Lecture 13: Design Theory (cont’d), Indexing
Database Design Process

Conceptual Model:

- Conceptual Schema
- Physical storage details

Relational Model:
- Tables + constraints
- And also functional dep.

Normalization:
- Eliminates anomalies
Relational Schema Design (or Logical Design)

How do we do this systematically?

• Start with some relational schema

• Find out its *functional dependencies* (FDs)

• Use FDs to *normalize* the relational schema
Functional Dependencies (FDs)

**Definition**

If two tuples agree on the attributes

\[ A_1, A_2, \ldots, A_n \]

then they must also agree on the attributes

\[ B_1, B_2, \ldots, B_m \]

Formally:

\[ A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \]
**Definition**  \( A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n \) holds in \( R \) if:

\[
\forall t, t' \in R, (t.A_1 = t'.A_1 \land \ldots \land t.A_m = t'.A_m \rightarrow t.B_1 = t'.B_1 \land \ldots \land t.B_n = t'.B_n )
\]

---

\( R \) | \( A_1 \) | \( \ldots \) | \( A_m \) | \( B_1 \) | \( \ldots \) | \( B_n \)  
---

| \( t \) | | | | | |
| \( t' \) | | | | | |

**Text:**

if \( t, t' \) agree here then \( t, t' \) agree here
Closure Algorithm

\[ X = \{ A_1, \ldots, A_n \} \]

Repeat until \( X \) doesn’t change do:

\[
\text{if } B_1, \ldots, B_n \rightarrow C \text{ is a FD and } B_1, \ldots, B_n \text{ are all in } X \text{ then add } C \text{ to } X.
\]

Example:

1. name \( \rightarrow \) color
2. category \( \rightarrow \) department
3. color, category \( \rightarrow \) price

\[ \{ \text{name, category} \}^+ = \{ \text{name, category, color, department, price} \} \]

Hence: name, category \( \rightarrow \) color, department, price
Keys

• A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for any other attribute $B$, we have $A_1, \ldots, A_n \rightarrow B$

• A **key** is a minimal superkey
  – A superkey and for which no subset is a superkey
Computing (Super)Keys

- For all sets $X$, compute $X^+$
- If $X^+ = \text{[all attributes]}$, then $X$ is a superkey
- Try only the minimal $X$’s to get the key
Eliminating Anomalies

Main idea:

• $X \rightarrow A$ is OK if $X$ is a (super)key

• $X \rightarrow A$ is not OK otherwise
  – Need to decompose the table, but how?

Boyce-Codd Normal Form
There are no “bad” FDs:

**Definition.** A relation R is in BCNF if:

Whenever X → B is a non-trivial dependency, then X is a superkey.

Equivalently:

**Definition.** A relation R is in BCNF if:

∀ X, either X⁺ = X or X⁺ = [all attributes]
BCNF Decomposition Algorithm

Normalize(R)

find X s.t.: X ≠ X+ and X+ ≠ [all attributes]

if (not found) then “R is in BCNF”

let Y = X+ - X; Z = [all attributes] - X+

decompose R into R1(X ∪ Y) and R2(X ∪ Z)

Normalize(R1); Normalize(R2);
Decompositions in General

\[ R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p) \]

\[ S_1(A_1, \ldots, A_n, B_1, \ldots, B_m) \quad S_2(A_1, \ldots, A_n, C_1, \ldots, C_p) \]

\[ S_1 = \text{projection of } R \text{ on } A_1, \ldots, A_n, B_1, \ldots, B_m \]
\[ S_2 = \text{projection of } R \text{ on } A_1, \ldots, A_n, C_1, \ldots, C_p \]
Lossless Decomposition

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

- **Table 1:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
</tr>
</tbody>
</table>

- **Table 2:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Camera</td>
</tr>
</tbody>
</table>

CS 457 - Fall 2018
Lossy Decomposition

What is lossy here?

