7042 Datenbanken

Prof. O. Nierstrasz

Wintersemester 1997/98

Table of Contents

1. 7042 Datenbanken

Lecturer:	Prof. O. Nierstrasz Schützenmattstr. 14/103, Tel. 631.4618
Secr.:	Frau I. Huber, Tel. 631.4692
Assistants:	J.-G. Schneider S. Kneubühl
WWW:	http://iamwww.unibe.ch/~scg/Lectures/

Principle Text:

- A. Silberschatz, H.F. Korth and S. Sudarshan, *Database System Concepts*, 3d edition, McGraw Hill, 1997.

Supplementary texts:

- R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Benjamin/ Cummings, 1994.
- A. Kemper, A. Eickler, *Datenbanksysteme*, Oldenbourg Verlag, 1996.
- G. Vossen, *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*, Addison-Wesley, 1994.

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?

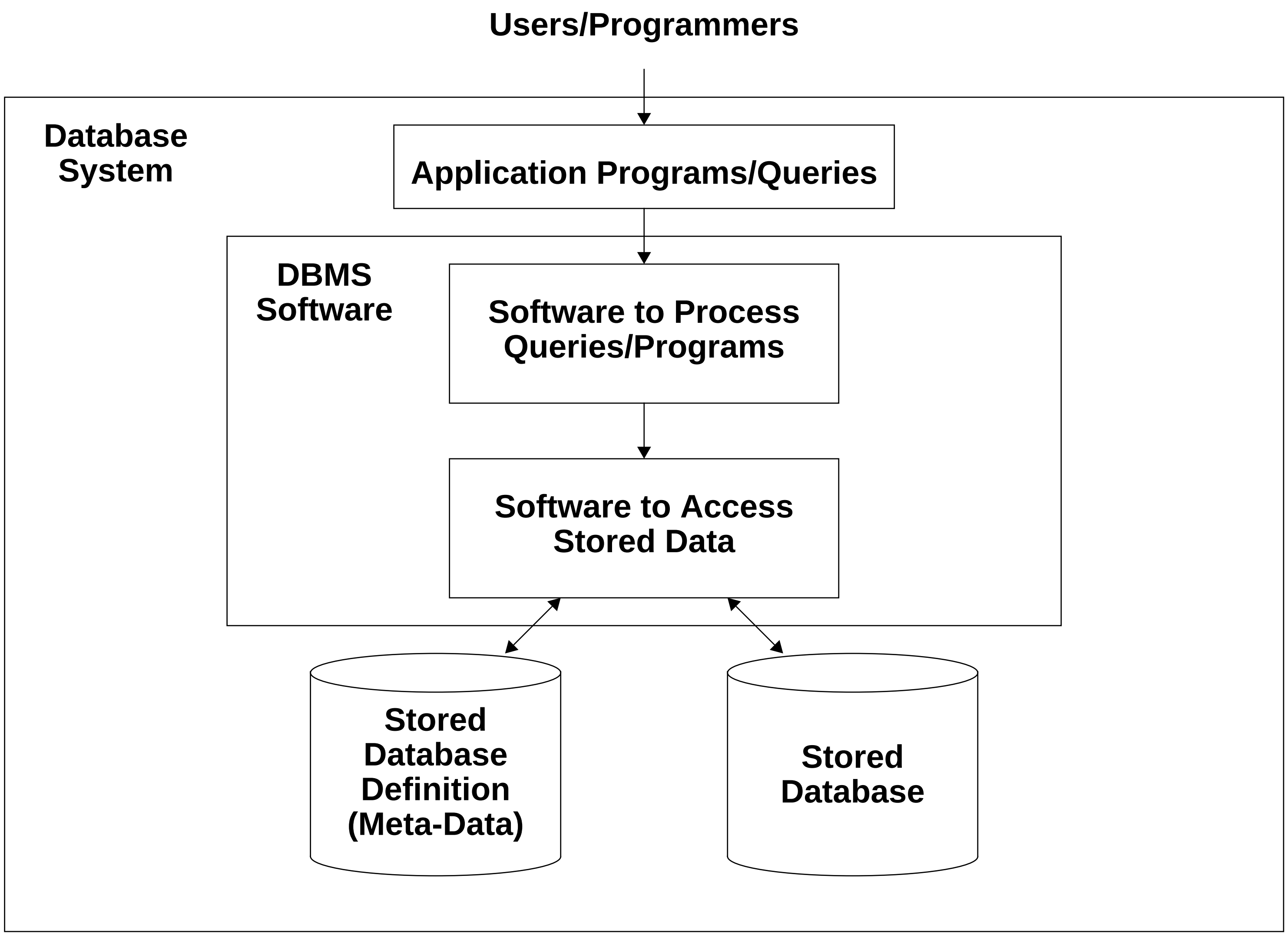
What is a Database System?

- ❑ 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?



Example

borrow

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

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

customer

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

branch

branch-name	*assets*	*branch-city*
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

client

customer-name	*banker-name*
Turner	Johnson
Hayes	Jones
Johnson	Johnson

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
- ...

Data Models

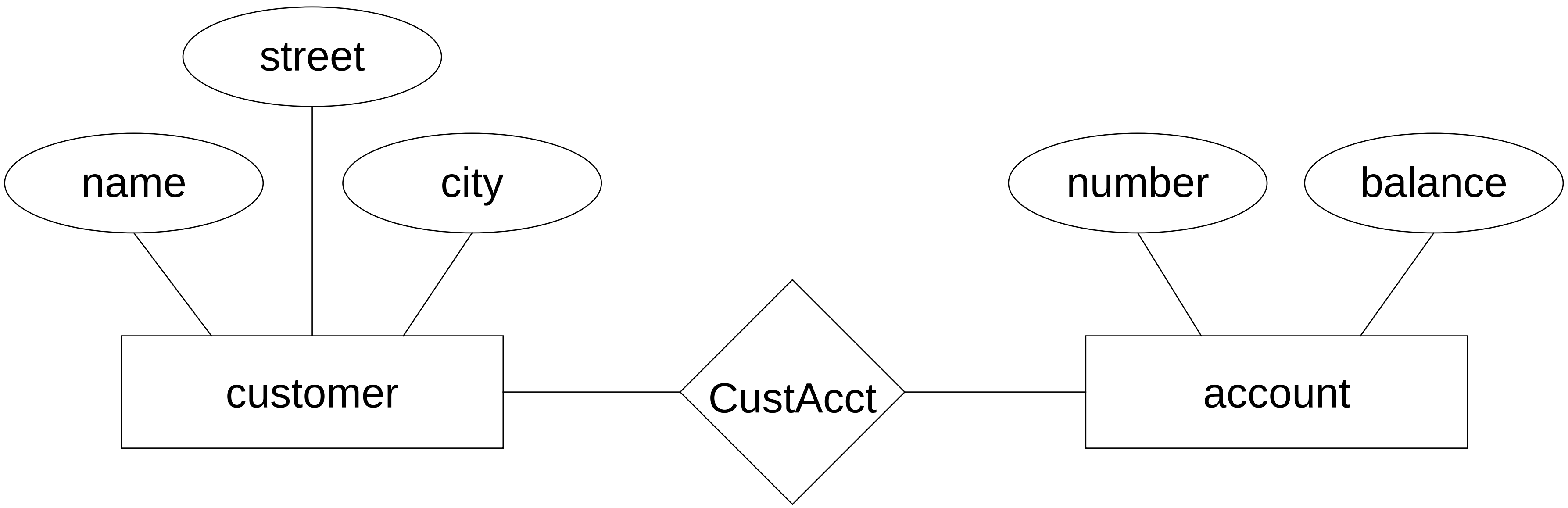
"A data model is a set of concepts that can be used to describe the structure of a database." (E&N)

- 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)



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

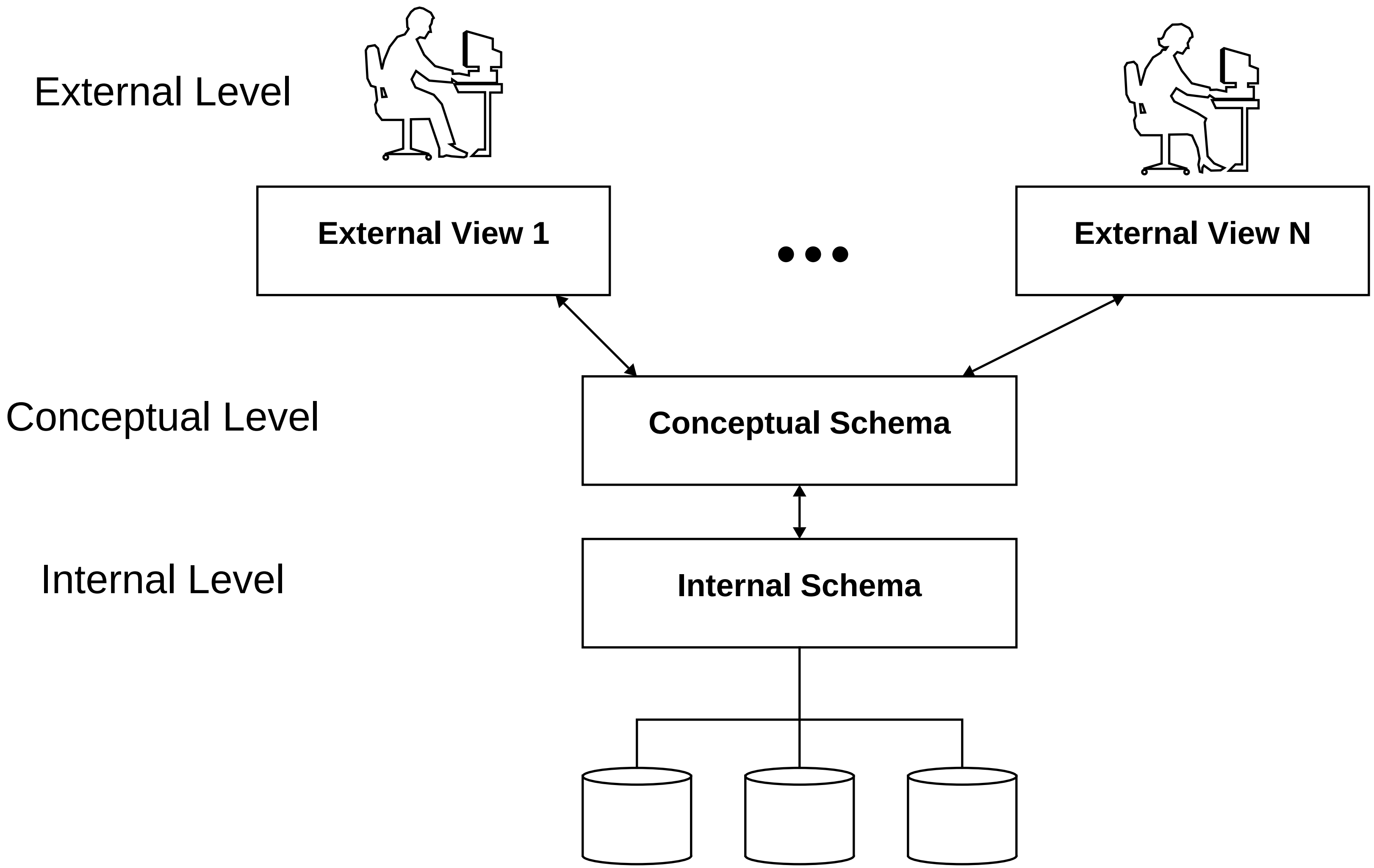
Database Schema

- ☞ 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



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 }

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	135-79-7	Nassau	Austin

259	1000
630	2000
401	700
700	1500
199	500
467	900
115	1200
183	1300
118	2000
225	2500
210	2200

Attributes

- special value *null*
- multi-valued attributes: $A = 2^V$
- atomic and composite attributes
- derived attributes

