Based on 3 Key Features

- Simple data structures: 2-dimensional tables – physical data independence
- Solid foundation for consistency – normalization, integrity rules
- Set-oriented manipulation of relations – relational algebra, calculus, and SQL
Basic Concepts

- A database is a structured collection of data related to some real-life enterprise.
- A relational database structures the data in the form of tables.
- A relation $R$, defined over $n$ sets $D_1, \ldots, D_n$, is a set of tuples $<d_1, d_2, \ldots, d_n>$ such that $d_i$ belongs to $D_i$.

Example

- Employees relation:
  - EMP(ENO, ENAME, TITLE, SAL, PNO, RESP, DUR)
  - PROJ(PNO, PNAME, BUDGET)
RDBMS terms

- Tables or relations
- Schema
- Attributes
- Tuples
- Null Value

Keys

- A key is a minimal nonempty subset of attributes of a relation that uniquely identify each tuple of the relation.
- A superkey is a superset of a key
- A primary key is a specially designated key of the relation.
- All others keys are called candidate keys.
Normalization

• Possible anomalies in the schema include:
  – Repetition anomaly: e.g. name, title, and salary of every employee repeated for each project – wastage of space.
  – Update anomaly: due to repetition, multiple rows need to be changed
  – Insertion anomaly: it may not be possible to add new information to the database, e.g. a new employee can’t be added unless assigned to a project
  – Deletion anomaly: unnecessary deletion of data, e.g. if an employee works only on one project, and it is cancelled, then we lose information about the employee.

Normalization

• Normalization transforms arbitrary schemas into ones without these problems.
• Achieved through decomposition, and functional dependencies.
• Several normal forms are defined, e.g. 3NF, BCNF, etc.
• Want to have a lossless-join, and dependency preserving decomposition.
Normalization

• We say that A functionally determines B if (where A and B are sets of attributes of relation R), for each value of A in R, there is only one value of B in R, or $A \rightarrow B$.
  – $PNO \rightarrow (PNAME, BUDGET)$
  – $(ENO, PNO) \rightarrow (ENAME, TITLE, SAL, RESP, DUR)$

• Decomposition attempts to eliminate dependencies within relations by breaking them up into multiple relations.

Integrity Rules

• Specified through keys
• Every relation has a primary key – no duplicate combinations of these attributes
• Key attributes cannot be NULL
• Foreign keys ensure referential integrity
• Domain constraints.
• Explicit constraints can also be specified – e.g. CHECK (SAL < 200000)
Relational Data Languages

- **Relational Algebra**
  - Operational: describes *how* to get the answer
- **Relational Calculus**
  - Declarative: describes *what answer is desired*, not how to obtain it
- **SQL**
  - Practical query language supported by RDBMS products.

Example Instances

- “Sailors” and “Reserves” relations for our examples.
- We’ll use positional or named field notation, assume that names of fields in query results are ‘inherited’ from names of fields in query input relations.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Relational Algebra

- **Basic operations:**
  - *Selection* ($\sigma$) Selects a subset of rows from relation.
  - *Projection* ($\pi$) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.

- **Additional operations:**
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.

- Since each operation returns a relation, operations can be composed (Algebra is “closed”.)

**Projection**

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it. (Why not?)

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

$$\pi_{sname, rating}(S2)$$

<table>
<thead>
<tr>
<th>age</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>55.5</td>
<td></td>
</tr>
</tbody>
</table>

$$\pi_{age}(S2)$$
Selection

- Selects rows that satisfy selection condition.
- No duplicates in result! (Why?)
- Schema of result identical to schema of (single) input relation.
- Result relation can be the input for another relational algebra operation! (Operator composition.)

\[
\sigma_{\text{rating} > 8}(S_2)
\]

\[
\pi_{\text{name, rating}}(\sigma_{\text{rating} > 8}(S_2))
\]

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be union-compatible:
  - Same number of fields.
  - 'Corresponding' fields have the same type.
- What is the schema of the result?

\[
S_1 \cup S_2
\]

\[
S_1 \cap S_2
\]

\[
S_1 - S_2
\]
Cross-Product

- Each row of S1 is paired with each row of R1.
- **Result schema** has one field per field of S1 and R1, with field names ‘inherited’ if possible.
  - **Conflict**: Both S1 and R1 have a field called `sid`.

### Cross-Product

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Renaming operator:** \( \rho (C(l \rightarrow t_1, s \rightarrow t_2), S \times !l) \)
Joins

- **Condition Join**: \( R \bowtie_{c} S = \sigma_{c} (R \times S) \)

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **Result schema** same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.

