Design Objectives

- Obtain the theoretically “best” design (normalize)
  - Remove redundancy and update anomalies
  - Remove nulls
  - Minimize the number of schemes and maximize their size
  - Make the design faithful to the specification
    - preserve information
    - preserve constraints
- Use cost analysis to adjust, if necessary (denormalize)
  - The theoretically “best” is often the best.
  - Adjust for application-dependent time and space considerations.

Update Anomalies

<table>
<thead>
<tr>
<th>Guest</th>
<th>Room Nr</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>1</td>
<td>Kennedy</td>
</tr>
<tr>
<td>G2</td>
<td>1</td>
<td>Kennedy</td>
</tr>
<tr>
<td>G3</td>
<td>1</td>
<td>Kennedy</td>
</tr>
<tr>
<td>G4</td>
<td>5</td>
<td>Green</td>
</tr>
<tr>
<td>G5</td>
<td>3</td>
<td>Carter</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Nixon</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Blue</td>
</tr>
</tbody>
</table>

Modification Anomaly: e.g., change Kennedy to Clinton – must update all redundant values consistently.
Deletion Anomaly: e.g., G4 cancels reservation – the fact that the Green room is room 5 is lost.
Insertion Anomaly: e.g., add a new room (the Gold room) – necessarily yields a null.

Update anomalies and redundancy are two sides of the same coin.
Join Dependencies – Example

Let \( r = A \times B \times C = A \times B \times C \\times A \times C \) 

\[
\begin{array}{ccc}
1 & a & x \\
1 & a & y \\
1 & b & x \\
1 & b & y \\
2 & a & y \\
2 & b & y \\
\end{array}
\]

Observe: \( r = \pi_{AB} r \times \pi_{AC} r \)

Note: \( r \) is the cross product of \( B \) and \( C \) wrt \( A \).

This always holds when we build \( r \) by joining relationship sets in this way. In general, however, if we arbitrarily create a relation, this may not happen. Add \(<2, a, x>\) to \( r \), for example, then \( r \neq \pi_{AB} r \times \pi_{AC} r \) because the join also yields \(<2, b, x>\), which is not in \( r \).

Join Dependencies – Definitions

- A join dependency (JD) denoted \( \times (R_1, \ldots, R_n) \) holds for a relation \( r(R) \) if \( r = \pi_{R_1} r \times \pi_{R_2} r \times \ldots \times \pi_{R_n} r \). (e.g., \( \times (AB, AC) \))

- When \( n = 2 \), we call a JD a Multivalued Dependency (MVD) and write \( X \rightarrow Y \) or \( X \rightarrow Z \) or \( X \rightarrow Y \mid Z \) where \( X = R_1 \cap R_2 \), \( Y = R_1 - R_2 \), and \( Z = R_2 - R_1 \). (e.g., \( A \rightarrow B \) or \( A \rightarrow C \) or \( A \rightarrow B \mid C \))
Redundancy

- We (usually) want to remove redundancy.
  - Space savings: no need to store duplicate values.
  - Time savings: no need for extra processing to avoid update anomalies.

- Basic Idea:
  - A data value $v$ is redundant if we can “erase” $v$ and then from the remaining data values and the constraints uniquely determine $v$.
  - The constraints we consider: FDs, MVDs, JDs.

FD Redundancy

If $B \rightarrow C$, the circled data values are redundant.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
MVD Redundancy

If \( A \rightarrow B | C \), the circled data values are redundant.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

JD Redundancy

If \( |x|(AB, BC, AC) \), the circled data values are redundant.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 10 - 9

**Nulls**

### Incongruent

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>_</td>
</tr>
<tr>
<td>3</td>
<td>_</td>
</tr>
<tr>
<td>4</td>
<td>_</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>_</td>
</tr>
</tbody>
</table>

### Congruent

<table>
<thead>
<tr>
<th>A</th>
<th>A'</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 10 - 10

**Minimize the Number of Schemes**

- Combine object and relationship sets
- BUT only if there is no possibility of:
  - redundancy
  - nulls
- Preserve information and constraints
Sample Combinations with Redundancy

\[ A \rightarrow B \]
\[ B \rightarrow C \]
\[ A \rightarrow C \]

\[
\begin{array}{c|ccc|c|c|ccc}
\hline
& A & B & C & D & E & F & G & H \\
\hline
A & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
B & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
\hline
\end{array}
\]

Sample Combinations with No Redundancy

\[ A \rightarrow B \]
\[ B \rightarrow C \]
\[ A \rightarrow C \]

\[
\begin{array}{c|ccc|c|c|ccc}
\hline
& A & B & C & D & E & F & G & H \\
\hline
A & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\
B & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 2 \\
\hline
\end{array}
\]
Canonical ORM Hypergraph