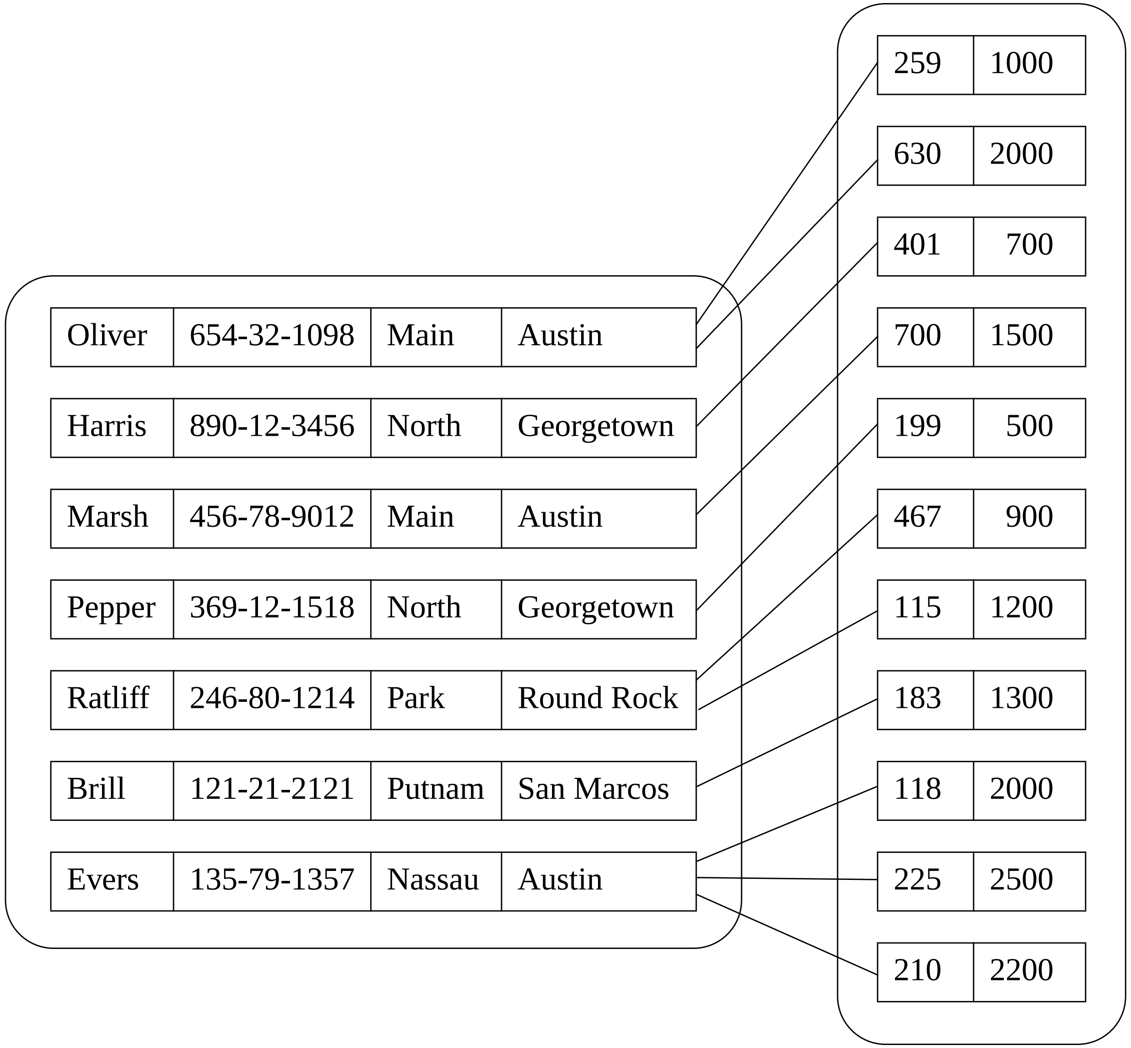
Relationships

A *relationship* is an association among several (n > 2) entities.

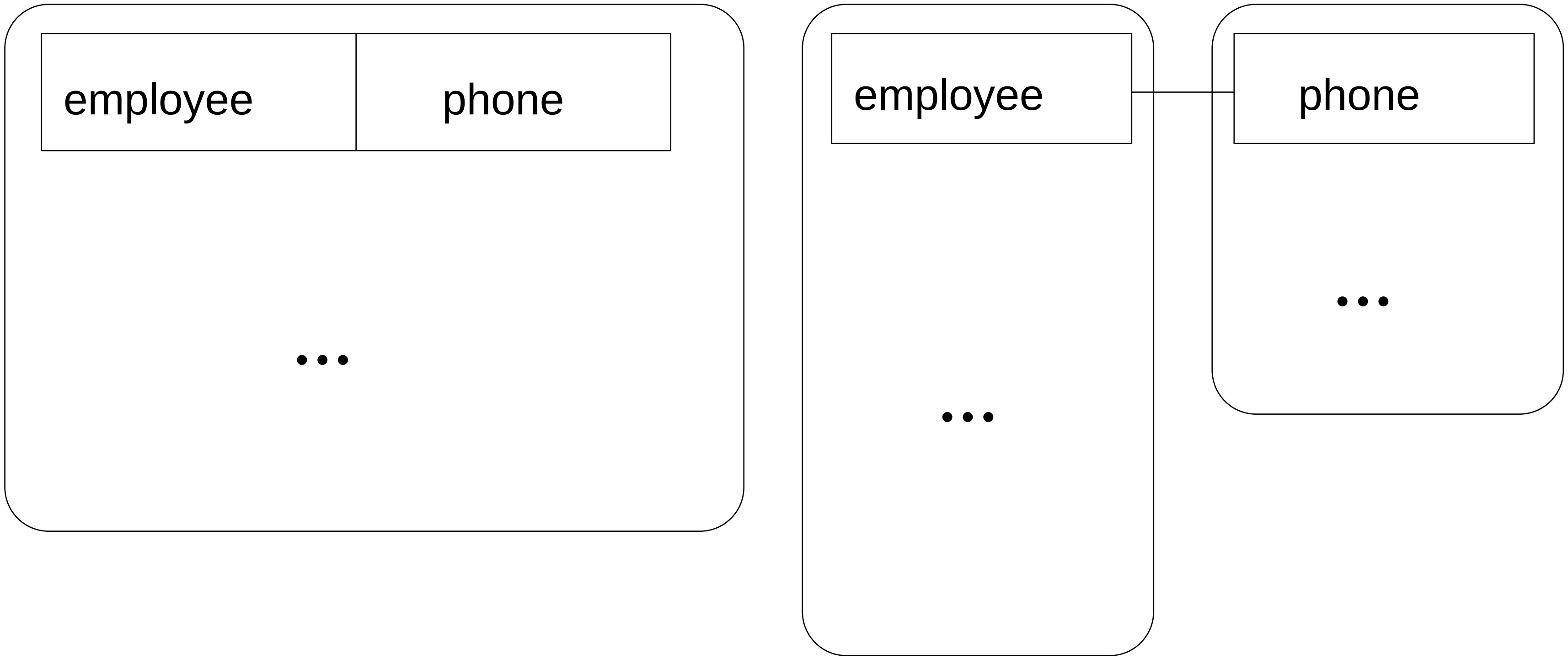
A *relationship set* is a set of relationships of the same type.

Formally, $R \subseteq E_1 \times E_2 \times ... \times E_N$

Relationships and relationship sets

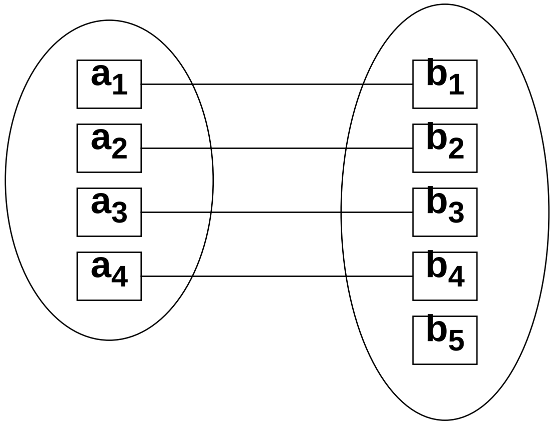


Attributes vs. Entities

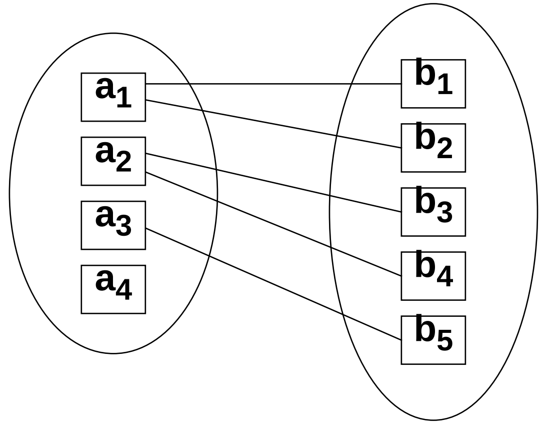


When should an attribute be modelled as a separate entity?

Mapping Constraints



One-to-one relationship



One-to-many relationship

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

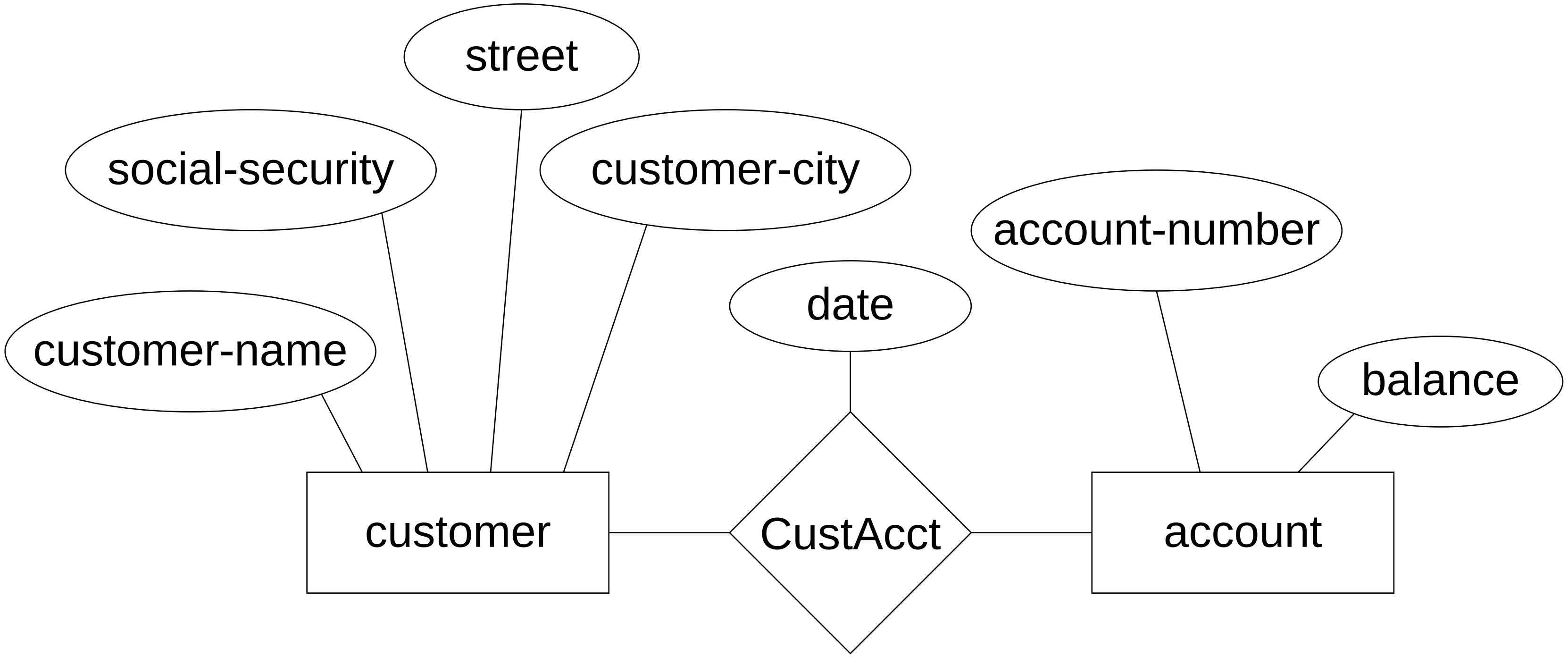
A **transaction** is *existence-dependent* on an **account**.

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



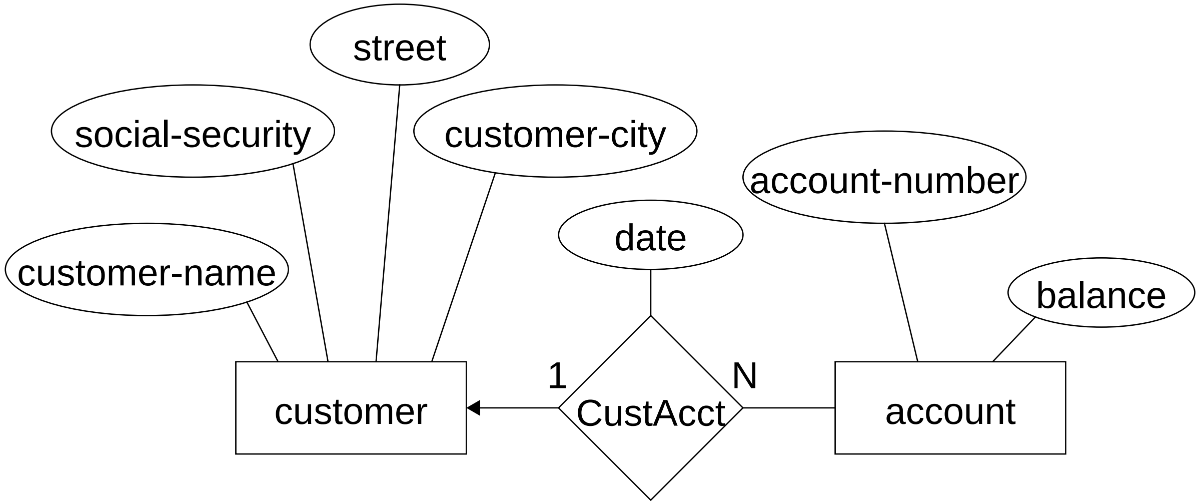
Rectangles represent *entity sets*

Ellipses represent *attributes*

Diamonds represent *relationship sets*

Lines connect attributes to their entity/relationship sets
and entities to their relationship sets

