RA

Some key basic definitions:

- relation
- attribute
- schema
- tuple
- domain? `Movies(title:string, year:integer, length:integer, genre:string)`
- key, primary key
- data-definition language (ddl) & data-manipulation language (dml)
- tables, views and temporary tables

data types: `CHAR(n), VARCHAR(n)` (e.g. `foo` is valid for `CHAR(5)`), `BOOLEAN`, `INT`, `FLOAT`, `DATE` and `TIME`

create a relation

```
CREATE TABLE Movies (
  title       CHAR(100),
  year        INT,
  length      INT,
  genre       CHAR(10),
  studioName  CHAR(10),
  producerC#  INT
);
```

delete a relation R

```
DROP TABLE R;
```

modify the schema of existing relation

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

or

```
ALTER TABLE MovieStar DROP birthdate;
```

default values (right after the data type when declared)

```sql
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

declaring keys, `PRIMARY KEY` or `UNIQUE`, can be declared either after the data type or at the end of `CREATE TABLE`

some more about operations of RA: set operations (union, intersection, difference), selection $\sigma_{Condition}(R)$ (eliminate rows/tuples), projection $\pi_{Attr1,Attr2,...}(R)$ (eliminate columns/attributes), Cartesian product/various joins (combine/(selectively pair) tuples), renaming

Cartesian product $R \times S$, natural joins $R \bowtie S$, Theta-joins $R \bowtie_C S$ (which is equal to product, then selection, note that it's Cartesian product not natural product)

Renaming R as S: $\rho_{S(A_1,A_2,...,A_n)}(R)$

also, left join, right join

constraints on relations (2 types):

- if R is an expression of relational algebra, the $R = \emptyset$ is a constraint.
- if R and S..., $R \subseteq S$

aggregation operators, `SUM, AVG, MIN, MAX, COUNT`

SQL

The simple example `S(ELECT)-F(ROM)-W(HERE)`

```sql
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

projection in SQL (with renaming and even calculation)

```sql
SELECT title AS name, length*0.5 AS halfDuration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

selection in SQL

- 6ix comparison operators: `=, <>, <, >, <=, >=`

```sql
SELECT title
FROM Movies
WHERE (year > 1970 OR length <90) AND studioName = 'MGM';
```

comparison of strings: `'bar' < 'bargain'`,`'fodder' < 'foo'`
pattern matching:

- `WHERE title LIKE 'Star ____';`
- `WHERE title LIKE '%''s$';`

`UNKNOWN, TRUE` and `FALSE`: treat `UNKNOWN` as `0.5`.
`x AND y` := min(x, y)
`x OR y` := max(x, y)
negation of `x` := 1-x

order by (`DESC` means descending, by default is ascending)

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title DESC;
```

products and joins in SQL

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
---with disambiguity---
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

tuple variables

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
    AND Star1.name < Star2.name;
```

union, intersection and difference of queries

```
(SELECT name, address
 FROM MovieStar
 WHERE gender = 'F')
    INTERSECT
(SELECT name, address
 FROM MovieExec
 WHERE netWorth > 100000000);
---- can switch INTERSECT to UNION/EXCEPT ----
```

subqueries

```sql
--- produce scalr values ---
SELECT name
FROM MovieExec
WHERE cert# =
      (SELECT producerC#
       FROM Movies
       WHERE title = 'Star Wars'
      );

--- conditions involving relations ---
  -- EXISTS R is a condition that is true iff R is not empty
  -- s IN R is true iff s is equal to one of the values in R
  -- s > ALL R is true iff greater than every value in unary relation R
  -- s > ANY R is true iff greater than at least one value in unary relation R

--- conditions involving tuples ---
--- find all Movie executors of Harrison Ford's movies
SELECT name
FROM MovieExec
WHERE cert# IN
    (SELECT producerC#
     FROM Movies
     WHERE (title, year) IN
        (SELECT movieTitle, movieYear
         FROM StarsIn
         WHERE starName = 'Harrison Ford'
        )
    );
--- a simpler version without nested subqueries ---
SELECT name
FROM MovieExec, Movies, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';

--- correlated subqueries ---
--- find the titles that have been used for two or more movies ---
SELECT title
FROM Movies Old
WHERE year < ANY
    (SELECT year
     FROM Movies
     WHERE title = Old.title
    );

--- subqueries in `FROM` clauses ---
--- same Harrison Ford's problem
SELECT name
FROM MovieExec, (SELECT producerC#
                 FROM Movies, StarsIn
                 WHERE title = movieTitle AND
                       year = movieYear AND
                       starName = 'Harrison Ford'
                ) Prod
```

```
WHERE cert# = Prod.producerC#;

--- SQL join expressions ---
--- natural joins ---
--- outerjoins ---
```

full-relation opertaions

