7042 Datenbanken

Prof. O. Nierstrasz

Wintersemester 1997/98
Table of Contents

1. 7042 Datenbanken 1
   Schedule 2
   What you will be expected to learn: 3
   Definitions? 4
   In Search of a Definition ... 5
   What is a Database? 6
   Example 7
   Why Do We Need Database Systems? 8
   When Do We Need Database Systems? 9
   When Do We Not Need Database Systems? 10
   Kinds of Database Systems 11
   Data Models 12
   E-R Model 13
   Relational Model 14
   OO Model 15
   Schemas and Instances 16
   The Three Schema Architecture 17
   Data Independence 18
   Architecture 19
   Implementation issues 20
   Classification of Database Systems 21
   Summary 22

2. The Entity-Relationship Model 23
   Entities and Attributes 24
   Entities & Attributes 25
   Attributes 26
   Relationships 27
   Relationships and relationship sets 28
   Attributes vs. Entities 29
   Mapping Constraints 30
   Existence Constraints 31
   E-R Diagrams — Example 32
   One-to-one, one-to-many 33
   Ternary Relationships 34
   Roles 35
   Summary 36

3. Entity-Relationship Diagrams 37
   Primary Keys 38
   Strong & Weak Entity Sets 39
   Relationship keys 40
   ER Diagrams 41
   Generalisation 42
   Aggregation 43
   Reducing E-R Diagrams to Tables 44
   Reducing Weak Entity Sets 45
   Design Decisions 46
   Summary 47

4. The Relational Model 48
   History 49
   Example: The Bank Database Schema 50
   Relational Databases 51
   Notation 52
   Schemas and instances 53
   Common attributes 54
   Query Languages 55
   Relational Algebra 56
   Example: The Bank Database 57
   Select 58
   Project 59
   Cartesian product 60
   Renaming 61
   Union 62
   Set-difference 63
   Summary 64

5. The Relational Model (Continued) 65
   Derived operators 66
   Example: The Bank Database 67
   Intersection 68
   Natural Join 69
   Division 70
   Insertions and Deletions 71
   Updates 72
   The Tuple Relational Calculus 73
   Examples 74
   Safety 75
   The Domain Relational Calculus 76
   Examples 77
   Summary 78

6. SQL 79
   SQL 80
   SQL Syntax Summary: Queries 81
   SQL Syntax Summary: DDL 82
   Basic Structure 83
   Set Operations: Union 84
   Set Operations: Intersection and Minus 85
   Predicates and Joins 86
   Logical Connectives 87
   String matching 88
   Tuples 89
   Tuple Variables 90
   Set comparison 91
   Set containment 92
   Testing for empty relations 93
   Ordering 94
   Summary 95
# Table of Contents

## 7. SQL, QBE and Quel
- Aggregate Functions: 97
- Group Predicates: 98
- Restrictions: 99
- Updates: 100
- Null Values: 101
- Views: 102
- Data Definition: 103
- Summary: 104
- Aggregate Functions: 98
- Group Predicates: 99
- Restrictions: 101
- Updates: 102
- Null Values: 103
- Views: 104
- Data Definition: 105
- Summary: 106
- Query-by-example: 107
- Simple queries: 108
- Variable unification: 109
- Set Difference: 110
- Result Relations: 111
- Other features: 112
- Quel: 113
- Differences between Quel and SQL: 114
- Queries: 115
- Other Features: 116
- Summary: 117

## 8. Integrity Constraints
- Domain Constraints: 118
- Foreign keys: 119
- Referential Integrity: 120
- Referential Integrity in SQL: 121
- Functional Dependencies: 122
- Example FDs: 123
- Example FDs in the Bank Database: 124
- Closure of a set of FDs: 125
- Example — using closures: 126
- Derived Rules: 127
- Closure of an attribute set: 128
- Finding Keys: 129
- Example — finding keys: 130
- Canonical Covers: 132
- Assertions: 133
- Triggers: 134
- Summary: 135
- Data Dictionary Storage: 167
- Buffer Management: 168
- Buffer Management: 169
- Summary: 170

## 9. Database Design
- Example: 136
- Repetition of Information: 137
- Lossy Joins: 138
- Lossy Joins: 139
- Normalisation: 140
- BCNF Decomposition Algorithm: 141
- Shortfalls of BCNF: 142
- Third Normal Form: 143
- 3NF Decomposition Algorithm: 144
- BCNF vs. 3NF: 145
- Summary: 146
- Physical Storage Media: 154
- Disk Storage: 155
- File Organisation: 156
- Fixed-length records: 157
- Insertions and deletions: 158
- Variable length records: 159
- Byte String Representation: 160
- Fixed-Length Representation: 161
- Anchor/overflow block organization: 162
- Organizing Records into Blocks: 163
- Sequential Files: 164
- Mapping Relational Data to Files: 165
- Non-serializable Schedules: 196
- Conflict Serializability: 197
- Serializing Schedules: 198
- Testing for Conflict Serializability: 199
- Sorting Precedence Graphs: 200
- Locks: 201

## 10. File and System Structure
- Example: 152
- Physical Storage Media: 155
- Disk Storage: 156
- File Organisation: 157
- Fixed-length records: 158
- Insertions and deletions: 159
- Variable length records: 160
- Byte String Representation: 161
- Fixed-Length Representation: 162
- Anchor/overflow block organization: 163
- Organizing Records into Blocks: 164
- Sequential Files: 165
- Mapping Relational Data to Files: 166

## 11. Indexing and Hashing
- Basic Concepts: 137
- Indexing: 138
- Dense and sparse indices: 139
- Indices: 140
- Secondary indices: 141
- B+ Tree Insertion Files: 142
- B+ Tree Insertion Files: 143
- B+ Tree Deletions: 144
- B+ Tree Index Files: 145
- B+ Tree Index Files: 146
- Hash Functions: 147
- Static hash functions: 148
- Dynamic hash functions: 149
- Dynamic Hashing example: 150
- Hashing vs. Indexing: 151
- Summary: 152

## 12. Transactions and Concurrency Control
- Transactions: 188
- Transaction States: 189
- Aborted Transactions: 190
- Recovery Logs: 191
- Deferred Database Modification: 192
- Immediate Database Modification: 193
- Log Record Buffering: 194
- Concurrent and Serializable Schedules: 195
- Non-serializable Schedules: 196
- Conflict Serializability: 197
- Serializing Schedules: 198
- Testing for Conflict Serializability: 199
- Sorting Precedence Graphs: 200
- Locks: 201
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-phase Locking Protocol</td>
<td>202</td>
</tr>
<tr>
<td>Locking Protocols</td>
<td>203</td>
</tr>
<tr>
<td>Deadlock</td>
<td>204</td>
</tr>
<tr>
<td>Deadlock Recovery</td>
<td>205</td>
</tr>
<tr>
<td>Summary</td>
<td>206</td>
</tr>
<tr>
<td><strong>13. Query Processing</strong></td>
<td></td>
</tr>
<tr>
<td>Equivalence of Expressions</td>
<td>208</td>
</tr>
<tr>
<td>Selection</td>
<td>209</td>
</tr>
<tr>
<td>Conjunctions</td>
<td>210</td>
</tr>
<tr>
<td>Projections</td>
<td>211</td>
</tr>
<tr>
<td>Natural Joins</td>
<td>212</td>
</tr>
<tr>
<td>Other transformations</td>
<td>213</td>
</tr>
<tr>
<td>Estimation of Query-Processing Cost</td>
<td>214</td>
</tr>
<tr>
<td>Joins</td>
<td>215</td>
</tr>
<tr>
<td>Indices</td>
<td>216</td>
</tr>
<tr>
<td>Query Strategies Using Indices</td>
<td>217</td>
</tr>
<tr>
<td>Join Strategies</td>
<td>218</td>
</tr>
<tr>
<td>Simple vs. Block-oriented Iteration</td>
<td>219</td>
</tr>
<tr>
<td>Merge Join (Sorted Join Attributes)</td>
<td>220</td>
</tr>
<tr>
<td>Computing Joins with Indices</td>
<td>221</td>
</tr>
<tr>
<td>Summary</td>
<td>222</td>
</tr>
</tbody>
</table>
1. 7042 Datenbanken

Lecturer: Prof. O. Nierstrasz  
Schützenmattstr. 14/103, Tel. 631.4618

Secr.: Frau I. Huber, Tel. 631.4692

Assistant: J.-G. Schneider  
S. Kneubühl

WWW: http://iamwww.unibe.ch/~scg/Lectures/

Principle Text:

Supplementary texts:
Schedule

1.  10.22  Introduction
2.  10.29  E-R Model
3.  11.05  E-R Model, continued
4.  11.12  The Relational Model
5.  11.19  The Relational Model, continued
6.  11.26  Query Languages
7.  12.03  Query Languages
8.  12.10  Integrity Constraints
   ☞  12.12  Midterm Test
9.  12.17  Database Design
10.  01.07  File and System Structure
11.  01.14  Indexing and Hashing
12.  01.21  Transactions and Concurrency Control
13.  01.28  Query processing
   ☞  02.04  Final Exam
What you will be expected to learn:

- How to draw and interpret *E-R diagrams*
- How to realize E-R schemas as *relational databases*
- How to pose queries using *relational algebra* and the relational tuple calculus
- How to write *SQL* queries
- How to express and interpret *functional dependencies* (FDs)
- How to use FDs in *database design*
- How databases are *physically organized* for optimal performance
- How *concurrent databases accesses* are managed
- How queries are *evaluated*
Definitions?

What is a Database?

- Definition?
- Examples?

What is a Database System?

- Services, functionality?
- Difference with File Systems?
In Search of a Definition ...

- A **database** is a logically coherent collection of data with some inherent meaning. A database is designed, built and populated with data for a specific purpose and represents some aspect of the real world. [Elmasri, p. 3]

- A **database management system** consists of a collection of interrelated data and a set of programs to access data. The collection of data is usually referred to as the database. [Korth, p. 1]

- A **database system** is essentially nothing more than a computerized record-keeping system. The database itself can be regarded as a kind of electronic filing cabinet — that is, a repository for a collection of computerized data files. [Date, p. 3]

- A **database** can be defined as a set of master files, organized and administered in a flexible way, so that the files of the database can be easily adapted to new, unforeseen tasks.

- A **database** is a structured collection of operational data together with a description of that data. [Stranczyk, p. 4]

- A **database system** is a collection of programs that run on a computer and that help the user to get information, to update information, to protect information, in general to manage information. [Paradaens, p. 1]
What is a Database?

Database System

Software to Access Stored Data

Stored Database

Stored Database Definition (Meta-Data)

DBMS Software

Software to Process Queries/Programs

Application Programs/Queries

Users/Programmers
### Example

#### borrow

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>

#### deposit

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Perryridge</td>
<td>201</td>
<td>Williams</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
</tr>
<tr>
<td>Downtown</td>
<td>105</td>
<td>Green</td>
<td>850</td>
</tr>
</tbody>
</table>

#### customer

<table>
<thead>
<tr>
<th>customer-name</th>
<th>street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
<tr>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Glenn</td>
<td>Sand Hill</td>
<td>Woodside</td>
</tr>
<tr>
<td>Brooks</td>
<td>Senator</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Green</td>
<td>Walnut</td>
<td>Stamford</td>
</tr>
</tbody>
</table>

#### branch

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Mianus</td>
<td>4000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
</tr>
</tbody>
</table>

#### client

<table>
<thead>
<tr>
<th>customer-name</th>
<th>banker-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turner</td>
<td>Johnson</td>
</tr>
<tr>
<td>Hayes</td>
<td>Jones</td>
</tr>
<tr>
<td>Johnson</td>
<td>Johnson</td>
</tr>
</tbody>
</table>
Why Do We Need Database Systems?

To avoid:

- Redundancy
- Inconsistency
- Inflexibility
- Concurrent access anomalies

To provide:

- Security
- Integrity
- Standards
When Do We Need Database Systems?

- Large, complex database
- Persistent data
- Multiple Users
- Frequent updates
- Ad hoc queries
- Large, open class of applications
- Security and authorization
- Integrity constraints
- Backup and recovery
When Do We Not Need Database Systems?