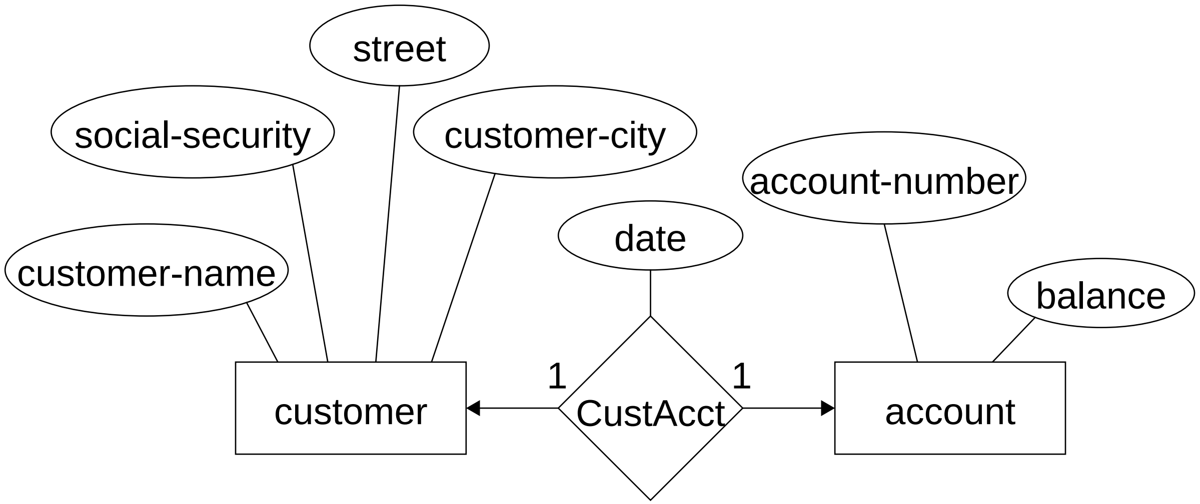
One-to-one, one-to-many



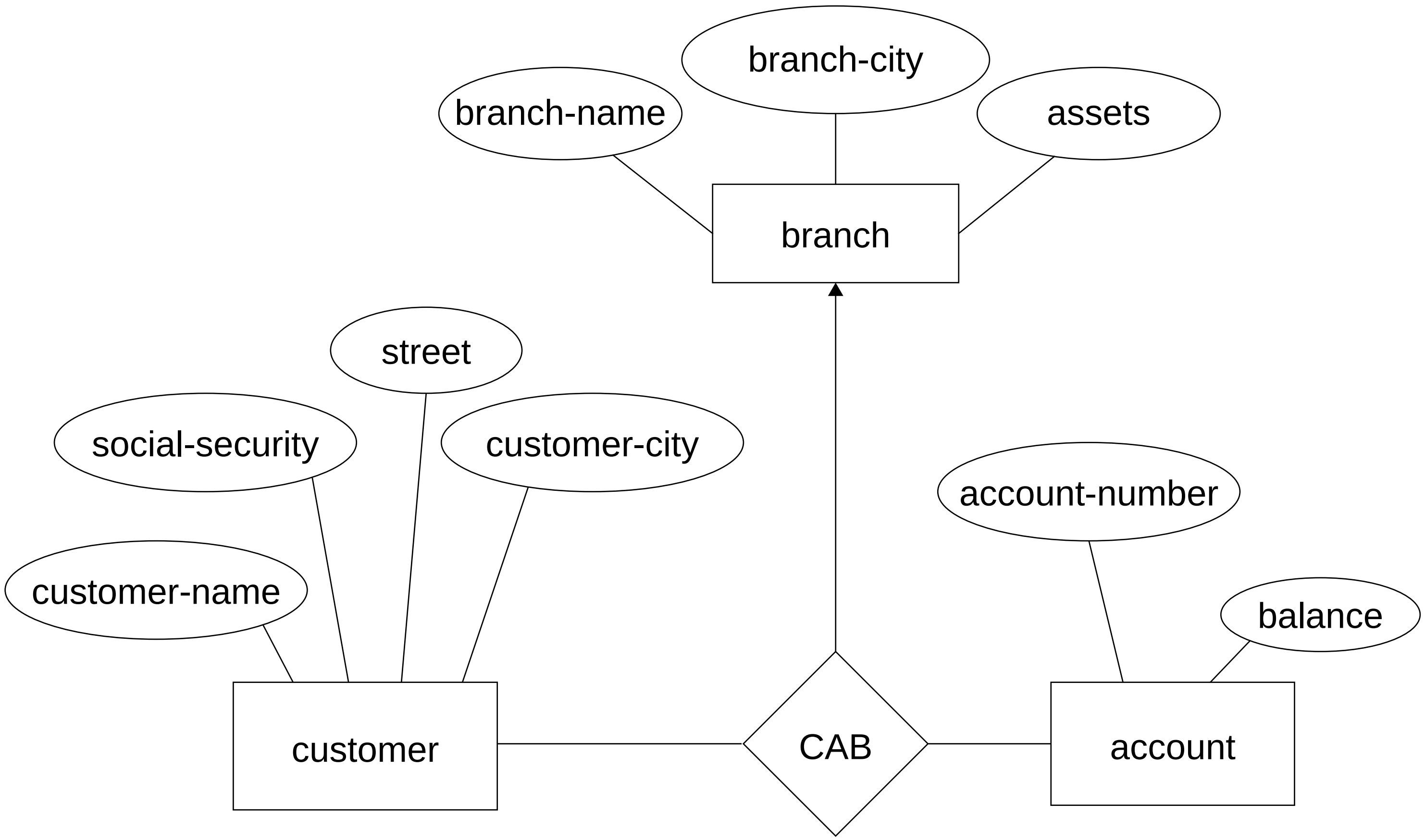
One-to-many: Every account belongs to at most one customer

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

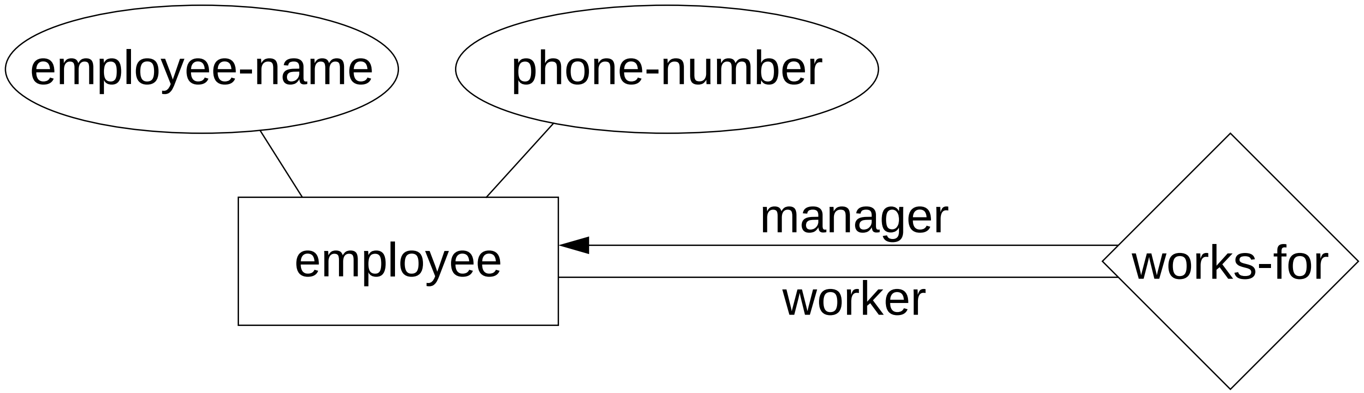
One-to-one: Every customer has at most one account, and vice versa



Ternary Relationships



Roles


employee-name
phone-number
employee
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?*

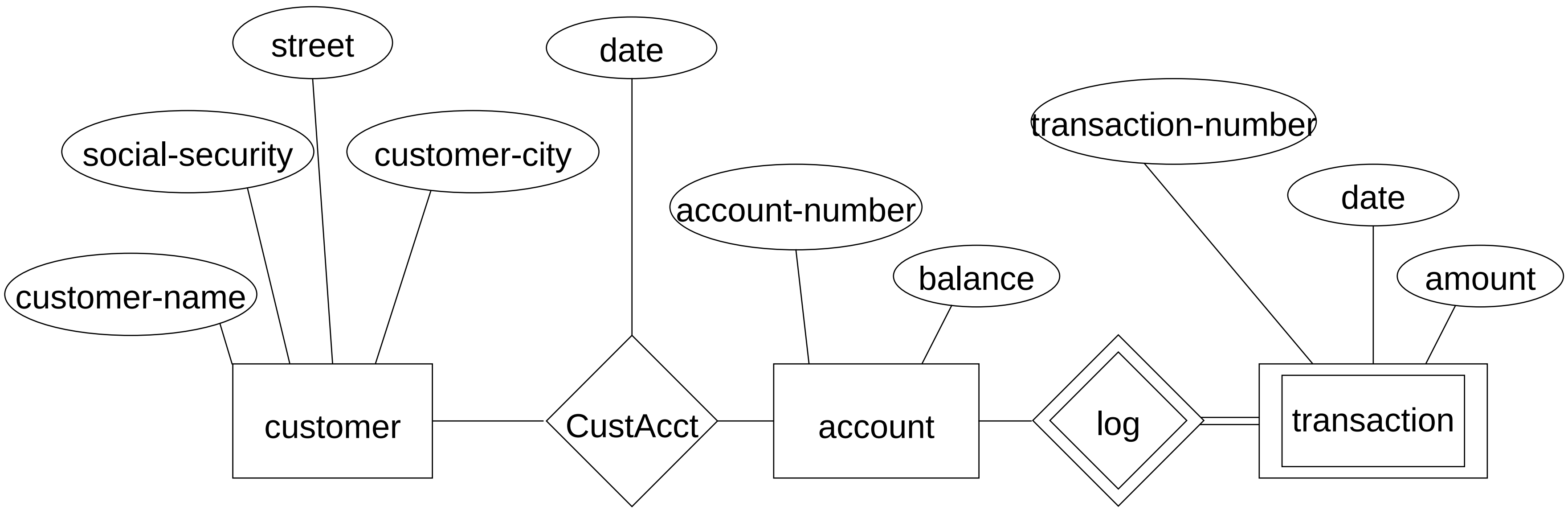
A *candidate key* is a *minimal* superkey.
A *primary key* is chosen by design

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

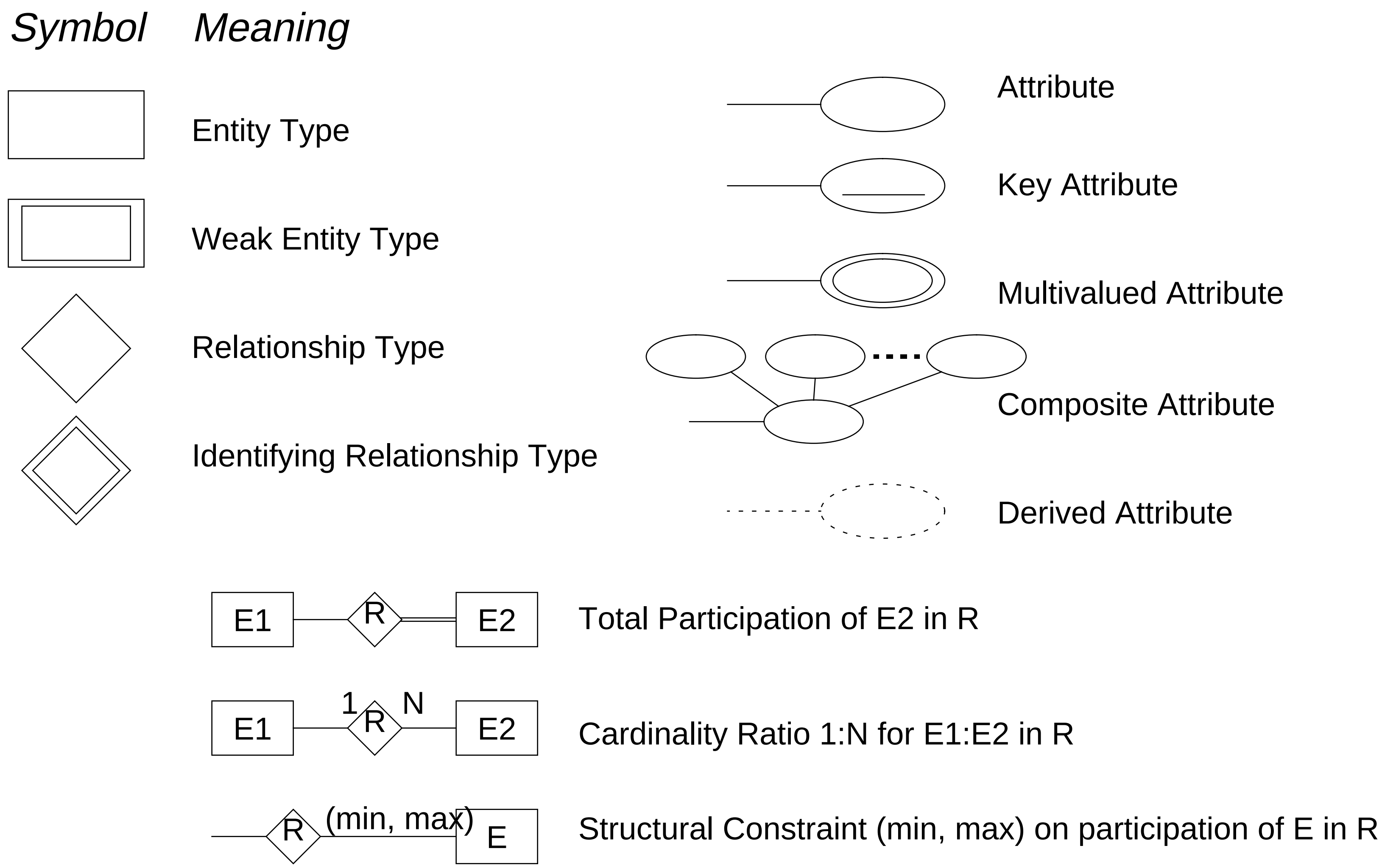
$$R \subseteq E_1 \times E_2 \times \ldots \times E_N$$