```
-- eliminating duplicates
SELECT DISTINCT name

-- duplicates in unions, intersections, and differences
-- they (set operations) automatically eliminate duplicates;
-- unlike SELECT (which preserves duplicates, that's why we need DISTINCT)
-- if we want to keep duplicates, use `ALL` after these set operations
(SELECT title, year FROM Movies)
  UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);

R INTERSECT ALL S
R EXCEPT ALL S

-- aggregation operators (SUM, AVG, MIN, MAX, COUNT)
SELECT COUNT(DISTINCT starName)
FROM StarsIn;

-- grouping
  -
- whatever aggregation operators are used in the SELECT clause are applied only within grou
ps
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;

-- `NULL` doesn't contribute to a sum; COUNT(*) is NULL-inclusive

-- HAVING
-
- sometimes we want to choose our groups based on some aggregate property of the group itse
lf, then we follow the GROUP BY clause with a HAVING clause. HAVING is followed by a condit
ion about the group.
  -- an aggregation in a HAVING clause applies only to the tuples of the group being tested
  -
- any attribute of relations in the FROM clause may be aggregated in the HAVING clause, but
 only those attributes that are in the GROUP BY list may appear unaggregated in the HAVING
clause (same rule for SELECT)
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

database modifications

```sql
-- insert tuples into a relation: INSERT INTO R(A1,...An) VALUES (v1,...vn);
INSERT INTO StarsIn(movieTitle, movieYear, starName) -
- we can even skip the attributes here
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');

-
- another example: add to the relation Studio(name, address, presC#) all movie studios that are mentioned in the relation Movies(title, year, length, genre, studioName, producerC#) without naming address and presC# (just use NULL)
INSERT INTO Studio(name)
    SELECT DISTINCT studioName
    FROM Movies
    WHERE studioName NOT IN
        (SELECT name
        FROM Studio);

-- delete certain tuples from a relation: DELETE FROM R WHERE <condition>;
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
    movieYear = 1942 AND
    starName = 'Sydney Greenstreet';

-- update values of certain components of certain existing tuples: UPDATE R SET <new-value assignment> WHERE <condition>;
UPDATE MovieExec
SET name = 'Pres. ' || name
WHERE cert# IN (SELECT presC# FROM Studio);
```

keys and foreign keys

```sql
-- declaring foreign-key constraints: REFERENCES <table>(<attribute>) or "start a new line"
-
- the referenced attribute(s) of the other relation must be declared UNIQUE or the PRIMARY KEY
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MoiveExec(cert#)
        ON DELETE SET NULL -- set-NULL policy
        ON UPDATE CASCADE -- cascade policy
);
-- or
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
        ON DELETE SET NULL -- set-NULL policy
        ON UPDATE CASCADE -- cascade policy
);
```

constraints on attributes and tuples

```sql
-- not-NULL constraints (attribute-based)
presC# INT REFERENCES MoiveExec(cert#) NOT NULL

-- attribute-based CHECK constraints
gender CHAR(1) CHECK (gender IN ('F', 'M')),
presC# INT REFERENCES MoiveExec(cert#)
        CHECK (presC# >= 1000000)
presC# INT CHECK (presC# IN (SELECT cert# FROM MovieExec))

-- tuple-based CHECK constraints
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1)
    birthdate DATE,
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.$')
);

-- add and drop constraint separately
ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;
ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey PRIMARY KEY (name);
ALTER TABLE MovieStar ADD CONSTRAINT NoAndro CHECK (gender IN ('F', 'M'));
```

virtual views

```sql
-- declare views: CREATE VIEW <view-name> AS <view-definition>;
CREATE VIEW ParamountMovies AS
    SELECT title, year
    FROM Movies
    WHERE studioName = 'Paramount';

CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;

-- views can be used together with other regular tables:
SELECT DISTINCT starName
FROM ParamountMovies, StarsIn
WHERE title = movieTitle AND year = movieYear;

-- renaming attributes
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

modifying views

```sql
DROP VIEW ParamountMovies -- only affect this view
DROP TABLE Movies -- affect not only Movies, also make the previous view unusable
```

```sql

CRETE VIEW ParamountMovies AS
    SELECT studioName, title, year
    FROM Movies
    WHERE studioName = 'Paramount';

-- add

INSERT INTO ParamountMovies
VALUES('Paramount', 'Star Trek', 1979);
-- is equivalent to
INSERT INTO Movies(title, year)
VALUES('Paramount', 'Star Trek', 1979);

-- delete

DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
-- is equivalent to
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';

-- udpate

UPDATE ParamountMovies
SET year = 1979
WHERE title = 'STar Trek the Movie';
-- is equivalent to
UPDATE Movies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND studioName = 'Paramount';
```

XML

- XML (Extensible Markup Language), tag-based
- two different modes
    - well-formed XML: invent your own tags
    - valid XML involves a 'DTD' (Document Type Definition) that specifies the allowable tags and gives a grammar for how they may be nested

