A database is

- a collection of highly structured data
- along with a set of access and control mechanisms

We deal with them every day:

- bank account information,
- airline reservation systems,
- store inventories,
- library catalogs,
- telephone billing etc.
Goals of a Database Management System

- Provide users with a meaningful view of data:
  Hide from them irrelevant detail → provide an abstract view of data
- Support various operations on data
  queries: getting answers from databases
  updates: changing information in databases
- Coordinate access to data
  concurrency control

The Relational Data Model: Named Perspective

- Data is organized in relations ("tables")
- A relational database schema consists of
  a set of relation names
  a list of attributes for each relation
- Notation: <relation name>: <list of attributes>
- Examples:
  Account: number, branch, customerId
  Movie: title, director, actor
  Schedule: theater, title

- Relations have different names
- Attributes within a relation have different names
We assume three disjoint countably infinite sets of symbols:

- $\text{att}$, the possible attributes
  
  ...we assume there is a total ordering $\leq_{\text{att}}$ on $\text{att}$

- $\text{dom}$, the possible constants
  
  $\text{dom}$ is called the domain

- $\text{relname}$, the possible relation names

Relations have a sort and an arity, formalized as follows:

- For every relation name $R$ there is a finite set of attributes $\text{sort}(R)$.
  
  That is, $\text{sort}$ is a function

  $$\text{sort}: \text{relname} \rightarrow \mathcal{P}^{\text{fin}}(\text{att})$$

  We assume as well: $\text{sort}^{-1}(U)$ is infinite, for each $U \in \mathcal{P}^{\text{fin}}(\text{att})$

  What does this mean?
• The \textit{arity} of a relation is the number of attributes: \(\text{arity}(R) = |\text{sort}(R)|\)

• Notation: Often \(R[U]\) where \(U = \text{sort}(R)\), or \(R: A_1, \ldots, A_n\) if \(\text{sort}(R) = \{A_1, \ldots, A_n\}\) and \(A_1 \leq \text{att} \cdots \leq \text{att} A_n\).

Example: \(\text{sort}(\text{Account}) = \{\text{number, branch, customerId}\}\)

is denoted \(\text{Account: number, branch, customerId}\)

Relations and databases have schemas:

• A \textit{relation schema} is a relation name

• A \textit{database schema} \(R\) is a nonempty finite set of relation schemas

Example: Database schema \(C = \{\text{Account, Movie, Schedule}\}\)

\begin{align*}
\text{Account: number, branch, customerId} \\
\text{Movie: title, director, actor} \\
\text{Schedule: theater, title}
\end{align*}

\textbf{Tuples}

• A \textit{tuple} is a function

\[t: U \rightarrow \text{dom}\]

mapping a finite set \(U \subseteq \text{att}\) (a sort) to constants.

Example: Tuple \(t\) on \(\text{sort}(\text{Movie})\) such that

\begin{align*}
\text{t(title) } &= \text{Shining} \\
\text{t(director) } &= \text{Kubrick} \\
\text{t(actor) } &= \text{Nicholson}
\end{align*}

• For \(U = \emptyset\), there is only one tuple: the empty tuple, denoted \(\langle \rangle\)

• If \(U \subseteq V\), then \(t[V]\) is the restriction of \(t\) to \(V\)

Example:

\(\langle \text{title: Shining, director: Kubrick, actor: Nicholson} \rangle\)
The Relational Model: Unnamed Perspective

Alternative view: We ignore names of attributes, relations have only arities

- Tuples are elements of a Cartesian product of \( \text{dom} \)
- A tuple \( t \) of arity \( n \geq 0 \) is an element of \( \text{dom}^n \), for example
  \[
  t = \langle \text{Shining}, \text{Kubrick}, \text{Nicholson} \rangle
  \]
- We access components of tuples via their position \( i \in \{1, \ldots, n\} \):
  \[
  t(2) = \text{Kubrick}
  \]
- Note: Because of “\( \leq \text{att} \)”, unnamed and named perspective naturally correspond

Instances of Relations and Databases

- A relation or relation instance of a relation schema \( R[U] \) is a finite set of tuples on \( U \).
- A database instance of database schema \( \mathcal{R} \) is a mapping \( I \) that assigns to each \( R \in \mathcal{R} \) a relation instance.

Other perspectives:

Logic programming
First-order logic
Logic Programming Perspective

- A fact over relation $R$ with arity $n$ is an expression $R(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n \in \text{dom}$.
- A relation (instance) is a finite set of facts over $R$.
- A database instance $I$ of $R$ is the union of relation instances for each $R \in \text{R}$.

Example:

$$I = \{ \text{Movie(Shining,Kubrick,Nicholson)}, \text{Movie(Player,Altman,Robbins)}, \text{Movie(Chinatown,Polanski,Nicholson)}, \text{Movie(Chinatown,Polanski,Polanski)}, \text{Movie(Repulsion,Polanski,Deneuve)}, \text{Schedule(Le Champo,Shining)}, \text{Schedule(Le Champo,Chinatown)}, \text{Schedule(Le Champo,Player)}, \text{Schedule(Odeon,Chinatown)}, \text{Schedule(Odeon,Repulsion)} \}$$

First-Order Logic: Database Instances as Theories

- For a database instance $I$, construct an extended relational theory $\Sigma_I$ consisting of:
  - Atoms $R_i(\vec{a})$ for each $\vec{a} \in I(R_i)$;
  - Extension Axioms $\forall \vec{x}(R_i(\vec{x}) \iff \vec{x} = \vec{a}_1 \lor \cdots \lor \vec{x} = \vec{a}_m)$, where $\vec{a}_1, \ldots, \vec{a}_m$ are all elements of $R_i$ in $I$, and “=$ ranges over tuples of the same arity;
  - Unique Name Axioms: $\neg (c_i = c_j)$ for each pair $c_i, c_j$ of distinct constants occurring in $I$;
  - Domain Closure Axiom: $\forall x(x = c_1 \lor \cdots \lor x = c_n)$, where $c_1, \ldots, c_n$ is a listing of all constants occurring in $I$.