Costs:
- investment in hardware, software and training
- generality
- overhead for security, concurrency control, recovery and integrity

When not to use:
- DB + applications are simple, well-defined, and won’t evolve
- very small database
- stringent real-time constraints
- multiple-use (update?) access not required
Kinds of Database Systems

- Legacy: Network, Hierarchical...
- Relational
- Object-Oriented
- CAD
- Deductive
- Knowledge bases
- ...

University of Bern
Data Models

“A data model is a set of concepts that can be used to describe the structure of a database.” (E&N)

- data types
- relationships
- constraints
- basic operations (retrieval & update)
- behaviour
**E-R Model**

Formal model and Graphical notation

- **Entity sets** (rectangles)
- **Attributes** (ellipses)
- **Relationship sets** (diamonds)

```
account
customer

CustAcct

value

number
balance

name
city
street
```
Relational Model

Record-based model
- Named tables of tuples
- Named, typed fields
- No pointers
- No nesting
- No behaviour
OO Model

Comparable to, but distinct from objects in OO programming languages

- Nested objects
- Instance variables
- Methods
- Classes
- Messages
Schemas and Instances

Database Schema
- describes the structure of the database
- consists of “meta-data”

Database Instance (or State)
- snapshot of a database at some point in time
The Three Schema Architecture

External Level

External View 1

Conceptual Level

Conceptual Schema

Internal Level

Internal Schema

Datenbanken 7042 — WS 97/98
Data Independence

Physical data independence
☞ the ability to modify the physical scheme without affecting client applications

Logical data independence
☞ the ability to modify the conceptual scheme without affecting client applications or external schemas
Architecture

Data Definition Language (DDL)
- Used by Database Administrator to define schema
- Compiled into a *data dictionary* containing all meta-data and storage details (file names, mappings, constraints)
- A separate *Storage Definition Language* may exist for specifying the physical schema...

Data Manipulation Language (DML)
- Queries *and* Updates (insertion, modification, deletion)
- Procedural: specifies *how* to get data (navigational)
- Non-procedural: specifies *what* data to get

Database Interfaces
- Menus; graphical interfaces for e.g., schema design; forms; natural language; canned operations; canned DBA operations
- Report generators; 4GLs; Office systems (forms, workflows...)

Database Manager
- Data storage, security, concurrency etc.
Implementation issues

- File Organisation
- File Re-organisation
- Query Processing
- Concurrency Control
- Transactions
- Recovery
- Performance monitoring
- Data conversion (import/export)
- Distribution
Classification of Database Systems

- Data model
- Number of Users
- Number of sites
- Cost
- Types of Access Path
- General/Special-purpose
Summary

You should know the answers to these questions:

❑ What are the distinctions between a database, a database system and a database management system?
❑ When are database systems (not) needed?
❑ What is a data model?
❑ What is a database schema/instance?
❑ What are the main parts of a database system?

Can you answer the following questions?

✎ Would you use a DBMS to implement a personal address database? Why (not)?
✎ What are the main functions of a database administrator?
✎ What differences would you expect between a DBMS for a PC user and one for a large corporation?
✎ What major steps would you go through to set up a database system for a particular enterprise?
✎ What is the difference between physical and logical data independence? Give examples.
2. The Entity-Relationship Model

Overview

- Entities, Attributes and Relationships
- Attributes vs. Entities
- Mapping Constraints
- E-R diagrams — an introduction
Entities and Attributes

An **entity** is an object that exists and is distinguishable from other objects.

An **entity-set** is a set of entities of the same type. — E&N

An entity is represented by a set of **attributes**, which is formally a function $a : E \to A$
**Entities & Attributes**

**Customer:** \{ name, social security, street, city \}

**Account:** \{ account-number, balance \}

<table>
<thead>
<tr>
<th>Name</th>
<th>Account Number</th>
<th>Street</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver</td>
<td>654-32-1098</td>
<td>Main</td>
<td>Austin</td>
</tr>
<tr>
<td>Harris</td>
<td>890-12-3456</td>
<td>North</td>
<td>Georgetown</td>
</tr>
<tr>
<td>Marsh</td>
<td>456-78-9012</td>
<td>Main</td>
<td>Austin</td>
</tr>
<tr>
<td>Pepper</td>
<td>369-12-1518</td>
<td>North</td>
<td>Georgetown</td>
</tr>
<tr>
<td>Ratliff</td>
<td>246-80-1214</td>
<td>Park</td>
<td>Round Rock</td>
</tr>
<tr>
<td>Brill</td>
<td>121-21-2121</td>
<td>Putnam</td>
<td>San Marcos</td>
</tr>
<tr>
<td>Evers</td>
<td>135-79-7</td>
<td>Nassau</td>
<td>Austin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Account Number</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
<td>1000</td>
</tr>
<tr>
<td>630</td>
<td>2000</td>
</tr>
<tr>
<td>401</td>
<td>700</td>
</tr>
<tr>
<td>700</td>
<td>1500</td>
</tr>
<tr>
<td>199</td>
<td>500</td>
</tr>
<tr>
<td>467</td>
<td>900</td>
</tr>
<tr>
<td>115</td>
<td>1200</td>
</tr>
<tr>
<td>183</td>
<td>1300</td>
</tr>
<tr>
<td>118</td>
<td>2000</td>
</tr>
<tr>
<td>225</td>
<td>2500</td>
</tr>
<tr>
<td>210</td>
<td>2200</td>
</tr>
</tbody>
</table>
Attributes

- special value *null*
- multi-valued attributes: $A = 2^V$
- atomic and composite attributes
- derived attributes
A *relationship* is an association among several \((n > 2)\) entities.

A *relationship set* is a set of relationships of the same type.

Formally, \(R \subseteq E_1 \times E_2 \times \ldots \times E_N\)
Relationships and relationship sets

Oliver 654-32-1098 Main Austin
Harris 890-12-3456 North Georgetown
Marsh 456-78-9012 Main Austin
Pepper 369-12-1518 North Georgetown
Ratliff 246-80-1214 Park Round Rock
Brill 121-21-2121 Putnam San Marcos
Evers 135-79-1357 Nassau Austin
Attributes vs. Entities

When should an attribute be modelled as a separate entity?

When should an attribute be modelled as a separate entity?
**Mapping Constraints**

**One-to-one:** An entity A is associated with *at most one* entity in B, and vice versa.

**One-to-many:** An entity in A is associated with any number of entities in B. An entity in B, however, can be associated with at most one entity in A. (I.e., a function from B to A)

**Many-to-one:** (reverse of one-to-many)

**Many-to-many:** An entity in A is associated with any number of entities in B, and vice versa.
Existence Constraints

A transaction is existence-dependent on an account.

Account is a dominant entity set whereas transaction is a subordinate entity set.

The entity-set transaction must totally participate in some relationship with account.

(If there is no existence constraint between entity-sets, then participation in mutual relationships is said to be partial.)
**E-R Diagrams — Example**

Rectangles represent *entity sets*

Ellipses represent *attributes*

Diamonds represent *relationship sets*

Lines connect attributes to their entity/relationship sets and entities to their relationship sets
One-to-one, one-to-many

One-to-many: Every account belongs to at most one customer

NB: can use either arrow or explicit 1:N labelling

One-to-one: Every customer has at most one account, and vice versa
Ternary Relationships

- Customer
  - Social-security
  - Customer-name
  - Customer-city

- Account
  - Account-number
  - Balance

- Branch
  - Branch-name
  - Branch-city
  - Assets

- CAB

The Entity-Relationship Model
The Entity-Relationship Model

Roles

- employee
- phone-number
- employee-name
- manager
- worker
- works-for
Summary

You should know the answers to these questions:
- What are entities, entity sets, attributes and relationships?
- How can these be represented formally?
- What are null values?
- What does a one-to-many relationship mean?
- How can a database schema be represented as an E-R diagram?

Can you answer the following questions?
- How are existence constraints represented in an E-R diagram?
- How should relationships be represented in a database?
3. Entity-Relationship Diagrams

Overview
- Primary Keys
- Strong & Weak Entity Sets
- E-R Diagrams
- Generalisation and Aggregation
- Reducing E-R Diagrams to Tables
- Design Decisions
Primary Keys

A superkey is set of attributes that uniquely identifies an entity.

How can you formally define a superkey?

A candidate key is a minimal superkey.
A primary key is chosen by design.
Strong & Weak Entity Sets

An entity set that lacks a superkey is a weak entity set, otherwise the entity set is strong.

A weak entity set depends existentially on a strong entity set:

- transaction depends on its identifying owner account

transaction has a partial key transaction-number and primary key (account-number, transaction-number)
**Relationship keys**

If

\[ R \subseteq E_1 \times E_2 \times \ldots \times E_N \]

then

\[ \text{attr}(R) = \text{prim}_\text{key}(E_1) \cup \ldots \cup \text{prim}_\text{key}(E_N) \cup \text{desc}_\text{attr}(R) \]

The candidate (i.e., minimal) keys of a relationship will depend on the cardinality mappings. If these are many-to-many, then all prim_keys will be needed; if some are one-to-many or many-to-one, then some prim_keys will not be needed!
**ER Diagrams**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="symbol" alt="Entity Type" /></td>
<td>Entity Type</td>
</tr>
<tr>
<td><img src="symbol" alt="Weak Entity Type" /></td>
<td>Weak Entity Type</td>
</tr>
<tr>
<td><img src="symbol" alt="Relationship Type" /></td>
<td>Relationship Type</td>
</tr>
<tr>
<td><img src="symbol" alt="Identifying Relationship Type" /></td>
<td>Identifying Relationship Type</td>
</tr>
<tr>
<td><img src="symbol" alt="Attribute" /></td>
<td>Attribute</td>
</tr>
<tr>
<td><img src="symbol" alt="Key Attribute" /></td>
<td>Key Attribute</td>
</tr>
<tr>
<td><img src="symbol" alt="Multivalued Attribute" /></td>
<td>Multivalued Attribute</td>
</tr>
<tr>
<td><img src="symbol" alt="Composite Attribute" /></td>
<td>Composite Attribute</td>
</tr>
<tr>
<td><img src="symbol" alt="Derived Attribute" /></td>
<td>Derived Attribute</td>
</tr>
</tbody>
</table>

- **Total Participation of E2 in R**
- **Cardinality Ratio 1:N for E1:E2 in R**
- **Structural Constraint (min, max) on participation of E in R**
Generalisation

account

IS-A

savings-account

interest-rate

checking-account

overdraft-amount

balance

account-number
E-R diagram with redundant relationships

E-R diagram with aggregation
Reducing E-R Diagrams to Tables

- **Strong entity sets**: one column per attribute
- **Relationships**: (between strong entity sets) one column per attribute in attr(R)

<table>
<thead>
<tr>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
<td>1000</td>
</tr>
<tr>
<td>630</td>
<td>2000</td>
</tr>
<tr>
<td>401</td>
<td>700</td>
</tr>
<tr>
<td>700</td>
<td>1500</td>
</tr>
<tr>
<td>199</td>
<td>500</td>
</tr>
<tr>
<td>467</td>
<td>900</td>
</tr>
<tr>
<td>115</td>
<td>1200</td>
</tr>
<tr>
<td>183</td>
<td>1300</td>
</tr>
<tr>
<td>118</td>
<td>2000</td>
</tr>
<tr>
<td>225</td>
<td>2500</td>
</tr>
<tr>
<td>210</td>
<td>2200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>social-security</th>
<th>account-number</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>654-32-1098</td>
<td>259</td>
<td>17 June 1990</td>
</tr>
<tr>
<td>654-32-1098</td>
<td>630</td>
<td>17 May 1990</td>
</tr>
<tr>
<td>890-12-3456</td>
<td>401</td>
<td>23 May 1990</td>
</tr>
<tr>
<td>456-78-9012</td>
<td>700</td>
<td>28 May 1990</td>
</tr>
<tr>
<td>369-12-1518</td>
<td>199</td>
<td>13 June 1990</td>
</tr>
<tr>
<td>246-80-1214</td>
<td>467</td>
<td>7 June 1990</td>
</tr>
<tr>
<td>246-80-1214</td>
<td>115</td>
<td>7 June 1990</td>
</tr>
<tr>
<td>121-21-2121</td>
<td>183</td>
<td>13 June 1990</td>
</tr>
<tr>
<td>135-79-1357</td>
<td>118</td>
<td>17 June 1990</td>
</tr>
<tr>
<td>135-79-1357</td>
<td>225</td>
<td>19 June 1990</td>
</tr>
<tr>
<td>135-79-1357</td>
<td>210</td>
<td>27 June 1990</td>
</tr>
</tbody>
</table>
Reducing Weak Entity Sets