then

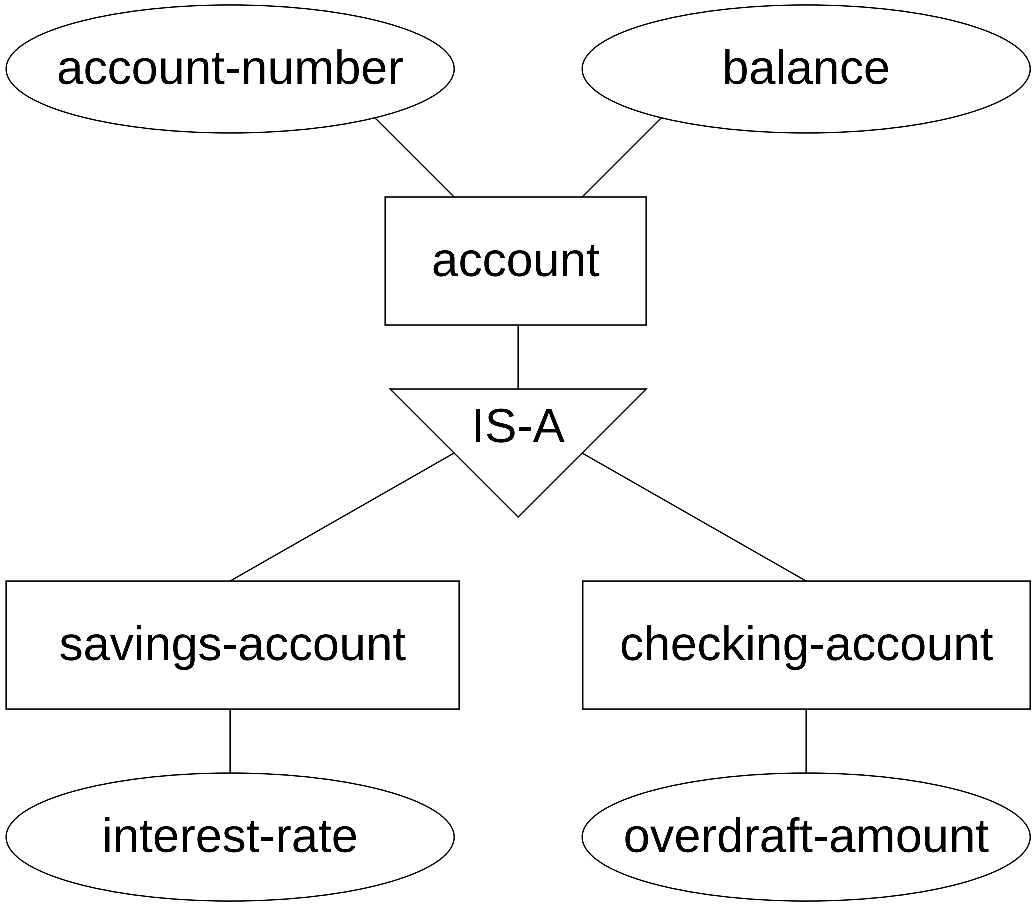
$$attr(R) = prim\_key(E_1) \cup \ldots \cup prim\_key(E_N) \cup desc\_attr(R)$$

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!

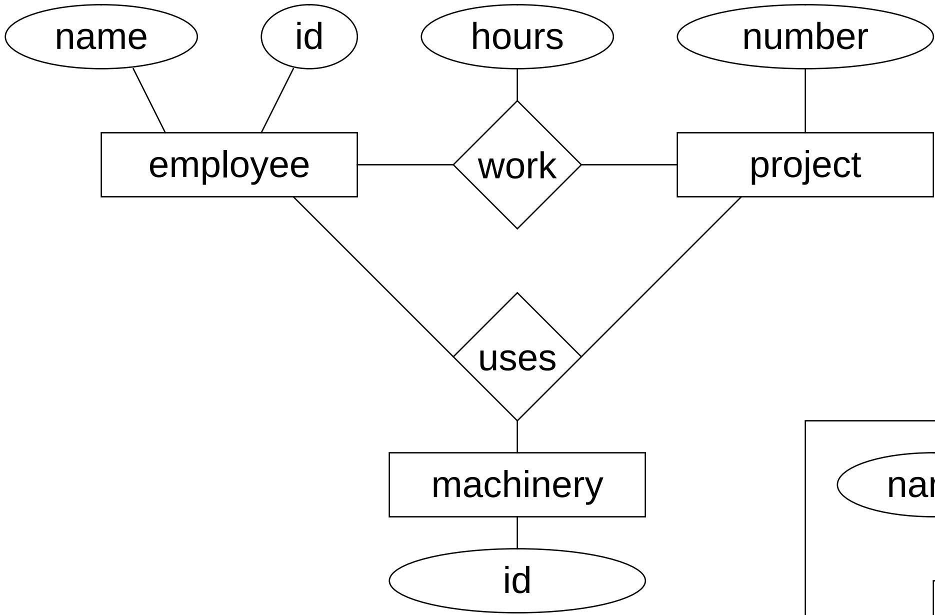
ER Diagrams


Symbol
Meaning
Entity Type
Weak Entity Type
Relationship Type
Identifying Relationship Type
Attribute
Key Attribute
Multivalued Attribute
Composite Attribute
Derived Attribute
E1
R
E2
Total Participation of E2 in R
1
N
Cardinality Ratio 1:N for E1:E2 in R
(min, max)
E
Structural Constraint (min, max) on participation of E in R

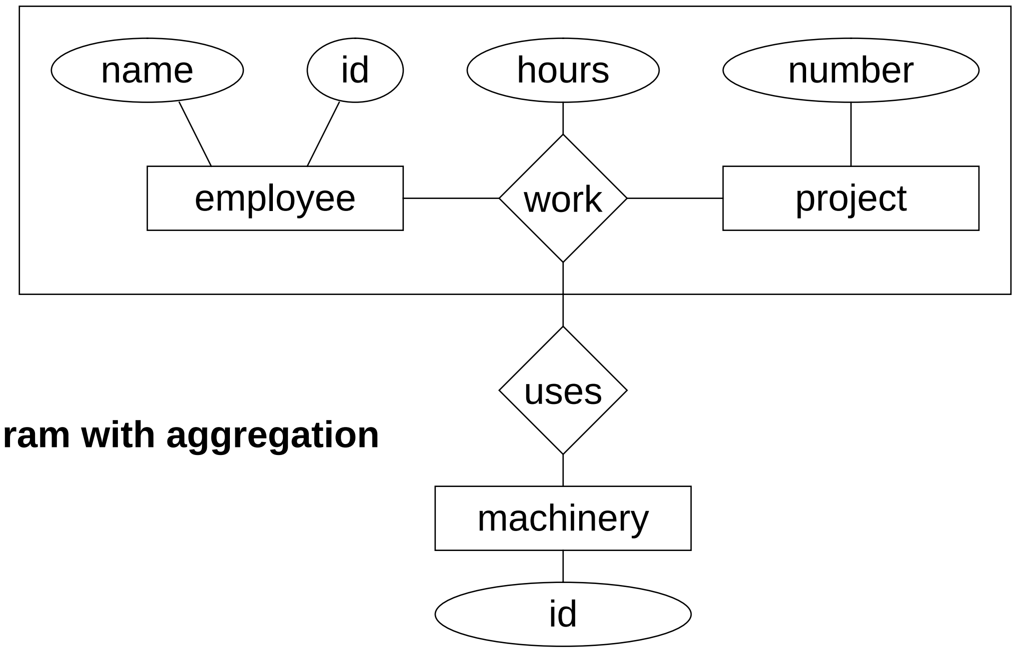
Generalisation



Aggregation



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)

account-number	*balance*
259	1000
630	2000
401	700
700	1500
199	500
467	900
115	1200
183	1300
118	2000
225	2500
210	2200

social-security	*account-number*	*date*
654-32-1098	259	17 June 1990
654-32-1098	630	17 May 1990
890-12-3456	401	23 May 1990
456-78-9012	700	28 May 1990
369-12-1518	199	13 June 1990
246-80-1214	467	7 June 1990
246-80-1214	115	7 June 1990
121-21-2121	183	13 June 1990
135-79-1357	118	17 June 1990
135-79-1357	225	19 June 1990
135-79-1357	210	27 June 1990

Reducing Weak Entity Sets

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

account-number	*transaction-number*	*date*	*amount*
259	5	11 May 1990	+50
630	11	17 May 1990	+70
401	22	23 May 1990	-300
700	69	28 May 1990	-500
199	103	3 June 1990	+900
259	6	7 June 1990	-44
115	53	7 June 1990	+120
199	104	13 June 1990	-200
259	7	17 June 1990	-79

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

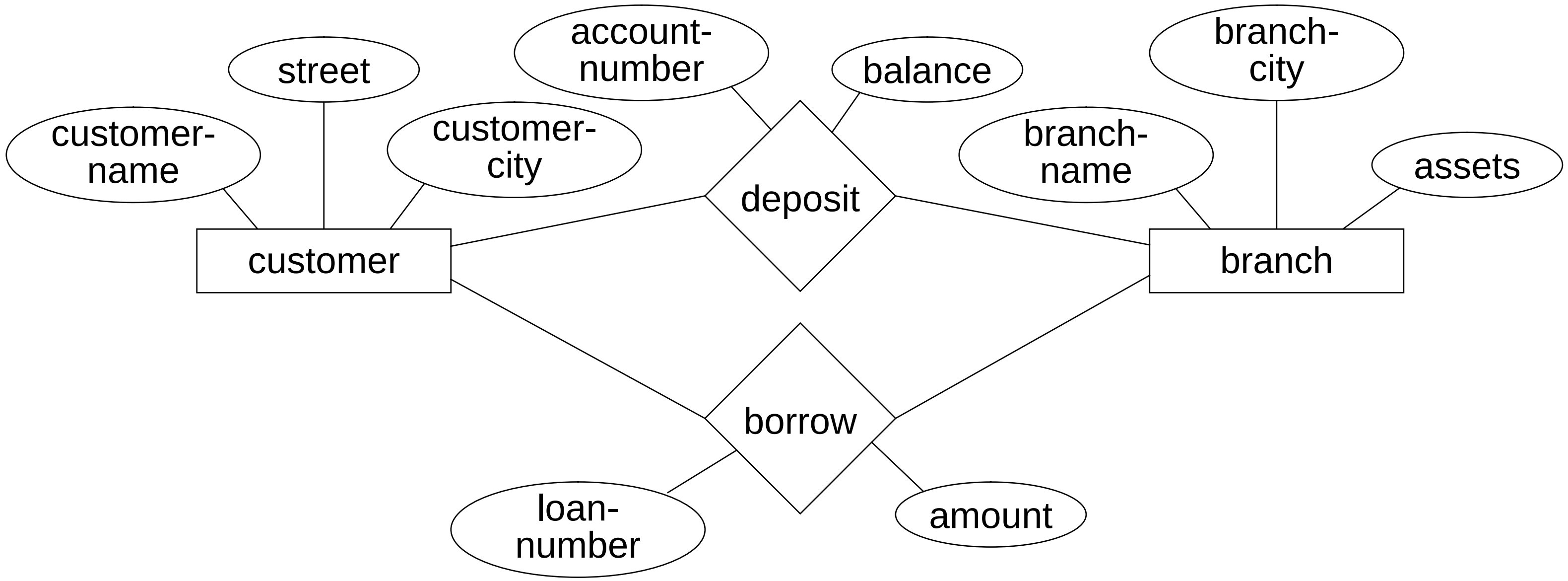
Overview

- ❑ 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*:

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

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 $t[a_i] \equiv t[i]$

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

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

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

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: $temp \leftarrow \langle \text{expression} \rangle$ (for intermediate expressions)
- ❑ *Derived:* intersection, natural join, division

Example: The Bank Database

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

customer

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

borrow

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

branch

branch-name	*assets*	*branch-city*
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

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?*

client

customer-name	*banker-name*
Turner	Johnson
Hayes	Jones
Johnson	Johnson

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:

- ✎ *Which branches are in the same city as the Perryridge branch?*
- ✎ *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

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

customer

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

borrow

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

branch

branch-name	*assets*	*branch-city*
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

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

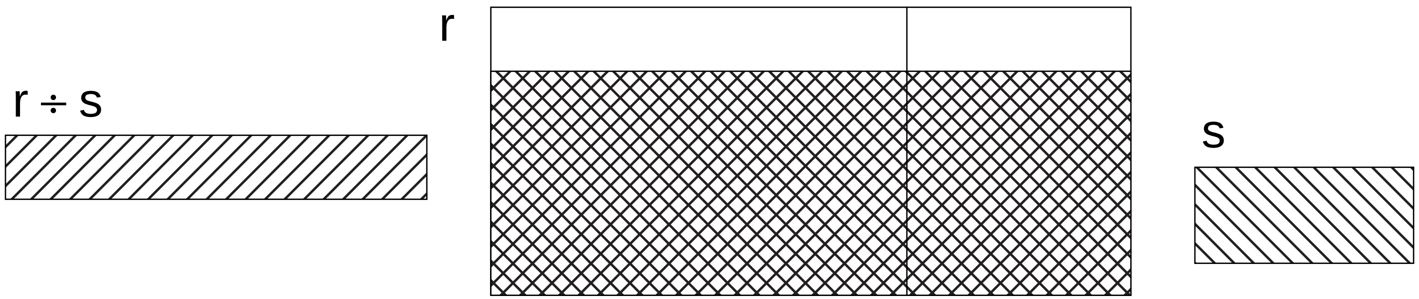
Express the following queries:

- ✎ *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

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

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

Atoms:

- $s \in r$
- $s[x] \Theta r[y]$
- $s[x] \Theta c$

Formulae P_i :

- atoms
- $\neg P$, (P), $P_1 \wedge P_2$, $P_1 \vee P_2$, $P_1 \Rightarrow P_2$
- $\exists s \in r(P(s))$, $\forall s \in r(P(s))$, *s* free in *P(s)*

Examples

Which loans are over $1200?

$$\{ t \mid t \in borrow \wedge t[amount] > 1200 \}$$

What are the names of customers with loans over $1200?

$$\{ t \mid \exists s \in borrow (t[cn] = s[cn] \wedge s[amount] > 1200) \}$$

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: $\{t \mid \neg(t \in borrow)\}$

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

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

Atoms:

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

Formulae P_i :

- atoms
- $\neg P$, (P), $P_1 \wedge P_2$, $P_1 \vee P_2$, $P_1 \Rightarrow P_2$
- $\exists x (P(x))$, $\forall x (P(x))$

Examples

Which loans are over $1200?

$$\{\langle b, l, c, a\rangle \mid \langle b, l, c, a\rangle \in borrow \wedge a > 1200\}$$

What are the names of customers with loans over $1200?

$$\{\langle c\rangle \mid \exists b, l, a(\langle b, l, c, a\rangle \in borrow \wedge a > 1200)\}$$

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:

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 $A_1, A_2, \ldots, A_N$
from $r_1, r_2, \ldots, r_m$
where P

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

Examples:

```
select   branch-name
from     deposit
```

```
select   distinct branch-name
from     deposit
```

Set Operations: Union

Find all customers with accounts or loans at Perryridge

```
(   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

$$\Pi_{customer-name,\, customer-city}\,(borrow \bowtie customer)$$

Express as:

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

Comparisons may be: $<$, $\leq$, $=$, $\neq$, $\geq$, $>$

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 “\”

```
select  customer-name
from    customer
where   street like “%Main%”
```

Set Membership

✎ *What does the following query represent?*

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

Tuples

✎ *What does the following query represent?*

```
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

```
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”
         )
```

Set containment

✎ *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

```
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:

```
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

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

Find the number of depositors for each branch

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

Group Predicates

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

May not compose aggregate functions!

```
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*

```
delete   deposit
where    customer-name = “Smith”
```

```
delete   borrow
```

Insertion:

```
insert into deposit
           values (“Perryridge”, 9732, “Smith”, 1200)
```

```
insert into deposit (  account-number,
                       customer-name,
                       branch-name,
                       balance )
           values (9732, “Smith”, “Perryridge”, 1200)
```

Restrictions

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

INVALID SQL:

☞ **delete** *deposit*
where *balance* < (**select avg** (*balance*)
from *deposit*)

INVALID SQL:

☞ **insert into** *deposit*
select * from *deposit*

Updates

```
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:

☞ **update** *deposit*
set *balance* = *balance* * 1.05
where *balance* > **select avg**(*balance*)
from *deposit*

Null Values

```
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 <u>single</u> base relation.

```
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:

create table r (A_1 D_1, ... , A_n D_n)

Removing tables:

drop table r

Adding new attributes:

alter table r **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:

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	"Perryridge"		P._*x*	

In the domain relational calculus:

$$\{\langle x\rangle \mid \exists b, l, a(\langle b, l, x, a\rangle \in deposit \wedge b = \text{"Perryridge"})\}$$

Variable unification

Which customers have accounts at both Perryridge and Redwood?

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	"Perryridge"		P._*x*	
	"Redwood"		_*x*	

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

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	"Perryridge"		P._*x*	
	"Redwood"		P._*y*	

Set Difference

✎ *What does this query express?*

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	"Perryridge"		P._*x*	

borrow	*branch-name*	*loan-number*	*customer-name*	*amount*
¬	"Perryridge"		_*x*	

✎ *What would it mean if the negation were removed?*

Result Relations

✎ *What does this query express?*

deposit	*branch-name*	*account-number*	*customer-name*	*balance*
	"Perryridge"	_z	_x	

customer	*customer-name*	*street*	*customer-city*
	_x		_y

result	*customer_name*	*customer-city*	*account-number*
P.	_x	_y	_z

Other features

- Condition boxes
- Ordering display of tuples
- Aggregate operations
- Deletion, Insertion and Update operators (D., I. and U.)

Quel

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 $(t_{i1}.A_{j1}, \ldots t_{in}.A_{jn})$
where P

Differences between Quel and SQL

Equivalent expressive power, but:

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

Queries

```
range of s is borrow
range of t is deposit
retrieve unique (s.customer-name)
where        t.branch-name = "Perryridge"
        and  s.branch-name = "Perryridge"
        and  t.customer-name = s.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

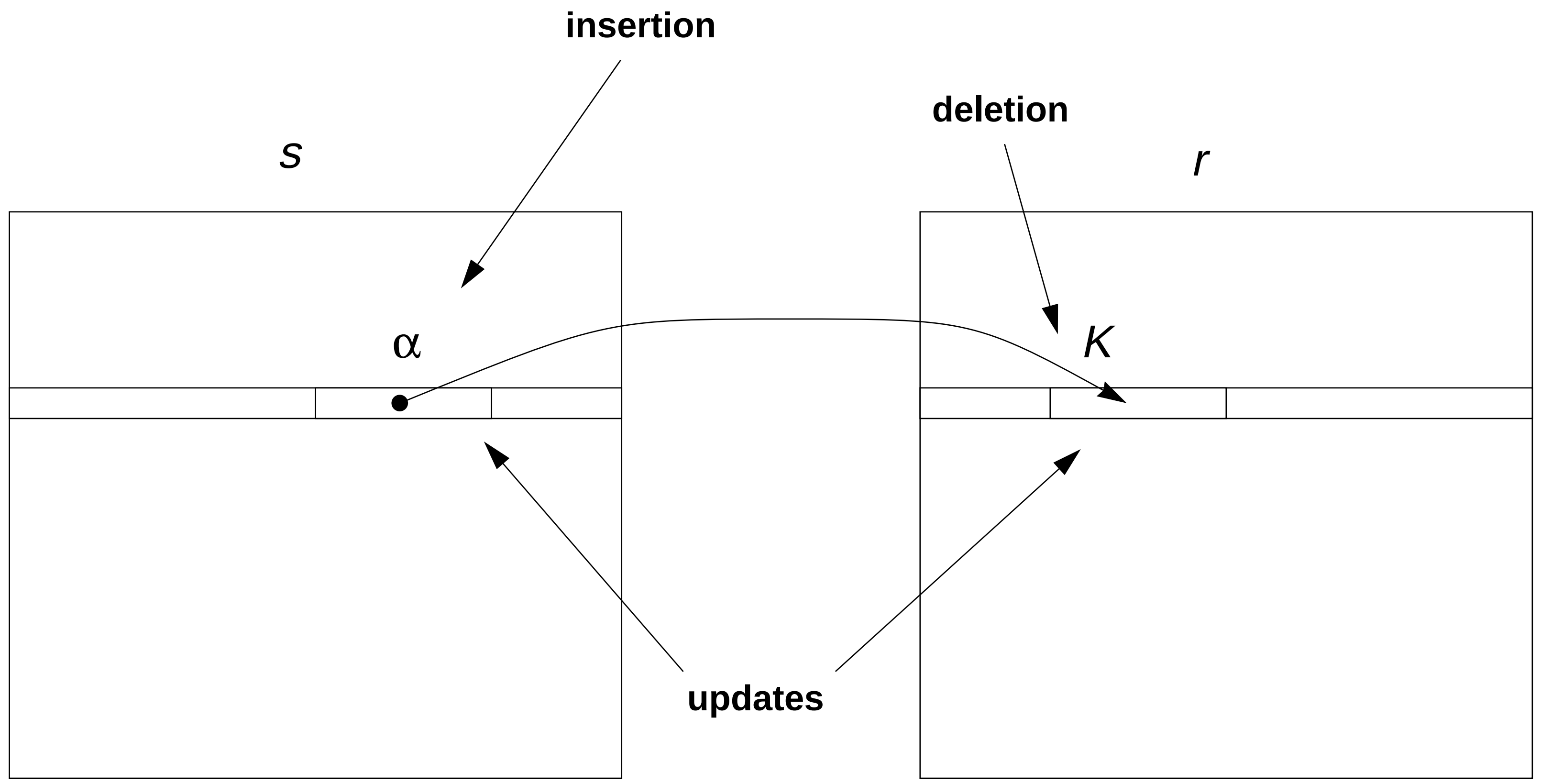
Then $\alpha \subseteq S$ is a *foreign key* if

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:



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

```
create table deposit
   (  branch-namechar(15) not null,
      account-numberchar(10),
      customer-namechar(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

A	*B*	*C*	*D*
a1	*b1*	*c1*	*d1*
a1	*b2*	*c1*	*d2*
a2	*b2*	*c2*	*d2*
a2	*b3*	*c2*	*d3*
a3	*b3*	*c2*	*d4*

Example FDs in the Bank Database

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

customer

customer-name	*street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

borrow

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

branch

branch-name	*assets*	*branch-city*
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

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:

$A \rightarrow B$	(1)
$A \rightarrow C$	(2)
$CG \rightarrow H$	(3)
$CG \rightarrow I$	(4)
$BC \rightarrow H$	(5)

✎ *Can we also conclude A→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* α is the set α^+ of all attributes functionally determined by α

Example:

AG	$\rightarrow$	ABG	(1)
	$\rightarrow$	ABCG	(2)
	$\rightarrow$	ABCGH	(3)
	$\rightarrow$	ABCGHI	(4)

Problem:

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

Solution:

Compute α^+ 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:

$AB \rightarrow C$	(1)
$B \rightarrow D$	(2)
$E \rightarrow F$	(3)
$CE \rightarrow A$	(4)

- ✎ *Does BE → DF?*
- ✎ *Does BE → FC?*
- ✎ *Is BE a superkey?*
- ✎ *Is BE a candidate key?*
- ✎ *What are all the candidate keys?*
- ✎ *Can you prove that you have found all of them?*

Canonical Covers

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 α
3. Each $\alpha \rightarrow \beta$ in F_c contains no extraneous attributes in β
4. For each $\alpha \rightarrow \beta$ in F_c, α 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 α or β (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.acount-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

borrow

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

Branch-Scheme

branch

branch-name	*assets*	*branch-city*
Downtown	9000000	Brooklyn
Redwood	2100000	Palo Alto
Perryridge	1700000	Horseneck
Mianus	400000	Horseneck
Round Hill	8000000	Horseneck
Pownal	300000	Bennington
North Town	3700000	Rye
Brighton	7100000	Brooklyn

Repetition of Information

Lending-Scheme

branch $\bowtie$ borrow

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

Lossy Joins

Consider decomposing Borrow-Scheme as follows:

Amt-Scheme *Loan-Scheme*

amt

branch-name	*loan-number*	*amount*
Downtown	17	1000
Redwood	23	2000
Perryridge	15	1500
Downtown	14	1500
Mianus	93	500
Round Hill	11	900
Pownal	29	1200
North Town	16	1300
Downtown	18	2000
Perryridge	25	2500
Brighton	10	2200

loan

amount	*customer-name*
1000	Jones
2000	Smith
1500	Hayes
1500	Jackson
500	Curry
900	Turner
1200	Williams
1300	Adams
2000	Johnson
2500	Glenn
2200	Brooks

Lossy Joins

amt $\bowtie$ loan

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

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

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \dots \bowtie \Pi_{R_n}(r)$$

Normalisation

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

with FDs:

branch-name $\rightarrow$ assets branch-city

loan-number $\rightarrow$ amount branch-name

Decompose into:

Branch-scheme = (<u>branch-name</u>, assets, branch-city)

Loan-info-scheme = (branch-name, <u>loan-number</u>, amount)

Customer-loan-scheme = (<u>loan-number, customer-name</u>)

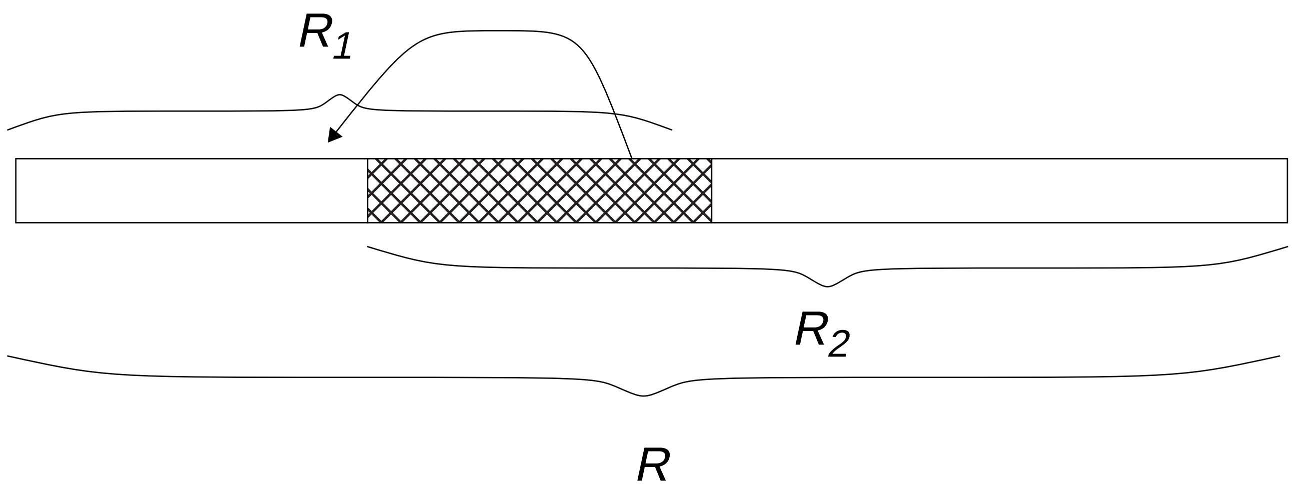
Lossless Join Decomposition

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

- $R_1 \cap R_2 \to R_1$
- $R_1 \cap R_2 \to R_2$

is in F^+.



Lossless Join Decomposition

Use

branch-name → *assets branch-city*

to decompose *Lending-scheme* into

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

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

Then, use

loan-number → *amount branch-name*

to decompose *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 $\bigcup_i F_i$ 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. α 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)

branch-name $\rightarrow$ *assets branch-city*

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

loan-number $\rightarrow$ *amount branch-name*

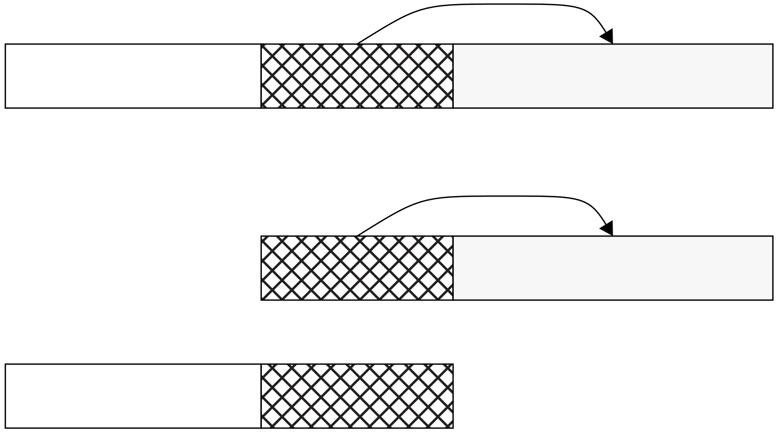
BCNF Decomposition Algorithm

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 $(R-\beta)$

Replace

by

and



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 $\rightarrow$ *branch-name*

customer-name branch-name $\rightarrow$ banker-name

Decompositions are not necessarily unique.

Consider: a $\rightarrow$ b c, b d $\rightarrow$ 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. α is a superkey for *R*, or
3. each attribute *A* in β 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

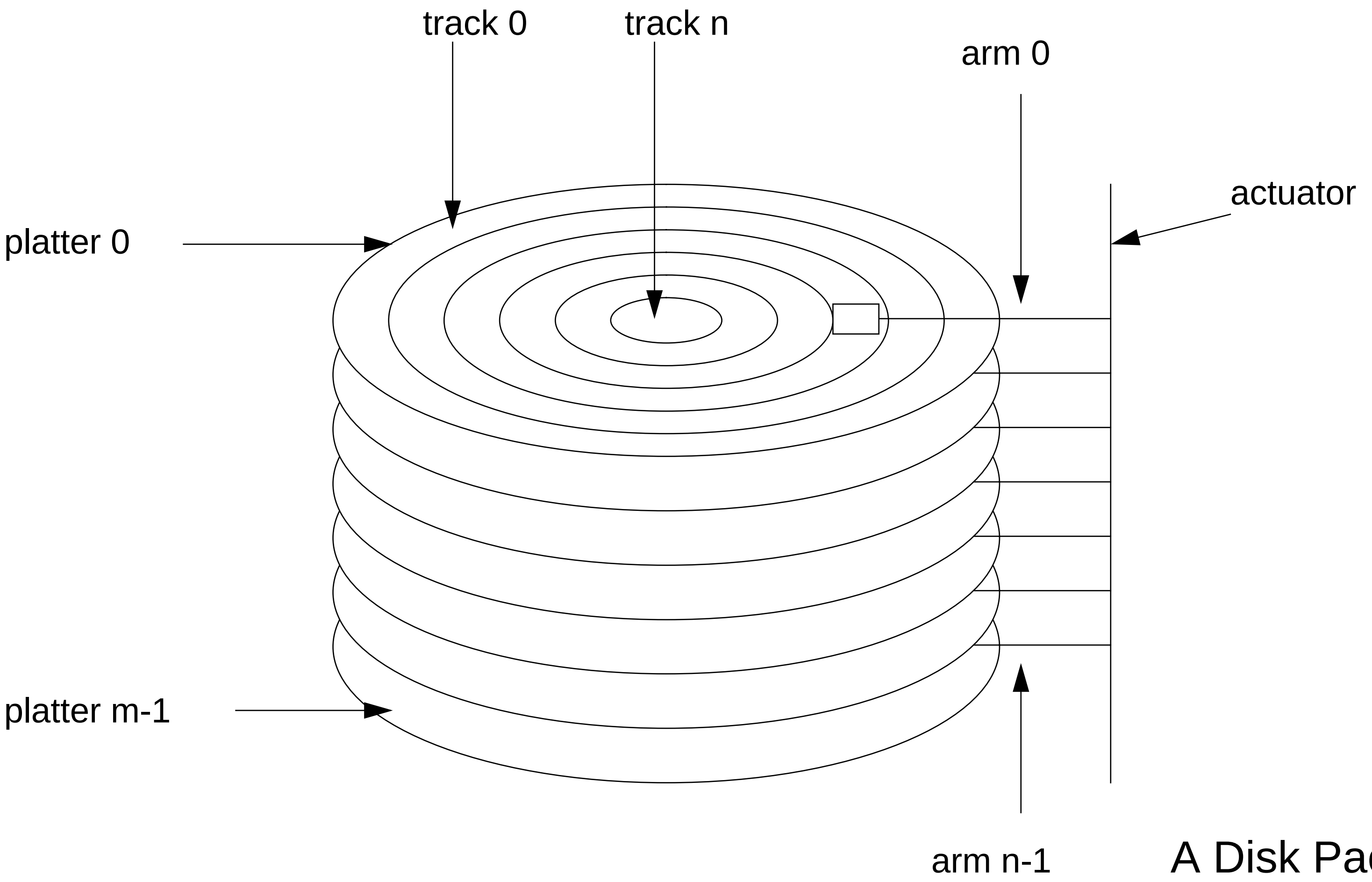
Disk storage:

☞ slower, large, persistent

Tape storage:

☞ slow, sequential, archival, cheap

Disk Storage



A Disk Pack

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

```
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.

header					
record 0	-	Downtown	101	Johnson	500
record 1	-	Mianus	215	Smith	700
record 2					
record 3	-	Round Hill	305	Turner	350
record 4	-	Perryridge	201	Williams	900
record 5					
record 6	-	Brighton	217	Green	750
record 7	-	Downtown	105	Green	850

Variable length records

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

