What is a Database Management System?

- A *Database Management System (DBMS)* is a software package designed to store and manage databases:
  1. Manages very large amounts of data.
  2. Supports efficient access to very large amounts of data.
  3. Supports concurrent access to very large amounts of data.
     - Example: bank and its ATM machines.
  4. Supports secure, atomic access to very large amounts of data.
     - Contrast two people editing the same UNIX file – last to write "wins" – with the problem if two people deduct money from the same account via ATM machines at the same time – new balance is wrong whichever writes last.

Example: Online Bookseller

- Data = information on books (including categories, bestsellers, etc.), customers, pending orders, order histories, trends and preferences, etc.
  - Massive: many gigabytes at a minimum for medium-size bookseller, more if keep all order histories over all time, even more if keep images of book covers and sample pages

    => Far too big for memory
  - Persistent: data outlives programs that operate on it
  - Multi-user: many people/programs accessing same database, or even same data, simultaneously

    => Need careful controls

Files vs. DBMS

- Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)
- Special code for different queries
- Must protect data from inconsistency due to multiple concurrent users
- Crash recovery
- Security and access control

What is a Relational Database?

- Based on the relational model (tables):

acct #	name	balance
12345	Sally	1000.21
34567	Sue	285.48
…	…	…

- Today used in *most* DBMS's.

The DBMS Marketplace

- Relational DBMS companies – Oracle, Sybase – are among the largest software companies in the world.
- IBM offers its relational DB2 system. With IMS, a nonrelational system, IBM is by some accounts the largest DBMS vendor in the world.
- Microsoft offers SQL-Server, plus Microsoft Access for the cheap DBMS on the desktop, answered by “lite” systems from other competitors.
- Relational companies also challenged by “object-oriented DB” companies.
- But countered with “object-relational” systems, which retain the relational core while allowing type extension as in OO systems.

Three Aspects to Studying DBMS's

1. Modeling and design of databases.
   - Allows exploration of issues before committing to an implementation.
2. Programming: queries and DB operations like update.
   - SQL = “intergalactic dataspeak.”
3. DBMS implementation.

Query Languages

Employee	
Name	Dept

Department	
Dept	Manager

SQL

```
SELECT Manager
FROM Employee, Department
WHERE Employee.name = "Clark Kent”
        AND Employee.Dept = Department.Dept
```

Query Language

Data definition language (DDL) ~ like type defs in C or Pascal

Data Manipulation Language (DML)

Query (SELECT)

UPDATE < relation name >

SET <attribute> = < new-value>

WHERE <condition>

Host Languages

C, C++, Java, Lisp, COBOL

- Host language is completely general (Turing complete)
- Query language—less general "non procedural" and optimizable

Relational Model

Relational model is good for:

- Large amounts of data —> simple operations
- Navigate among small number of relations

Difficult Applications for relational model:

- VLSI Design (CAD in general)
- CASE
- Graphical Data

Other Models

Where number of "relations" is large, relationships are complex

- Object Data Model
- Logic Data Model

<u>OBJECT DATA MODEL</u>

1. Complex Objects – Nested Structure (pointers or references)
2. Encapsulation, set of Methods/Access functions
3. Object Identity
4. Inheritance – Defining new classes like old classes

Object model: usually find objects via explicit navigation
Also query language in some systems

Other Models

<u>LOGIC (Horn Clause) DATA MODEL</u>

- Prolog, Datalog:

  if A1 and A2 then B

  B:- A1 and A2

- Functions s(5) = 6 (successor)
- Predicates with Arguments: sum(X,Y,Z) $\leftarrow$ X + Y = Z

  sum(X,0,X) means X + 0 = X (always true for all X)

  sum(X,s(Y),s(Z)):-sum(X,Y,Z) means X+(Y+1) = (Z+1) if X + Y = Z

- More powerful than relational

  Can Compute Transitive Closure

  edge(X,Y).

  path(X,Y) :- edge(X,Y).

  path(X,Z) :- path(X,Y) & edge(Y,Z).

Data Models

60’s

70's

80's

90’s

now

Hierarchical

Network

Relational

Choice for most new applications

Object Bases

Knowledge Bases

Why Use a DBMS?

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.

Data Independence *

- Applications insulated from how data is structured and stored.
- *Logical data independence*: Protection from changes in *logical* structure of data.
- *Physical data independence*: Protection from changes in *physical* structure of data.

✉ *One of the most important benefits of using a DBMS!*

Levels of Abstraction

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.