- **Weak entity sets:** (W dependent on S)
  - One column per attribute in attr(W) U prim_key(S)

<table>
<thead>
<tr>
<th>account-number</th>
<th>transaction-number</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>259</td>
<td>5</td>
<td>11 May 1990</td>
<td>+50</td>
</tr>
<tr>
<td>630</td>
<td>11</td>
<td>17 May 1990</td>
<td>+70</td>
</tr>
<tr>
<td>401</td>
<td>22</td>
<td>23 May 1990</td>
<td>-300</td>
</tr>
<tr>
<td>700</td>
<td>69</td>
<td>28 May 1990</td>
<td>-500</td>
</tr>
<tr>
<td>199</td>
<td>103</td>
<td>3 June 1990</td>
<td>+900</td>
</tr>
<tr>
<td>259</td>
<td>6</td>
<td>7 June 1990</td>
<td>-44</td>
</tr>
<tr>
<td>115</td>
<td>53</td>
<td>7 June 1990</td>
<td>+120</td>
</tr>
<tr>
<td>199</td>
<td>104</td>
<td>13 June 1990</td>
<td>-200</td>
</tr>
<tr>
<td>259</td>
<td>7</td>
<td>17 June 1990</td>
<td>-79</td>
</tr>
</tbody>
</table>
Design Decisions

- ternary vs. pairs of binary relationships?
- representing concepts by entity sets or relationships?
- representing properties by attributes or entities?
- using strong or weak entity sets?
- generalisation?
- aggregation?
Summary

You should know the answers to these questions:

- What are keys, superkeys, candidate keys, minimal keys and primary keys?
- What are strong and weak entity sets?
- How can you determine the keys of a relationship?
- When can you use generalization and aggregation?
- How can you translate E-R diagrams to tables?

Can you answer the following questions?

- Can an entity have more than one minimal key?
- When can an entity be inserted into or deleted from a weak entity set?
- Is a totally participating entity set necessarily a weak entity set?
- When should you use generalization and aggregation?
- How many tables will result from an E-R diagram?
4. The Relational Model

Overview

- Relations: Schemas and instances
- Relational Algebra
  - Basic operators: select, project, product, renaming, union, difference
History

- 1970: Proposed by Codd (IBM, San José)
- 1970s: Various research prototypes
  - System R (IBM, San José)
  - Ingres (UC Berkeley)
  - Query-by-Example (IBM, TJ Watson) ...
- Late 1970s: Relational theory matures
- Early 1980s: commercial presence established
Example: The Bank Database Schema
Relational Databases

- Relational Database = set of (named) tables
- Table = set of rows
- Rows represent relationships amongst values
- Columns represent (named, typed) attributes

| deposit  |
|------------------|-----------------|-----------------|-----------|
| branch-name | account-number | customer-name   | balance  |
| Downtown    | 101             | Johnson         | 500      |
| Mianus      | 215             | Smith           | 700      |
| Perryridge  | 102             | Hayes           | 400      |
| Round Hill  | 305             | Turner          | 350      |
| Perryridge  | 201             | Williams        | 900      |
| Redwood     | 222             | Lindsay         | 700      |
| Brighton    | 217             | Green           | 750      |
| Downtown    | 105             | Green           | 850      |
Notation

Formally, a relation:

$$r \subseteq R, \quad R = D_1 \times \ldots \times D_N$$

where each $$D_i$$ is atomic

Each attribute $$a_i: R \rightarrow D_i$$

But, for $$t \in r$$ we write $$t[a_i]$$ rather than $$a_i(t)$$ ... since $$t[a_i] = t[i]$$
**Schemas and instances**

A relation $r$ is an *instance* of a relation *scheme* $R = D_1 \times \ldots \times D_N$.

A relation scheme is defined by, e.g.:

```
Deposit-scheme = ( 
    branch-name : string, 
    account-number : integer, 
    customer-name : string, 
    balance : integer 
)
```
Common attributes

Customer shares attributes with Deposit, allowing relations to be associated.
Query Languages

- Procedural vs. non-procedural
- Formal vs. commercial
  - relational algebra, tuple & domain relation calculi
  - SQL, QBE, Quel ...
Relational Algebra

Basic unary & binary operators over relations: \( r \otimes s = u \)
- Select: \( \sigma_p(r) \)
- Project: \( \Pi_A r \)
- Cartesian product: \( r \times s \)
- Renaming: \( \rho_s(r) \)
- Union: \( r \cup s \)
- Set-difference: \( r - s \)

Other operators
- Assignment: \( \text{temp} \leftarrow \langle \text{expression} \rangle \) (for intermediate expressions)
- Derived: intersection, natural join, division
### Example: The Bank Database

#### customer

<table>
<thead>
<tr>
<th>customer-name</th>
<th>street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
<tr>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Glenn</td>
<td>Sand Hill</td>
<td>Woodside</td>
</tr>
<tr>
<td>Brooks</td>
<td>Senator</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Green</td>
<td>Walnut</td>
<td>Stamford</td>
</tr>
</tbody>
</table>

#### deposit

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Perryridge</td>
<td>201</td>
<td>Williams</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
</tr>
<tr>
<td>Downtown</td>
<td>105</td>
<td>Green</td>
<td>850</td>
</tr>
</tbody>
</table>

#### branch

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Mianus</td>
<td>400000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
</tr>
</tbody>
</table>

#### borrow

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>
Select

\[ \sigma_p(r) \] selects \( t \) in \( r \) satisfying predicate \( p \)

(Where \( p \) is a Boolean expression using comparisons \( =, \neq, <, \leq, >, \geq \) over attributes and values, and connectives \( \wedge \) (and) and \( \vee \) (or))

Express the following queries:

- What are all the branches in Horseneck?
- Which loans at Perryridge are over 1200?
- Which bankers have accounts at their own branches?

<table>
<thead>
<tr>
<th>client</th>
<th>customer-name</th>
<th>banker-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turner</td>
<td>Johnson</td>
<td></td>
</tr>
<tr>
<td>Hayes</td>
<td>Jones</td>
<td></td>
</tr>
<tr>
<td>Johnson</td>
<td>Johnson</td>
<td></td>
</tr>
</tbody>
</table>
The Relational Model

Project

\[ \Pi_A r \] projects attributes in \( A \) from all tuples in \( r \)

Express the following queries:

- What are the account numbers of all deposits?
- Who are our customers?
- Which customers have loans?
- In which cities do we have branches?
- Which bankers have accounts at their own branches?
**Cartesian product**

$r \times s$ generates the set of tuples obtained by concatenating each possible pair of tuples from $r$ and $s$.

Express the following queries:

- What are the home towns of the customers with deposits at the Downtown branch?
- What are the names and home cities of all the clients of Johnson?
Renaming

\[ \rho_s(r) \] renames relation \( r \) as \( s \)

Express the following queries:

1. Which branches are in the same city as the Perryridge branch?
2. What are the names of all customers who live on the same street of the same city as Smith?
Union

$r \cup s$ generates the union of $r$ and $s$

Express the following queries:

Who are the customers of the Perryridge branch?
Set-difference

$r - s$ generates the set of tuples in $r$ but not in $s$

Express the following queries:

✎ Which customers have loans out but no deposits?
✎ Which customers do not have branches in their home cities?
✎ Which customer has the largest balance?
Summary

You should know the answers to these questions:

- What are relations, tables, relation schemes?
- What are the operators of the relational algebra?
- How can operators be combined to express queries over multiple relations?
- What is a join operation?

Can you answer the following questions?

- How can the relational operators be defined formally?
- What are the cardinalities and the relation schemes of the results of each operator?
- Why do we need the renaming operator?
- Can union be expressed in terms of the other operators? (Why, or why not?)
- How can you formulate the query: “Which customers have exactly one deposit?”
5. The Relational Model (Continued)

Overview

- Relational Algebra
  - Derived operators: intersect, join, division, assignment
- Deletions, Insertion and Updates
- Views, view updates and null values
- The Tuple and Domain Relational Calculi
Derived operators

- Intersection: \( r \cap s \)
- Natural Join: \( r \bowtie s \)
- Division: \( r \div s \)
### Example: The Bank Database

#### branch

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Mianus</td>
<td>400000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
</tr>
</tbody>
</table>

#### deposit

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Perryridge</td>
<td>201</td>
<td>Williams</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
</tr>
<tr>
<td>Downtown</td>
<td>105</td>
<td>Green</td>
<td>850</td>
</tr>
</tbody>
</table>

#### customer

<table>
<thead>
<tr>
<th>customer-name</th>
<th>street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
<tr>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Glenn</td>
<td>Sand Hill</td>
<td>Woodside</td>
</tr>
<tr>
<td>Brooks</td>
<td>Senator</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Green</td>
<td>Walnut</td>
<td>Stamford</td>
</tr>
</tbody>
</table>

#### borrow

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>
Intersection

$r \cap s$ extracts all tuples in both $r$ and $s$

Express the following queries:

- Which customers have both deposits and loans at Perryridge?
Natural Join

$r \bowtie s$ extracts pairs of tuples from $r$ and $s$ with common attributes and forms new tuples with those attributes identified.

Express the following queries:

- What are the names and home cities of all customers with a loan?
- What are the assets and names of branches with depositors in Stamford?
- Which customers have both deposits and loans at Perryridge?
Division

$r \div s$ yields the remainder of tuples in $r$ whose product with $s$ is in $r$

NB: $(r \div s) \times s \subseteq r$ always holds. $r \div s$ is the maximal such relation.

Express the following queries:

📝 Which customers have an account at all branches in Brooklyn?
Insertions and Deletions

Insertion: $ r \leftarrow r \cup E$

- Open a new account 9732 for Smith with $1200 at Perryridge

Deletion: $ r \leftarrow r - E$

- Delete all of Smith’s accounts
- Delete all accounts at branches in Needham
Updates: \( \delta_A \leftarrow E(r) \)

Express the following updates:

- Add 5% interest to accounts with balance over $1000
- Add 6% interest to accounts with a balance over $10,000, and %5 to the rest
The Tuple Relational Calculus

\{ t \mid P(t) \} \text{ selects all tuples } t \text{ such that } P(t) \text{ holds }

Atoms:
- \( s \in r \)
- \( s[x] \Theta r[y] \)
- \( s[x] \Theta c \)

Formulae \( P_i \):
- atoms
- \( \neg P, (P), P_1 \land P_2, P_1 \lor P_2, P_1 \Rightarrow P_2 \)
- \( \exists s \in r(P(s)), \forall s \in r(P(s)) , s \text{ free in } P(s) \)
Examples

Which loans are over $1200?
\{ t | t \in borrow \land t[amount] > 1200 \}

What are the names of customers with loans over $1200?
\{ t | \exists s \in borrow (t[cn] = s[cn] \land s[amount] > 1200) \}

Express the following queries:

- What are the names and home cities of customers with loans at Perryridge?
- Which customers have either deposits or loans at Perryridge?
- Which customers have both deposits and loans at Perryridge?
- Which customers have deposits but no loans at Perryridge?
- Which customers have deposits at all branches in Brooklyn?
**Safety**

**Consider:** \( \{ t \mid \neg (t \in \text{borrow}) \} \)

This expression is not safe since it includes a potentially infinite number of tuples. Formally, the domain of a formula is the set of all values it references. If the result generates values outside the domain, the formula is **unsafe**.
The Domain Relational Calculus

\{ \langle x_1, \ldots, x_n \rangle \mid P(x_1, \ldots, x_n) \} \text{ selects all tuples } \langle x_1, \ldots, x_n \rangle \text{ such that } P(x_1, \ldots, x_n) \text{ holds}

Atoms:

- \langle x_1, \ldots, x_n \rangle \in r
- x \Theta y
- x \Theta c

Formulae \( P_i \):

- atoms
- \neg P, (P), P_1 \land P_2, P_1 \lor P_2, P_1 \Rightarrow P_2
- \exists x(P(x)), \forall x(P(x))
Examples

Which loans are over $1200?
\[ \{ \langle b, l, c, a \rangle | \langle b, l, c, a \rangle \in \text{borrow} \land a > 1200 \} \]

What are the names of customers with loans over $1200?
\[ \{ \langle c \rangle | \exists b, l, a(\langle b, l, c, a \rangle \in \text{borrow} \land a > 1200) \} \]

Express the following queries:

- What are the names and home cities of customers with loans at Perryridge?
- Which customers have either deposits or loans at Perryridge?
- Which customers have both deposits and loans at Perryridge?
- Which customers have deposits at all branches in Brooklyn?
Summary

You should know the answers to these questions:

- How can intersection, natural join and division be derived from the basic operators of the relational algebra?
- When are joins useful? Division?
- How are modifications expressed in the relational algebra?
- How can updates be made to views?
- How can queries be expressed in the tuple and domain relational calculi?

