# Review of Relational Databases

- Relational model
- Schemas
- Relational algebra
- Relational calulus
- SQL
- Constraints (keys, foreign keys)

# The relational model

- Data is organized in relations (tables)
- Relational database schema:

  set of table names

  list of attributes for each table
- Tables are specified as: `<table name>:<list of attributes>`
- Examples:

  `Account: number, branch, customerId`

  `Movie: title, director, actor`

  `Schedule: theater, title`
- Attributes within a table have different names
- Tables have different names

# Declarative vs Procedural

- In our queries, we ask **what** we want to see in the output.
- But we do not say **how** we want to get this output.
- Thus, query languages are **declarative**: they specify what is needed in the output, but do not say how to get it.
- Database system figures out **how** to get the result, and gives it to the user.
- Database system operates internally with different, **procedural** languages, which specify how to get the result.

# Declarative vs Procedural: example

Declarative:

$\{$ title $|$ (title, director, actor) $\in$ movies $\}$

Procedural:

```
for each tuple T=(t,d,a) in relation movies do
  output t
end
```

In relational algebra: $\pi_{title}(Movies)$.

in SQL:

```
SELECT title FROM Movies
```

# Relational Calculus

- Codd 1970: Relational databases are queried using first-order predicate logic.
- Relational calculus: another name for it. Queries written in the logical notation using:

  relation names (e.g., Movies)

  constants (e.g., 'Shining', 'Nicholson')

  conjunction $\wedge$, disjunction $\vee$, negation $\neg$

  existential quantifiers $\exists$, universal quantifiers $\forall$

- $\wedge, \exists, \neg$ suffice:

$$\begin{aligned} \forall x F(x) &= \neg \exists x \neg F(x) \\ F \vee G &= \neg(\neg F \wedge \neg G) \end{aligned}$$

## Relational Calculus cont'd

- Bound variable: a variable $x$ that occurs in $\exists x$ or $\forall x$
- Free variable: a variable that is not bound.
- Free variables are those that go into the output of a query.
- Two ways to write a query:

  $Q(\vec{x}) \; = \; F$, where $\vec{x}$ is the tuple of free variables

  $\{\vec{x} \mid F\}$
- Examples:

  $\{x, y \mid \exists z \; (R(x, z) \wedge S(z, y))\}$

  $\{x \mid \forall y R(x, y)\}$

  { dir | $\forall$ (th, tl) $\in$ schedule

  $\exists$ (tl', act): (tl',dir,act) $\in$ movies $\wedge$ (th, tl') $\in$ schedule }

# Relational Algebra

- Procedural language
- Six ( $= 5+\ 1$ ) operations:
  - Projection $\pi$
  - Selection $\sigma$
  - Cartesian product $\times$
  - Union $\cup$
  - Difference $-$
  - Renaming $\rho$
- Renaming changes names of attributes
- $\rho_{A \leftarrow C, B \leftarrow D}(R)$ turns a relation with attributes $C, D$ into a relation with attributes $A, B$.

# Relational Algebra cont'd

- Projection: chooses some attributes in a relation
- $\pi_{A_1,\ldots,A_n}(R)$: only leaves attributes $A_1, \ldots, A_n$ in relation $R$.
- Selection: Chooses tuples that satisfy some condition
- $\sigma_c(R)$: only leaves tuples $t$ for which $c(t)$ is true
- Conditions: conjunctions of

  $R.A = R.A'$ – two attributes are equal

  $R.A = constant$ – the value of an attribute is a given constant

  Same as above but with $\neq$ instead of $=$

# Relational Algebra cont'd

- Cartesian Product: puts together two relations
- $R_1 \times R_2$ puts together each tuple $t_1$ of $R_1$ and each tuple $t_2$ of $R_2$
- Example:

$$\begin{array}{c|cc} R_1 & A & B \\ \hline & a_1 & b_1 \\ & a_2 & b_2 \end{array} \quad \times \quad \begin{array}{c|cc} R_2 & A & C \\ \hline & a_1 & c_1 \\ & a_2 & c_2 \\ & a_3 & c_3 \end{array} \quad = \quad \begin{array}{cccc} R_1.A & R_1.B & R_2.A & R_2.C \\ \hline a_1 & b_1 & a_1 & c_1 \\ a_1 & b_1 & a_2 & c_2 \\ a_1 & b_1 & a_3 & c_3 \\ a_2 & b_2 & a_1 & c_1 \\ a_2 & b_2 & a_2 & c_2 \\ a_2 & b_2 & a_3 & c_3 \end{array}$$

# Relational Algebra cont'd

- Union $R \cup S$ is the union of relations $R$ and $S$
- $R$ and $S$ must have the same set of attributes.
- Difference $R - S$: tuples in $R$ but not in $S$.