✉ *Schemas are defined using DDL; data is modified/queried using DML.*

Concurrency Control

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- Interleaving actions of different user programs can lead to inconsistency: e.g., cheque is cleared while account balance is being computed.
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

Transaction: An Execution of a DB Program

- Key concept is *transaction*, which is an *atomic* sequence of database actions (reads/writes).
- Each transaction, executed completely, must leave the DB in a *consistent state* if DB is consistent when the transaction begins.
  - Users can specify some simple *integrity constraints* on the data, and the DBMS will enforce these constraints.
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
  - Thus, ensuring that a transaction (run alone) preserves consistency is ultimately the user's responsibility!

Scheduling Concurrent Transactions

- DBMS ensures that execution of {T1, ... , Tn} is equivalent to some *serial* execution T1' ... Tn'.
  - Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock. All locks are released at the end of the transaction. (Strict 2PL locking protocol.)
  - Idea: If an action of Ti (say, writing X) affects Tj (which perhaps reads X), one of them, say Ti, will obtain the lock on X first and Tj is forced to wait until Ti completes; this effectively orders the transactions.
  - What if Tj already has a lock on Y and Ti later requests a lock on Y? (Deadlock!) Ti or Tj is aborted and restarted!

Ensuring Atomicity

- DBMS ensures *atomicity* (all-or-nothing property) even if system crashes in the middle of a Xact.
- Idea: Keep a *log* (history) of all actions carried out by the DBMS while executing a set of Xacts:
  - Before a change is made to the database, the corresponding log entry is forced to a safe location. (*WAL protocol*; OS support for this is often inadequate.)
  - After a crash, the effects of partially executed transactions are *undone* using the log. (Thanks to WAL, if log entry wasn't saved before the crash, corresponding change was not applied to database!)

The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: The old value and the new value.
    - Log record must go to disk <u>*before*</u> the changed page!
  - *Ti commits/aborts*: A log record indicating this action.
- Log records chained together by Xact id, so it's easy to undo a specific Xact (e.g., to resolve a deadlock).
- Log is often *duplexed* and *archived* on "stable" storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Databases make these folks happy

- End users and DBMS vendors
- DB application programmers
  - e.g., smart webmasters
- *Database administrator (DBA)*
  - Designs logical /physical schemas
  - Handles security and authorization
  - Data availability, crash recovery
  - Database tuning as needs evolve

Must understand how a DBMS works!

Structure of a DBMS

- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.

Figure 1.3 Architecture of a DBMS

Summary

- DBMS used to maintain, query large datasets.
- Benefits include recovery from system crashes, concurrent access, quick application development, data integrity and security.
- Levels of abstraction give data independence.
- A DBMS typically has a layered architecture.
- DBAs hold responsible jobs and are well-paid! ☺
- DBMS R&D is one of the broadest, most exciting areas in CS.

The Entity-Relationship Model

Data Model

- Move from informal description of what user wants to
- Precise description of what can be implemented in a DBMS

Payroll officer needs access to staff details
Customer service advisor needs access to customer accounts
Sales manager needs access to stock levels
DBMS
Central database

Steps in developing a database

- Requirements analysis →
- Conceptual Database design →
- Logical Database design →
- Schema refinement →
- Physical Database design
- Applications and security

Conceptual design: (ER Model is used at this stage.)

- What are the *entities* and *relationships* in the enterprise?
- What information about these entities and relationships should we store in the database?
- What are the *integrity constraints* or *business rules* that hold?
- A database `schema' in the ER Model can be represented pictorially (*ER diagrams*).
- Can map an ER diagram into a relational schema.

- The ER diagram is just an approximate description of the data, constructed through a subjective evaluation of the information collected during requirements analysis.

- A more careful analysis can often refine the logical schema obtained at the end of Step 3

ER Model Basics

- *Entity:* Real-world object distinguishable from other objects. An entity is described using a set of *attributes*. Each attribute has a *domain*.
- *Entity Set*: A collection of similar entities. E.g., all employees.
  - All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
  - Each entity set has a *key*.

- Two classes of entities are there:
  - Regular Entity or Strong Entity
  - Weak Entity

Attribute

Properties/characteristics which describe entities are called attributes.

Key Attribute

Multivalued Attributes

Derived Attribute

Relationships

Associations between entities are called relationships

Keys

- Minimal set of attributes which uniquely identify an instance of a entity
- Many candidate keys choose one to be a primary keys
- SSN vs Name … key must be unique

Keys

Types of keys in DBMS

Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table. (ex: In a student table SRN or studID)

Super Key – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table. (ex: **Employee table (Emp_SSN, Emp_num, Emp_name))**

Candidate Key – A super key with no redundant attribute is known as candidate key, Candidate key is also called as minimal super key. Primary keys can be selected from Candidate keys.

Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

for better understanding of the key. Table "**Branch Info**"

	Branch_Id	Branch_Name	Branch_Code
	1	Computer Scie...	CSE
	2	Electronics	ECE
	3	Mechnical	MCE
	4	Information Te...	ITE
	5	Civil	CVE
▸*	NULL	NULL	NULL

Table: Branch_Info

Possible Candidate Keys in Branch_Info table.

- Branch_Id
- Branch_Name
- Branch_Code

Among the above candidate keys one key can be primary key which uniquely identifies the entire relation. (Branch_Id), Alternate Key (Secondary Key) is Branch_name or Branch_code.

- Unique Key

ER Model Basics (Contd.)

- *Relationship*: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
- *Relationship Set*: Collection of similar relationships.
  - An n-ary relationship set R relates n entity sets E1 ... En; each relationship in R involves entities e1 E1, ..., en En
  - Same entity set could participate in different relationship sets, or in different "roles" in same set.

Key Constraints

- Consider Works_In: An employee can work in many departments; a dept can have many employees.

So many to many

Key Constraints

- In contrast, each dept has at most one manager, according to the *key constraint* on Manages.

- * arrow indicates that given a dept it uniquely determines the Manages relationship in which it appears

Cardinality of a Relationship

- Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivity's as given below.
- 1. One to one (1:1) relationship
- 2. One to many (1:N) relationship
- 3. Many to one (M:1) relationship
- 4. Many to many (M:N) relationship

Examples

Employee
1
Assigned With
1
Parking Space
Organization
1
has
N
Employee
Employee
M
Works in
1
Organization
Student
M
enrolls
N
Course

Degree of a Relationship

- Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n.
- Special cases are unary, binary, and ternary where the degree is 1, 2, and 3, respectively.

- Example for unary relationship : An employee is a manager of another employee
- Example for binary relationship : An employee works-for department.
- Example for ternary relationship : customer purchase item from a shop keeper

Participation Constraints

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
- Every Department entity must appear in an instance of the relationship Works_In (have an employee) and every Employee must be in a Department
- Both Employees and Departments participate totally in Works_In

Weak Entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.

ISA (`is a') Hierarchies

- As in C++, attributes can be inherited.

- If we declare A **ISA** B, every A entity is also considered to be a B entity.

- Upwards is generalization. Down is specialization

Constraints in ISA relation

- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)

- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*

- Reasons for using ISA:
  - To add descriptive attributes specific to a subclass.
  - To identify entitities that participate in a relationship.

Aggregation

- Used when we have to model a relationship involving (entitity sets and) a *relationship set*.

Aggregation allows us to treat a relationship set as an entity set for purposes of participation in (other) relationships.

Aggregation vs. ternary relationship:

- Monitors in last example is a distinct relationship, with a descriptive attribute.
- Also, can say that each sponsorship is monitored by at most one employee.

Conceptual Design Using the ER Model

- Design choices:
  - Should a concept be modeled as an entity or a relationship?
  - Identifying relationships: Binary or ternary? Aggregation?
- Constraints in the ER Model:
  - A lot of data semantics can (and should) be captured.
  - But some constraints cannot be captured in ER diagrams.

Entity vs. Attribute

- Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?
- Depends upon the use we want to make of address information, and the semantics of the data:
    - If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued).
    - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity (since attribute values are atomic).

Entity vs. Attribute (Contd.)

- Works_In4 does not allow an employee to work in a department for two or more periods.

- Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, Duration.

Entity vs. Relationship

- ER diagram OK if a manager gets a separate discretionary budget for each dept.
- What if a manager gets a discretionary budget that covers *all* managed depts?
  - Redundancy: *dbudget* stored for each dept managed by manager.
  - Misleading: Suggests *dbudget* associated with department-mgr combination.

This fixes the problem!

Binary vs. Ternary Relationships

- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints do we need?

Better design

- Key constraint
- Total participation of policies in purchaser relationship

Binary vs. Ternary Relationships (Contd.)

- Previous example illustrated a case when two binary relationships were better than one ternary relationship.
- An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*. No combination of binary relationships is an adequate substitute:
  - S "can-supply" P, D "needs" P, and D "deals-with" S does not imply that D has agreed to buy P from S.
  - How do we record *qty*?

Summary of Conceptual Design

- *Conceptual design* follows *requirements analysis*,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: *entities*, *relationships*, and *attributes* (of entities and relationships).
- Some additional constructs: *weak entities*, *ISA hierarchies*, and *aggregation*.
- Note: There are many variations on ER model.

Schema Diagram for University Database

Summary of ER (Contd.)

- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap/covering constraints* for ISA hierarchies. Some *foreign key constraints* are also implicit in the definition of a relationship set.
  - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
  - Constraints play an important role in determining the best database design for an enterprise.

Summary of ER (Contd.)

- ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
  - Entity vs. attribute, entity vs. relationship, binary or n-ary relationship, whether or not to use ISA hierarchies, and whether or not to use aggregation.
- Ensuring good database design: resulting relational schema should be analyzed and refined further. FD information and normalization techniques are especially useful.

Structured Query Language (SQL)

SEQUEL-XRM and System-R projects (1974–1977).

Current ANSI/ISO standard for SQL, which is called SQL-92.

Structured Query Language (SQL)

DDL, DML, Embedded SQL and dynamic SQL, Triggers.

Example Instances

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Base tables used in Textbook

- Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
- Boats(*bid:* integer, *bname:* string, *color:* string)
- Reserves(*sid:* integer, *bid:* integer, *day:* date)

- The basic form of SQL Query

```
SELECT    [ DISTINCT ] select-list
FROM      from-list
WHERE     qualification
```

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance $B1$ of Boats

- SQL Query corresponds to a relational algebra expression which involves selections, projections and cross-products.

- *Find the names and ages of all sailors.*
- *Sol: select distinct s.name, s.age from sailor s;*
- *Find all sailors with a rating above 7.*
- *Sol:* SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

- *Find the names of sailors who have reserved boat number 103.*
- *Sol:* SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103;
- *Find the sids of sailors who have reserved a red boat.*
- *Sol:* SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND B.color = ‘red’
- *Find the names of sailors who have reserved a red boat.*

  SELECT S.sname FROM Sailors S, Reserves R, Boats B
- WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

- *Find the colors of boats reserved by Lubber.*
- *Sol:* SELECT B.color FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = ‘Lubber’

Find the names of sailors who have reserved at least one boat.

- *Sol:* SELECT S.sname FROM Sailors S, Reserves R
- WHERE S.sid = R.sid

Basic SQL Query

SELECT	[DISTINCT] *target-list*
FROM	*relation-list*
WHERE	*qualification*

- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of $<, >, =, \leq, \geq, \neq$) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

OR

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

It is good style, however, to use range variables always!

Find sailors who've reserved at least one boat

```
SELECT  S.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT  S.age, age1=S.age-5, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.

Find sid's of sailors who've reserved a red <u>or</u> a green boat

- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
  AND (B.color='red' OR B.color='green')
```

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
            AND B.color='red'
UNION
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
            AND B.color='green'
```

Find sid's of sailors who've reserved a red and a green boat

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT  S.sid
FROM  Sailors S, Boats B1, Reserves R1,
          Boats B2, Reserves R2
WHERE  S.sid=R1.sid AND R1.bid=B1.bid
 AND  S.sid=R2.sid AND R2.bid=B2.bid
 AND (B1.color='red' AND B2.color='green')
```

Key field!

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
          AND B.color='red'
INTERSECT
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
          AND B.color='green'
```

Nested Queries

Find names of sailors who've reserved boat #103:

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  (SELECT  R.sid
                  FROM  Reserves R
                  WHERE  R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a *nested loops* evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

```
SELECT  S.sname
FROM  Sailors S
WHERE  EXISTS  (SELECT  *
                FROM  Reserves R
                WHERE  R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* IN $>,<,=,\geq,\leq,\neq$
- Find sailors whose rating is greater than that of some sailor called Horatio:

```
SELECT  *
FROM  Sailors S
WHERE  S.rating > ANY  (SELECT  S2.rating
                        FROM  Sailors S2
                        WHERE S2.sname='Horatio')
```

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
        AND S.sid IN  (SELECT  S2.sid
                       FROM  Sailors S2, Boats B2, Reserves R2
                       WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                              AND  B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division in SQL

(1)

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
          ((SELECT  B.bid
            FROM  Boats B)
           EXCEPT
            (SELECT  R.bid
             FROM  Reserves R
             WHERE  R.sid=S.sid))
```

Find sailors who've reserved all boats.

- Let's do it the hard way, without EXCEPT:

(2)

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
                                        FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                          AND R.sid=S.sid))
```

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B

Aggregate Operators

```
COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```

single column

- Significant extension of relational algebra.