- Congruent
- Nonrecursive
- Head and Tail Reduced
- Object-Set Reduced (Lexical & Merged)
- Non-FD-edge Reduced
- Embedded-FD Reduced
- Separately Linked (Semantically Separate Eq. Classes)
- Minimally Consolidated
- Semantically Head Consistent

---

Semantically Separate Eq. Class

<table>
<thead>
<tr>
<th>Room</th>
<th>Room Name</th>
<th>Prior Room Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Kennedy</td>
<td>Nixon</td>
</tr>
<tr>
<td>R2</td>
<td>Nixon</td>
<td>Kennedy</td>
</tr>
<tr>
<td>R3</td>
<td>Carter</td>
<td>Carter</td>
</tr>
<tr>
<td>R4</td>
<td>Blue</td>
<td>Green</td>
</tr>
<tr>
<td>R5</td>
<td>Green</td>
<td>Blue</td>
</tr>
</tbody>
</table>
## Semantically Head Consistent

- **IsDoing(Guest, Activity)**
- **NextDoes(Guest, Activity)**

<table>
<thead>
<tr>
<th>Guest</th>
<th>Current Activity</th>
<th>Next Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>4-Wheeling</td>
<td>Hot Tub</td>
</tr>
<tr>
<td>G2</td>
<td>Horse Riding</td>
<td>4-Wheeling</td>
</tr>
<tr>
<td>G3</td>
<td>Hot Tub</td>
<td>Horse Riding</td>
</tr>
</tbody>
</table>

## Scheme Synthesis

- **Input**: a canonical ORM hypergraph.
- **Output**: a set of relation schemes with keys.

- Equivalence classes (including trivial equivalence classes) with FDs – each equivalence-class element is a key
- Nontrivial equivalence classes without FDs – each equivalence-class element is a key
- Non-FD edges – all the attributes together constitute a composite key
- Stand-alone object sets – the lone attribute is a key
Scheme Synthesis – Example

Case 1: \(A \rightarrow B \rightarrow C \rightarrow D \rightarrow F \rightarrow D\)
Case 2: \(E \rightarrow F \rightarrow G\)
Case 3: \(B \rightarrow E\)
Case 4: \(H\)

Inclusion Dependencies – Generation of Foreign Keys

- Input: a canonical ORM hypergraph and a set of schemes generated by the scheme-synthesis algorithm
- Output: a set of inclusion dependencies
- Generalization/specialization pairs
- Multiple appearances
- Subset constraints among relationship sets
Inclusion Dependencies – Example

Inclusion dependencies:

- Database scheme: \( q(A, B), r(A, C), s(D, E) \)
- Inclusion dependencies:
  - Case 1: \( q[A] = s[D], r[A] = s[D] \)
  - Case 2: \( q[A] = r[A] \)
  - Case 3: \( r[A, C] = s[D, E] \)

B&B Example – ORM Diagram

Database scheme: \( q(A, B), r(A, C), s(D, E) \)

- Inclusion dependencies:
  - Case 1: \( q[A] = s[D], r[A] = s[D] \)
  - Case 2: \( q[A] = r[A] \)
  - Case 3: \( r[A, C] = s[D, E] \)

B&B Example – ORM Diagram

- Database scheme: \( q(A, B), r(A, C), s(D, E) \)
- Inclusion dependencies:
  - Case 1: \( q[A] = s[D], r[A] = s[D] \)
  - Case 2: \( q[A] = r[A] \)
  - Case 3: \( r[A, C] = s[D, E] \)
B&B Example – ORM Hypergraph

B&B Example – Congruent
B&B Example – Canonical

B&B Example – Generated Database Scheme

Room(RoomNr, RoomName, NrBeds, Cost)

Guest(GuestNr, GuestName, StreetNr, City)

Reservation(GuestNr, RoomNr, ArrivalDate, NrDays)

Room[RoomNr] Ↄ Reservation[RoomNr]
Guest[GuestNr] = Reservation[GuestNr]
Keys and FDs

Let U be a set of object sets, and let F be a set of FDs over U. Let R ⊆ U be a relation scheme. A subset K of R (K need not be a proper subset of R) is a superkey of R if K → R ∈ F+ and is a candidate key of R if there does not exist a proper subset K′ of K such that K′ → R ∈ F+.

