WEEK 2

Integrity Constraints (ICs)

- **IC**: condition that must be true for *any* instance of the database; e.g., domain constraints.
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!
Primary Key Constraints

- A set of fields is a key for a relation if:
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
    - If part 2 is false, choose a superkey.
    - If there's >1 key for a relation, one of the keys is chosen (by DBA) to be the primary key.
- E.g., S# is a key for S. (What about name?) The set {S#, name} is a superkey.

- Key value cannot be null.
- Functional dependency of other attributes on the PK, exists.

For example, for a relation R(A1, A2, A3, ...., An)

If (A1, A2) is PK then

\[(A1,A2) \rightarrow Ai \quad i = 3, ..., n\]

The maximum size of R is \(|A1| \times |A2|\)
Primary and Candidate Keys in SQL

- Possibly many candidate keys (specified using UNIQUE), one of which is chosen as the primary key.

- "For a given supplier and part, there is a single quantity" vs. "Supplier can supply only one part and same part supplied by different suppliers cannot have the same quantity."

- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

CREATE TABLE SP
(s# char(5),
p# char(5),
qty integer,
PRIMARY KEY (s#, p#))

For a given supplier and part, there is a single quantity" vs. "Supplier can supply only one part and same part supplied by different suppliers cannot have the same quantity."

Used carelessly, an IC can prevent the storage of database instances that arise in practice!

CREATE TABLE SP
(s# char(5),
p# char(5),
qty integer,
PRIMARY KEY (s#),
UNIQUE (p#, qty))

Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to "refer" to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a "logical pointer."

- E.g. S# is a foreign key referring to Supplier:
  - SP(S#, p#: quantity)
  - If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references.
  - Can you name a data model w/o referential integrity?

  - Links in HTML!
Foreign Keys in SQL

- Only p# listed in the P relation should be allowed to supplied by a supplier.

```sql
CREATE TABLE SP
    (s# char(5), p# char(5), qty INTEGER,
     PRIMARY KEY (s#,p#),
     FOREIGN KEY (p#) REFERENCES P,
     FOREIGN KEY (s#) REFERENCES S)
```

Enforcing Referential Integrity

- Consider S and SP; s# in SP is a foreign key that references S.
- What should be done if an SP tuple with a non-existent s# is inserted? (Reject it!)
- What should be done if a S tuple is deleted?
  - Also delete all SP tuples that refer to it.
  - Disallow deletion of a S tuple that is referred to.
  - Set s# in SP tuples that refer to it to a default s#.
  - (In SQL, also: Set s# in SP tuples that refer to it to a special value null, denoting 'unknown' or 'inapplicable'.)
- Similar if primary key of S tuple is updated.
Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.
- We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.
  - An IC is a statement about all possible instances!
  - From example, we know `sname` is not a key, but the assertion that `s#` is a key is given to us.
- Key and foreign key ICs are the most common; more general ICs supported too.

Views

- A view is just a relation, but we store a definition, rather than a set of tuples.

  ```sql
  CREATE VIEW Small_Suppliers
  AS SELECT s#
  FROM SP
  WHERE qty < 500.
  ```

- Views can be dropped using DROP VIEW command.
  - How to handle DROP TABLE if there's a view on the table?
    - DROP TABLE command has options to let the user specify this.
Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given Small_Suppliers, we can find the s# of the suppliers who supply parts with quantity less than 500, without disclosing the type of part.

Chapter 5: Relational Model
Design of DB

Universe of discourse

- Intensional (fixed) Eg. \( S(S\#, \text{Sname}, \text{Status}, \text{city}) \)
- Instance (Extensional) \((x_1, x_2, x_3, x_4)\)

Modified Slides from Database Management Systems, R. Ramakrishnan

The Supplier Relation \( S \)

<table>
<thead>
<tr>
<th>S#</th>
<th>Name</th>
<th>Status</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

Modified Slides from Database Management Systems, R. Ramakrishnan
Relational Model

- **Relation**
  
  Given $D_1, D_2, D_3, \ldots, D_N$ sets (not necessarily distinct). $R$ is a relation on these sets if it is a set of ordered $n$-tuples ($n \leq N$)

  $$<d_1, d_2, d_3, \ldots, d_n>$$

  such that $d_i \in D_i$
  
  - $D_i$’s are the domains of $R$
  - $n$ is the degree of $R$

Relational Model: Summary

- A tabular representation of data.
- Simple and intuitive, currently the most widely used.
- Integrity constraints can be specified by the DBA, based on application semantics. DBMS checks for violations.
  - Two important ICs: primary and foreign keys
  - In addition, we always have domain constraints.
- Powerful and natural query languages exist.
Relational Algebra

Chapter 6

Relational Query Languages

- **Query languages**: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages \(\neq\) programming languages!
  - QLs not expected to be “Turing complete”.
  - QLs not intended to be used for complex calculations.
  - QLS support easy, efficient access to large data sets.
**Formal Relational Query Languages**

Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:

- **Relational Algebra**: More operational, very useful for representing execution plans.
- **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-operational, declarative.)

Understanding Algebra & Calculus is key to understanding SQL, query processing!

