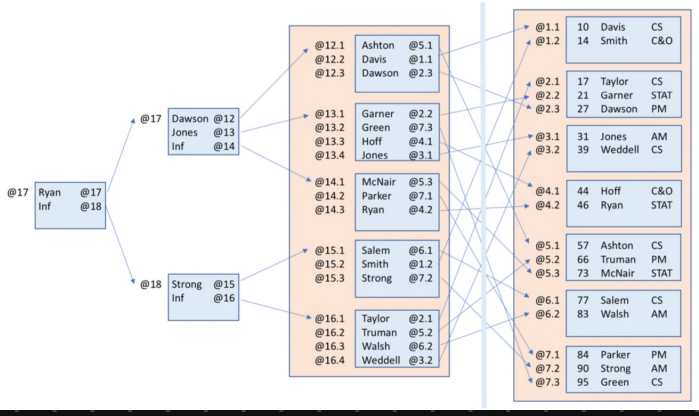
Example (cont'd): *The last names of all professors:*

$$\text{elim } \pi_{\#2}(\texttt{PROF-SECONDARY})$$

Example (cont'd): *Btree index* `PROF-PRIMARY` *on* `pnum`*:*

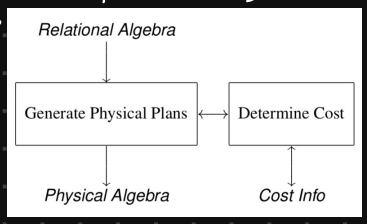


Example (cont'd): *Btree index* `PROF-SECONDARY` *on* `lname`*:*



QUERY OPTIMIZATION

- generate all physical plans equiv to query
  - ↳ choose plan having lowest cost
  - ↳



- in practice:
  - ↳ consider only plans of certain form (e.g. restrictions on search space)
  - ↳ focus on eliminated rlly bad query plans

  to determine best plan, estimate cost based on statiscal metadata collected by DBMS on db instances
- assume:
  - ↳ uniformity: all possible vals of attr are equally likely to appear in rltn
  - ↳ independence: likelihood that an attr has particular val in tuple don't depend on vals of other attrs

COST ESTIMATION

- for materialized rltn R w/ attr A, we keep:
  - ↳ |R| as cardinality of R
    - i.e. # tuples in R
  - ↳ b(R) as blocking factor for index R (i.e. # entries that can fit in single block)
  - ↳ min(R, A) as min val for A in R
  - ↳ max(R, A) as max val for A in R
  - ↳ distinct(R, A) as # distinct vals in R

e.g.

Example: *Consider the following case:*

- *R* has the signature:

  MARK/(studnum, course, assignnum, mark).

- *E* is the query:

  $$\pi_{\#1,\#4}(\sigma_{\#2=\#7}(\sigma_{\#1=\#6}(\sigma_{\#4>\#5}(\text{MARK} \times 90) \times 100) \times \text{PHYS}))$$

- The query is obtained by a translation of the following SQL query.

```
select studnum, mark from MARK
where course = 'PHYS' and studnum = 100 and mark > 90
```

Note: the result of the query can be a multiset.

- The physical design for MARK:
  1. a Btree primary index on course:

     MARK-PINDEX/(RID-P, studnum, course, assignnum, mark)
  2. a Btree secondary index on studnum:

     MARK-SINDEX/(RID-S, studnum, RID-P)

- The following statistical metadata:
  1. $|\text{MARK}| = 10000$
  2. $b(\text{MARK-PINDEX}) = 50$
  3. $\text{distinct}(\text{MARK}, \text{studnum}) = 500$ (number of different students)
  4. $\text{distinct}(\text{MARK}, \text{course}) = 100$ (number of different courses)
  5. $\text{distinct}(\text{MARK}, \text{mark}) = 100$ (number of different marks)

↳ compare costs for 3 diff strategies

1) primary index

Query plan:

$$\pi_{\#2,\#5}(\sigma_{\#2=\#7}(\sigma_{\#5>\#6}(\sigma_{\#3='\text{PHYS}'}(\text{MARK-PINDEX}) \times 90) \times 100))$$

Cost in number of block reads:

- Assuming a uniform distribution of tuples over the courses, there will be about |MARK|/100 = 100 tuples with course = PHYS.
- Searching the MARK-PINDEX Btree has a cost of 2, and retrieving the 100 matching tuples adds a cost of $100/b(\text{MARK-PINDEX})$ data blocks.†
- The total cost is therefore 4 block reads.