Can you answer the following questions?

- Could set difference be replaced by intersection as a basic operator of the relational algebra? (Would it still be possible to express the same queries?)
- What is the join of a relation with itself?
- How can a join be efficiently implemented?
- Does selection distribute over join? (I.e., can the evaluation order be swapped?)
- How can relational algebra queries be transformed to the tuple/domain calculi?
6. SQL

Overview

- SQL
  - Basic structure: product, select and project
  - Union, Intersection, Minus
  - Predicates and Joins
  - Set membership
  - Ordering

To be continued ...
**SQL**

Not “just a query language”
- Data Definition Language
- Data Manipulation Language
- Embedded DML
- View Definition
- Authorization
- Integrity
- Transaction Control
**SQL Syntax Summary: Queries**

Queries and updates:

```sql
select [ distinct ] attribute-list
  from table-name { alias } { , table-name { alias } }
  [ where condition ]
  [ group by grouping-attributes [ having group-selection-condition ] ]
  [ order by column-name [ order ] { , column-name [ order ] } ]

attribute-list ::= ( * | ( column-name | function ( ( [ distinct ] column-name | * )))
  { , ( column-name | function ( ( [ distinct ] column-name | * ))) } )

grouping-attributes ::= column-name { , column-name }

order ::= ( asc | desc )

insert into table-name ( ( column-name { , column-name } ) )
  ( values ( constant-value { , constant-value } ) { , ( constant-value { , constant-value } ) } |
    select-statement )

delete from table-name [ where selection-condition ]

update table-name
  set column-name = value-expression { , column-name = value-expression } 
  [ where selection-condition ]
```
**SQL Syntax Summary: DDL**

**DDL operations:**

- `create table table-name (column-name column-type [attribute-constraint] {, column-name column-type [attribute-constraint]} [table-constraint {, table-constraint}])`

- `drop table table-name`

- `alter table table-name add column-name column-type`

- `create [unique] index index-name on table-name (column-name [order] {, column-name [order]}) [cluster]`

- `drop index [index-name]`

- `create view view-name [ (column-name {, column-name}) ] as select-statement`

- `drop view view-name`

*Adapted from Elmasri and Navathe, p. 226*

NB: this is only a summary; differences may exist between different versions of SQL
### Basic Structure

```
select A_1, A_2, ..., A_N
from r_1, r_2, ..., r_m
where P
```

equivalent to: \( \Pi_{A_1, A_2, ..., A_n}(\sigma_P(r_1 \times r_2 \times ... \times r_m)) \)

#### Examples:
```
select branch-name
from deposit
```
```
select distinct branch-name
from deposit
```
**Set Operations: Union**

Find all customers with accounts or loans at Perryridge

```sql
( select customer-name
  from deposit
  where branch-name = "Perryridge"
 )

union

( select customer-name
  from borrow
  where branch-name = "Perryridge"
 )
```
Set Operations: Intersection and Minus

Find all customers with both deposits and loans at Perryridge.

```
( select  distinct customer-name
  from    deposit
  where   branch-name = "Perryridge"
 )

intersect

( select  distinct customer-name
  from    borrow
  where   branch-name = "Perryridge"
 )
```
Predicates and Joins

Recall:

Find names and home cities of all customers with a loan

\[ \Pi_{\text{customer-name, customer-city}} (\text{borrow} \Join \text{customer}) \]

Express as:

\[
\text{select distinct customer.customer-name, customer-city}
\text{from borrow, customer}
\text{where borrow.customer-name = customer.customer-name}
\]

Comparisons may be: <, ≤, =, ≠, ≥, >
Logical Connectives

```
select distinct customer.customer-name, customer-city
from borrow, customer
where borrow.customer-name = customer.customer-name
    and branch-name = "Perryridge"
```

Differences with Relational Algebra:

- Connectives: and, or, not
- Comparisons: between
- Arithmetic operators: +, -, *, /
String matching

- %  - (percent) matches arbitrary substrings
- _  - (underscore) matches any character
- \  - (backslash) escapes “%”, “_” or “\”

```sql
select customer-name
from customer
where street like "%Main%"
```
**Set Membership**

What does the following query represent?

```sql
select distinct customer-name
from borrow
where branch-name = "Perryridge"
and customer-name in
  ( select distinct customer-name
    from deposit
    where branch-name = "Perryridge"
  )
```
What does the following query represent?

```sql
select distinct customer-name
from borrow
where branch-name = "Perryridge"
and <branch-name, customer-name> in
    ( select branch-name, customer-name
        from deposit
    )
```
**Tuple Variables**

```sql
select distinct C.customer-name, customer-city
from borrow B, customer C
where B.customer-name = C.customer-name
```

Express the following query:

✧ *Find all customers who have an account at some branch where Johnson has an account*
**Set comparison**

Can compare attributes against *sets* of values (*compare all* or *compare some*):

```
select branch-name
from branch
where assets > some
  ( select assets
    from branch
    where branch-city = "Brooklyn"
  )
```
What does the following query represent?

```
select  distinct  S.customer-name
from    deposit S
where   ( select  T.branch-name
          from    deposit T
          where   S.customer-name = T.customer-name
          )
       contains
       ( select  branch-name
         from    branch
         where   branch-city = "Brooklyn"
         )
```
Testing for empty relations

```sql
select distinct customer-name
from customer
where exists
  ( select *
      from deposit
      where customer.customer-name = deposit.customer-name
        and branch-name = "Perryridge"
  )
and exists
  ( select *
      from borrow
      where customer.customer-name = borrow.customer-name
        and branch-name = "Perryridge"
  )
```
**Ordering**

Query results may be sorted in ascending or descending order by selected attributes:

```sql
select * 
from borrow 
order by amount desc, 
loan-number asc
```
Summary

You should know the answers to these questions:

- How do you express selections, projections and joins?
- How can you compare relations? (union, intersection, etc.)
- How do you form complex predicates?
- How do you express string matching predicates?
- When are tuple variables needed?
- How can query results be sorted?

Can you answer the following questions?

- How can a relational algebra query be translated to SQL?
- When is the distinct keyword needed?
- How do you express the RA division operator in SQL?
7. SQL, QBE and Quel

Overview

- SQL
  - Aggregate functions and group predicates
  - Restrictions, null values and views
- Query-by-Example
- Quel
Aggregate Functions

Aggregate functions apply to groups with common attributes:

- `avg` — average
- `min` — minimum
- `max` — maximum
- `sum` — total
- `count` — cardinality

Find the average account balance at each branch

```sql
select branch-name, avg(balance)
from deposit
group by branch-name
```

Find the number of depositors for each branch

```sql
select branch-name, count(distinct customer-name)
from deposit
group by branch-name
```
Group Predicates

```sql
select branch-name, avg (balance)
from deposit
group by branch-name
having avg (balance) > 1200
```

May not compose aggregate functions!

```sql
select branch-name
from deposit
group by branch-name
having avg (balance) ≥ all( select avg(balance)
    from deposit
    group by branch-name )
```

✎ Find the average balance of all depositors who live in Harrison and have at least three accounts
Modification

Deletion: delete \( r \) where \( P \)

\[
\text{delete} \quad \text{deposit} \\
\text{where} \quad \text{customer-name} = \text{"Smith"}
\]

\[
\text{delete} \quad \text{borrow}
\]

Insertion:

\[
\text{insert into} \quad \text{deposit} \\
\quad \text{values} \quad \text{"Perryridge", 9732, "Smith", 1200}
\]

\[
\text{insert into} \quad \text{deposit ( account-number, customer-name, branch-name, balance )} \\
\quad \text{values} \quad (9732, \text{"Smith", "Perryridge", 1200})
\]
Restrictions

A deletion or insertion may not include an embedded select that accesses the relation being modified

**INVALID SQL:**

```sql
delete deposit
where balance < ( select avg (balance) from deposit )
```

**INVALID SQL:**

```sql
insert into deposit
select * from deposit
```
Updates

```sql
update deposit
set balance = balance * 1.05

update deposit
set balance = balance * 1.06
where balance > 10000

update deposit
set balance = balance * 1.05
where balance ≤ 10000
```

INVALID SQL:

```sql
update deposit
set balance = balance * 1.05
where balance > select avg(balance)
from deposit
```
Null Values

```sql
insert into deposit
  values ("Perryridge", null, "Smith", 1200)

select *
from deposit
where account-number = 1700

select distinct customer-name
from borrow
where amount is null

select sum (amount)
from borrow
```
### Views

**create view** `<view-name>` **as** `<query-expression>`

View names may be used anywhere that relation names appear **EXCEPT** modifications may only be applied to views constructed from a single base relation.

```sql
create view loan-info as
    select    branch-name, loan-number, customer-name
    from      borrow

insert into loan-info
    values ("Perryridge", 3, "Ruth")
```
Data Definition

Defining new tables:
\[ \text{create table } r( A_1 \ D_1, \ldots, A_n \ D_n ) \]

Removing tables:
\[ \text{drop table } r \]

Adding new attributes:
\[ \text{alter table } r \text{ add } A \ D \]
Summary

You should know the answers to these questions:

- How to compute (aggregate) functions over sets of values in SQL?
- What is the difference between a `where` clause and a `having` clause?
- How do you express deletion, insertions and updates in SQL?
- What restrictions must be obeyed in update commands?
- What test can be performed with null values?
- How do you define a view?
- What kind of views may be updated? How?

Can you answer the following questions?

- How can you compute the average of the maximum balance at each branch?
- Why can’t views defined over multiple relations be updated by simply propagating the update to the base relations?
- What is the difference between `delete r` and `drop table r`?
Query-by-example

Developed by Zloof & de Jong, IBM TJ Watson, early 1970s

- Two-dimensional syntax representing tables
- Queries expressed “by example” by entering constraints into “skeleton” tables
- Domain variables preceded by underscores: _x
- Complex queries via variable unification
- Explicit print command (P.) to obtain results
Simple queries

In QBE:

<table>
<thead>
<tr>
<th>deposit</th>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Perryridge&quot;</td>
<td>P_x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

In the domain relational calculus:

\[
\{ \langle x \rangle | \exists b, l, a(\langle b, l, x, a \rangle \in deposit \land b = "Perryridge") \}
\]
### Variable unification

Which customers have accounts at both Perryridge and Redwood?

<table>
<thead>
<tr>
<th>deposit</th>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Perryridge”</td>
<td>P.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Redwood”</td>
<td>_x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which customers have accounts at either Perryridge or Redwood (or both)

<table>
<thead>
<tr>
<th>deposit</th>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Perryridge”</td>
<td>P.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Redwood”</td>
<td>P.y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Set Difference

What does this query express?

<table>
<thead>
<tr>
<th>deposit</th>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Perryridge&quot;</td>
<td>P_x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What would it mean if the negation were removed?

<table>
<thead>
<tr>
<th>borrow</th>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬</td>
<td>&quot;Perryridge&quot;</td>
<td>_x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Result Relations

What does this query express?