```
SELECT  COUNT (*)
FROM  Sailors S
```

```
SELECT  AVG (S.age)
FROM  Sailors S
WHERE  S.rating=10
```

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.rating= (SELECT  MAX(S2.rating)
                  FROM  Sailors S2)
```

```
SELECT  COUNT (DISTINCT S.rating)
FROM  Sailors S
WHERE S.sname='Bob'
```

```
SELECT  AVG ( DISTINCT S.age)
FROM  Sailors S
WHERE  S.rating=10
```

Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT  S.sname, MAX (S.age)
FROM  Sailors S
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX (S2.age)
           FROM  Sailors S2)
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  (SELECT  MAX (S2.age)
          FROM  Sailors S2)
          = S.age
```

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For i = 1, 2, ... , 10:

```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.rating = i
```

Queries With GROUP BY and HAVING

SELECT	[DISTINCT] *target-list*
FROM	*relation-list*
WHERE	*qualification*
GROUP BY	*grouping-list*
HAVING	*group-qualification*

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age ≥18, for each rating with at least 2 <u>such</u> sailors

```
SELECT  S.rating,  MIN (S.age)
                          AS minage
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age ≥18, for each rating with at least 2 <u>such</u> sailors.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age ≥18, for each rating with at least 2 <u>such</u> sailors and with every sailor under 60.

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

Find age of the youngest sailor with age ≥18, for each rating with at least 2 sailors between 18 and 60.

```
SELECT  S.rating,  MIN (S.age)
                        AS minage
FROM  Sailors S
WHERE  S.age >= 18 AND S.age <= 60
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

For each red boat, find the number of reservations for this boat

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?

Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age > 18
GROUP BY  S.rating
HAVING  1  <  (SELECT  COUNT (*)
               FROM  Sailors S2
               WHERE  S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested! WRONG:

```
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN (AVG (S2.age)) FROM Sailors S2)
```

❖ Correct solution (in SQL/92):

```
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG (S.age) AS avgage
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN (Temp.avgage)
                     FROM Temp)
```

Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null*.
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE Sailors
    ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK ( rating >= 1
            AND rating <= 10 )
```

```
CREATE TABLE Reserves
    ( sname CHAR(10),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid,day),
    CONSTRAINT noInterlakeRes
    CHECK (`Interlake' <>
            ( SELECT B.bname
            FROM Boats B
            WHERE B.bid=bid)))
```

Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

```
CREATE TABLE Sailors
( sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Number of boats plus number of sailors is < 100

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

SQL Joins

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

- Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Employee table

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmpID1.pic
2	Fuller	Andrew	2/19/1952	EmpID2.pic
3	Leverling	Janet	8/30/1963	EmpID3.pic

- Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.
- Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

-

- Different Types of SQL JOINs
- Here are the different types of the JOINs in SQL:
- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table

INNER JOIN
table1
table2
LEFT JOIN
table1
table2
RIGHT JOIN
table1
table2
FULL OUTER JOIN
table1
table2