- If the “$=$” are not available, the intended meaning can be emulated with equality axioms.

- Theorem: The interpretations of dom and $R$ that satisfy $\Sigma_I$ are isomorphic to $I$.

- Corollary: A set of sentences $\Gamma$ is satisfied by $I$ iff $\Sigma_I \cup \Gamma$ is satisfiable.

Other view: database instance $I$ as finite relational structure (finite universe of discourse; considered later)
**Database Queries: Examples**

- **“What are the titles of current movies?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shining</td>
</tr>
<tr>
<td></td>
<td>Player</td>
</tr>
<tr>
<td></td>
<td>Chinatown</td>
</tr>
<tr>
<td></td>
<td>Repulsion</td>
</tr>
</tbody>
</table>

- **“Which theaters are showing movies directed by Polanski?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>theater</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Le Champo</td>
</tr>
<tr>
<td></td>
<td>Odéon</td>
</tr>
</tbody>
</table>

- **“Which theaters are showing movies featuring Nicholson?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>theater</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Le Champo</td>
</tr>
<tr>
<td></td>
<td>Odéon</td>
</tr>
</tbody>
</table>

- **“Which directors acted themselves?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>director</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Polanski</td>
</tr>
</tbody>
</table>

- **“Who are the directors whose movies are playing in all theaters?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>director</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Polanski</td>
</tr>
</tbody>
</table>

- **“Which theaters show only movies featuring Nicholson?”**
  
<table>
<thead>
<tr>
<th>answer</th>
<th>theater</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  ...but if Le Champo stops showing 'Player', the answer contains 'Le Champo'.

*Relational Query Languages*
How To Ask a Query?

- Query languages
  
  Commercial: SQL
  
  Theoretical: Relational Algebra, Relational Calculus, datalog etc.

- Query results: Relations constructed from relations in the database

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.

- A database management system figures out **how** to get the result, and gives it to the user.

- A database management system operates internally with an **algebra** that takes into account how data is stored.

- Finally, queries in that algebra are translated into a **procedural** language.
Declarative vs Procedural: Example

Declarative:

\{
\text{title} \mid (\text{title, director, actor}) \in \text{Movie}\}

Procedural:

\text{for each tuple } T= (t, d, a) \text{ in relation Movie do}
\text{output } t
\text{end}

 Conjunctive Queries

- Conjunctive queries are a simple form of declarative, \textit{rule-based queries}
- A \textit{rule} says \textit{when} certain elements belong to the answer.

- \textbf{Example}: “What are the titles of current movies?”
  As a conjunctive query:

  \text{answer}(tl) :– \text{Movie}(tl, dir, act)

  That is, while (tl, dir, act) ranges over relation Movies, output tl (the title attribute)
Conjunctive Queries: One More Example

“Which theaters are showing movies directed by Polanski?”

As a conjunctive query:

\[
\text{answer}(\text{th}) :\neg \text{Movie}(\text{tl}, \text{'Polanski'}, \text{act}), \text{Schedule}(\text{th}, \text{tl})
\]

While (\text{tl}, \text{dir}, \text{act}) range over tuples in \text{Movie}, check if \text{dir} is \text{'Polanski'};
   
   if not, go to the next tuple,

   if yes, look at all tuples (\text{th}, \text{tl}) in Schedule
       
       corresponding to the title \text{tl} in relation \text{Movie},

   and output \text{th}.

Conjunctive queries are probably the most **common** type of queries
and are **building blocks** for all other queries over relational databases.

Conjunctive Queries: Another Example

“Which theaters are showing movies featuring Nicholson?”

Very similar to the previous example:

\[
\text{answer}(\text{th}) :\neg \text{Movie}(\text{tl}, \text{dir}, \text{'Nicholson'}), \text{Schedule}(\text{th}, \text{tl})
\]

While (\text{tl}, \text{dir}, \text{act}) range over tuples in \text{Movie}, check if \text{act} is \text{'Nicholson'};
   
   if not, go to the next tuple,

   if yes, look at all tuples (\text{th}, \text{tl}) in Schedule
       
       corresponding to the title \text{tl} in relation \text{Movie},

   and output \text{th}.
Conjunctive Queries: Still One More . . .

“Which directors acted in one of their own movies?”:

\[
\text{answer}(\text{dir}) \leftarrow \text{Movie}(\text{tl}, \text{dir}, \text{act}), \text{dir} = \text{act}
\]

While (tl, dir, act) ranges over tuples in movie,
check if dir is the same as act,
and output it if that is the case.