<table>
<thead>
<tr>
<th>deposit</th>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Perryridge”</td>
<td>_z</td>
<td>_x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer</th>
<th>customer-name</th>
<th>street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>_x</td>
<td></td>
<td>_y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
<th>customer_name</th>
<th>customer-city</th>
<th>account-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>_x</td>
<td>_y</td>
<td>_z</td>
</tr>
</tbody>
</table>
Other features

- Condition boxes
- Ordering display of tuples
- Aggregate operations
- Deletion, Insertion and Update operators (D., I. and U.)
Based on tuple relational calculus:

- range of $t_1$ is $r_1$
- range of $t_2$ is $r_2$
- ... 
- range of $t_m$ is $r_m$
- retrieve $(t_{i1}.A_{j1}, ... t_{in}.A_{jn})$
- where $P$
Differences between Quel and SQL

Equivalent expressive power, but:

- No set operations (intersection, union, minus)
- No nested retrieve-where clauses
Queries

\textbf{range of } s \textit{ is } \textbf{borrow}
\textbf{range of } t \textit{ is } \textbf{deposit}
\textbf{retrieve unique } (s.\text{customer-name})
\textbf{where } t.\text{branch-name} = \text{“Perryridge”}
\textbf{and } s.\text{branch-name} = \text{“Perryridge”}
\textbf{and } t.\text{customer-name} = s.\text{customer-name}
Other Features

Aggregate functions:
☞ count, sum, avg, any ...

Deletion:
☞ delete t [ where P ]

Updates:
☞ replace t [ where P ]

Temporary relations:
☞ retrieve into, append to
Summary

You should know the answers to these questions:

- What are QBE and Quel?
- How can a query in the tuple relational calculus be expressed in QBE?
- How do you express selections, projections, products, joins etc. in QBE?
- How can a query in the domain relational calculus be expressed in Quel?

Can you answer the following questions?

Are there queries that are easier to express in QBE than in SQL? Vice versa?
8. Integrity Constraints

Kinds of integrity constraints:
- Key declarations
- Mapping constraints
- Domain constraints
- Functional dependencies
- Assertions
- Triggers
Domain Constraints

SQL types
- fixed length strings
- fixed point numbers
- integers
- small integers
- floating point numbers
- floating and double-precision

Null values
- not null declaration
Foreign keys

Suppose \( s(S) \) and \( r(R) \) are relations with key attributes \( K_S \) and \( K_R \)

Then \( \alpha \subseteq S \) is a foreign key if

for every \( t_1 \) in \( s \) there is a (unique) \( t_2 \) in \( r \) such that \( t_1[\alpha] = t_2[K_R] \).

Alternatively, if \( \Pi_\alpha(s) \subseteq \Pi_{K_R}(r) \).
**Referential Integrity**

A *referential integrity* constraint requires that a foreign key in one relation refers to an actual, existing tuple in another relation:

![Diagram showing referential integrity](image)
Referential Integrity in SQL

Table creation constraints:

- **primary key** — list of attributes
- **unique key** — list of attributes
- **foreign key** — list of attributes referenced relation name

```sql
create table deposit
(
  branch-name char(15) not null,
  account-number char(10),
  customer-name char(20) not null,
  primary key (account-number, customer-name),
  foreign key (branch-name) references branch,
  foreign key (customer-name) references customer
)
```
Functional Dependencies

Let $\alpha, \beta \subseteq R$.
Then the functional dependency $\alpha \rightarrow \beta$ holds on $R$ if for all $t_1, t_2$ in $r(R)$

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$
## Example FDs

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
<tr>
<td>a1</td>
<td>b2</td>
<td>c1</td>
<td>d2</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
<td>d2</td>
</tr>
<tr>
<td>a2</td>
<td>b3</td>
<td>c2</td>
<td>d3</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c2</td>
<td>d4</td>
</tr>
</tbody>
</table>
**Example FDs in the Bank Database**

### deposit

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>customer-name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Perryridge</td>
<td>201</td>
<td>Williams</td>
<td>900</td>
</tr>
<tr>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
</tr>
<tr>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
</tr>
<tr>
<td>Downtown</td>
<td>105</td>
<td>Green</td>
<td>850</td>
</tr>
</tbody>
</table>

### customer

<table>
<thead>
<tr>
<th>customer-name</th>
<th>street</th>
<th>customer-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
<tr>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Glenn</td>
<td>Sand Hill</td>
<td>Woodside</td>
</tr>
<tr>
<td>Brooks</td>
<td>Senator</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Green</td>
<td>Walnut</td>
<td>Stamford</td>
</tr>
</tbody>
</table>

### branch

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Mianus</td>
<td>400000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
</tr>
</tbody>
</table>

### borrow

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>
Closure of a set of FDs

The closure of a set $F$ of FDs is the set $F^+$ of all FDs logically implied by $F$.

Armstrong’s Axioms

- Reflexivity: $\beta \subseteq \alpha \Rightarrow \alpha \rightarrow \beta$
- Augmentation: $\alpha \rightarrow \beta \Rightarrow \alpha \gamma \rightarrow \beta \gamma$
- Transitivity: $\alpha \rightarrow \beta, \beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$
Example — using closures

Consider:

\begin{align*}
A & \rightarrow B \quad (1) \\
A & \rightarrow C \quad (2) \\
CG & \rightarrow H \quad (3) \\
CG & \rightarrow I \quad (4) \\
BC & \rightarrow H \quad (5)
\end{align*}

\(\checkmark\quad \text{Can we also conclude } A \rightarrow H?\)
Derived Rules

The following rules can be derived from Armstrong’s Axioms:

- **Union:** \( \alpha \rightarrow \beta, \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta \gamma \)
- **Decomposition:** \( \alpha \rightarrow \beta \gamma \Rightarrow \alpha \rightarrow \beta, \alpha \rightarrow \gamma \)
- **Pseudotransitivity:** \( \alpha \rightarrow \beta, \beta \gamma \rightarrow \delta \Rightarrow \alpha \gamma \rightarrow \delta \)
**Closure of an attribute set**

The *closure of an attribute set* $\alpha$ is the set $\alpha^+$ of all attributes functionally determined by $\alpha$.

**Example:**

\[
\begin{align*}
AG & \rightarrow ABG & (1) \\
& \rightarrow ABCG & (2) \\
& \rightarrow ABCGH & (3) \\
& \rightarrow ABCGHI & (4)
\end{align*}
\]

**Problem:**

Given a set $F$ of FDs, show that $\alpha \rightarrow \beta$ is in $F^+$.

**Solution:**

Compute $\alpha^+$ and check that $\beta \subseteq \alpha^+$.
Finding Keys

We can now redefine a *key* of a relation R as a set of attributes K such that $K^+ = R$. A *candidate* key is a minimal such K (i.e., for any $A \in K$, $(K \setminus \{A\})^+ \neq R$)

**Problem:**
Given a relation R with FDs F, find a candidate key for R.

**Solution:**
Start with $K = R$. Remove elements from K until a minimal key is identified.

**Alternative solution:**
Find the set M of all attributes *not appearing on the RHS* of any FD in F.
If $M^+ = R$, done
else let $K = M \cup (R \setminus M^+)$
Clearly $K^+ = R$. Remove elements from K until a minimal key is identified.
Example — finding keys

Consider:

\begin{align*}
\text{AB} & \rightarrow \text{C} \quad (1) \\
\text{B} & \rightarrow \text{D} \quad (2) \\
\text{E} & \rightarrow \text{F} \quad (3) \\
\text{CE} & \rightarrow \text{A} \quad (4)
\end{align*}

- Does \text{BE} \rightarrow \text{DF}?
- Does \text{BE} \rightarrow \text{FC}?
- Is \text{BE} a superkey?
- Is \text{BE} a candidate key?
- What are all the candidate keys?
- Can you prove that you have found all of them?
A **canonical cover** $F_c$ of $F$, is a set of FDs such that

1. $F_c^+ = F^+$
2. Each $\alpha \rightarrow \beta$ in $F_c$ contains no extraneous attributes in $\alpha$
3. Each $\alpha \rightarrow \beta$ in $F_c$ contains no extraneous attributes in $\beta$
4. For each $\alpha \rightarrow \beta$ in $F_c$, $\alpha$ is unique

[Attributes are *extraneous* if they can be removed without affecting the closure.]

To compute the canonical cover, use the union rule repeatedly to join common $\alpha \rightarrow \beta_i$ (4). Then check each $\alpha \rightarrow \beta$ for extraneous attributes in $\alpha$ or $\beta$ (2,3). Repeat until stable.

🍁  Find the canonical cover for: $A \rightarrow BC$, $B \rightarrow C$, $A \rightarrow B$, $AB \rightarrow C$
 Assertions

Assertions in SQL:

```
assertion assertion-name on relation-name : predicate

assertion banker-constraint on client :
  customer-name ≠ employee-name

assertion address-constraint on insertion to deposit :
  exists ( select*
    from customer
    where customer.customer-name = deposit.customer-name
  )
```
Triggers

define trigger overdraft
  on update of deposit T
  ( if new T.balance < 0
    then ( insert into borrow
              values ( T.branch-name, T.account-number,
                       T.customer-name, - new T.balance )
              update deposit S
              set S.balance = 0
              where S.account-number = T.account-number
        )
  )
Summary

You should know the answers to these questions:

- What kinds of integrity constraints are important in database systems?
- What is a foreign key?
- What is referential integrity, and how is it guaranteed?
- How is referential integrity specified in SQL?
- What is a functional dependency?
- How do you compute the closure of a set of FDs? Of an attribute set?
- How can you show that a particular FD holds?
- How can you test if a set of FDs is a canonical cover?
- How do you compute a canonical cover for a set of FDs?

Can you answer the following questions?

- Can you tell what functional dependencies hold just by examining the database?
- How would you prove Armstrong’s Axioms?
- What is an efficient algorithm for computing the closure of an attribute set?
- How can you find a candidate key for a relation?
9. Database Design

Seek to avoid:

- Repetition of information
- Inability to represent certain information
- Loss of information

Overview

- Lossless joins
- Normalization
- Dependency preservation
- Boyce-Codd Normal Form (BCNF)
- Third Normal Form (3NF)
**Example**

### Borrow-Scheme

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>

### Branch-Scheme

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Mianus</td>
<td>400000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
</tr>
</tbody>
</table>
Repetition of Information

Lending-Scheme

<table>
<thead>
<tr>
<th>branch-name</th>
<th>assets</th>
<th>branch-city</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>2100000</td>
<td>Palo Alto</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>400000</td>
<td>Horseneck</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>8000000</td>
<td>Horseneck</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>300000</td>
<td>Bennington</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>3700000</td>
<td>Rye</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>9000000</td>
<td>Brooklyn</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1700000</td>
<td>Horseneck</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>7100000</td>
<td>Brooklyn</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
</tbody>
</table>
**Lossy Joins**

Consider decomposing Borrow-Scheme as follows:

**Amt-Scheme**

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>2200</td>
</tr>
</tbody>
</table>

**Loan-Scheme**

<table>
<thead>
<tr>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Jones</td>
</tr>
<tr>
<td>2000</td>
<td>Smith</td>
</tr>
<tr>
<td>1500</td>
<td>Hayes</td>
</tr>
<tr>
<td>1500</td>
<td>Jackson</td>
</tr>
<tr>
<td>500</td>
<td>Curry</td>
</tr>
<tr>
<td>900</td>
<td>Turner</td>
</tr>
<tr>
<td>1200</td>
<td>Williams</td>
</tr>
<tr>
<td>1300</td>
<td>Adams</td>
</tr>
<tr>
<td>2000</td>
<td>Johnson</td>
</tr>
<tr>
<td>2500</td>
<td>Glenn</td>
</tr>
<tr>
<td>2200</td>
<td>Brooks</td>
</tr>
</tbody>
</table>
Lossy Joins

amt \Join loan

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>customer-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Mianus</td>
<td>93</td>
<td>Curry</td>
<td>500</td>
</tr>
<tr>
<td>Round Hill</td>
<td>11</td>
<td>Turner</td>
<td>900</td>
</tr>
<tr>
<td>Pownal</td>
<td>29</td>
<td>Williams</td>
<td>1200</td>
</tr>
<tr>
<td>North Town</td>
<td>16</td>
<td>Adams</td>
<td>1300</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>25</td>
<td>Glenn</td>
<td>2500</td>
</tr>
<tr>
<td>Brighton</td>
<td>10</td>
<td>Brooks</td>
<td>2200</td>
</tr>
<tr>
<td>Perryridge</td>
<td>15</td>
<td>Jackson</td>
<td>1500</td>
</tr>
<tr>
<td>Downtown</td>
<td>14</td>
<td>Hayes</td>
<td>1500</td>
</tr>
<tr>
<td>Redwood</td>
<td>23</td>
<td>Johnson</td>
<td>2000</td>
</tr>
<tr>
<td>Downtown</td>
<td>18</td>
<td>Smith</td>
<td>2000</td>
</tr>
</tbody>
</table>
**Decomposition**

A *decomposition* of a relation scheme $R$ is a set of relation schemes \{ $R_1$, ..., $R_n$ \} such that $R = \bigcup_i R_i$.