```
SELECT * FROM Orders LEFT JOIN Customers;
```

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

Note: The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

- In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

- SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
  FROM Orders
  RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
  ORDER BY Orders.OrderID;

- **Note:** The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

- The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.
- **Note:** FULL OUTER JOIN can potentially return very large result-sets!

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers FULL OUTER JOIN Orders ON
Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

SQL Group By, Having, Exists, Any, All

The SQL GROUP BY Statement

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

SELECT *column_name(s)*
FROM *table_name*
WHERE *condition*
GROUP BY *column_name(s)*
ORDER BY *column_name(s);*

- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country;

- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country
  ORDER BY COUNT(CustomerID) DESC;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

- The SQL HAVING Clause
- The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

SELECT *column_name(s)*
FROM *table_name*
WHERE *condition*
GROUP BY *column_name(s)*
HAVING *condition*
ORDER BY *column_name(s);*

- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country
  HAVING COUNT(CustomerID) > 5;

- SELECT COUNT(CustomerID), Country
  FROM Customers
  GROUP BY Country
  HAVING COUNT(CustomerID) > 5
  ORDER BY COUNT(CustomerID) DESC;

- The following SQL statement lists the employees that have registered more than 10 orders:

- SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOf Orders FROM (Orders
  INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)
  GROUP BY LastName
  HAVING COUNT(Orders.OrderID) > 10;

Relational Algebra

Chapter 4, Part A

Relational Query Languages

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

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Relational Algebra

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$$\pi_{sname,rating}(S2)$$

age
35.0
55.5

$$\pi_{age}(S2)$$

Selection

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

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- *Renaming operator*: $\rho\,(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$

Joins

- *Condition Join*: $R \bowtie_c S = \sigma_c (R \times S)$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*. $S1 \bowtie_{S1.sid < R1.sid} R1$

Joins

- *Equi-Join*: A special case of condition join where the condition *c* contains only ***equalities.***

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$$S1 \bowtie_{sid} R1$$

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

  Find sailors who have reserved ***<u>all</u>*** *boats*.

- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\{\langle x \rangle \mid \exists \langle x, y \rangle \in A \ \ \forall \langle y \rangle \in B\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *<u>every</u>* *y* tuple (boat) in *B*, there is an *xy* tuple in *A***.
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.

- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and *x* $\cup$ *y* is the list of fields of *A*.

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Expressing A/B Using Basic Operators

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

Disqualified *x* values: $\pi_x((\pi_x(A) \times B) - A)$

A/B: $\pi_x(A)$ − all disqualified tuples

Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103} \text{Reserves}) \bowtie Sailors)$

❖ Solution 2: $\rho\ (Temp1,\ \sigma_{bid=103} \text{Reserves})$

$\rho\ (Temp2,\ Temp1 \bowtie Sailors)$

$\pi_{sname}(Temp2)$

❖ Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \bowtie Sailors))$

Find names of sailors who've reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie \mathrm{Re}serves \bowtie Sailors)$$

❖ A more efficient solution:

$$\pi_{sname}(\pi_{sid}((\pi_{bid} \sigma_{color='red'} Boats) \bowtie \mathrm{Re}s) \bowtie Sailors)$$

A query optimizer can find this, given the first solution!

Find sailors who've reserved a red or a green boat

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}\ Boats))$$

$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

- Can also define Tempboats using union! (How?)
- What happens if $\vee$ is replaced by $\wedge$ in this query?

Find sailors who've reserved a red <u>and</u> a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname}(Tempsids \bowtie Sailors)$$

❖ To find sailors who've reserved all 'Interlake' boats:

$$\ldots\ldots \quad / \ \pi_{bid}(\sigma_{bname='Interlake'} Boats)$$

Additional information on Relational Algebra

A
Basic Relational Algebra

- A data model must include a set of operation to manipulate the database.
- This basic set of operation for relational model is Relational Algebra.
- A sequence of relational algebra operations forms a Relational Algebra expression.

Importance of Relational Algebra:

- It provides a formal foundation for relational model operation.
- It is used as a basis for implementing and optimizing queries in RDBMS.
- Some of its concepts are incorporated into the SQL standard query language for RDBMS.

Two groups of operations