† selection of N tuples from rltn R using clustered primary index has cost of $2 + \frac{N}{b(R)}$

2) secondary index

Query plan:

$$\pi_{\#5,\#8}(\sigma_{\#6=\#10}(\sigma_{\#8>\#9}((\sigma_{\#2=100}(\text{MARK-SINDEX}) \times \sigma_{\#1=\#3\ell}(\text{MARK-PINDEX})) \times 90) \times \text{PHYS}))$$

- NOTE: Part in red expresses a *nested index join*: "#3ℓ" refers to column 3 of the ℓeft argument of the nested cross product operator "×".

Cost in number of block reads:

- Assuming a uniform distribution of tuples over student numbers, there will be about |MARK|/500 = 20 tuples for each student.
- Searching the MARK-SINDEX Btree has a cost of 2. Since this is not a clustered index, we will make the pessimistic assumption that each matching record in the MARK-PINDEX Btree is on a separate data block, i.e., 20 blocks will need to be read.†
- The total cost is therefore 22 block reads.

↳ selection of N tuples from r|tn R using unclustered secondary index has cost 2+N

3) scan primary index

Query plan:

$$\pi_{\#2,\#5}(\sigma_{\#2=\#8}(\sigma_{\#5>\#7}(\sigma_{\#3=\#6}(\texttt{MARK-PINDEX} \times \text{PHYS}) \times 90) \times 100))$$

Cost in number of block reads:

- There are 10,000/50 = 200 MARK-PINDEX Btree data pages.†
- The total cost is therefore 200 block reads.

↳ selection of N tuples from r|tn R by exhaustive scan of its primary index has cost $\frac{|R|}{b(R)}$

- costs of physical ops in block reads & writes:

- selection

$$\text{cost}(\sigma_\varphi(E)) = (1 + \epsilon_\varphi)\,\text{cost}(E).$$

- nested loop join (R is the outer relation):

$$\text{cost}(R \times S) = \text{cost}(R) + (|R|/b)\,\text{cost}(S)$$

- nested index join (R is the outer relation, S is the inner relation, and Btree has depth d_S):

$$\text{cost}(R \times \sigma_\varphi(S)) = \text{cost}(R) + d_S|R|$$

- sort-merge join:

$$\text{cost}(R \bowtie_\varphi S) = \text{cost}(\text{sort}(R)) + \text{cost}(\text{sort}(S))$$

where

$$\text{cost}(\text{sort}(E)) = \text{cost}(E) + (|E|/b)\log(|E|/b).$$

- cost estimation requires estimate of size of results of ops
  - ↳ use selectivity estimates for cond $\sigma_\varphi(R)$
    - $\text{sel}(\sigma_\varphi(R)) = \frac{|\sigma_\varphi(R)|}{|R|}$
    - optimizer estimates selectivity using rules:

$$\text{sel}(\sigma_{A=c}(R)) \approx \frac{1}{\text{distinct}(R, A)}$$

$$\text{sel}(\sigma_{A\le c}(R)) \approx \frac{c - \min(R, A)}{\max(R, A) - \min(R, A)}$$

$$\text{sel}(\sigma_{A\ge c}(R)) \approx \frac{\max(R, A) - c}{\max(R, A) - \min(R, A)}$$

- size estimation for joins:
  - ↳ general join where φ is equality on col w/attr name A of R & col w/attr name B of S is $|R \bowtie_\varphi S| = |R| \frac{|S|}{\text{distinct}(S,B)} = |S| \frac{|R|}{\text{distinct}(R,A)}$
  - ↳ foreign key join is $|R \bowtie_\varphi S| = |R| \frac{|S|}{|S|} = |R|$
- transformations that are always good:
  - ↳ push selections: $\sigma_\varphi(E_1 \bowtie_\theta E_2) = \sigma_\varphi(E_1) \bowtie_\theta (E_2)$
    - φ involves cols of E_1 only
  - ↳ push projections: $\pi_V(R \bowtie_\theta S) = \pi_V(\pi_{V_1}(R) \bowtie_\theta \pi_{V_2}(S))$
    - V_1 is set of all cols of R involved in θ & V
    - same for V_2
  - ↳ replace products by joins: $\sigma_\varphi(R \times S) = R \bowtie_\varphi S$
- e.g.