Let $C$ be a set of constraints (e.g., functional dependencies) over a database. A decomposition \{ $R_1$, ..., $R_n$ \} of relation scheme $R$ is a *lossless-join decomposition* if for every relation $r$ that satisfies $C$, it is true that

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie ... \bowtie \Pi_{R_n}(r)$$
**Normalisation**

Lending-scheme = (branch-name, assets, branch-city, loan-number, customer-name, amount)

with FDs:

- branch-name → assets
- branch-city
- loan-number → amount
- branch-name

*Decompose into:*

- Branch-scheme = (branch-name, assets, branch-city)
- Loan-info-scheme = (branch-name, loan-number, amount)
- Customer-loan-scheme = (loan-number, customer-name)
**Lossless Join Decomposition**

Suppose $F$ is a set of functional dependencies over $R$. Then $R = R_1 \cup R_2$ is a lossless-join decomposition if either of

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

is in $F^+$. 
Lossless Join Decomposition

Use

\[ \text{branch-name} \rightarrow \text{assets branch-city} \]

to decompose \textit{Lending-scheme} into

Branch-scheme = (branch-name, assets, branch-city)
Borrow-scheme = (branch-name, loan-number, customer-name, amount)

Then, use

\[ \text{loan-number} \rightarrow \text{amount branch-name} \]

to decompose \textit{Borrow-scheme} into

Loan-info-scheme = (branch-name, loan-number, amount)
Customer-loan-scheme = (loan-number, customer-name)
**Dependency Preservation**

**Goal:**
- avoid taking joins to check integrity constraints upon updates

**Approach:**
- ensure that functional dependencies restricted to individual relation schemes are equivalent to the original set of FDs

The *restriction* of $F$ to $R_i$, where \( \{R_1, \ldots, R_n\} \) is a decomposition of $R$, is the set $F_i$ of FDs in $F^+$ including only attributes in $R_i$.

\( \{R_1, \ldots, R_n\} \) is a *dependency-preserving decomposition* of $R$ if the closure of $\bigcup F_i$ is equal to the closure $F^+$ of $F$. 
Normal Forms

Repetition of information typically occurs when FDs $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ occur within the same relation. Various normal forms have been introduced to avoid these problems.

*Boyce-Codd Normal Form*

- only allow superkey FDs to occur in relation schemes

*Third Normal Form*

- also allow transitive FDs

*Fourth Normal Form*

- like BCNF, but applied to “multivalued dependencies”
**Boyce-Codd Normal Form**

A relation scheme $R$ is in *Boyce-Codd Normal Form* if for every FD $\alpha \rightarrow \beta$ holding over $R$, either

1. $\alpha \rightarrow \beta$ is a trivial FD (i.e., $\beta \subseteq \alpha$), or
2. $\alpha$ is a superkey for $R$

A database schema is in *BCNF* if each relation scheme is in BCNF.

Branch-scheme = (branch-name, assets, branch-city)

$\text{branch-name} \rightarrow \text{assets} \text{ branch-city}$

Borrow-scheme = (branch-name, loan-number, customer-name, amount)

$\text{loan-number} \rightarrow \text{amount} \text{ branch-name}$
BCNF Decomposition Algorithm

while some R is not in BCNF
  select non-trivial $\alpha \rightarrow \beta$ holding on R where
  $\alpha \rightarrow R$ is not in $F^+$ and $\alpha \cap \beta = \emptyset$
  replace R by $\alpha \cup \beta$ and (R-$\beta$)

The algorithm terminates, generates a BCNF schema, and satisfies lossless join.
Shortfalls of BCNF

BCNF schemas are *not* necessarily dependency preserving! ...

Consider:

Banker-scheme = (branch-name, customer-name, banker-name)

*banker-name* → *branch-name*

*customer-name* *branch-name* → *banker-name*

Decompositions are not necessarily unique.

Consider: a → b c, b d → a
Third Normal Form

A relation scheme $R$ is in Third Normal Form if for every FD $\alpha \rightarrow \beta$ holding over $R$, either

1. $\alpha \rightarrow \beta$ is a trivial FD (i.e., $\beta \subseteq \alpha$), or
2. $\alpha$ is a superkey for $R$, or
3. each attribute $A$ in $\beta$ is contained in a candidate key for $R$.

A database schema is in 3NF if each relation scheme is in 3NF.
3NF Decomposition Algorithm

Given F in canonical form for relation scheme R:

\[ D = \emptyset \]

for each \( \alpha \rightarrow \beta \) in F

if no scheme in D contains \( \alpha \beta \)
then add \( \alpha \beta \) to D

if no scheme in D contains a candidate key for R
then add any candidate key for R to D

Guarantees 3NF, lossless join, and dependency preservation.
**BCNF vs. 3NF**

- BCNF is preferable if the resulting schema is also dependency-preserving.

- Otherwise 3NF is preferable, to reduce the cost of maintaining integrity constraints.

- In the presence of transitive FDs, 3NF may introduce redundancies and may require null values.

**Goal:**

- BCNF + lossless join + dependency preservation

*If not possible, accept:*

- 3NF + lossless join + dependency preservation
Summary

You should know the answers to these questions:
- What is a lossy join? What is lost in a lossy join?
- What is a lossless-join decomposition?
- What is dependency preservation?
- What is BCNF? How does the BCNF decomposition algorithm work?
- What is 3NF? How does the 3NF decomposition algorithm work?

Can you answer the following questions?
- Why does lossless join decomposition work correctly?
- Why is the BCNF decomposition algorithm correct?
- Is it possible for a relation scheme to be in BCNF yet not guarantee a lossless join?
- What about 3NF?
- Does BCNF imply 3NF?
- Why is it always possible to find a 3NF decomposition that is lossless-join and dependency preserving, but not always a BCNF one?
- Are 3NF schemas necessarily dependency preserving?
10. File and System Structure

Overview

❑ Storage media
❑ File Organization
❑ Buffer Management
Physical Storage Media

Main memory:
- fast, small, volatile, expensive

Disk storage:
- slower, large, persistent

Tape storage:
- slow, sequential, archival, cheap
Disk Storage

A Disk Pack

track 0
track n
arm 0
actuator
platter 0
platter m-1
arm n-1
File Organisation

Blocks are fixed-size units of memory on a disk.

A file is organized logically as a sequence of records mapped onto disk blocks. Records within a given file may be either fixed or variable length.

- Fixed-length records: simple and efficient to implement; inflexible for representing complex information

- Variable length records: more flexible; problems with memory fragmentation, wasted storage, slower searching
Fixed-length records

```haskell
type deposit =
  record
    branch-name : char (20) ;
    account-number : integer ;
    customer-name : char (20) ;
    balance : real ;
  end
```

Record length = 52 bytes (20 + 4 + 20 + 8)
☞ alignment with block boundaries?
☞ insertions and deletions?
Insertions and deletions

Rather than moving data when records are deleted, a free list of deleted records is maintained: deletions and insertions occur at the head of the list. When the list is empty, new records are inserted at the end of the file.

<table>
<thead>
<tr>
<th></th>
<th>Downtown</th>
<th>101</th>
<th>Johnson</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>record 0</td>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
</tr>
<tr>
<td>record 1</td>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>record 2</td>
<td>Perryridge</td>
<td>201</td>
<td>Williams</td>
<td>900</td>
</tr>
<tr>
<td>record 3</td>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
</tr>
<tr>
<td>record 4</td>
<td>Downtown</td>
<td>105</td>
<td>Green</td>
<td>850</td>
</tr>
</tbody>
</table>
Variable length records

- multiple record types per file
- repeating fields
- variable length fields

```pascal
type deposit-list =
    record
        branch-name : char (20) ;
        account-info : array [1 .. ] of
            record
                account-number : integer ;
                customer-name : char (20) ;
                balance : real ;
            end
        end
    end
```
## Byte String Representation

<table>
<thead>
<tr>
<th></th>
<th>Perryridge</th>
<th>Hayes</th>
<th>400</th>
<th>201</th>
<th>Williams</th>
<th>900</th>
<th>218</th>
<th>Lyle</th>
<th>700</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
<td>110</td>
<td>Peterson</td>
<td>600</td>
<td>⊥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use special end-of-record marker (⊥)

- Hard to reuse space; can lead to fragmentation
- Costly to handle record growth
**Fixed-Length Representation**

1. **Reserved space:** requires fixed maximum space for records

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
<td>201</td>
<td>Williams</td>
</tr>
<tr>
<td>1</td>
<td>Round Hill</td>
<td>305</td>
<td>Turner</td>
<td>350</td>
<td>⊥️</td>
<td>⊥️</td>
</tr>
<tr>
<td>2</td>
<td>Mianus</td>
<td>215</td>
<td>Smith</td>
<td>700</td>
<td>⊥️</td>
<td>⊥️</td>
</tr>
<tr>
<td>3</td>
<td>Downtown</td>
<td>101</td>
<td>Johnson</td>
<td>500</td>
<td>110</td>
<td>Peterson</td>
</tr>
<tr>
<td>4</td>
<td>Redwood</td>
<td>222</td>
<td>Lindsay</td>
<td>700</td>
<td>⊥️</td>
<td>⊥️</td>
</tr>
<tr>
<td>5</td>
<td>Brighton</td>
<td>217</td>
<td>Green</td>
<td>750</td>
<td>⊥️</td>
<td>⊥️</td>
</tr>
</tbody>
</table>

2. **Pointers:** represent variable length record by chain of fixed-length records

   ![Pointer diagram]
Anchor/overflow block organization

To save space, records can be separated into anchor blocks and overflow blocks:

- Anchor block:
  - Perryridge 102
  - Round Hill 305
  - Mianus 215
  - Downtown 101
  - Redwood 222
  - Brighton 217

- Overflow block:
  - Williams 900
  - Peterson 600
  - Lyle 700

Example:

- Perryridge 102 Hayes 400
- Round Hill 305 Turner 350
- Mianus 215 Smith 700
- Downtown 101 Johnson 500
- Redwood 222 Lindsay 700
- Brighton 217 Green 750
- Williams 900
- Peterson 600
- Lyle 700
Organizing Records into Blocks

To reduce seek-time for retrieving related records, organize into chained blocks. Related blocks should be stored on the same, or nearby cylinders. Need separate free lists to maintain closeness with insertions and updates.

Trade-off between time and space efficiency.
Sequential Files

Sequential files are pre-sorted to support fast retrieval by a search key.

Deletions are kept on a free list for each block. Insertions are made to free slots on the same block if possible, otherwise to an overflow block.

Requires occasional reorganisation.
Mapping Relational Data to Files

- Tuples are usually fixed-length records, and relations can be mapped to simple file structures.

- For very large databases assignments of records to blocks can have a critical impact on performance, and more complex file structures may be needed.

- Large-scale database systems may bypass the operating system’s file management by storing the entire database in a single system file. Related tuples in separate relations may be clustered together to efficiently implement commonly expected joins — e.g., deposit customer — though this may slow down other queries ...
Data Dictionary Storage

The Data Dictionary may itself be accessed as a database

Database schema:
- Names of relations; names and domains of attributes
- Names and definitions of views
- Integrity constraints for each relation (e.g., keys)

Users:
- User names and authorization; accounting information

Statistics and Technical details:
- Number of tuples per relation; types of queries
- Storage method used per relation (e.g., clustered)
- Indexed relations and attributes; types of indices
Buffer Management

Main memory (buffer)

When the buffer is full, blocks to be read in must replace existing blocks. What strategy should be used (LRU, MRU ...)?

Modified blocks must be written out in a controlled fashion to maintain consistency. Output may be forced, or even restricted for temporarily pinned blocks.

Permanent Storage
Buffer Management

- Commonly accessed information (data dictionary, indices) should remain in memory.