- Set Operations:
  - Union
  - Intersection
  - Difference
  - Cartesian product
- Developed specially for relational database:
  - Select
  - Project
  - Join

- Unary operation:
  - Select
  - Project

- Binary operation:
  - All set operations
  - Join

Select Operation

- Used to select a subset of the tuples from a relation that satisfy a selection condition.

  example

- It is a horizontal partitioning of the relation:

  - One partition satisfying the condition and will be shown as result.
  - Other partition not satisfying the condition will be discarded.

Select Operation

- σ <selection condition> (R)
  - Where σ (sigma) is for select operation
  - selection condition is a boolean expression specified on the attributes of Relation R

    Example
- Clauses of selection condition
  - <Attribute name> <comparison operator> <constant value>
  - <Attribute name> <comparison operator> <Attribute name>

- Comparison operator
  - =, <, >, ≠, ≥,≤ for numeric value or date
  - =, ≠ for strings of characters

- Constant value
  - Value from attribute domain

- Degree of resulting relation is same as that of R
- No. of tuples in resulting relation <= no. of tuples in R
  - i.e. $\sigma_c(R) <= R$
- Selection condition may be more than one
  - i.e. (Cond1 and Cond2) → true if both are true
  - i.e. (Cond1 or Cond2) → true if any one or both are true
  - Not Cond →true if cond is false

- Select operation is Commutative:
  - $\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$
  - A sequence of Selects can be applied in any order.

    Example
- Cascade of Selects can be used through one select using Conjunctive (AND)
  - $\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots\ldots))$

  $= \sigma_{<cond1> \text{ AND} <cond2> \text{AND}\ldots\ldots \text{ AND} <condn>}(R)$

  Example

Project Operation

- Project operation selects certain column from relation and discards rest.

Example

- It is a vertical partitioning of the relation:
  - One partition satisfying the condition and will be shown as result.
  - Other partition not satisfying the condition and will be discarded.

- Π <attribute list> (R) operation
  - Where Π (pi) is for project
- Attribute list is the desired list of attributes from relation R
- Resulting relation has attribute only those specified in list and in same order

- Degree of resulting relation is same as no. of attributes in list
- No. of tuples in resulting relation?
  - If attribute list has Primary key, the no. of tuples will be same as in Relation R
  - If attribute list does not have Primary key, the duplicate tuples will be removed to give valid relation
  - No. of tuples in resulting relation =< no. of tuples in relation R

- Project operation is not Commutative:
  - $\Pi_{<list1>} (\Pi_{<list2>} (R)) \neq \Pi_{<list2>} (\Pi_{<list1>} (R))$
  - $\Pi_{<list1>} (\Pi_{<list2>} (R)) = \Pi_{<list1>} (R)$

    Example

Sequence of operations

- Several algebraic expressions:
  - As a single relational expression by nesting operations
  - $\Pi_{<list1>}(\sigma_{<cond1>}(R))$ Example

  Or
  - Name the intermediate relation and get final result as a series of operations
  - temp $\leftarrow \sigma_{<cond1>}(R)$
  - result $\leftarrow \Pi_{<list1>}(temp)$ Example

Rename

- Attributes can be renamed in resulting relation
  - temp ← $\sigma_{<cond1>}$ (Employee)
  - Emp(empid, empname, empsalary) ← $\Pi_{<eid, ename, salary>}$ (temp)

- If no renaming is not applied, resulting relation will have same names and order of attributes as parent relation has.

Formal Rename operation

- Renaming both relation name and attributes

  $\rho_{S(B1, B2,....Bn)} (R)$

- Renaming only relation name

  $\rho_{S} (R)$

- Renaming only attributes

  $\rho_{(B1, B2,....Bn)} (R)$

Set Theoretic Operation

- Binary operation
- Both relational should be union Compatible
  - i.e. for R(A1, A2, ….An) & S(B1,B2,….Bn)
  - dom(Ai) = dom(Bi) for 1 <=i<=n
- UNION (RUS)
- INTERSECTION (R∩S)
- MINUS (R-S)

- UNION (R∪S)
  - It includes all tuples that are either in R or S or in both
  - Duplicate tuples are eliminated.

    Example
- INTERSECTION (R∩S)
  - It includes all tuples that are in R & S both

    Example

- Set difference (MINUS)
- (R-S) example
  - It includes all tuples that are in R but not in S
- (S-R) example
  - It includes all tuples that are in S but not in R

- (R-S) ≠ (S-R)

example

- Union & Intersection are:

1. Commutative
   - (R∪S) = (S∪R)
   - (R∩S) = (S∩R)

- Associative
   - (R∪(S∪T) = (R∪S)∪T)
   - (R∩(S ∩T) = (R∩S)∩T)

example

Cartesian Product

- Binary Operation
- Union Compatibility not required
- R x S example
  - no. of tuples in Resultant relation = no. of tuples in R x no. of tuples in S
  - It includes one tuple from each combination
  - No. of attributes in Resultant relation =n + m
  - R(A1, A2,.....,An) x S(B1, B2,....,Bn) = Q(A1, A2,.....,An, B1, B2,....,Bn)
- Result will have some irrelevant tuple which can be further processed by select & project operation

Join operation

- R ⋈ S It is used to combine related tuples from two relations into single tuple.
- It allows to process relationships among relations
- R <join condition> S example
  - Attributes in Resultant relation = n + m
  - R(A1, A2,.....,An) ⋈ S(B1, B2,....,Bn) = Q(A1, A2,.....,An, B1, B2,....,Bn)