Example: U = ABCDE and F = {A → B, B → A, AB → C, D → BC}.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Candidate Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>A, B</td>
</tr>
<tr>
<td>CE</td>
<td>CE</td>
</tr>
<tr>
<td>ABCD</td>
<td>D</td>
</tr>
<tr>
<td>ABCDE</td>
<td>DE</td>
</tr>
</tbody>
</table>

Generated Keys are Candidate Keys (Thm 10.3)

Superkeys:

A → B

Minimal Keys: Suppose not, then tail reducible.
Generated Schemes have no Potential Redundancy* (Thm 10.4)

Canonical hypergraphs do not have edges that cause redundancy.

*Except (possibly) for schemes that have a nontrivial, inextricably embedded JD.

Inextricably Embedded JDs

ABCD has redundancy within its ABC component, but cannot be decomposed losslessly into ABC and any other scheme.
No Nulls (Thm 10.5)
Canonical hypergraphs are congruent.

Synthesis Preserves Information (Thm 10.6)
Generated Scheme: \[
\begin{array}{ccc}
A & B & C \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Original Object and Relationship Sets:
\[
\begin{array}{ccc}
A & B & C \\
2 & 3 & 1 \\
3 & 2 & 1 \\
\end{array}
\]

Join/Project always returns the original.
Minimal Number of Schemes*
(Thm 10.7)

A → B → C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A → Q → C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

*Without potential redundancy or nulls and assuming more than one tuple per relation is possible.

---

Minimal Number of Attributes in Schemes (Prop. 10.1 & 10.2)

- Hard to guarantee no fewer:
  - Do we count replacing two attributes, say Name and Address, by a single combined attribute, say Name-Address?
  - Perhaps a different way of deriving attributes for schemes might yield fewer.
- Can guarantee:
  - Proposition 10.1: We can’t make fewer by lexicalization or by one-to-one merges of nonlexical object sets.
  - Proposition 10.2: We can’t make fewer by consolidation within equivalence classes.
Synthesis Preserves Constraints
(Thms 10.8 & 10.9)

- **Theorem 10.8:** We keep all constraints of the canonical hypergraph.
  - Some constraints become key constraints.
  - Some constraints become foreign-key constraints.
  - General constraints given or generated, including generated co-occurrence constraints for embedded FD reductions, remain intact.

- **Theorem 10.9:** Sometimes all constraints become key constraints or foreign-key constraints.
  - We can represent these constraints in SQL DDL.
  - Database systems efficiently check these constraints for us (no extra code need be written to check these constraints).

Cost Analysis
Rule-of-Thumb Guidelines

- As a guide, consider denormalizing if:
  - nulls are applicable but unknown (e.g., address information)
  - redundancy is minimal and update anomalies are not expected (e.g., StreetNr City State → Zip)
  - replicated objects are large (e.g., images in View)
  - join frequencies are very high when compared to updates (e.g., approximate costs in foreign currencies)

- Using actual application characteristics, estimate space and time requirements for various possibilities and compare costs.
Cost Estimation – B & B (Space)

Assume: 5 rooms, 100 reservations, and 80 guests.

Case 1: Room(RoomNr, RoomName, NrBeds, Cost)
        Guest(GuestNr, GuestName, StreetNr, City)
        Reservation(GuestNr, RoomNr, ArrivalDate, NrDays)

vs. Case 2: Room(RoomNr, RoomName, NrBeds, Cost)
           Reservation(GuestNr, RoomNr, ArrivalDate, NrDays)

vs. Case 3: Guest(GuestNr, GuestName, StreetNr, City)
            Reservation(GuestNr, RoomNr, ArrivalDate, NrDays,
                        RoomName, NrBeds, Cost)

vs. Case 4: Reservation(GuestNr, GuestName, StreetNr, City,
                RoomNr, ArrivalDate, NrDays,
                RoomName, NrBeds, Cost)

\[\begin{array}{c|c|c}
\text{Case} & \text{Cost} & \text{Cost} \\
\hline
1 & 5 \times 4 = 20 & 80 \times 4 = 320 \\
\hline
\text{vs.} & 740 & \\
\hline
2 & 5 \times 4 = 20 & 100 \times 4 = 400 \\
\hline
\text{vs.} & 740 & \\
\hline
3 & 80 \times 4 = 320 & 100 \times 7 = 700 \\
\hline
\text{vs.} & 740 & \\
\hline
4 & 100 \times 10 = 1000 & \\
\hline
\end{array}\]

Cost Estimation – B & B (Time)

Case 1: Room(RoomNr, RoomName, NrBeds, Cost)
        Guest(GuestNr, GuestName, StreetNr, City)
        Reservation(GuestNr, RoomNr, ArrivalDate, NrDays)

vs. Case 2: Room(RoomNr, RoomName, NrBeds, Cost)
           Reservation(GuestNr, GuestName, StreetNr, City,
                        RoomNr, ArrivalDate, NrDays)

Most important queries/updates:
1. (40%) What rooms are available?
2. (30%) Make a reservation.
3. (10%) Change a reservation.
4. (10%) Cancel a reservation.
5. (the rest) Miscellaneous.