```
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
```

Byte String Representation

0	Perryridge	102	Hayes	400	201	Williams	900	218	Lyle	700	⊥
1	Round Hill	305	Turner	350	⊥						
2	Mianus	215	Smith	700	⊥						
3	Downtown	101	Johnson	500	110	Peterson	600	⊥			
4	Redwood	222	Lindsay	700	⊥						
5	Brighton	217	Green	750	⊥						

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

0	Perryridge	102	Hayes	400	201	Williams	900	218	Lyle	700
1	Round Hill	305	Turner	350	⊥	⊥	⊥	⊥	⊥	⊥
2	Mianus	215	Smith	700	⊥	⊥	⊥	⊥	⊥	⊥
3	Downtown	101	Johnson	500	110	Peterson	600	⊥	⊥	⊥
4	Redwood	222	Lindsay	700	⊥	⊥	⊥	⊥	⊥	⊥
5	Brighton	217	Green	750	⊥	⊥	⊥	⊥	⊥	⊥

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

0		Perryridge	102	Hayes	400
1		Round Hill	305	Turner	350
2		Mianus	215	Smith	700
3		Downtown	101	Johnson	500
4		Redwood	222	Lindsay	700
5			201	Williams	900
6		Brighton	217	Green	750
7			110	Peterson	600
8			218	Lyle	700

Anchor/overflow block organization

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

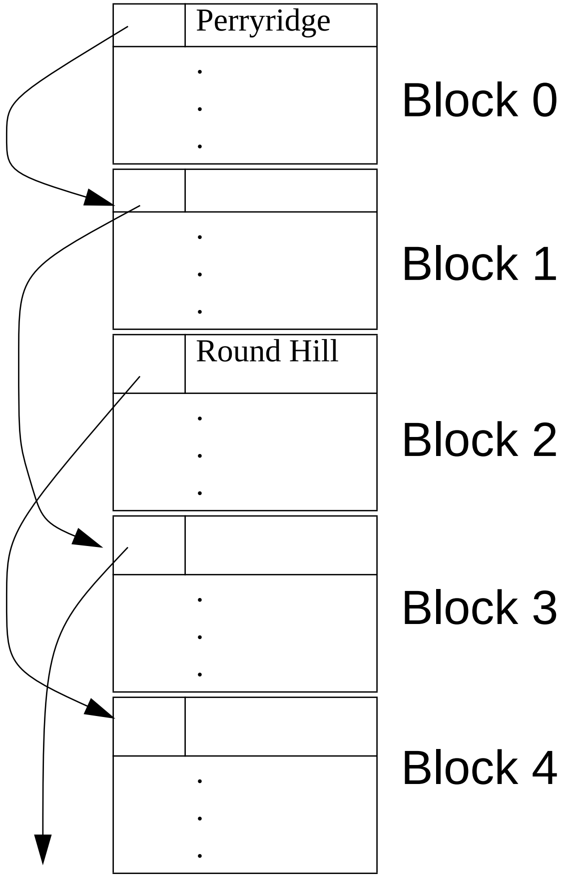
Anchor block

	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

Overflow block

	201	Williams	900
	110	Peterson	600
	218	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

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

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.

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

	Mianus	888	Adams	800

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 $\bowtie$ 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:

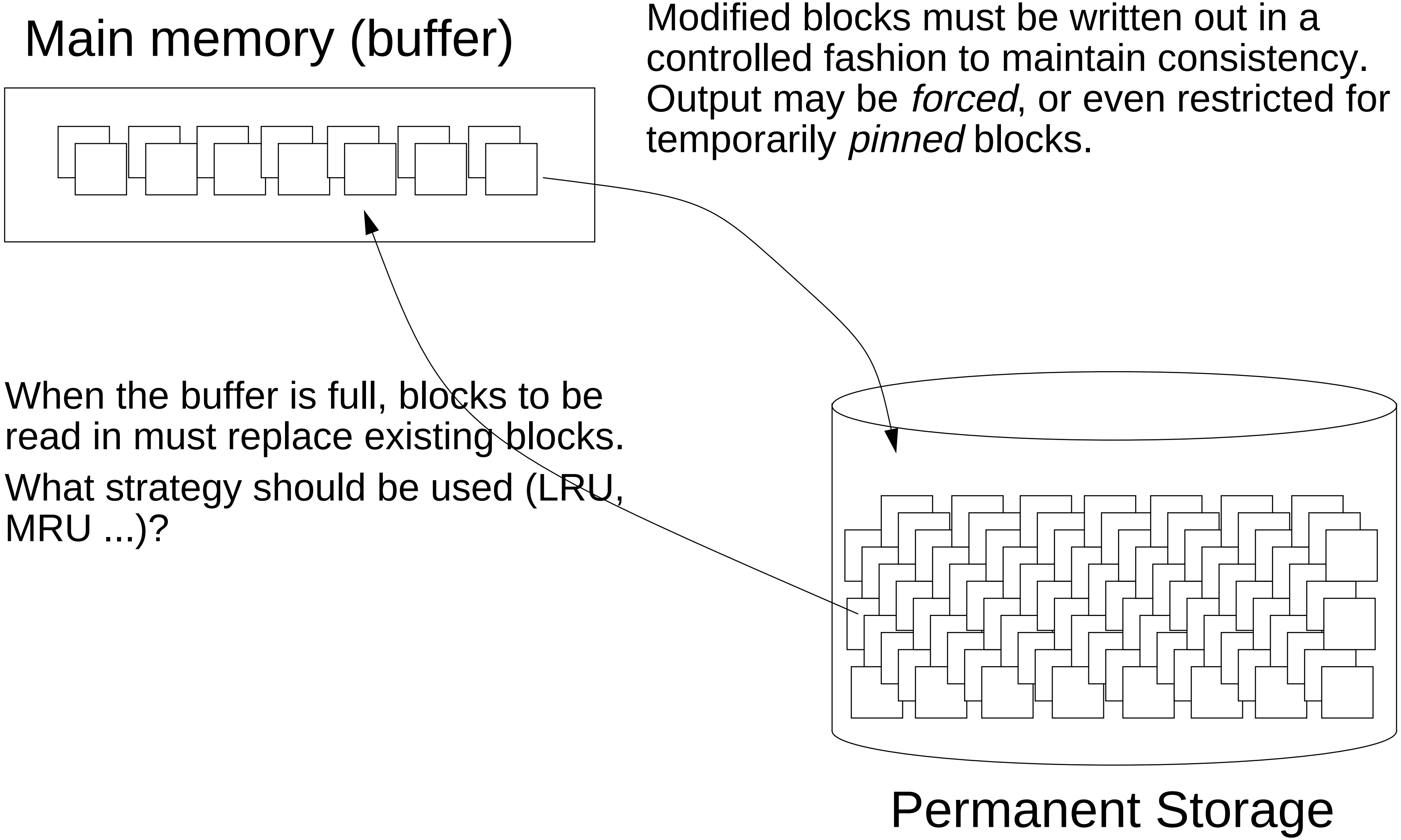
- ❑ User names and authorization; accounting information

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

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



When the buffer is full, blocks to be read in must replace existing blocks.

What strategy should be used (LRU, MRU ...)?

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	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

	Brighton	217	Green	750
	Downtown	101	Johnson	500
	Downtown	110	Peterson	600
	Mianus	215	Smith	700
	Perryridge	102	Hayes	400
	Perryridge	201	Williams	900
	Perryridge	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	
Mianus	
Redwood	

	Brighton	217	Green	750
	Downtown	101	Johnson	500
	Downtown	110	Peterson	600
	Mianus	215	Smith	700
	Perryridge	102	Hayes	400
	Perryridge	201	Williams	900
	Perryridge	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

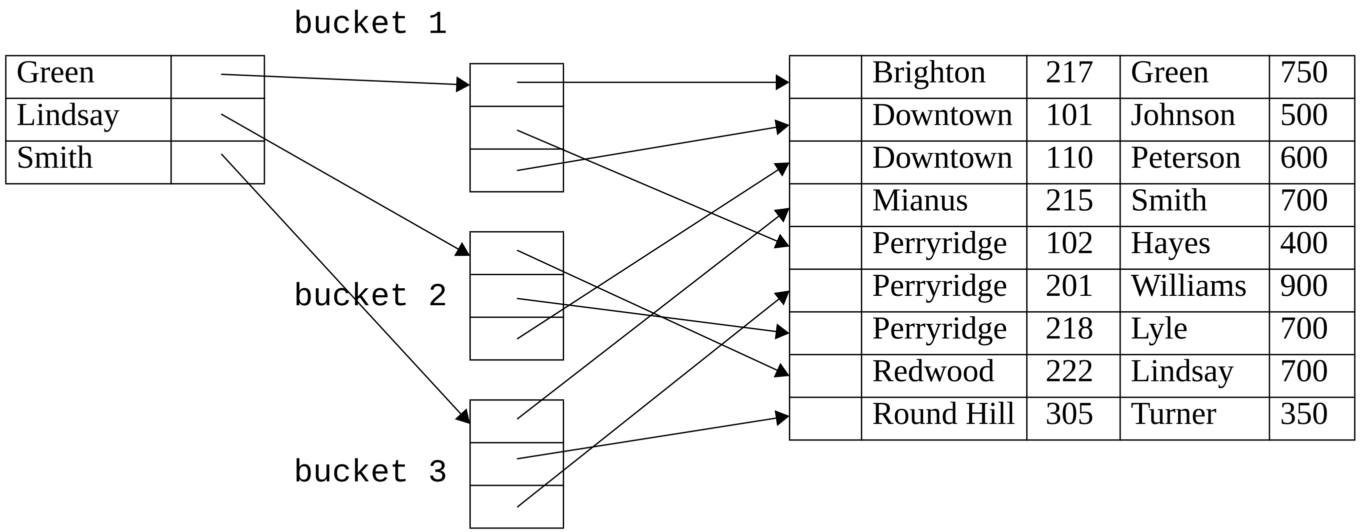
Indices

- 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

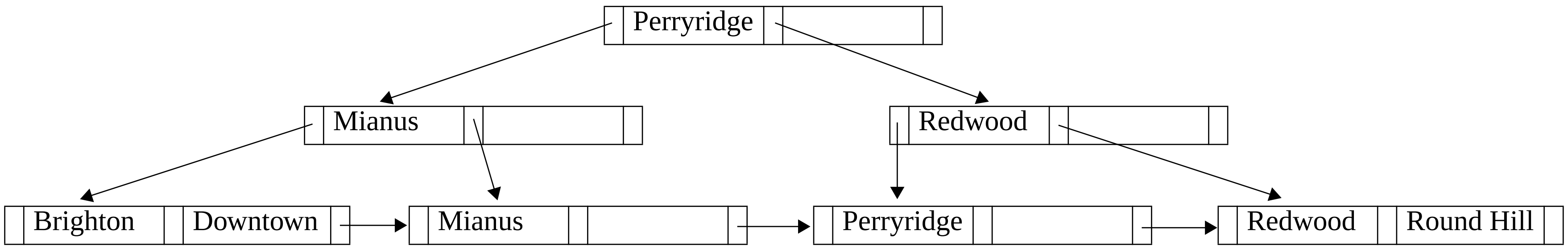
Index-sequential files perform poorly as database grows;
B^+ Trees perform better under frequent modifications

- 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)

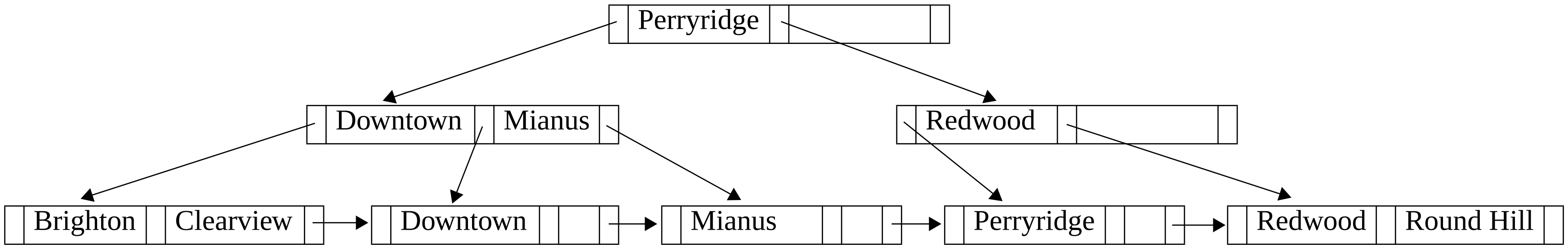
P_1	K_1	P_2	...	P_{m-1}	K_{m-1}	P_m	...	P_{n-1}	K_{n-1}	P_n

- 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

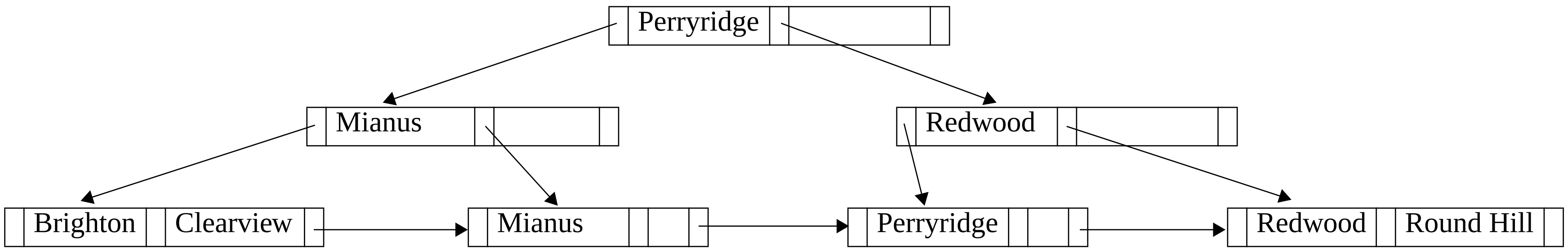


Insertion of "Clearview" causes leaf node to split

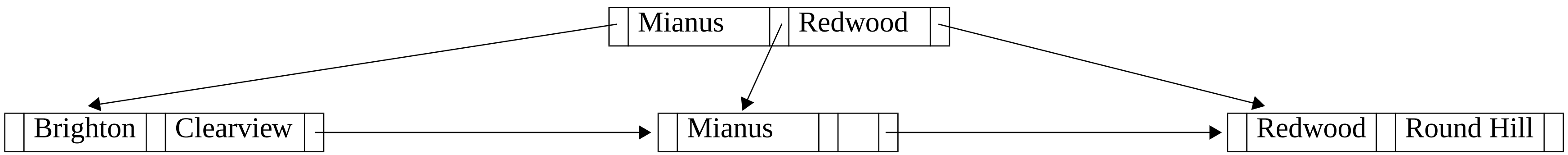


B+ Tree Deletions

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)

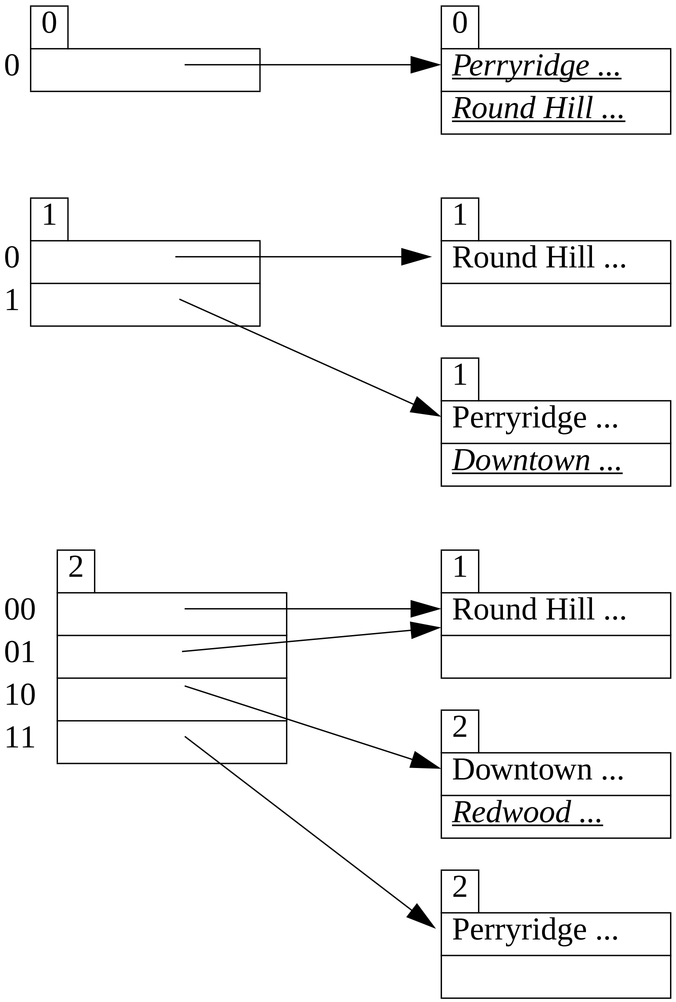
- The hash function and number of buckets must be fixed in advance; space is wasted if too many buckets are chosen, but performance will suffer if the buckets become too full

    - ☞ 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

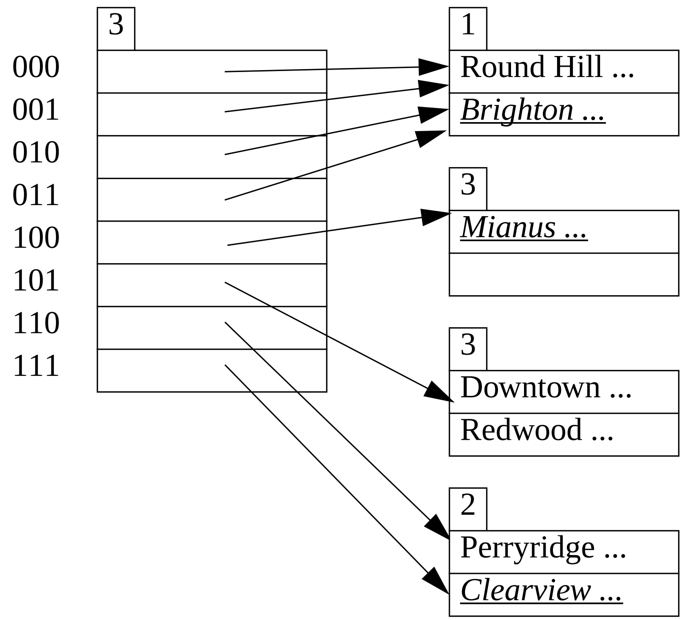


Hash function (abbreviated)

Brighton	0010
Clearview	1101
Downtown	1010
Mianus	1000
Perryridge	1111
Redwood	1011
Round Hill	0101

Sample deposit file

Brighton	217	Green	750
Downtown	101	Johnson	500
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Redwood	222	Lindsay	700
Round Hill	305	Turner	350
Clearview	117	Throggs	295



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 [c1,c2])
  - 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

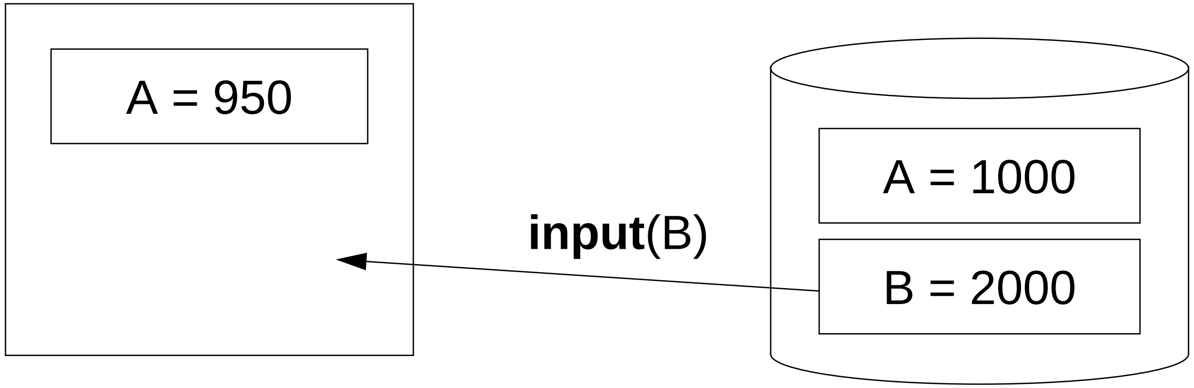
Transactions

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

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

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



Transaction States

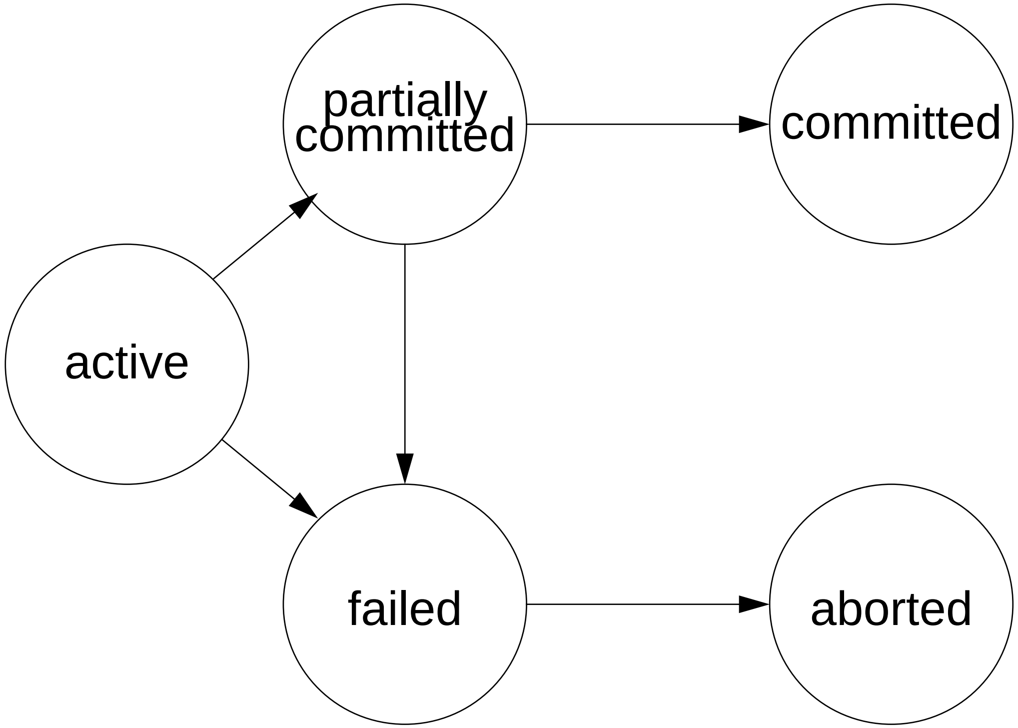
Active — the initial state

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:read(A)
   A := A - 50
   write(A)
   read(B)
   B := B + 50
   write(B)

T2:read(C)
   C := C - 100
   write(C)
```