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Camera</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>
Lossy Decomposition

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>
Decomposition in General

$R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p)$

Let:

$S_1 = \text{projection of } R \text{ on } A_1, \ldots, A_n, B_1, \ldots, B_m$

$S_2 = \text{projection of } R \text{ on } A_1, \ldots, A_n, C_1, \ldots, C_p$

The decomposition is called **lossless** if $R = S_1 \bowtie S_2$

Fact: If $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ then the decomposition is lossless

It follows that every BCNF decomposition is lossless
Schema Refinements
= Normal Forms

• 1st Normal Form = all tables are flat
  – No nested attributes…
• 2nd Normal Form = obsolete
• Boyce Codd Normal Form = no bad FDs
• 3rd Normal Form = see book
  – BCNF is lossless but can cause loss of ability to check some FDs
  – 3NF fixes that (is lossless and dependency-preserving), but some tables might not be in BCNF – i.e., they may have redundancy anomalies
How to split relations in SQL?
Views

• A view in SQL =
  – A table computed from other tables, s.t., whenever the base tables are updated, the view is updated too

• More generally:
  – A view is derived data that keeps track of changes in the original data

• Compare:
  – A function computes a value from other values, but does not keep track of changes to the inputs
A Simple View

Create a view that returns for each store the prices of products purchased at that store

CREATE VIEW StorePrice AS
SELECT DISTINCT x.store, y.price
FROM Purchase x, Product y
WHERE x.product = y.pname

This is like a new table StorePrice(store, price)
We Use a View Like Any Table

- A "high end" store is a store that sell some products over 1000.
- For each customer, return all the high end stores that they visit.

```
SELECT DISTINCT u.customer, u.store
FROM Purchase u, StorePrice v
WHERE u.store = v.store
AND v.price > 1000
```
Types of Views

- **Virtual views**
  - Computed only on-demand – slow at runtime
  - Always up to date

- **Materialized views**
  - Pre-computed offline – fast at runtime
  - May have stale data (must recompute or update)
  - Indexes *are* materialized views

- A key component of physical tuning of databases is the selection of materialized views and indexes
Vertical Partitioning

Resumes

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Resume</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
<td>Clob1…</td>
<td>Blob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
<td>Clob2…</td>
<td>Blob2…</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
<td>Clob3…</td>
<td>Blob3…</td>
</tr>
<tr>
<td>432432</td>
<td>Ann</td>
<td>Portland</td>
<td>Clob4…</td>
<td>Blob4…</td>
</tr>
</tbody>
</table>

T1

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Huston</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T2

<table>
<thead>
<tr>
<th>SSN</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Clob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Clob2…</td>
</tr>
</tbody>
</table>

T3

<table>
<thead>
<tr>
<th>SSN</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Blob1…</td>
</tr>
<tr>
<td>345345</td>
<td>Blob2…</td>
</tr>
</tbody>
</table>

**T2**. SSN is a key *and* a foreign key to **T1**. SSN. Same for **T3**. SSN.
Vertical Partitioning

CREATE VIEW Resumes AS
SELECT T1.ssn, T1.name, T1.address,
       T2.resume, T3.picture
FROM T1, T2, T3
WHERE T1.ssn = T2.ssn AND T1.ssn = T3.ssn
Vertical Partitioning

CREATE VIEW Resumes AS
SELECT T1.ssn, T1.name, T1.address,
     T2.resume, T3.picture
FROM T1, T2, T3
WHERE T1.ssn = T2.ssn AND T1.ssn = T3.ssn

SELECT address
FROM Resumes
WHERE name = 'Sue'
Vertical Partitioning

CREATE VIEW Resumes AS
SELECT T1.ssn, T1.name, T1.address,
	T2.resume, T3.picture
FROM T1, T2, T3
WHERE T1.ssn = T2.ssn AND T1.ssn = T3.ssn

SELECT address
FROM Resumes
WHERE name = ‘Sue’

Original query:
SELECT T1.address
FROM T1, T2, T3
WHERE T1.name = ‘Sue’
AND T1.SSN = T2.SSN
AND T1.SSN = T3.SSN
Vertical Partitioning

CREATE VIEW Resumes AS
SELECT T1.ssn, T1.name, T1.address,
T2.resume, T3.picture
FROM T1, T2, T3
WHERE T1.ssn = T2.ssn
AND T1.ssn = T3.ssn

SELECT address
FROM Resumes
WHERE name = 'Sue'

Final query:
SELECT T1.address
FROM T1
WHERE T1.name = 'Sue'

Modified query:
SELECT T1.address
FROM T1, T2, T3
WHERE T1.name = 'Sue'
AND T1.SSN = T2.SSN
AND T1.SSN = T3.SSN
Vertical Partitioning Applications