Assume indexed on primary keys.

1. Case 1 & 2: Retrieve reservations that could overlap the requested date and determine room availability. (Case 1 insignificantly better.)

...
B & B – Semantic Change?

Case 1: Room(RoomNr, RoomName, NrBeds, Cost)
         Guest(GuestNr, GuestName, StreetNr, City)
         Reservation(GuestNr, RoomNr, ArrivalDate, NrDays)
vs. Case 2: Room(RoomNr, RoomName, NrBeds, Cost)
         Reservation(GuestNr, GuestName, StreetNr, City,
                     RoomNr, ArrivalDate, NrDays)

Most important queries/updates: Assume indexed on primary keys.
... 2. (30%) Make a reservation. 2. Case 1: Insert tuple in Reservation (1 read & 1 write) and insert tuple in Guest, if necessary, (1 read and (usually) 1 write).
... Case 2: Insert tuple in Reservation (1 read & 1 write); check duplicate guest information (read file, or add secondary index).

(Developer) Do we really need to check duplicate guest information?
(Proprietor) Hmmm, maybe not; it doesn’t matter if it is different.
(Developer) Does a guest always need the same guest number?
(Proprietor) Not really; there are no guest numbers in our manual system.
(Developer) Aha! Great, this really lets us save – watch this.

Unique GuestNr in Reservation

Observe that we have a new equivalence class:
   { (GuestNr), (RoomNr, ArrivalDate) }
And thus a new generated database scheme:
   Reservation(GuestNr, GuestName, City, Street, RoomNr, ArrivalDate, NrDays)
   Room(RoomNr, RoomName, NrBeds, Cost)
Nested Schemes

- Flat schemes often have replicated data values.
- Nested schemes allow us to collapse some of these replicated data values.

<table>
<thead>
<tr>
<th>NrBeds</th>
<th>RoomNr</th>
<th>NrBeds</th>
<th>(RoomNr)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Redundancy in Nested Schemes

- The redundancy definition is the same as for flat relations.
- If a value change causes a constraint violation, the value is redundant.
Algorithm 10.3

Input: a canonical, acyclic, binary ORM hypergraph.
Output: a set of nested schemes with no potential redundancy.

Repeat
   Mark an unmarked node in as the first attribute in a new nested scheme.
   While an unmarked edge is incident on a marked node A:
      Mark the edge.
      If A → B: Add B with A; Mark B.
      If A ⇑ B: Add B with A; Mark B if all B’s incident edges are marked.
      If A ← B: Nest B under A; Mark B.
   Else (A − B): Nest B under A; Mark B if all B’s incident edges are marked.
Until all nodes have been marked

Nested Scheme Generation Example

1. NrBeds (RoomNr, RoomName, Cost (View)* (GuestNr, GuestName)* )*
2. RoomNr, RoomName, Cost, NrBeds (View)* (GuestNr, GuestName)*
3. GuestNr GuestName RoomNr
   RoomNr, RoomName, Cost, NrBeds (View)*
Redundancy Prevention

This replication ... ... causes this redundancy.

Generalization of Algorithm 10.3 for N-ary Relationship Sets

- “Composite nodes” can be treated as a node (in Algorithm 10.3).
  - B C (A)* (D)*
  - D (B C)*; A B C
- NNF (see Exercise 10.35), basically:
  - Schemes should be constructed along hypergraph paths.
  - Schemes should not violate the natural 1-many hierarchical structure.
Guidelines for Selecting Nested Schemes

- Select “important nodes” as the initial nodes for nested-scheme generation – e.g., Scheme 3 or 2 in earlier Bed-&-Breakfast example.
- Maximize the size of schemes.
  - Select nodes included in the largest number of FD closures (i.e., when Algorithm 10.3 requires a new node to be arbitrarily selected, compute the set of unmarked nodes in the FD closure of every unmarked node and choose a node included in at least as many sets as any other node) – e.g., Scheme 1 in earlier example.
  - When possible, adjust these generated maximal schemes by placing the most important node first – e.g., Scheme 2 in earlier example.

Cost Analysis for Nested Schemes

- Nested schemes impose variable-length records.
- Recall variable-length record implementation strategies:
  - Reserve enough space for maximum.
  - Chain each nested record.
  - Reserve space for the expected number and chain the rest.
- Insertion, deletion, modification, retrieval tradeoffs.