---

**Preliminaries**

- A query is applied to relation instances, and the result of a query is also a relation instance.
  - Schemas of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the result of a given query is also fixed! Determined by definition of query language constructs.
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 or 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?)
**Projection**

- Projection: (vertical subset, remove duplicates)
  \[ \pi_{i_1, i_2, \ldots, i_n} (R) \]
- Book notation:
  - \[ R (i_1, i_2, \ldots, i_n) \]
- A convenient notation: if only a few attributes are “projected away”
  - \[ P \{ \text{ALL BUT weight} \} \]

---

**Selection**

- Selection (Horizontal subset): Selects rows that satisfy selection condition.
  \[ \sigma_F(R) \]
  - \[ F = \text{formula (first-order predicate calculus)} \]
  - A. Comparison operators: \[ <, \leq, >, \geq, =, \neq \]
  - B. Logical operators: \[ \text{AND, OR, NOT} \]
- Book notation
  - \[ R \ WHERE \ condition \ F \]
Selection

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

```
S#  SNAME  STATUS  CITY
S1  Smith  20      London
S4  Clark  20      London

σ_{city = 'London'} (S)
```

```
P#  PNAME  COLOR  WEIGHT  CITY
P1  Nut    Red    12.0     London
P5  Cam    Blue   12.0     Paris

σ_{weight < 14.0} (P)
```

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.
- Union
  \[ R \cup S \rightarrow \text{tuples in either } R \text{ or } S \text{ or both} \]
- Set Difference
  \[ R - S \rightarrow \text{tuples in } R \text{ after excluding tuples from } R \cap S \]
- Intersection
  \[ R \cap S = R - (R - S) \rightarrow \text{tuples in both } R \& S \]
- What is the schema of result?
Union, Intersection, Set-Difference

Example (Figure 6.2)

- Book notation
  - $R \text{ UNION } S$
  - $R \text{ MINUS } S$
  - $R \text{ INTERSECT } S$
Cross-Product (Cartesian Product)

- Cartesian product
  \[ R \times S \] concatenation of tuples (attributes names must be distinct)
- Each row of \( R \) is paired with each row of \( S \).
- Result schema has one field per field of \( R \) and \( S \), with field names `inherited` if possible.
- Book notation: \( R \times S \)
Cross-Product

- **Conflict**: Both S1 and R1 have a field called sid.

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\[ R \times S \]

\[ (sid) \quad \text{sid} \]

\[ (sid) \quad \text{sname} \quad \text{rating} \quad \text{age} \]

\[ \rho (R \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), S1 \times R1 \]

**Renaming operator**: \[ \rho (C(1 \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), S1 \times R1) \]

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>45.0</td>
<td>58</td>
<td>58</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>58</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\[ S1 \bowtie S1.\text{sid} < R1.\text{sid} \]

- 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**. (book notation R TIMES S WHERE condition)
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>45.0</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>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 (book notation is \( S1 \ JOIN \ R1 \))

Semi-Join

- **R** has attribute set A
- **S** has attribute set B

Take subset of tuples of \( R \) that participate in JOIN of \( R \& S \). That is a subset of \( R \bowtie_F S \), where \( F \) is a predicate.

\[
\pi_A (R \bowtie_F S) = \pi_A (R \bowtie_F \pi_{A\cap B} (S)) = R \bowtie_F \pi_{A\cap B} (S)
\]
Advantages of Semi-Join

- Less attribute before join
  \[ \Rightarrow \text{less disk-accesses & lesser data transmission between sites to evaluate queries.} \]

Note:

\[ (R \bowtie_F S) \neq (S \bowtie_F R) \]
Division

- Not supported as a primitive operator, but useful for expressing queries like:
  - Find suppliers who supply all parts.
- Let A have 2 fields, x and y; B have only field y:
  - \( A/B = \{\langle x \rangle | \exists \langle x, y \rangle \in A \land \forall \langle y \rangle \in B\} \)
  - i.e., \( A/B \) contains all x tuples (suppliers) such that for every y tuple (part) in B, there is an xy tuple in A.
  - Or: If the set of y values (parts) associated with an x value (supplier) in A contains all y values in B, the x value is in A/B.
- In general, x and y can be any lists of fields; y is the list of fields in B, and \( x \cup y \) is the list of fields of A.

Examples of Division A/B

```
<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
<th>pno</th>
<th>pno</th>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td></td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td></td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td></td>
<td>p4</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td>sno</td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td></td>
<td></td>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
<td></td>
<td></td>
<td>s3</td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A         A/B1     A/B2     A/B3
```

Modified Slides from Database Management Systems, R. Ramakrishnan
Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- Idea: For A/B, compute all x values that are not 'disqualified' by some y value in B.
  - x value is disqualified if by attaching y value from B, we obtain an xy tuple that is not in A.

Disqualified x values:  \[ \pi_x((\pi_x(A) \times B) - A) \]

A/B:  \[ \pi_x(A) \] - all disqualified tuples

Expressing A/B Using Basic Operators

- x = set of attributes of A not in B [(r-s) tuples]
- A/ B = \[ \pi_x(A) - \pi_x((\pi_x(A) \times B) - A) \]

C = A/ B, such that for every \( y_i \in B \) there exists \( x_i \) \( x_i \in C \) iff \((x_i, y_i) \in A\) for every \( y_i \in B\)
**Division (Book Notation)**

**M DIVIDED BY N PER Q**

- Q / N

- Output is those attributes of Q that are in M.
Union

Intersection

Difference

(Natural) Join

a1 b1
a2 b1
a3 b2

b1 c1
b2 c2
b3 c3

a1 b1 c1
a2 b1 c1
a3 b2 c2

(a x)
(b y)
(c z)

Divide

a
b

(x)
(y)
(z)

a

a1 b1
a2 b1
a3 b2

b2 c2
b3 c3

a1 b1 c1
a2 b1 c1
a3 b2 c2

a

(x)
(y)
(z)

a

Modified Slides from Database Management Systems, R. Ramakrishnan 43