Alternative formulation:

\[
\text{answer}(\text{dir}) \leftarrow \text{Movie}(\text{tl}, \text{dir}, \text{dir})
\]

Conjunctive Queries: Definition

A rule-based conjunctive query with (in)equalities is an expression of form

\[
\text{answer}(\vec{x}) \leftarrow R_1(\vec{x}_1), \ldots, R_n(\vec{x}_n),
\]

where \( n \geq 0 \) and

- “answer” is a relation name not in \( R \cup \{ =, \neq \} \)
- \( R_1, \ldots, R_n \) are relation names from \( R \cup \{ =, \neq \} \)
- \( \vec{x} \) is a tuple of distinct variables with length \( = \) \( \text{arity}(\text{answer}) \)
- \( \vec{x}_1, \ldots, \vec{x}_n \) are tuples of variables and constants of suitable (?!?) length
- each variable occurring somewhere in the query must also occur in some atom \( R_i(\vec{x}_i) \) where \( R_i \in R \)

Note: Equality “=” can be eliminated if we change the definition slightly

How?
Conjunctive Queries: Semantics

Let $q$ be a conjunctive query of the form (1) and let $I$ be a database instance.

- A valuation $\nu$ over $\text{var}(q)$ is a mapping
  
  $\nu : \text{var}(q) \cup \text{dom} \rightarrow \text{dom}$
  
  that is the identity on $\text{dom}$.

- The result (aka image) of $q$ on $I$ is
  
  $q(I) = \{ \nu(\vec{x}) \mid \nu \text{ is a valuation over } \text{var}(q), \text{ and } \nu(\vec{x}_i) \in I(R_i), \text{ for all } 1 \leq i \leq n \}$

Example: $q$: answer(dir) :– Movie(tl, dir, act), dir=act

For $I$ from above, we obtain

$q(I) = \{ \langle \text{Polanski} \rangle \}$

Elementary Properties of Conjunctive Queries

Proposition. Let $q$ be a conjunctive query of form (1). Then:

- the result $q(I)$ is finite, for any database instance $I$;

- $q$ is monotonic,
  
  i.e., $I \subseteq J$ implies $q(I) \subseteq q(J)$, for all database instances $I$ and $J$;

- if $q$ contains neither “=” nor “≠”, then $q$ is satisfiable,
  
  i.e., there exists some $I$ such that $q(I) \neq \emptyset$
Beyond Conjunctive Queries?

"Who are the directors whose movies are playing in all theaters?"

- Recall the notation from mathematical logic:
  \( \forall \) means ‘for all’, \( \exists \) means ‘exists’, ‘\( \land \)’ is conjunction (logical ‘and’)

- We write the query above as
  \[
  \{ \text{dir} \mid \forall \text{th} \left( \exists \text{tl}' \left( \text{Schedule}(\text{th}, \text{tl}') \rightarrow \right. \right.
  \exists \text{tl}, \text{act} \left( \text{Movie}(\text{tl}, \text{dir}, \text{act}) \land \text{Schedule}(\text{th}, \text{tl}) \right) \}
  \]

- That is, to see if director dir is in the answer, for each theater name th, check
  that there exists a tuple (tl, dir, act) in Movie, and a tuple (th, tl) in Schedule

  *Is there something missing?*

  *Can we formulate this as a conjunctive query?*

Structured Query Language: SQL

- De-facto standard for all relational RDBMs
- Latest versions: SQL:1999 (also called SQL3), SQL:2003 (supports XML),
  SQL:2006 (more XML support), SQL:2008
  Each standard covers well over 1,000 pages

  “The nice thing about standards is that you have so many to choose from.”
  – Andrew S. Tanenbaum.

- Query structure:
  \[
  \text{SELECT} \quad \text{attribute list } \langle R_i.A_j \rangle \\
  \text{FROM} \quad R_1, \ldots, R_n \\
  \text{WHERE} \quad \text{condition } C
  \]
  In the simplest case, \( C \) is a conjunction of equalities/inequalities
### SQL Examples

- “Which theaters are showing movies directed by Polanski?”:

  ```sql
  SELECT Schedule.Theater
  FROM Schedule, Movie
  WHERE Movie.Title = Schedule.Title AND
        Movie.Director = 'Polanski'
  ```

- “Which theaters are playing the movies of which directors?”

  ```sql
  SELECT Movie.Director, Schedule.Theater
  FROM Movie, Schedule
  WHERE Movie.Title = Schedule.Title
  ```

### Relational Algebra

- We start with a subset of relational algebra that suffices to capture queries defined by simple rules, SQL `SELECT-FROM-WHERE` statements.
- The subset has three operations:
  - Projection \( \pi \)
  - Selection \( \sigma \)
  - Cartesian Product \( \times \)
- This fragment of Relational Algebra is called **SPC Algebra**
- Sometimes we also use **renaming** of attributes, denoted as \( \rho \)
Projection

- Restricts tuples of a relation \( R \) to a subset of \( \text{sort}(R) \)
- \( \pi_{A_1, \ldots, A_n}(R) \) returns a new relation with sort \( \{ A_1, \ldots, A_n \} \)
- Example:

\[
\begin{pmatrix}
\text{title} & \text{director} & \text{actor} \\
\text{Shining} & \text{Kubrick} & \text{Nicholson} \\
\text{Player} & \text{Altman} & \text{Robbins} \\
\text{Chinatown} & \text{Polanski} & \text{Nicholson} \\
\text{Chinatown} & \text{Polanski} & \text{Polanski} \\
\text{Repulsion} & \text{Polanski} & \text{Deneuve}
\end{pmatrix}
= \begin{pmatrix}
\text{title} & \text{director} \\
\text{Shining} & \text{Kubrick} \\
\text{Player} & \text{Altman} \\
\text{Chinatown} & \text{Polanski} \\
\text{Repulsion} & \text{Polanski}
\end{pmatrix}
\]