- Assume the following.
  1. There are $|S| = 1000$ students,
  2. enrolled in $|C| = 500$ classes.
  3. The enrollment table is $|E| = 5000$,
  4. and, on average, each student is registered for five courses.
- Then:

$$\text{cost}(\sigma_{\text{name}='\text{Smith}'}(S \bowtie (E \bowtie C)))$$

greatly exceeds

$$\text{cost}(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C)).$$

- joins are associative so $R \bowtie S \bowtie T \bowtie U$ can be expressed as:
  - $((R \bowtie S) \bowtie T) \bowtie U$
  - $(R \bowtie S) \bowtie (T \bowtie U)$
  - $R \bowtie (S \bowtie (T \bowtie U))$
- e.g.

1. $\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C)$

   This order evaluates $E \bowtie C$, which has one tuple for each course registration (by any student) $\sim$ 5000 tuples.

2. $(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie E) \bowtie C$

   This join order produces an intermediate relation which has one tuple for each course registration by a student named Smith.

   If there are only a few Smith's among the 1,000 students (say there are 10), this relation will contain about 50 tuples.

TRANSACTION AND RECOVERY MANAGEMENT

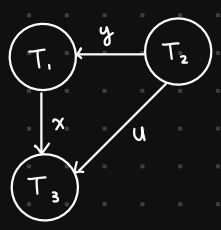
CONCURRENCY CONTROL

- **transaction** is seq of indivisible DML requests
  - ↳ apps access db via transactions
  - ↳ has **ACID** properties:
    - atomicity: transaction occurs entirely or not at all
    - consistency: each transaction preserves consistency of db
    - isolation: concurrent transactions don't interfere w/ each other
    - durability: once completed, transaction's changes are permanent
- db is finite set of indep physical objs x_j that are read / written by transactions T_i
  - ↳ $r_i[x_j]$ means T_i reads obj x_j
  - ↳ $w_i[x_j]$ means T_i reads obj x_j
- **transaction T_i** is seq of read & write ops on db, end w/ commit request of T_i
  - ↳ e.g. $T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$
- **schedule** S is execution order of ops in set of transactions $\{T_1, \ldots, T_k\}$
  - ↳ every op $o_i \in T_i$ also appears in S
  - ↳ T_i's ops in S are ordered same as in T_i
  - ↳ goal is to prod schedule w/ **maximal parallelism**
- schedule is **serializable** if it's equiv to some other serial execution of same transactions
  - ↳ e.g.

> ▶ An interleaved execution of two transactions:
> $$S_a = w_1[x], r_2[x], w_1[y], c_1, r_2[y], c_2$$
> ▶ An equivalent serial execution (T_1 , T_2):
> $$S_b = w_1[x], w_1[y], c_1, r_2[x], r_2[y], c_2$$
> ▶ An interleaved execution with no equivalent serial execution:
> $$S_c = w_1[x], r_2[x], r_2[y], c_2, w_1[y], c_1$$

- 2 ops **conflict** if they belong to diff transactions & mutate same data item x
  - ↳ read - write
  - ↳ write - write
- 2 schedules are **conflict equiv** when all conflictings ops are ordered same way
- S_1 is **conflict serializable** if it's conflict equiv to another serial schedule S_2
  - ↳ to determine, make **serialization graph SG(N, E)**:
    - nodes $T_i \in N$ correspond to transactions
    - directed edges $T_i \to T_j \in E$ whenever op $o_i[x]$ occurs before op $o_j[x]$ & they're conflicting ops
  - ↳ schedule is serializable iff SG is **acyclic**
    - e.g.