---

Joins

- **Equi-Join**: A special case of condition join where the condition \( c \) contains only *equalities*.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **Result schema** similar to cross-product, but only one copy of fields for which equality is specified.
- **Natural Join**: Equijoin on *all* common fields.
Relational Calculus

- Comes in two flavours: **Tuple relational calculus** (TRC) and **Domain relational calculus** (DRC).
- Calculus has variables, constants, comparison ops, logical connectives and quantifiers.
  - **TRC**: Variables range over (i.e., get bound to) tuples.
  - **DRC**: Variables range over domain elements (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called **formulas**. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to **true**.

Domain Relational Calculus

- **Query** has the form:
  \[
  \{ x_1,x_2,...,x_n | p(x_1,x_2,...,x_n) \}
  \]
  - **Answer** includes all tuples \((x_1,x_2,...,x_n)\) that make the formula \(p(x_1,x_2,...,x_n)\) true.
  - **Formula** is recursively defined, starting with **simple atomic formulae** (getting tuples from relations or making comparisons of values), and building bigger and better formulas using **logical connectives**.
DRC Formulas

- **Atomic formula:**
  - $\langle x_1, x_2, ..., x_n \rangle \in \text{name}$, or $X \ op \ Y$, or $X \ op \ \text{constant}$
  - $op$ is one of $<, >, =, \leq, \geq, \neq$
- **Formula:**
  - an atomic formula, or
  - $\neg p$, $p \land q$, $p \lor q$, where $p$ and $q$ are formulae, or
  - $\exists X (p(X))$, where variable $X$ is free in $p(X)$, or
  - $\forall X (p(X))$, where variable $X$ is free in $p(X)$
- The use of quantifiers $\exists X$ and $\forall$ is said to **bind** $X$.
  - A variable that is not bound is free.

Free and Bound Variables

- The use of quantifiers $\exists X$ and $\forall$ in a formula is said to **bind** $X$.
  - A variable that is not bound is free.
- Let us revisit the definition of a query:
  \[
  \{ x_1, x_2, ..., x_n \} \ p\{ x_1, x_2, ..., x_n \}
  \]
- There is an important restriction: the variables $x_1, ..., x_n$ that appear to the left of ‘|’ must be the **only** free variables in the formula $p(...)$.
Find all sailors with a rating above 7

\[\{I,N,T,A\} | \{I,N,T,A\} \in \text{ailors} \land T > 7\]

- The condition \(\{I,N,T,A\} \in \text{ailors}\) ensures that the domain variables \(I, N, T\) and \(A\) are bound to fields of the same Sailors tuple.
- The term \(\{I,N,T,A\}\) to the left of ‘|’ (which should be read as such that) says that every tuple \(\{I,N,T,A\}\) that satisfies \(T > 7\) is in the answer.
- Modify this query to answer:
  - Find sailors who are older than 18 or have a rating under 9, and are called ‘Joe’.

Find sailors who’ve reserved all boats

\[\{I,N,T,A\} | \{I,N,T,A\} \in \text{ailors} \land
\forall B,BN,C \left(\neg \{B,BN,C\} \in \text{Boats}\right) \lor
\exists Ir,Br,D \left(\{Ir,Br,D\} \in \text{Reserves} \land I = Ir \land Br = B\right)\]

- Find all sailors \(I\) such that for each 3-tuple \(\{B,BN,C\}\) either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor \(I\) has reserved it.
Unsafe Queries, Expressive Power

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called unsafe.
  - e.g., \( S \neq \{ i \in \text{ailors} \} \)
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC/ TRC; the converse is also true.
- **Relational Completeness**: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

---

Basic SQL Query

- **relation-list** A list of relation names (possibly with a range-variable after each name).
- **target-list** A list of attributes of relations in relation-list
- **qualification** Comparisons (Attr op const or Attr1 op Attr2, where op is one of \(<, >, =, \leq, \geq, \neq\) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are **not** eliminated!
Conceptual Evaluation Strategy

• Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  – Compute the cross-product of \textit{relation-list}.
  – Discard resulting tuples if they fail \textit{qualifications}.
  – Delete attributes that are not in \textit{target-list}.
  – If \textsc{distinct} is specified, eliminate duplicate rows.

• This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute \textit{the same answers}.

Other Operations

• \textsc{union}, \textsc{intersect}, \textsc{except}
• Nested Queries
• \textsc{any}, \textsc{all}, \textsc{in}
• Aggregate operators: \textsc{sum}, \textsc{count}, \textsc{max}, \textsc{min}, \textsc{avg}
• Grouping: \textsc{group by}, \textsc{having}
• Embedded SQL