- Creates a view of the original data that hides some attributes

Selection

- Chooses tuples of \( R \) that satisfy some condition \( C \)
- \( \sigma_C(R) \) returns a new relation with the same sort as \( R \), and with the tuples \( t \) of \( R \) for which \( C(t) \) is true
- Conditions are conjunctions of elementary conditions of the form
  \( R.A = R.A' \) (equality between attributes)
  \( R.A = \text{constant} \) (equality between an attribute and a constant)
  same as above but with \( \neq \) instead of \( = \)
- Examples:
  \[
  \text{Movie.Actor} = \text{Movie.Director}
  \]
  \[
  \text{Movie.Actor} = \text{Movie.Director} \land \text{Movie.Actor} \neq \text{Nicholson}
  \]
- Creates a view of data by hiding tuples that do not satisfy the condition
### Selection: Example

\[ \sigma_{\text{actor}=\text{director} \land \text{director}='Polanski'}(\text{title, director, actor}) = \]

<table>
<thead>
<tr>
<th>title</th>
<th>director</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shining</td>
<td>Kubrick</td>
<td>Nicholson</td>
</tr>
<tr>
<td>Player</td>
<td>Altman</td>
<td>Robbins</td>
</tr>
<tr>
<td>Chinatown</td>
<td>Polanski</td>
<td>Nicholson</td>
</tr>
<tr>
<td>Chinatown</td>
<td>Polanski</td>
<td>Polanski</td>
</tr>
<tr>
<td>Repulsion</td>
<td>Polanski</td>
<td>Deneuve</td>
</tr>
</tbody>
</table>

### Cartesian Product

- \( R_1 \times R_2 \) is a relation with \( \text{sort}(R_1 \times R_2) = \text{sort}(R_1) \cup \text{sort}(R_2) \) and the tuples are all possible combinations \((t_1, t_2)\) of \(t_1\) in \(R_1\) and \(t_2\) in \(R_2\).

- **Example:**

  \[
  \begin{array}{c|c|c} 
  R_1 & A & B \\
  \hline 
  a_1 & b_1 \\
  a_2 & b_2 \\
  \end{array} \times \begin{array}{c|c|c} 
  R_2 & A & C \\
  \hline 
  a_1 & c_1 \\
  a_2 & c_2 \\
  a_3 & c_3 \\
  \end{array} = 
  \begin{array}{c|c|c|c|c} 
  \hline 
  a_1 & b_1 & a_1 & c_1 \\
  a_1 & b_1 & a_2 & c_2 \\
  a_1 & b_1 & a_3 & c_3 \\
  a_2 & b_2 & a_1 & c_1 \\
  a_2 & b_2 & a_2 & c_2 \\
  a_2 & b_2 & a_3 & c_3 \\
  \end{array}
  \]

- We assume that the cartesian product operator automatically renames attributes so as to include the name of the relation: in the resulting table, all attributes must have different names.
**Cartesian Product: Example**

“Which theaters are playing movies directed by Polanski?”

answer(th) :- Movie(tl,dir,act), Schedule(th,tl), dir='Polanski'

- **Step 1:** Let $R_1 = \text{Movie} \times \text{Schedule}$

  We don’t need all tuples, only those in which titles are the same, so:

- **Step 2:** Let $R_2 = \sigma_C(R_1)$ where $C$ is “Movie.title = Schedule.title”

  We are only interested in movies directed by Polanski, so

- **Step 3:** $R_3 = \sigma_{\text{director}='Polanski'}(R_2)$

  In the output, we only want theaters, so finally

- **Step 4:** Answer = $\pi_{\text{theater}}(R_3)$

**Summing up, the answer is**

$$\pi_{\text{theater}}(\sigma_{\text{director}='Polanski'}(\sigma_{\text{Movie}.\text{title}=\text{Schedule}.\text{title}}(\text{Movie} \times \text{Schedule})))$$

- **Merging selections, this is equivalent to**

$$\pi_{\text{theater}}(\sigma_{\text{director}='Polanski'} \land \text{Movie}.\text{title}=\text{Schedule}.\text{title}(\text{Movie} \times \text{Schedule}))$$
Renaming

- Let $R$ be a relation that has attribute $A$ but does not have attribute $B$.
- $\rho_{B\leftarrow A}(R)$ is the “same” relation as $R$ except that $A$ is renamed to be $B$.
  