- Statistics may help to determine which relations are likely to be accessed.

- The way in which queries are processed may affect the order in which blocks should be read and replaced.

- In the presence of concurrent users, certain requests may need to be delayed to maintain consistency — loading of blocks needed by these requests can therefore be delayed.

- Writing of modified blocks must be coordinated by the crash recovery system. (Updates must be atomic in the presence of system failures.)
Summary

You should know the answers to these questions:

- How should related disk blocks be organized to speed up access?
- Why are variable-length records harder to manage than fixed-length?
- What is a free list? How is it used?
- What is fragmentation? How does it arise?
- What are “anchor” and “overflow” blocks? Why are they useful?
- How can sequential files speed up access time?
- What is the role and function of the database buffer?

Can you answer the following questions?

- How is a free list initialized?
- Can variable length records arise in relational databases?
- Why must one often trade-off time against space efficiency?
- Why do many database systems need to bypass the file system?
- What kind of information can be used to fine-tune database performance?
- How must modified blocks be written to disk to guarantee atomicity?
11. Indexing and Hashing

Overview
- Index Sequential Files; primary and secondary indices
- \(B^+\)-trees and B-trees
- Hashing; static and dynamic hashing
Basic Concepts

Access time:
☞ How long does it take to find items?

Insertion time:
☞ How long does it take to insert items (including time to update index structure)?

Deletion time:
☞ How long to delete items (and update index structure)?

Space overhead:
☞ What is the cost of extra space?
Indexing

Primary index:
- file is sorted by primary search key
- all matching records are in the same or nearby blocks

Secondary index:
- index on other attributes
- matching records may be in arbitrary blocks
- “buckets” of pointers point to actual records

Dense index:
- index record for every search-key value

Sparse index:
- index record only for selected search-key values
  e.g., first record of each block/bucket
**Dense and sparse indices**

### Dense index

- **Brighton**
  - 217 Green 750
- **Downtown**
  - 101 Johnson 500
  - 110 Peterson 600
- **Mianus**
  - 215 Smith 700
- **Perryridge**
  - 102 Hayes 400
  - 201 Williams 900
  - 218 Lyle 700
- **Redwood**
  - 222 Lindsay 700
- **Round Hill**
  - 305 Turner 350

**Deletion:** Look up and delete record; if this is the last record with this search value, also delete search key in index

**Insertion:** Lookup and insert record; add search-key to index if needed

### Sparse index

- **Brighton**
  - 217 Green 750
- **Downtown**
  - 101 Johnson 500
  - 110 Peterson 600
- **Mianus**
  - 215 Smith 700
- **Perryridge**
  - 102 Hayes 400
  - 201 Williams 900
  - 218 Lyle 700
- **Redwood**
  - 222 Lindsay 700
- **Round Hill**
  - 305 Turner 350

**Deletion:** Look up and delete record; replace search key in index by that of next record (or delete if already in index)

**Insertion:** Lookup and insert record; add new search key to index only if new block is created
Indices

- Records can be retrieved more quickly with dense indices, but these may take up a great deal of space.

- Cost of searching in memory is low compared to cost of reading a block; so sparse indices are used to locate blocks to read. (One search-key entry per block.)

- If the primary index does not fit into memory, a second-level sparse index may be constructed (even for very large databases, two levels usually suffice).
Secondary indices

Buckets group together pointers to records with nearby secondary “keys”.

Bucket entries may also contain the search key value to reduce the cost of retrieving individual records.
**B+ Tree Index Files**

Index-sequential files perform poorly as database grows; B+ Trees perform better under frequent modifications

- Tree of ranges of search key values
- Nodes contain search keys and pointers to nodes/records
- Each node has $m$ children, between $\lceil n/2 \rceil$ and $n$ ($n$ is fixed)
- Search key values are in sort order
- Leaf nodes point to records (for primary keys) or to buckets
- Pointer $P_n$ is also used to chain together leaf nodes
- Insertions/deletions may cause nodes to split/coalesce if $m$ leaves the range $(\lceil n/2 \rceil, n)$
B+ Tree Insertions

Insertion of “Clearview” causes leaf node to split
B+ Tree Deletions

Deletion of “Downtown”:

Deletion of “Perryridge”: 
B-Tree Index Files

Similar to B+ Trees, except:

- Every node contains pointers to records/buckets, not just leaf nodes (additional pointers needed); so some records can be found more quickly
- Leaf nodes are not chained
- Deletions are more complicated since non-leaf nodes that become too small will require local reorganizations

Advantages are marginal for large indices, so B+ trees are usually preferred.
**Hash Functions**

A *hash function* $h$ maps search keys $K$ to bucket addresses $B$.

To perform a lookup on search key $k_i$, compute $h(k_i)$, and scan the bucket for the key value.

A *good* hash function assigns search keys to buckets:
- with uniform distribution (*over the entire space* $K$)
- with random distribution (*for arbitrary subsets of* $K$)
Static hash functions

- Insertion and deletion are straightforward (lookup the bucket and insert or delete)

- The hash function and number of buckets must be fixed in advance; space is wasted if too many buckets are chosen, but performance will suffer if the buckets become too full
  - choose hash function based on current file size (performance will degrade with time)
  - choose hash function based on anticipated file size (initially wastes space)
  - periodically reorganize the hash structure (time-consuming and disruptive)
Dynamic hash functions

- **Extendible** hash function computes a value for a very large number of buckets, e.g., $2^{32}$

- First $k$ bits of hash value are used to look up the actual bucket in a *bucket address table*

- Multiple entries may point to the same bucket

- As buckets grow too big and are split, the bucket address table is modified accordingly

- When the table can no longer accommodate split buckets, $k$ is incremented and the table is expanded
Dynamic Hashing example

Hash function (abbreviated)

Sample deposit file

<table>
<thead>
<tr>
<th>Location</th>
<th>City</th>
<th>Street</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Brighton</td>
<td></td>
<td>217</td>
</tr>
<tr>
<td>0</td>
<td>Clearview</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td>0</td>
<td>Downtown</td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td>0</td>
<td>Mianus</td>
<td></td>
<td>215</td>
</tr>
<tr>
<td>0</td>
<td>Perryridge</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>Redwood</td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td>0</td>
<td>Round Hill</td>
<td></td>
<td>0101</td>
</tr>
<tr>
<td>1</td>
<td>Brighton</td>
<td></td>
<td>217</td>
</tr>
<tr>
<td>1</td>
<td>Clearview</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td>Downtown</td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>Mianus</td>
<td></td>
<td>215</td>
</tr>
<tr>
<td>1</td>
<td>Perryridge</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>Redwood</td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td>1</td>
<td>Round Hill</td>
<td></td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>Brighton</td>
<td></td>
<td>217</td>
</tr>
<tr>
<td>2</td>
<td>Clearview</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>Downtown</td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td>2</td>
<td>Mianus</td>
<td></td>
<td>215</td>
</tr>
<tr>
<td>2</td>
<td>Perryridge</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>Redwood</td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td>2</td>
<td>Round Hill</td>
<td></td>
<td>0101</td>
</tr>
</tbody>
</table>
Hashing vs. Indexing

What kinds of queries will be most common?

- Hashing is more efficient for equality selections (attribute = key value)
  - index lookup takes time $O(\log(n))$ for $n$ values
  - hash lookup is constant time (though worst case is $O(n)$)

- Indexes are more efficient for range selections (attribute in range $[c_1, c_2]$)
  - since indices use sorted files or buckets, ranges are easy to find
  - not so for hash structures; order-preserving hash functions are hard to find (conflicts with uniformity and randomness!)
Summary

You should know the answers to these questions:

- What are primary and secondary indices?
- How are insertions and deletions handled with dense/sparse indices?
- What is the structure of a valid B+ tree?
- When must nodes be split/coalesced in a B+ tree?
- How are hash functions used to find key values?
- What are the limitations of static hash functions?
- What are the relative advantages of indexing and hashing?

Can you answer the following questions?

- Why do secondary indices point to buckets rather than individual records?
- When must node values be redistributed in a B+ tree?
- What is the space overhead for a B+ tree?
- What are examples of good/bad hash functions?
12. Transactions and Concurrency Control

Overview

- Transactions
- Recovery logs
- Serializability
- Two-phase locking
Transactions

A transaction must satisfy the “ACID” properties:

- **Atomicity**: either all transaction operations must complete or none
- **Consistency**: correct execution must ensure database consistency
- **Isolation**: intermediate states are not visible to other transactions
- **Durability**: once committed, a transaction is resistant to failures

```
T: read(A, a)
   a := a - 50
write(A, a)
read(B, b)
   b := b + 50
write(B, b)
```

**read** and **write** operations to memory may trigger **input** from disk; **output** to disk must ensure database consistency
**Transaction States**

*Active* — the initial state

*Partially committed* — after the last statement has been executed

*Failed* — after normal execution is no longer possible

*Aborted* — after the transaction is rolled back

*Committed* — after successful completion
Aborted Transactions

Aborted Transactions must leave the (permanent) database in a consistent state.

Two options after abortion:

- **Restart**: only possible if the transaction was aborted for external reasons (e.g., crash, deadlock, etc.)

- **Kill the transaction**: should only occur if it is logically impossible to complete the transaction (e.g., unavailable data, bad input, etc.)


**Recovery Logs**

*Principle idea:* Achieve atomicity by logging all modifications and transaction state changes to stable storage *without* modifying the database until a transaction commits.

Committed transactions can be safely *redone* after a crash by re-running the logged modifications. (*Redo must be idempotent.*)

Log entries may contain:
- Transaction name
- Data item name
- Old value
- New value
- Transaction state changes (*start* and *commit*)
Deferred Database Modification

T1: \texttt{read}(A)
A := A - 50
\texttt{write}(A)
\texttt{read}(B)
B := B + 50
\texttt{write}(B)

T2: \texttt{read}(C)
C := C - 100
\texttt{write}(C)

<table>
<thead>
<tr>
<th>Log</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T1 starts&gt;</td>
<td>A = 1000</td>
</tr>
<tr>
<td>&lt;T1, A, 950&gt;</td>
<td>B = 2000</td>
</tr>
<tr>
<td>&lt;T1 commits&gt;</td>
<td>C = 700</td>
</tr>
<tr>
<td>&lt;T2 starts&gt;</td>
<td>A = 950</td>
</tr>
<tr>
<td>&lt;T2, C, 600&gt;</td>
<td>B = 2050</td>
</tr>
<tr>
<td>&lt;T2 commits&gt;</td>
<td>C = 600</td>
</tr>
</tbody>
</table>
**Immediate Database Modification**

Logged updates can be immediately reflected in stable storage if both *old* and *new* values are logged: after failure, uncompleted transactions must first be *undone* by restoring old values, and then completed transactions must be *redone.*

<table>
<thead>
<tr>
<th>Log</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T1 starts&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;T1, A, 1000, 950&gt;</td>
<td>A = 950</td>
</tr>
<tr>
<td>&lt;T1, B, 2000, 2050&gt;</td>
<td>B = 2050</td>
</tr>
<tr>
<td>&lt;T1 commits&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;T2 starts&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;T2, C, 700, 600&gt;</td>
<td>C = 600</td>
</tr>
<tr>
<td>&lt;T2 commits&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Log Record Buffering

- All log records for $T$ must be output to stable storage before the $<T\ commit>$ log record is output.

- Transaction $T$ enters the commit state after the $<T\ commit>$ log record has been output to stable storage.

- All log records pertaining to a block of data in memory must be output to stable storage before the block itself is output.