$S_4 = w_1[x], w_2[u], w_2[y], w_1[y], w_3[x], w_3[u], w_1[z]$



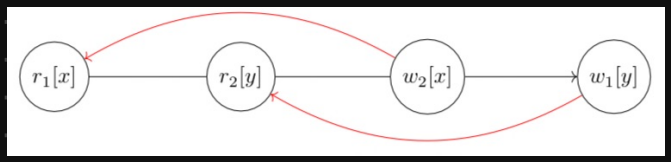
T_1	T_2	T_3
w[x]		
	w[u]	
	w[y]	
w[y]		
		w[x]
		w[u]
w[z]		

Since S_4 serialization graph is acyclic, it's conflict serializable. An equivalent serial schedule would be $S_4' = w_2[u], w_2[y], w_1[x], w_1[y], w_1[z], w_3[x], w_2[u]$ or just $T_2 \rightarrow T_1 \rightarrow T_3$.

- serializability guarantees correctness but there's other undesirable situations
  - ↳ **recoverable schedules (RC)**: T_j reads val T_i has written, T_j commits, then T_i attempts to abort
    - must undo effects of committed T_j
    - soln is to allow commit in order of read-from dependencies
  - ↳ **cascadeless schedules (ACA)**: T_j reads val T_i has written, then T_i attempts to abort
    - must also abort T_j, which may lead to cascade of further aborts
    - soln is to disallow transactions to read uncommitted objs

TWO PHASE LOCKING

- DBMS **scheduler** coordinates execution of scheduled ops by:
  - ↳ executing it
  - ↳ delaying it
  - ↳ rejecting it (i.e. aborting)
  - ↳ ignoring it
- 2 kinds of schedulers:
  - ↳ **conservative** ones favour delaying
  - ↳ **aggressive** ones favour rejection
- conservative schedulers are **lock based**, meaning transactions are required to have lock on objs they want to access
  - ↳ **shared** lock to read
  - ↳ **exclusive** lock to write
- **two phase locking protocol (2PL)** is when transaction has to acquire all locks before being allowed to release them
  - ↳ still allows ops to proceed to cascading aborts ? deadlocks
  - ↳ use **strict 2PL**, meaning locks are held until a commit / abort op
    - ensures ACA, but can still result in deadlock
- **deadlock** is when 2+ transactions are unable to proceed b/c they're waiting for each other to release locks
  - ↳ e.g. transaction seq is (T_2, T_1): $r_1[x], r_2[y], w_2[x], w_1[y]$



  - ↳ to solve:
    - **prevention**: locks granted only if they don't lead to deadlock
      - → locks granted in order of objs
    - **detection**: use wait for graphs ? cycle detection
      - → system aborts one of offending transactions (i.e. **involuntary abort**)
- neither 2PL nor strict 2PL handles cases when creation / deletion of physical objs is required
  - ↳ e.g.

> Example: *Consider the following pair of transactions:*
> 1. *Based on a sufficient fixed budget, transaction T_1 gives each employee a bonus.*
> 2. *As a consequence of a new hire, transaction T_2 adds a new employee (who should not receive the bonus).*

    - **phatom tuple problem**
  - ↳ soln is that DML requests that modify db need larger scale locks (e.g. locks on entire indices, predicate locks on tables) w/ lock compatibility rules
- SQL std addresses **4 isolation lvls** for transaction:
  - ↳ lvl 3 (**serializability**): realized by strict 2PL w/ table lvl locks

↳ lvl 2 (repeatable read): realized by strict 2PL w/ tuple lvl locks
- phantom tuples may occur

↳ lvl 1 (cursor stability): realized by strict 2PL w/ tuple lvl exclusive locks
- same query executed 2x in transaction can prod diff ans

↳ lvl 0: realized by no locking
- transaction queries may read uncommitted updates

- transaction manager is responsible for consistency & isolation, which is based on serializability

FAILURE RECOVERY

- recovery manager is responsible for durability and atomicity
  - ↳ input is 2PL & ACA schedule of ops prod by transaction manager
  - ↳ output is schedule of obj reads, writes, & forced writes

2 essential approaches to recovery:

↳ shadowing means to copy-on-write & merge-on-commit
- poor clustering
- not used in modern systems

↳ logging means using log files on separate stable media
- good use of buffers
- preserves og clusters

- log is read/append-only data struct
  - ↳ recovery manager appends transaction log records
  - ↳ aborting & failure recovery reads transaction log records
- log records record diff types of info:
  - ↳ UNDO: old versions of objs that have been modified by transaction
    - used to undo db changes made by transaction that aborts
  - ↳ REDO: new versions of objs that have been modified by transaction
    - used to redo work done by transaction that commits
  - ↳ BEGIN
  - ↳ COMMIT
  - ↳ ABORT
- e.g.