  **Example:**
  
  $\rho_{\text{parent}\leftarrow \text{father}}$
  
  $\begin{pmatrix}
  \text{father} & \text{child} \\
  \text{George} & \text{Elizabeth} \\
  \text{Philip} & \text{Charles} \\
  \text{Charles} & \text{William}
  \end{pmatrix}
  =
  
  \begin{pmatrix}
  \text{parent} & \text{child} \\
  \text{George} & \text{Elizabeth} \\
  \text{Philip} & \text{Charles} \\
  \text{Charles} & \text{William}
  \end{pmatrix}$

- Simultaneous renaming $\rho_{A_1, \ldots, A_m \leftarrow B_1, \ldots, B_m}$, for distinct $A_1, \ldots, A_m$ resp. $B_1, \ldots, B_m$ can be defined from it.
- Prefixing the relation name to rename attributes is convenient (used in practice)
- Not all problems are solved by this (e.g., Cartesian Product $R \times R$)

Unnamed Perspective

- Renamings are for SPC immaterial, if we adopt the unnamed perspective
  
  **Why?**

- Example (again): “Which theaters are playing movies directed by Polanski?”
  
  Recall Movie: title, director, actor
  
  Schedule: theater, title

  $\pi_1(\sigma_{2=\text{Polanski}'} \land 1=5(Movie \times Schedule)))$

- SPC Algebra is often assumed to be based in the unnamed setting
- Other operations of Relational Algebra can only be defined for named perspective (e.g., natural join, to be seen later)
SQL and Relational Algebra

For execution, declarative queries are translated into algebra expressions

- Idea: SELECT is projection \( \pi \)
  
  FROM is Cartesian product \( \times \)
  
  WHERE is selection \( \sigma \)

- A simple case (only one relation in FROM):

  \[
  \begin{align*}
  & \text{SELECT } A, B, \ldots \\
  & \text{FROM } R \\
  & \text{WHERE } C
  \end{align*}
  \]

  is translated into

  \[
  \pi_{A,B,\ldots}(\sigma_C(R))
  \]

Translating Declarative Queries into Relational Algebra

We use rules as intermediate format

Example: “Which are the titles of movies?”

- SELECT Title
  
  FROM Movie

- answer(tl) :- Movie(tl,dir,act)

- \( \pi_{\text{title}}(\text{Movie}) \)

  \( \ldots \text{this was simply projection} \)
"Which theaters are showing movies directed by Polanski?"

- SELECT Schedule.Theater
  FROM Schedule, Movie
  WHERE Movie.Title = Schedule.Title AND
  Movie.Director = 'Polanski'

- First, translate into a rule:
  answer(th) ← Schedule(th, tl), Movie(tl', 'Polanski', act)

- Second, change the rule such that:
  - constants appear only in conditions
  - no variable occurs twice

- This gives us:
  answer(th) ← Schedule(th, tl), Movie(tl', dir, act), dir = 'Polanski', tl = tl'

Two relations $\implies$ Cartesian product
Conditions $\implies$ selection
Subset of attributes in the answer $\implies$ projection

- Step 1: $R_1 = \text{Schedule} \times \text{Movie}$
- Step 2: Make sure we talk about the same movie:
  $R_2 = \sigma_{\text{Schedule.title} = \text{Movie.title}}(R_1)$
- Step 3: We are only interested in Polanski's movies:
  $R_3 = \sigma_{\text{Movie.director} = \text{Polanski}}(R_2)$
- Step 4: We need only theaters in the output
  $\text{answer} = \pi_{\text{Schedule.theater}}(R_3)$
Summing up, the answer is:

\[ \pi_{\text{Schedule}.\text{theater}}(\sigma_{\text{Movie}.\text{director}=\text{Polanski}}(\sigma_{\text{Schedule}.\text{title}=\text{Movie}.\text{title}}(\text{Schedule} \times \text{Movie}))) \]

or, using the rule \( \sigma_{C_1} (\sigma_{C_2} (R)) = \sigma_{C_1 \land C_2} (R) \):

\[ \pi_{\text{Schedule}.\text{theater}}(\sigma_{\text{Movie}.\text{director}=\text{Polanski} \land \text{Schedule}.\text{title}=\text{Movie}.\text{title}}(\text{Schedule} \times \text{Movie})) \]

**Formal Translation: SQL to Rules**

\[
\begin{align*}
\text{SELECT} & \quad \text{attribute list} \langle R_i.A_j \rangle \\
\text{FROM} & \quad R_1, \ldots, R_n \\
\text{WHERE} & \quad \text{condition } C
\end{align*}
\]

is translated into:

\[
\begin{align*}
\text{answer}((R_i.A_j)) & : R_1(<\text{attributes}>,) \\
& \quad \ldots, \\
& \quad R_n(<\text{attributes}>,) \\
& \quad C
\end{align*}
\]

Note: Attributes become variables of rules
Rules to Relational Algebra

- Consider the rule
  \[
  \text{answer}(\bar{x}) : \text{-} R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)
  \]
  where, wlog ("without loss of generality"),
  
  \[R_1, \ldots, R_k \in \mathcal{R}, k \leq n,\]
  
  \[R_{k+1}, \ldots, R_n \in \{=, \neq\}.
  \]
  
  Let \(\text{conditions} := R_{k+1}(\bar{x}_{k+1}), \ldots, R_n(\bar{x}_n)\)

- **First transformation:** Ensure that each variable occurs at most once 
  in \(R_1(\bar{x}_1), \ldots, R_k(\bar{x}_k)\):

  If there are \(R_i(\ldots, x, \ldots)\) and \(R_j(\ldots, x, \ldots)\),
  rewrite them as \(R_i(\ldots, x', \ldots)\) and \(R_j(\ldots, x'', \ldots)\), and
  add \(x' = x''\) to the conditions and, if \(x\) occurs elsewhere, also \(x = x'\)