NB: if blocks in memory must be swapped out to make room for new blocks, all log records for the block to be swapped out must first be output to stable storage.
## Concurrent and Serializable Schedules

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong>(A)</td>
<td></td>
<td><strong>read</strong>(A)</td>
<td></td>
</tr>
<tr>
<td>A := A - 50</td>
<td></td>
<td>A := A - 50</td>
<td></td>
</tr>
<tr>
<td><strong>write</strong>(A)</td>
<td></td>
<td><strong>write</strong>(A)</td>
<td><strong>read</strong>(A)</td>
</tr>
<tr>
<td><strong>read</strong>(B)</td>
<td></td>
<td></td>
<td><strong>read</strong>(B)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td></td>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td><strong>write</strong>(B)</td>
<td></td>
<td><strong>write</strong>(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>temp := A * 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := A - temp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>write</strong>(A)</td>
<td></td>
<td><strong>write</strong>(A)</td>
</tr>
<tr>
<td></td>
<td><strong>read</strong>(B)</td>
<td></td>
<td><strong>read</strong>(B)</td>
</tr>
<tr>
<td></td>
<td>B := B + temp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>write</strong>(B)</td>
<td></td>
<td><strong>write</strong>(B)</td>
</tr>
</tbody>
</table>

Universität Bern
### Non-serializable Schedules

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A - 50</td>
<td>temp := A * 0.1</td>
</tr>
<tr>
<td></td>
<td>A := A - temp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>B := B + 50</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>

A non-serializable schedule is not equivalent to any serial schedule, and leaves the database in an inconsistent state.
Conflict Serializability

Read and write instructions $I_i$ and $I_j$ of separate transactions $T_i$ and $T_j$ within a schedule may be interchanged if they do not conflict.

$I_i$ and $I_j$ conflict if they refer to the same data item $Q$, and one of the two is a write operation.

$$
\begin{array}{c|c}
T_i & T_j \\
\hline
\text{read}(Q) & \text{read}(Q) \\
\end{array}
$$

$$
\begin{array}{c|c}
T_i & T_j \\
\hline
\text{read}(Q) & \text{read}(Q) \\
\end{array}
$$

$$
\begin{array}{c|c}
T_i & T_j \\
\hline
\text{write}(Q) & \text{write}(Q) \\
\end{array}
$$
Serializing Schedules

A schedule is conflict-serializable if it can be transformed into a serial schedule by interchanging non-conflicting instructions.
Testing for Conflict Serializability

A schedule is *conflict serializable* if there are no cycles in its *precedence graph*.

Construct a *precedence graph* by introducing one node for each transaction, and an edge from $T_i$ to $T_j$ if:

- $T_i$ executes $\text{write}(Q)$ before $T_j$ executes $\text{read}(Q)$, or
- $T_i$ executes $\text{read}(Q)$ before $T_j$ executes $\text{write}(Q)$, or
- $T_i$ executes $\text{write}(Q)$ before $T_j$ executes $\text{write}(Q)$.

![Diagram showing serializable and not serializable precedence graphs]
**Sorting Precedence Graphs**

A *topological sorting* of the precedence graph yields a possible serialization.
Locks

Serializability can be ensured by locking data items accessed by a transaction according to a locking protocol ...

Shared (read) locks:
- If transaction T obtains a shared lock (lock-S) on Q, it may read but not write Q
- A shared lock for Q may be obtained only if no exclusive lock for Q is already held by another transaction.

Exclusive (write) locks:
- If T obtains an exclusive lock (lock-X) on Q, it may both read and write Q
- An exclusive lock for Q may only be obtained if no lock for Q is held by another transaction.

A transaction may upgrade a shared lock to an exclusive lock if no other locks are held by other transactions.
Two-phase Locking Protocol

Two-phase locking ensures serializability by ensuring that inconsistent database states cannot be seen by other transactions.

- **Growing phase**: first, a transaction may obtain locks, but may not release them.
- **Shrinking phase**: then, a transaction may release locks, but may not obtain any new locks.

Two-phase locking guarantees conflict-serializability, but does not avoid deadlock...
## Locking Protocols

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td>lock-X(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>unlock(A)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>lock-X(B)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

Unserializable schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two-phase, serializable schedule
Deadlock

Two-phase locking is not sufficient to avoid deadlock.

Deadlock is detected by constructing a waits-for graph and checking for cycles.

Two-phase, deadlocking schedule

- T1
  - lock-X(A)
  - read(A)
  - write(A)
- T3
  - lock-X(B)
  - read(B)
  - write(B)
  - lock-X(A)

Waits-for graph
Deadlock Recovery

Deadlock is resolved by picking a *victim* to roll back:

- The victim should be selected to minimize the overall cost of rolling back and restarting the victim
  - computation time?
  - number of data items used so far? still needed?
  - how many transactions to roll back?

- Partial rollback may be sufficient

- Starvation must be avoided
Summary

You should know the answers to these questions:
- What properties must a transaction satisfy?
- When may an aborted transaction be restarted?
- How does a recovery log help to achieve atomicity?
- When can transaction updates actually be reflected in the database?
- How can you check if two transactions are conflict-serializable?
- How can you derive an equivalent serial schedule from a set of interleaved, but serializable transactions?
- How does two-phase locking ensure serializability?
- How can you detect and resolve deadlock?

Can you answer the following questions?
- Can two transactions be unserializable, yet still lead to a consistent database state?
- How can you avoid redoing all committed transactions after a failure?
- How can you avoid deadlock in the first place?
13. Query Processing

Overview
- Equivalence of expressions
- Estimation of query-processing cost
- Join strategies
Equivalence of Expressions

Textual queries in, e.g., SQL, are parsed and represented internally in a form based on relational algebra.

- Each R.A. expression determines a certain evaluation order
- Formally equivalent expressions may differ in efficiency
- Various rules can be applied to transform queries to more efficient forms
**Selection**

Customer-Scheme = (customer-name, street, customer-city)
Deposit-Scheme = (branch-name, account-number, customer-name, balance)
Branch-Scheme = (branch-name, assets, branch-city)

Consider:

\[
\Pi_{\text{branch-name}, \text{assets}} (\sigma_{\text{customer-city} = \text{"Port Chester"}} (\text{customer} \bowtie \text{deposit} \bowtie \text{branch}))
\]

vs.:

\[
\Pi_{\text{branch-name}, \text{assets}} (\sigma_{\text{customer-city} = \text{"Port Chester"}} (\text{customer}) \bowtie \text{deposit} \bowtie \text{branch})
\]

☞ Perform selections as early as possible

✎ How can you formalize this rule?
Conjunctions

\[ \Pi_{\text{branch-name, assets}} (\sigma_{\text{customer-city = "Port Chester" \land balance > 1000}} (\text{customer} \bowtie \text{deposit} \bowtie \text{branch})) \]

- Replace expressions of the form:
  \[ \sigma_{P_1 \land P_2}(e) \] by \[ \sigma_{P_1}(\sigma_{P_2}(e)) \]
Projections

Consider:
\[ \Pi_{\text{branch-name, assets}} (\sigma_{\text{customer-city = "Port Chester"}} (\text{customer} \bowtie \text{deposit} \bowtie \text{branch})) \]

vs.
\[ \Pi_{\text{branch-name, assets}} ( \Pi_{\text{branch-name}} (\sigma_{\text{customer-city = "Port Chester"}} (\text{customer} \bowtie \text{deposit} \bowtie \text{branch})) \]

☞ Perform projections early

✎ How can you formalize this rule?
**Natural Joins**

Consider:

\[ \sigma_{\text{customer-city} = "\text{Harrison}"} (\text{customer}) \bowtie (\text{deposit} \bowtie \text{branch}) \]

vs.

\[ (\sigma_{\text{customer-city} = "\text{Harrison}"} (\text{customer}) \bowtie \text{branch}) \bowtie \text{deposit} \]

vs.

\[ (\sigma_{\text{customer-city} = "\text{Harrison}"} (\text{customer}) \bowtie \text{deposit}) \bowtie \text{branch} \]

☞ Rearrange multiple joins to minimize temporary results
Other transformations

\[ \sigma_P(\sigma_Q(r)) = \sigma_Q(\sigma_P(r)) \]
\[ \sigma_P(r_1 \cup r_2) = \sigma_P(r_1) \cup \sigma_P(r_2) \]
\[ \sigma_P(r_1 - r_2) = \sigma_P(r_1) - r_2 = \sigma_P(r_1) - \sigma_P(r_2) \]
\[ \pi_A(\pi_B(...\pi_X(r))) = \pi_A(r) \]
\[ \pi_A(\sigma_{A=v}(r)) = \sigma_{A=v}(\pi_A(r)) \]
\[ r \bowtie s = s \bowtie r \]
\[ (r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3) \]
\[ r_2 \cup r_1 = r_1 \cup r_2 \]
Estimation of Query-Processing Cost

Need various statistics:

- $n_r$ — the number of tuples in relation $r$
- $s_r$ — the size of a tuple in relation $r$ (in bytes)
- $V(A, r)$ — the number of distinct values for attribute $A$ in $r$

Can assume that, on average, $\sigma_{A=a}(r)$ will have $\frac{n_r}{V(A, r)}$ tuples
**Joins**

Consider $r_1 \bowtie r_2$, where $r_1(R_1)$ and $r_2(R_2)$

1. If $R_1 \cap R_2 = \emptyset$ then size is $n_{r_1} \cdot n_{r_2}$
2. If $R_1 \cap R_2$ is a key for $R_1$, then size is at most $n_{r_2}$
3. If $A = R_1 \cap R_2$ is not a key, then a tuple in $r_1$ will join with at most $n_{r_2} \cdot \frac{1}{V(A, r_2)}$ tuples in $r_2$. By symmetry, the join contains at most

$$\min\left(\frac{n_{r_1} \cdot n_{r_2}}{V(A, r_1)}, \frac{n_{r_1} \cdot n_{r_2}}{V(A, r_2)}\right)$$
tuples.
Consider:

```sql
select account-number
from deposit
where branch-name = "Perryridge"
and customer-name = "Williams"
and balance > 1000
```

where
- 20 deposit tuples fit on one block
- \( V(\text{branch-name}, \text{deposit}) = 50 \)
- \( V(\text{customer-name}, \text{deposit}) = 200 \)
- \( V(\text{balance}, \text{deposit}) = 5000 \)
- \( n_{\text{deposit}} = 10000 \)
- there is a clustering B+ tree index for `branch-name`
- there is a non-clustering B+ tree index for `customer-name`
Query Strategies Using Indices

1. Use index on *branch-name*: 12 block accesses
   - 50 tuples occupy 3-5 leaf nodes (assume 20 entries per node) for a total of 2 block accesses (root + leaf)
   - 200 clustered tuples occupy 10 blocks

2. Use index on *customer-name*: 52 block accesses
   - 200 tuples occupy 10-20 leaf nodes: 2 block accesses
   - 50 non-clustered tuples occupy 50 blocks

3. Use both indices: 5 blocks
   - 4 blocks to retrieve *pointers* to 200 + 50 records
   - compute intersection to yield 1 in 50 × 200 = 10000 pointers: 1 more block to access
**Join Strategies**

Depends on:
- physical order of tuples
- presence and type (clustering) of indices
- cost of computing a temporary index for a single query

Consider: deposit $\bowtie$ customer
- $n_{deposit} = 10000$
- $n_{customer} = 200$
**Simple vs. Block-oriented Iteration**

**Simple Iteration**

for each tuple \( d \) in deposit

for each tuple \( c \) in customer

compare common attributes

500 blocks + 10000 tuples x 10 blocks = 100500 blocks

**Block-Oriented Iteration**

for each block in deposit

for each block in customer

compare common attributes of each pair of tuples in the two blocks

500 blocks + 500 blocks x 10 blocks = 5500 blocks

*NB:* if customer fits entirely into memory, then:

500 blocks + 10 blocks = 510 blocks
Merge Join (Sorted Join Attributes)

If relations to be joined are both sorted by their join attribute, the join can be efficiently computed by reading blocks in sort order.
Computing Joins with Indices

Assume tuples are physically unclustered; an unclustered index exists on customer-name for customer:

for each tuple d in deposit
    look up matching tuples in customer

10000 blocks + 10000 tuples x 3 blocks = 40000 block accesses
(vs. 100500 block accesses)

(2 index blocks + 1 record block = 3 block accesses)

NB: it may be worthwhile to construct a temporary index to compute large joins.
Summary

You should know the answers to these questions:

- What kinds of query transformation may speed up evaluation?
- Why should selections and projections be performed as early as possible?
- How can you estimate the cost of evaluating a query?
- What kinds of queries will indices help to speed up?
- How can multiple indices be used to speed up selections?
- When can merge join be used?
- When is it worthwhile computing a temporary index?

Can you answer the following questions?

- Can you prove that the transformations shown are correct?
- When can projection be commuted with natural join?
- How should one select the main relation to iterate over when computing a join?