Log	Database
	A = 1000
	B = 2000
	C = 700
<T1 **starts**>	
<T1, A, 950>	
<T1, B, 2050>	
<T1 **commits**>	
	A = 950
	B = 2050
<T2 **starts**>	
<T2, C, 600>	
<T2 **commits**>	
	C = 600

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*.

Log	Database
<T1 **starts**>	
<T1, A, 1000, 950>	
	A = 950
<T1, B, 2000, 2050>	
	B = 2050
<T1 **commits**>	
<T2 **starts**>	
<T2, C, 700, 600>	
	C = 600
<T2 **commits**>	

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

T1	T2
read(A)	
A := A - 50	
write(A)	
read(B)	
B := B + 50	
write(B)	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
	read(B)
	B:= B+ temp
	write(B)

T1	T2
read(A)	
A := A - 50	
write(A)	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
read(B)	
B := B + 50	
write(B)	
	read(B)
	B:= B+ temp
	write(B)

Non-serializable Schedules

T1	T2
read(A)	
A := A - 50	
	read(A)
	temp := A * 0.1
	A := A - temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B:= B+ temp
	write(B)

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 <u>conflict</u> if they refer to the same data item Q, and one of the two is a **write** operation.*

T_i	T_j
read(Q)	
	read(Q)

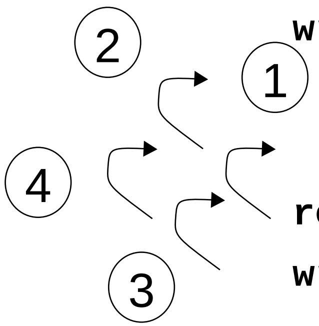
T_i	T_j
read(Q)	
	write(Q)

T_i	T_j
	write(Q)
read(Q)	

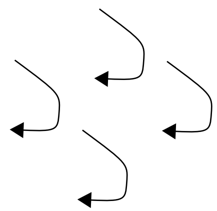
T_i	T_j
write(Q)	
	write(Q)

Serializing Schedules

A schedule is conflict-serializable if it can be transformed into a serial schedule by interchanging non-conflicting instructions



T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



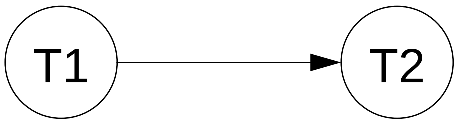
T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Testing for Conflict Serializability

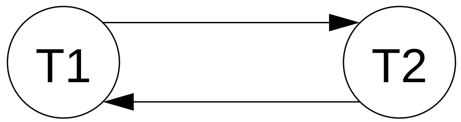
A schedule is ***conflict serializable*** if there are no cycles in its *precedence graph*.

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

- T_i executes **write**(Q) before T_j executes **read**(Q), *or*
- T_i executes **read**(Q) before T_j executes **write**(Q), *or*
- T_i executes **write**(Q) before T_j executes **write**(Q).



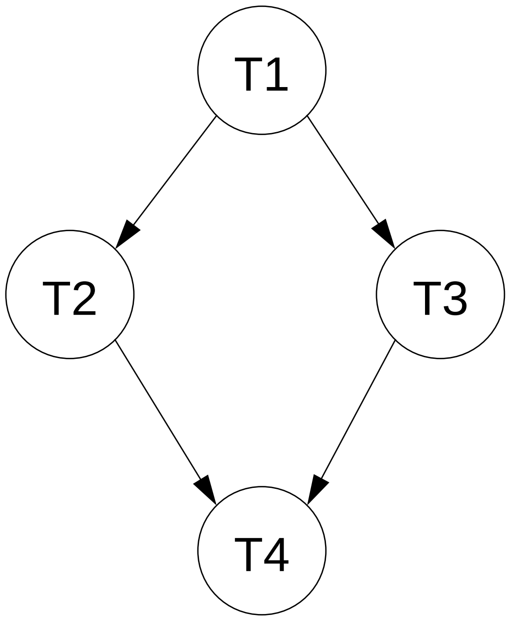
Serializable

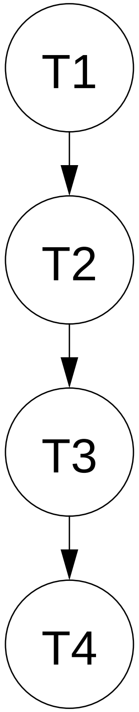


Not Serializable

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

T1	T2
lock-X(A)	
read(A)	
write(A)	
unlock(A)	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	lock-X(B)
	read(B)
	write(B)
	unlock(B)
lock-X(B)	
read(B)	
write(B)	
unlock(B)	

Unserializable schedule

T1	T2
lock-X(A)	
read(A)	
write(A)	
lock-X(B)	
unlock(A)	
	lock-X(A)
	read(A)
read(B)	
	write(A)
write(B)	
unlock(B)	
	lock-X(B)
	unlock(A)
	read(B)
	write(B)
	unlock(B)

Two-phase, serializable schedule

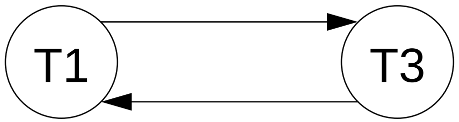
Deadlock

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

Two-phase, deadlocking schedule

Two-phase locking is not sufficient to avoid deadlock.

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



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:

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

vs.:

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

☞ Perform selections as early as possible

✎ *How can you formalize this rule?*

Conjunctions

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

$$\sigma_{\text{customer-city = "Port Chester" ^ balance > 1000}} \textbf{(customer} \bowtie \textbf{deposit)} \bowtie \textbf{branch}$$

$$\sigma_{\text{customer-city = "Port Chester"}} \textbf{(} \sigma_{\text{balance > 1000}} \textbf{(customer} \bowtie \textbf{deposit))}$$

$$\sigma_{\text{customer-city = "Port Chester"}} \textbf{(customer)} \bowtie \sigma_{\text{balance > 1000}} \textbf{(deposit))}$$

☞ Replace expressions of the form:

$\sigma_{P1 \wedge P2}(e)$ by $\sigma_{P1}(\sigma_{P2}(e))$

Projections

Consider:

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

vs.

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

☞ Perform projections early

✎ *How can you formalize this rule?*

Natural Joins

Consider:

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

vs.

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

vs.

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

☞ Rearrange multiple joins to minimize temporary results

Other transformations

- ☞ $\sigma_P(\sigma_Q(r)) = \sigma_Q(\sigma_P(r))$
- ☞ $\sigma_P(r_1 \cup r_2) = \sigma_P(r_1) \cup \sigma_P(r_2)$
- ☞ $\sigma_P(r_1 - r_2) = \sigma_P(r_1) - r_2 = \sigma_P(r_1) - \sigma_P(r_2)$
- ☞ $\pi_A(\pi_B(\ldots\pi_X(r))) = \pi_A(r)$
- ☞ $\pi_A(\sigma_{A = v}(r)) = \sigma_{A = v}(\pi_A(r))$
- ☞ $r \bowtie s = s \bowtie r$
- ☞ $(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$
- ☞ $r_2 \cup r_1 = r_1 \cup r_2$

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 $\dfrac{n_r}{V(A, r)}$ 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 $\frac{n_{r_2}}{V(A, r_2)}$ tuples in r_2. By symmetry, the join contains at most

$$min(\frac{n_{r_1} \cdot n_{r_2}}{V(A, r_1)}, \frac{n_{r_1} \cdot n_{r_2}}{V(A, r_2)}) \text{ tuples.}$$

Indices

Consider:

```
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(branch-name, deposit) = 50
- V(customer-name, deposit) = 200
- V(balance, deposit) = 5000
- $n_{deposit} = 10000$
- 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 $50 \times 200 = 10000$ 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)

deposit

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

join all tuples with same customer-name
advance pointer when join attribute falls behind

customer

customer-name	*street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

advance pointer after each join

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?*