Example:

\[
\text{answer}(\text{th}, \text{dir}) \text{-} \text{movie}(\text{tl}, \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl})
\]

is rewritten to

\[
\text{answer}(\text{th}, \text{dir}) \text{-} \text{movie}(\text{tl'}, \text{dir}, \text{act}), \text{schedule}(\text{th}, \text{tl''}), \text{tl'}=\text{tl''}
\]

- **Next step:** each occurrence of a constant \(a\) in a relational atom \(R_i(\ldots, a, \ldots)\), \(R_i \in \mathcal{R}\), is replaced by some variable \(x\) and add \(x = a\) to the conditions

- **Finally:** after the rule (2) is rewritten, it is translated into

\[
\pi_{\hat{x}}(\sigma_{\text{conditions}}(R_1 \times \cdots \times R_n))
\]

where \(\hat{x}\) maps

- a variable \(x\) occurring in some \(R_i(\ldots, x, \ldots)\), \(R_i \in \mathcal{R}\),
  to the corresponding attribute \(\hat{x}\) in sort\((R_i)\);

- an expression \(\alpha\) to the expression \(\hat{\alpha}\) where every \(x\) is replaced by \(\hat{x}\)
Putting it Together: SQL to Relational Algebra

Combine the two translation steps:

\[ SQL \mapsto \text{rule-based queries} \mapsto \text{relational algebra.} \]

This yields the following translation from SQL to relational algebra:

\[
\begin{align*}
\text{SELECT} & \quad \text{attribute list} \quad \langle R_i.A_j \rangle \\
\text{FROM} & \quad R_1, \ldots, R_n \\
\text{WHERE} & \quad \text{condition } C
\end{align*}
\]

becomes

\[
\pi_{\langle R_i.A_j \rangle}(\sigma_C(R_1 \times \ldots \times R_n))
\]

Another Example

“Which theaters show movies featuring Nicholson?”

\[
\begin{align*}
\text{SELECT} & \quad \text{Schedule.Theater} \\
\text{FROM} & \quad \text{Schedule, Movie} \\
\text{WHERE} & \quad \text{Movie.Title} = \text{Schedule.Title} \land \text{Movie.Actor} = \text{’Nicholson’}
\end{align*}
\]

- Translate into a rule:
  \[
  \text{answer(th)} :\leftarrow \text{movie(tl, dir, ’Nicholson’), schedule(th, tl)}
  \]

- Modify the rule:
  \[
  \text{answer(th)} :\leftarrow \text{movie(tl, dir, act), schedule(th, tl’), tl=tl’, act=’Nicholson’}
  \]
answer(th) :- movie(tl, dir, act), schedule(th, tl'), tl=tl', act="Nicholson"

- Step 1: $R_1 = \text{Schedule} \times \text{Movie}$
- Step 2: Make sure we talk about the same movie:
  $$R_2 = \sigma_{\text{Schedule}.\text{title}=\text{Movie}.\text{title}}(R_1)$$
- Step 3: We are only interested in movies with Nicholson:
  $$R_3 = \sigma_{\text{Movie}.\text{actor}=\text{Nicholson}}(R_2)$$
- Step 4: we need only theaters in the output
  $$\text{answer} = \pi_{\text{schedule}.\text{theater}}(R_3)$$

Summing up:

$$\pi_{\text{schedule}.\text{theater}}(\sigma_{\text{Movie}.\text{actor}=\text{Nicholson} \land \text{Schedule}.\text{title}=\text{Movie}.\text{title}}(\text{Schedule} \times \text{Movie}))$$

---

**SPC Algebra into SQL**

Should be easy, but is it?

Where's the difficulty?

- Direct proof in two steps:
  
  Show that for SPC queries there are normal forms
  $$\pi_{A_1, \ldots, A_n}(\sigma_c(R_1 \times \cdots \times R_m)),$$
  called “simple SPC queries”  
  \textit{(proof idea?)}

  Then map normal forms to SQL

- Indirect proof:
  
  SPC is equivalent to conjunctive queries
  
  Conjunctive queries are equivalent to single block SQL queries
**Extension: Natural Join**

- Combine all pairs of tuples $t_1$ and $t_2$ in relations $R_1$ resp. $R_2$ that agree on common attributes.
- The resulting relation $R = R_1 \bowtie R_2$ is the **natural join** of $R$ and $S$, defined on the set of attributes in $R_1$ and $R_2$.

**Example: Schedule $\bowtie$ Movie**

<table>
<thead>
<tr>
<th>title</th>
<th>director</th>
<th>actor</th>
<th>theater</th>
<th>title</th>
<th>director</th>
<th>actor</th>
<th>theater</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shining</td>
<td>Kubrick</td>
<td>Nicholson</td>
<td>Le Champo</td>
<td>Shining</td>
<td>Kubrick</td>
<td>Nicholson</td>
<td>Le Champo</td>
</tr>
<tr>
<td>Player</td>
<td>Altman</td>
<td>Robbins</td>
<td>Le Champo</td>
<td>Chinatown</td>
<td>Polanski</td>
<td>Polanski</td>
<td>Le Champo</td>
</tr>
<tr>
<td>Chinatown</td>
<td>Polanski</td>
<td>Nicholson</td>
<td>Le Champo</td>
<td>Player</td>
<td>Altman</td>
<td>Robbins</td>
<td>Le Champo</td>
</tr>
<tr>
<td>Chinatown</td>
<td>Polanski</td>
<td>Polanski</td>
<td>Odéon</td>
<td>Chinatown</td>
<td>Polanski</td>
<td>Polanski</td>
<td>Le Champo</td>
</tr>
<tr>
<td>Repulsion</td>
<td>Polanski</td>
<td>Deneuve</td>
<td>Odéon</td>
<td>Repulsion</td>
<td>Polanski</td>
<td>Deneuve</td>
<td>Odéon</td>
</tr>
</tbody>
</table>