Five transactions: $\{T_0, T_1, T_2, T_3, T_4\}$

Transaction status:
$\{T_1, T_3\}$ committed,
$\{T_2\}$ aborted, and
$\{T_0, T_4\}$ active.

log head → (oldest part)
T_0, BEGIN
T_0, UNDO, x_1, 99
T_0, REDO, x_1, 100
T_1, BEGIN
T_1, UNDO, x_2, 199
T_1, REDO, x_2, 200
T_2, BEGIN
T_2, UNDO, x_3, 51
T_2, REDO, x_3, 50
T_1, UNDO, x_4, 1000
T_1, REDO, x_4, 10
T_1, COMMIT
T_3, BEGIN
T_2, ABORT
T_3, UNDO, x_2, 200
T_3, REDO, x_2, 50
T_4, BEGIN
T_4, UNDO, x_4, 10
(newest part)
T_4, REDO, x_4, 100
log tail →
T_3, COMMIT

- to recover from system failure:
  - ↳ do all UNDOs in reverse order for aborted and active transactions
    - this aborts all these transactions
  - ↳ do all REDOs in order for committed transactions to reapply the effects of them on the db
- write-ahead logging (WAL) protocol avoids allowing indices on stable store to be updated prior to LOG b/c it leads to inconsistency
  - ↳ undo rule: log record for update is appended to LOG file before data pg is written to stable store
    - guarantees atomicity
  - ↳ redo rule: all log records for transaction are appended to LOG file before acknowledging commit
    - guarantees durability

DATABASE TUNING

WORKLOAD MODELLING

- db tuning entails adjusting db params & physical db design to address performance issues w/ transaction response time & thruput
- workload description contains:
  - ↳ critical queries & their freq
    - rltns accessed
    - attrs received
    - attrs that occur in selection/join conditions & how selective conds are
  - ↳ critical updates & their freq
    - type of update
    - rltns/attrs affected
    - attrs that occur in selection/join conditions & how selective conds are
  - ↳ desired performance goal for each query/update

PHYSICAL DATABASE DESIGN

- materialized view is precomputed & stored result (i.e. physical table) of complex db query to provide faster access to data
  - ↳ e.g. primary & secondary views
  - ↳ syntax:

    CREATE VIEW <view-name> [AS] (<query>)
    MATERIALIZED [AS <data-struct-choice>]
  - ↳ e.g

Example (from Module 10): *Relation name*

`PROF/(pnum, lname, dept)`

with the standard physical design consisting of:

1. the Btree primary index on `pnum` called `PROF-PRIMARY`

```
create view PROF-PRIMARY as (select * from PROF)
materialized as BTREE with search key (pnum)
```

2. and a Btree secondary index on `lname` called `PROF-SECONDARY`

```
create view PROF-SECONDARY
as (select lname, rid from PROF-PRIMARY)
materialized as BTREE with search key (lname)
```

- 2 indices are co-clustered if data pgs for indices are in commom (i.e. records encoding tuples in indices are interleaved in same data pgs)
  - ↳ useful when (1,N) rltnships exist btwn records for 2 indices
  - ↳ leverages concept of data locality
  - ↳ e.g.

Example: *If a foreign key exists from an* `EMPLOYEE` *table to a* `DEPARTMENT` *table, each record for the primary index of* EMPLOYEE *might be co-clustered with its related* DEPARTMENT *record in the data pages of the* DEPARTMENT *primary index. Records for indices on other tables, e.g.,* `PROJECTS` *and* `JOBS` *can in turn be respectively co-clustered with* `DEPARTMENT` *and* `EMPLOYEE`.