- Every declarative query has a procedural implementation:

$$\text{Relational Calculus} \quad = \quad \text{Relational Algebra}$$

# SQL

- Structured Query Language
- Developed originally at IBM in the late 70s
- First standard: SQL-86
- De-facto standard of the relational database world – replaced all other languages.

# Examples of SQL queries

- Find titles of current movies

```
SELECT Title
FROM Movies
```

- `SELECT` lists attributes that go into the output of a query
- `FROM` lists input relations

# Examples of SQL queries cont'd

- Find theaters showing movies in which Nicholson played:

```
SELECT Schedule.Theater
FROM Schedule, Movies
WHERE Movies.Title = Schedule.Title
      AND Movies.Actor='Nicholson'
```

Differences:

- `SELECT` now specifies which relation the attributes came from – because we use more than one.
- `FROM` lists two relations
- `WHERE` specifies the *condition* for selecting a tuple.

# Joining relations

- `WHERE` allows us to join together several relations
- Consider a query: list directors, and theaters in which their movies are playing

```
SELECT Movies.Director, Schedule.Theater
FROM Movies, Schedule
WHERE Movies.Title = Schedule.Title
```

- This operation is called **join**.
- Notation: Schedule $\bowtie$ Movies

## Join cont'd

- Join is not a new operation of relational algebra
- It is **definable** with $\pi, \sigma, \times$
- Suppose $R$ is a relation with attributes $A_1, \ldots, A_n,\ B_1, \ldots, B_k$
- $S$ is a relation with attributes $A_1, \ldots, A_n,\ C_1, \ldots, C_m$
- $R \bowtie S$ has attributes $A_1, \ldots, A_n,\ B_1, \ldots, B_k, C_1, \ldots, C_m$

$$
\begin{aligned}
& R \bowtie S \\
= \ & \pi_{A_1,\ldots,A_n,\ B_1,\ldots,B_k,C_1,\ldots,C_m}(\sigma_{R.A_1=S.A_1 \wedge \ldots \wedge R.A_n=S.A_n}(R \times S))
\end{aligned}
$$

# Conjunctive queries

- Also known as select-project-join queries
- Fragment of relational algebra that consists of $\sigma, \pi, \bowtie$ (or $\sigma, \pi, \times$)
- In logic, $\exists$ and $\wedge$
- Theaters showing movies where Nicholson played:

  $\pi_{theater}(\sigma_{actor=Nicholson}(Movies \bowtie Schedule))$

  (hence called SPJ – select, project, join – queries)

  $\exists t \exists d \; Movies(t, d, Nicholson) \wedge Schedule(t, th)$

  often write as **rules**:

  $Q(th) \;\; \text{:–} \;\; Movies(t, d, Nicholson), Schedule(t, th)$

## Beyond simple queries

- So far we mostly used $\pi, \sigma, \bowtie$ in relational algebra.
- It is harder to do queries with "for all conditions".
- Query: *Find directors whose movies are playing in* all theaters:

$$\pi_{\text{director}}(M) - \pi_{\text{director}}(\pi_{\text{theater}}(S) \times \pi_{\text{director}}(M) - \pi_{\text{theater,director}}(M \bowtie S))$$

- They don't look easy in relational algebra

# For all and negation in SQL

- Find directors whose movies are playing in all theaters.
- SQL's way of saying this: Find directors such that there does not exist a theater where their movies do not play.
- Because: $\forall x \; f(x) \; \Leftrightarrow \; \neg\exists x \, \neg f(x)$.

```
SELECT M1.Director
FROM Movies M1
WHERE NOT EXISTS (SELECT S.Theater
                  FROM Schedule S
                  WHERE NOT EXISTS (SELECT M2.Director
                                    FROM Movies M2
                                    WHERE M2.Title=S.Title
                                      AND
                                     M1.Director=M2.Director))
```