```xml
<!-- well-formed XML would have an outer structure like: -->
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<SomeTag>
  ...
</SomeTag>

<!-- attributes
name-value pairs, an alternative way to represent a leaf node of semistructured data
<!-->
<Movie year = 1977><Title>Star Wars</Title></Movie>
<!-- is equivalent to say -->
<Movie title = "Star Wars" year = 1977></Movie>
<!-- or -->
```

```
<Movie title = "Star Wars" year = 1977 />
```

Document Type Definitions

```
<!DOCTYPE root-tag [
    <!ELEMENT element-name (components)>
        more elements
]>
```

`(#PCDATA)` ("parsed character data") means that the element has a value that is text, and it has no elements nested within.
`EMPTY` with no parentheses, indicates that the element is one of those that has no matched closing tag. It has no subelements, nor does it have text as a value.

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movies*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

*: any number of times, including zero
+: one or more times
?: one or zero time
|: "or", `<!ELEMENT Genre (Comedy|Drama|SciFi|Teen)>`

using a DTD

```
<? xml verion = "1.0" encoding = "utf-8" standalone = "no"?>
<!DOCTYPE Stars SYSTEM "star.dtd">
```

attribute lists
`<!ATTLIST element-name attribute-name type>`
The most common type for attributes is CDATA (character-string data with special characters like `<` escaped as in #PCDATA)
data type can be either #REQUIRED or #IMPLIED

```
<!DOCTYPE StarMovieData [
    <!ELEMENT StarMovieData (Star*, Movie*)>
    <!ELEMENT Star (Name, Address+)>
        <!ATTLIST Star
            starId ID #REQUIRED
```

```
        starredIn INDREFS #IMPLIED
    >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (Street, City)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT Movie (Title, Year)>
    <!ATTLIST Movie
        movieId ID #REQUIRED
        starsOf IDREFS #IMPLIED
    >
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
]>
```

- XPath
  - document nodes `doc("movies.xml")`
  - path expression `/StarMovieData/Star/Name`
  - relative path expression
  - attributes `/StarMovieData/Star/@starID`
  - find all city subelements `//City`
  - wildcards `/StarMovieData/*/@*`
  - conditions in path expressions `/StarMovieData/Star[//City = "Malibu"]/name`, can also use `=, >=, !=, and, or` etc.
    - an integer `[i]` by itself is true only when applied the ith child of its parents
    - a tag `[T]` by itself is true only for elements that have one or more subelements with tag T.
    - an attribute `[A]` by itself is true only for elements that have a value for the attribute A.
- XQuery
  - functional language
  - FLWR expressions (for, let, where, return)
    - zero or more for- and let-clauses in any order
    - optional where-clause
    - exactly one return-clause
    - let-clause
      - `let variable := expression`
      - `let $stars := doc("stars.xml")`
    - for-clause
      - `for variable in expression`
      - `let $moive := doc("movies.xml")`
      - `for $m in $movies/Movies/Movie ... something done with each Movie element`
    - where-clause
      - `where condition`
      - is applied to an item and the condition is an expression evaluates to true or false
    - return-clause
      - `return expression`

replacement of variables by their values

```
let $movies := doc("movies.xml")
```

```
for $m in $movies/Movies/Movie
return <Movie title = {$m/@title}>{$m/Version/Star}</Movie>

let $starSeq := (
  let $movies := doc("movies.xml")
  for $m in $movies/Movies/Movie
  return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

joins in XQuery

- use built-in function `data(E)` to extract the value of an element E.

```
let $movies := doc("movies.xml"),
    $stars := doc("stars.xml)
for $s1 in $movies/Movies/Movie/Version/Star,
    $s2 in $stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

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

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street eq "123 Maple St." and
      $s/Address/City eq "Malibu"
return $s/Name
```

elimination of duplicates (apply `distinct-values` function)

```
let $starSeq := distinct-values(
  let $movies := doc("movies.xml")
  for $m in $movies/Movies/Movie
  return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

quantification:
`every variable in expression1 satisfies expression2`
`some variable in expression1 satisfies expression2`

aggregation: `count, sum, max`, etc.

branching: `if (expression1) then expression2 else expression3`

```
let $kk := doc("movies.xml")/Movies/Movie[@title = "King Kong"]
for $v in $kk/Version
return
```

```
  if ($v/@year = max($kk/Version/@year))
  then <Latest>{$v}</Latest>
  else <Old>{$v}</Old>
```

ordering the result of a query
`order list of expression`
default is descending

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie,
    $v in $m/Version
order $v/@year, $m/@title
return <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

Design and normalization

- functional dependency
- BCNF (Boyce-Codd normal form)
- 3NF (third normal form)
- E/R model