**Relational Query Languages**

**Natural Join cont’d**

Natural join is not a new operation of relational algebra

- It is **definable** with $\pi$, $\sigma$, $\times$ (*and renaming!*?)
- Suppose
  - $R$ is a relation with attributes $A_1, \ldots, A_n, B_1, \ldots, B_k$
  - $S$ is a relation with attributes $A_1, \ldots, A_n, C_1, \ldots, C_m$
  - $\Rightarrow$ $R \bowtie S$ has attributes $A_1, \ldots, A_n, B_1, \ldots, B_k, C_1, \ldots, C_m$
- Then
  $$R \bowtie S = \pi_{A_1, \ldots, A_n, B_1, \ldots, B_k, C_1, \ldots, C_m} (\sigma_{R.A_1 = S.A_1 \land \ldots \land R.A_n = S.A_n} (R \times S))$$

*Could a natural join be defined in the unnamed perspective?*
Select Project Join Queries (SPJ Queries)

Queries of the form

$$\pi_{A_1, \ldots, A_n} (\sigma_c (R_1 \Join \cdots \Join R_m))$$

are called Select-project-join queries.

- These are probably the most common queries (over databases with foreign keys).

**Example:** “Which theaters show movies directed by Polanski?”

- answer(th) :- schedule(th, tl), movie(tl, 'Polanski', act)
- As SPJ query:

  $$\pi_{\text{theater}} (\sigma_{\text{director} = \text{'Polanski'}} (\text{Movie} \Join \text{Schedule}))$$

- Why has the query become simpler compared to the earlier version

  $$\pi_{\text{schedule. theater}} (\sigma_{\text{Movie. director} = \text{'Polanski'} \land \text{Schedule. title} = \text{Movie. title}} (\text{Schedule} \times \text{Movie}))$$?

SPJ Queries cont’d

“Which theaters show movies featuring Nicholson?”

- As rule-based conjunctive query

  answer(th) :- movie(tl, dir, 'Nicholson'), schedule(th, tl)

- As SPJ query:

  $$\pi_{\text{theater}} (\sigma_{\text{actor} = \text{'Nicholson'}} (\text{Movie} \Join \text{Schedule}))$$
Translating SPJ Queries to Rules and Single Block SQL

- **SPJ Query**
  \[ Q = \pi_{A_1, \ldots, A_n}(\sigma_C(R \bowtie S)) \]

- **Equivalent SQL statement** \((B_1, \ldots, B_m = \text{common attributes in } R \text{ and } S)\):
  
  ```
  SELECT A_1, \ldots, A_n 
  FROM R, S 
  WHERE C \text{ AND } R.B_1 = S.B_1 \text{ AND } \ldots \text{ AND } R.B_m = S.B_m
  ```

- **Equivalent rule query** \((R \text{ resp. } S \text{ has attributes: } C_1, \ldots, C_k \text{ resp. } D_1, \ldots, D_l)\)
  
  ```
  answer(A_1, \ldots, A_n) :– R(C_1, \ldots, C_k), S(D_1, \ldots, D_l), 
  R.B_1 = S.B_1, \ldots, R.B_m = S.B_m, C
  ```

SPJ to SQL: Example

“Who are the directors of currently playing movies that feature Ford?”

- **In SPJ**:
  \[ \pi_{\text{director}}(\sigma_{\text{actor} = 'Ford'}(\text{Movie} \bowtie \text{Schedule})) \]

- **In SQL**:
  ```
  SELECT Movie.director 
  FROM Movie, Schedule 
  WHERE Movie.title = Schedule.title AND Movie.actor = 'Ford'
  ```
What We’ve Seen So Far

- Queries defined by SQL `SELECT–FROM–WHERE` statements (single block queries)
- These are the same as the queries definable by rules
- They are also the same as the queries definable by $\pi$, $\sigma$, $\times$ (and renaming) in relational algebra, i.e., the same as SPC queries
- Question: What about SPJ?
  - SPJ queries are not a normal form for the $\sigma$, $\pi$, $\times$-fragment
  - $\leadsto$ To prevent unwanted joins, we need renaming
- SPJR Algebra = $\sigma$, $\pi$, $\Delta$, $\rho$ — fragment of Relational Algebra

Equivalence of SPC and SPJR Algebras

**Proposition.** The SPC Algebra and the SPJR Algebra are equivalent.

**Note:**
- Cartesian Product can be easily emulated using renaming
- BTW, also SQL provides a renaming construct
  - New attribute names can be introduced in `SELECT` using keyword `AS`.
  ```sql
  SELECT Father AS Parent, Child FROM R
  ```
Nested SQL Queries: Simple Example