# Other features of SQL

- Datatypes, type-specific operations
- Table declaration, constraint enforcement
- Aggregation

# Simple aggregate queries

Count the number of tuples in Movies

```
SELECT COUNT(*)
FROM Movies
```

Add up all movie lengths

```
SELECT SUM(Length)
FROM Movies
```

Find the number of directors.

```
SELECT COUNT(DISTINCT Director)
FROM   Movies
```

# Aggregation and grouping

For each theaters playing at least one long (over 2 hours) movie, find the average length of all movies played there:

```
SELECT S.Theater, AVG(M.Length)
FROM Schedule S, Movies M
WHERE S.Title=M.Title
GROUP BY S.Theater
HAVING MAX(M.Length) > 120
```

## Database Constraints

- In our examples we assumed that the *title* attribute identifies a movie.
- But this may not be the case:

| title | director | actor |
|---|---|---|
| Dracula | Browning | Lugosi |
| Dracula | Fischer | Lee |
| Dracula | Badham | Langella |
| Dracula | Coppola | Oldman |

- Database constraints: provide additional semantic information about the data.
- Most common ones: functional and inclusion dependencies, and their special cases: keys and foreign keys.

## Constraints cont'd

- If we want the *title* to identify a movie uniquely (i.e., no Dracula situation),
we express it as a **functional dependency**

$$\text{title} \longrightarrow \text{director}$$

- In general, a relation $R$ satisfies a functional dependency $A \longrightarrow B$, where $A$ and $B$ are attributes, if for every two tuples $t_1, t_2$ in $R$:

$$\pi_A(t_1) = \pi_A(t_2) \quad \text{implies} \quad \pi_B(t_1) = \pi_B(t_2)$$

# Functional dependencies and keys

- More generally, a functional dependency is $X \to Y$ where $X, Y$ are sequences of attributes. It holds in a relation $R$ if for every two tuples $t_1, t_2$ in $R$:

$$\pi_X(t_1) \;=\; \pi_X(t_2) \quad \text{implies} \quad \pi_Y(t_1) \;=\; \pi_Y(t_2)$$

- A very important special case: **keys**
- Let $K$ be a set of attributes of $R$, and $U$ the set of **all** attributes of $R$. Then $K$ is a key if $R$ satisfies functional dependency $K \to U$.
- In other words, a set of attributes $K$ is a key in $R$ if for any two tuples $t_1$, $t_2$ in $R$,

$$\pi_K(t_1) = \pi_K(t_2) \quad \text{implies} \quad t_1 = t_2$$

- That is, a key is a set of attributes that uniquely identify a tuple in a relation.

# Inclusion constraints

- We expect every Title listed in Schedule to be present in Movies.
- These are **referential** integrity constraints: they talk about attributes of one relation (Schedule) but refer to values in another one (Movies).
- These particular constraints are called **inclusion dependencies** (ID).
- Formally, we have an inclusion dependency $R[A] \subseteq S[B]$ when every value of attribute $A$ in $R$ also occurs as a value of attribute $B$ in $S$:

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

- As with keys, this extends to sets of attributes, but they must have the same number of attributes.
- There is an inclusion dependency $R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$ when

$$\pi_{A_1,\ldots,A_n}(R) \quad \subseteq \quad \pi_{B_1,\ldots,B_n}(S)$$

# Foreign keys

- Most often inclusion constraints occur as a part of a **foreign key**
- Foreign key is a conjunction of a key and an ID:

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n] \quad \text{and}$$
$$\{B_1, \ldots, B_n\} \rightarrow \text{all attributes of } S$$

- Meaning: we find a key for relation $S$ in relation $R$.
- Example: Suppose we have relations:
  `Employee(EmplId, Name, Dept, Salary)`
  `ReportsTo(Empl1,Empl2)`.
- We expect both `Empl1` and `Empl2` to be found in `Employee`; hence:
  `ReportsTo[Empl1]` $\subseteq$ `Employee[EmplId]`
  `ReportsTo[Empl2]` $\subseteq$ `Employee[EmplId]`.
- If `EmplId` is a key for `Employee`, then these are foreign keys.