  - ↳ speeds up joins
  - ↳ sequential scans of either rltn are slower

ordered indices allow more general subqueries to be evaluated efficiently

- ↳ e.g. range queries

B-trees can also help evaluating a query with the form

```
select * from <table>
where <attribute> >= <constant>
```

- ↳ e.g. multi-attr search keys
  - create index on several attrs of same rltn

Example: *Assume relation name*

`PROF/(pnum,lname,fname,dept)`

with a secondary index now defined as follows:

```
create view PROF-SECONDARY
as (select lname, fname, rid from PROF-PRIMARY)
materialized as BTREE with search key (lname, fname)
```

- organized first by lname then by fname (if tuples have same fname)
- ▶ *The* `PROF-SECONDARY` *index would be useful for these queries*:

```
select *                          select *
from PROF                         from PROF
where lname = 'Smith'             where lname = 'Smith'
                                  and fname = 'John'
```

▶ *It would be very useful for these queries*:

```
select fname                      select fname, lname
from PROF                         from PROF
where lname = 'Smith'
```

▶ *It is unlikely to be useful for this query*:

```
select *
from PROF
where fname = 'John'
```

- in std physical design, secondary index is materialized view on primary index
- **join indices**: materialized view can be defined on conjunctive query
  - ↳ e.g

Example: *A join index for the bibliography database*:

```
create view AUTHOR-BOOK as (
    select author-rid, book-rid
    form AUTHOR-PRIMARY, WROTE-PRIMARY, BOOK-PRIMARY
    where aid = author and publication = pubid )
materialized as heap file
```

given query Q & physical db design consisting of set of arbitrary materialized views $\{V_1, \ldots, V_n\}$, **view-based query rewriting** problem is finding most efficient query plan P equiv to Q that only uses views in V

↳ thm: undecidable, even when Q & each V_i is conjunctive query & any P equiv to Q is acceptable

- use **db2expn** & **dynexpn** in DB2 to investigate what plan is chosen for query & its estimated cost
  - ↳ e.g.

Example: *Invoking these tools on the bibliography query*

```
select name from author, wrote where aid = author
```

generates the following:

(default physical design)

```
Estimated Cost        = 50
Estimated Cardinality = 120

Optimizer Plan:

     RETURN
       |
     MSJOIN
     /    \
 TBSCAN   TBSCAN
   |        |
  SORT     SORT
   |        |
 TBSCAN   TBSCAN
   |        |
 Table:   Table:
 AUTHOR   WROTE
```

(secondary index `AW` on `author` attribute of `WROTE`)

```
Estimated Cost        = 25
Estimated Cardinality = 120

Optimizer Plan:

       RETURN
         |
       MSJOIN
       /    \
 TBSCAN      IXSCAN
   |        /     \
  SORT   Index:  Table:
   |       AW     WROTE
 TBSCAN
   |
 Table:
 AUTHOR
```

- **guidelines** for physical design
  - ↳ don't index unless performance inc outweighs update overhead
  - ↳ attrs in WHERE clauses are candidates for index search keys
  - ↳ multi-attr search keys used when:
    - WHERE has several conds
    - enables index-only plans
  - ↳ chooses indices that benefit as many queries as possible

- ↳ each rltn can have at most 1 primary index
  - range queries benefit most from clustering
  - join queries benefit most from co-clustering

LOGICAL SCHEMA AND QUERY TUNING

- after physical tuning, might still be necessary to make changes to conceptual db designs
- **renormalization**: alt BCNF decomps that better fit queries in workload
  - ↳ speeds up simple updates & queries
  - ↳ slows dow complex updates & queries
- **denormalization**: merge rltnal schemata to inc redundancy
  - ↳ inc update overhead
  - ↳ dec query overhead
- **partitioning** table means to split it into multiple ones to reduce I/O cost & lock contention
  - ↳ **horizontal**: each partition has all og cols & subset of og rows
    - often used to separate operational from archival data
  - ↳ **vertical**: each partition has all og rows & subset of og cols
    - often used to separate frequently used cols from infrequently used ones
- to **tune queries**:
  - ↳ avoid unnecessary ORDER BY, DISTINCT, & GROUP BY clauses
    - sorting is expensive
  - ↳ replace subqueries w/joins
  - ↳ replace correlated subqueries w/uncorrelated ones
- to **tune apps**:
  - ↳ minimize construction costs
    - return fewest cols & rols necessary
    - update multiple rows w/WHERE clause instead of cursor
  - ↳ minimize lock contention & **hot-spots**
    - delay updates & ops on hot-spots as long as possible
    - shorten/split transactions
    - perform insertions/deletions/updates in batches
    - lower isolation lvls