1. Advantages
   - Speeds up queries that touch only a small fraction of columns
   - Single column can be compressed effectively, reducing disk I/O

1. Disadvantages
   - Updates are expensive!
   - Need many joins to access many columns
   - Repeated key columns add overhead
Horizontal Partitioning

Customers

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Houston</td>
</tr>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
</tr>
<tr>
<td>234234</td>
<td>Ann</td>
<td>Portland</td>
</tr>
<tr>
<td>--</td>
<td>Frank</td>
<td>Calgary</td>
</tr>
<tr>
<td>--</td>
<td>Jean</td>
<td>Montreal</td>
</tr>
</tbody>
</table>

CustomersInHouston

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>234234</td>
<td>Mary</td>
<td>Houston</td>
</tr>
</tbody>
</table>

CustomersInSeattle

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>345345</td>
<td>Sue</td>
<td>Seattle</td>
</tr>
<tr>
<td>345343</td>
<td>Joan</td>
<td>Seattle</td>
</tr>
</tbody>
</table>
Horizontal Partitioning

CREATE VIEW Customers AS
  CustomersInHouston
    UNION ALL
  CustomersInSeattle
    UNION ALL
  ...
Horizontal Partitioning

```
SELECT name
FROM Customers
WHERE city = 'Seattle'
```

Which tables are inspected by the system?
-- All
Horizontal Partitioning Applications

• Performance optimization
  – Especially for data warehousing
  – E.g. one partition per month

• Distributed and parallel databases

• Data integration
CS 457: Database Management Systems

Indexing
Basic Access Method: Heap File

API

• **Create** or **destroy** a file
• **Insert** a record
• **Delete** a record with a given rid (rid)
  – rid: unique tuple identifier (more later)
• **Get** a record with a given rid
  – Not necessary for sequential scan operator
  – But used with indexes
• **Scan** all records in the file
But Often Also Want….

- **Scan** all records in the file that match a predicate of the form `attribute op value`
  - Example: Find all students with GPA > 3.5

- Critical to support such requests efficiently
  - Why read all data from disk when we only need a small fraction of that data?

- This lecture and next, we will learn how
Searching in a Heap File

File is **not sorted** on any attribute

\[ \text{Student}(\text{sid: int, age: int, } \ldots) \]

<table>
<thead>
<tr>
<th>30</th>
<th>18 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>19</td>
</tr>
<tr>
<td>80</td>
<td>19</td>
</tr>
<tr>
<td>60</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
</tr>
</tbody>
</table>

1 page

1 record
Heap File Search Example

- 10,000 students
- 10 student records per page
- **Total number of pages: 1,000 pages**
- Find student whose sid is 80
  - Must read on average 500 pages
- Find all students older than 20
  - Must read all 1,000 pages
- Can we do better?
Sequential File

File sorted on an attribute, usually on primary key

Student(sid: int, age: int, ...)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
Sequential File Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Could do binary search, read \( \log_2(1,000) \approx 10 \) pages
- Find all students older than 20
  - Must still read all 1,000 pages
- Can we do even better?

- Note: Sorted files are inefficient for inserts/deletes
Outline

• Index structures
• Hash-based indexes
• B+ trees

Today

Next time
Indexes

• **Index**: data structure that organizes data records on disk to optimize selections on the *search key fields* for the index

• An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given search key value $k$

• Indexes are also access methods!
  – So they provide the same API as we have seen for Heap Files
  – And efficiently support scans over tuples matching predicate on search key
Indexes

- **Search key** = can be any set of fields
  - not the same as the primary key, nor a key
- **Index** = collection of data entries
- **Data entry** for key k can be:
  - The actual record with key k
    - In this case, the index is also a special file organization
    - Called: “indexed file organization”
  - (k, RID)
  - (k, list-of-RIDs)
Different Types of Files

• For the data inside base relations:
  – **Heap file** (tuples stored without any order)
  – **Sequential file** (tuples sorted some attribute(s))
  – **Indexed file** (tuples organized following an index)

• Then we can have additional **index files** that store (key, rid) pairs

• Index can also be a “**covering index**”
  – Index contains (search key + other attributes, rid)
  – Index suffices to answer some queries
Primary Index

- Primary index determines location of indexed records
- Dense index: sequence of (key, rid) pairs (record level)
Primary Index

- **Sparse** index (page level)