- So far in the **WHERE** clause we used comparisons between attributes
- In general, a **WHERE** clause can contain *another query*, and test some relationship between an attribute or a constant and the result of that query
- We call such queries with subqueries *nested* queries

**Example:** “Which theaters are showing Polanski’s movies?”

```
SELECT Schedule.theater
FROM    Schedule
WHERE   Schedule.title IN
        (SELECT Movie.title
         FROM      Movie
         WHERE     Movie.director = 'Polanski')
```

### Nested vs Unnested Queries

```
SELECT S.theater
FROM    Schedule S
WHERE   S.title IN
        (SELECT M.title
         FROM      Movie M
         WHERE     M.director = 'Polanski')

SELECT S.theater
FROM    Schedule S, Movie M
WHERE   S.title = M.title AND
        M.director = 'Polanski'
```

- Both queries capture the same question …
- … and return the same results over all instances (*.. or do they?*)
- Queries nested with IN can be flattened …
- … but others can’t *(which?)*
**Equivalence Theorem**

**Theorem.** The following languages define the same (?! sets of queries:

- SPJR Queries
- SPC Queries
- simple SPC queries
- (rule-based) conjunctive queries
- SQL SELECT-FROM-WHERE
- SQL SELECT-FROM-WHERE with IN-nesting

**Disjunction in Queries**

“Which actors played in movies directed by Kubrick OR Polanski”

- SELECT Actor
  FROM Movie
  WHERE director = 'Kubrick' OR director = 'Polanski'

- Can this be defined by a single rule?

- How do you prove your answer?
  
  *(Hint: What can you say about the constants in the query and in the database?)*
**Union in SQL**

- The way out: Disjunction can be represented using more than one rule
  
  \[
  \text{answer(act) :– movie(tl,dir,act), dir='Kubrick'}
  \]
  
  \[
  \text{answer(act) :– movie(tl,dir,act), dir='Polanski'}
  \]
  
- Semantics: compute answers to each of the rules, and then take their union (union of conjunctive queries)
  
- SQL has its own syntax (distinguishing between `UNION` and `UNION ALL`):
  
  ```sql
  SELECT Actor
  FROM Movie
  WHERE director = 'Kubrick'
  UNION
  SELECT Actor
  FROM Movie
  WHERE director = 'Polanski'
  ```

**Disjunction in Relational Algebra**

How can we translate a query with disjunction into relational algebra?

- \( \text{answer(act) :– movie(tl,dir,act), dir='Kubrick'} \)
  
  is translated into
  
  \[
  Q_1 = \pi_{\text{actor}}(\sigma_{\text{director}=\text{Kubrick}}(\text{Movie}))
  \]
  
- \( \text{answer(act) :– movie(tl,dir,act), dir='Polanski'} \)
  
  is translated into
  
  \[
  Q_2 = \pi_{\text{actor}}(\sigma_{\text{director}=\text{Polanski}}(\text{Movie}))
  \]
  
- The whole query is translated into \( Q_1 \cup Q_2 \), i.e.,

  \[
  \pi_{\text{actor}}(\sigma_{\text{director}=\text{Kubrick}}(\text{Movie})) \cup \pi_{\text{actor}}(\sigma_{\text{director}=\text{Polanski}}(\text{Movie}))
  \]
Union in Relational Algebra

- Union is another operation of relational algebra
  
  \[ R \cup S \] is the union of relations \( R \) and \( S \)

  \( R \) and \( S \) must have the same set of attributes (be “union-compatible”).

- We now have four relational algebra operations:

  \[ \pi, \sigma, \times, \cup \]

  (and of course \( \Join \), which is definable from \( \pi, \sigma, \times \))

- This fragment is called the SPCU-Algebra, or positive relational algebra.

  Would an intersection operator add something new?

  And what about set difference?

Identities Among Relational Algebra Operators

- \( \pi_{A_1, \ldots, A_n}(R \cup S) = \pi_{A_1, \ldots, A_n}(R) \cup \pi_{A_1, \ldots, A_n}(S) \)

- \( \sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S) \)

- \( (R \cup S) \times T = R \times T \cup S \times T \)

- \( T \times (R \cup S) = T \times R \cup T \times S \)
Normal Form of SPCU Queries

**Theorem.** Every SPCU query is equivalent to a union of SPC queries

Proof: propagate the union operation.

Example:

\[ \pi_A(\sigma_c((R \times (S \cup T)) \cup W)) \]

\[ = \pi_A(\sigma_c((R \times S) \cup (R \times T) \cup W)) \]

\[ = \pi_A(\sigma_c(R \times S) \cup \sigma_c(R \times T) \cup \sigma_c(W)) \]

\[ = \pi_A(\sigma_c(R \times S)) \cup \pi_A(\sigma_c(R \times T)) \cup \pi_A(\sigma_c(W)) \]

Another Equivalence Theorem

**Theorem.** The following languages define the same sets of queries

- Positive relational algebra (SPCU queries)
- unions of SPC queries
- queries defined by multiple rules
- unions of conjunctive queries
- SQL `SELECT-FROM-WHERE-UNION`
- SQL `SELECT-FROM-WHERE-UNION` with IN-nesting
- SPJRU queries \((\sigma, \pi, \Join, \rho, \cup)\)

Would intersection add anything new?
Readings


  This is not a theory book but it covers the languages we presented in a very elegant way. It also has a lot of examples.

Relational Query Languages