  - No. of tuples wherever join condition is satisfied.

⋈

- Join operation can be stated as Cartesian product operation followed by Select operation

- Theta join: example
  - $A_i \theta B_i$
  - $dom(A_i) = dom(B_i)$
  - θ is one of the $\{=,<,>,\geq,\leq,\neq\}$
  - Tuples whose join attributes are null do not appear in the result

Equi join:

- Most commonly used
- Operator is =
- One or more pairs of attributes that have identical values
- E.g. Mgrssn = ssn example

Natural join:

- R * S [example]

- No condition is required

- Both the tables should have one same attribute with same name

- If attribute is same but name is different (first perform rename than natural join)

Join selectivity

- If join condition is not satisfied → no tuple in resultant relation

- Tuples are in between zero and n_R x n_S

- Join selectivity ratio = expected size of join result / maximum size

m-way join

- ((Project $\bowtie_{DNUM = DNUMBER}$ Department) $\bowtie_{MGRSSN = SSN}$ Employee)

A complete set of relational algebra operation

- {σ, π, U, -, x) is a complete set
- R ∩ S = (RUS) – ((R-S) U (S-R))
- R ⋈ <condition> S = σ <condition> (R x S)
- Natural join = π…(σ…(ρ….(R x S)))

- But for convenience, different operations are used.

Division operation

- Denoted by ÷
- Operation is applied to two relations $R(Z) \div S(X)$

example

Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; useful as internal representation for query evaluation plans.
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.

Examples for relational algebra in the following slides

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000
105	Pam	18,000

Resultant table:

Employee

Empid	Name	Salary
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000
105	Pam	18,000

σ salary>20,000 (Employee)

Resultant table:

Employee

Empid	Name	Salary
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Employee

σ salary>20,000

(σ Dno=2 (Employee))

Empid	Name	Dno	Salary
101	Ram	1	20,000
102	Shyam	2	22,000
103	Geeta	3	24,000
104	Geeta	2	25,000
105	Pam	2	18,000

Resultant table:

Employee

Empid	Name	Dno	Salary
102	Shyam	2	22,000
104	Geeta	2	25,000

Employee

σ Dno=2

(σ salary>20,000 (Employee))

Empid	Name	Dno	Salary
101	Ram	1	20,000
102	Shyam	2	22,000
103	Geeta	3	24,000
104	Geeta	2	25,000
105	Pam	2	18,000

Resultant table:

Employee

Empid	Name	Dno	Salary
102	Shyam	2	22,000
104	Geeta	2	25,000

Employee

σ Dno=2 and

σ salary>20,000 (Employee)

Empid	Name	Dno	Salary
101	Ram	1	20,000
102	Shyam	2	22,000
103	Geeta	3	24,000
104	Geeta	2	25,000
105	Pam	2	18,000

Resultant table:

Employee

Empid	Name	Dno	Salary
102	Shyam	2	22,000
104	Geeta	2	25,000

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Π empid, salary (Employee)

Resultant table:

Employee

Empid	Salary
101	20,000
102	22,000
103	24,000
104	25,000

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	25,000
104	Geeta	25,000

Π name, salary (Employee)

Resultant table:

Employee

Name	Salary
Ram	20,000
Shyam	22,000
Geeta	25,000

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Π empid (Π empid, salary (Employee))

Resultant table:

Employee

Empid
101
102
103
104

Π salary (Π empid, salary (Employee))

Employee →

Salary
20,000
22,000
24,000
25,000

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Π empid (Π empid, salary (Employee))

Resultant table:

Employee

Empid
101
102
103
104

Π empid (Employee))

Employee →

Empid
101
102
103
104

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Π_{empid} ($\sigma_{\text{salary>20,000}}$ (Employee))

Resultant table:

Employee

Empid
102
103
104

Employee

Π empid (σ salary>20,000 (Employee))

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Π empid (temp)

Employee

temp

temp $\leftarrow$ σ salary>20,000 (Employee)

Empid	Name	Salary
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Empid
102
103
104

Employee

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

EMP(EID) ← Π empid (temp)

EMP

EID
102
103
104

Employee

temp ← σ salary>20,000 (Employee)

Empid	Name	Salary
101	Ram	20,000
102	Shyam	22,000
103	Geeta	24,000
104	Geeta	25,000

Student

Stu_id	Name
101	Ram
102	Shyam
103	Geeta
104	Rita
105	John

Instructor

Empid	Name
102	Shyam
106	Smith
107	Nita

Student U Instructor

Stu_id	Name
101	Ram
102	Shyam
103	Geeta
104	Rita
105	John
106	Smith
107	Nita

Student

Stu_id	Name
101	Ram
102	Shyam
103	Geeta
104	Rita
105	John

Instructor

Empid	Name
102	Shyam
106	Smith
107	Nita

Student ∩ Instructor

Stu_id	Name
102	Shyam

Student

Stu_id	Name
101	Ram
102	Shyam
103	Geeta
104	Rita
105	John

Instructor

Empid	Name
102	Shyam
106	Smith
107	Nita

Student - Instructor

Stu_id	Name
101	Ram
103	Geeta
104	Rita
105	John

Student

Stu_id	Name
101	Ram
102	Shyam
103	Geeta
104	Rita
105	John

Instructor

Empid	Name
102	Shyam
106	Smith
107	Nita

Instructor - Student

Stu_id	Name
106	Smith
107	Nita

Student - Instructor

Stu_id	Name
101	Ram
103	Geeta
104	Rita
105	John

Instructor - Student

Stu_id	Name
106	Smith
107	Nita

Student

Stu_id	Name
101	Ram
102	Shyam
103	Geeta

Player

Play_id	Name
103	Geeta
102	Shyam

Instructor

Empid	Name
102	Shyam
106	Smith
107	Nita

Student U (Instructor U Player)

Stu_id	Name
101	Ram
103	Geeta
104	Rita
105	John
102	Shyam

Cartesian Product

Employee

Emp_id	Name
101	Ram
102	Shyam
103	Geeta

Dependant

Dep_name	Bdate
Meena	23-02-1988
Raju	23-02-1990

Employee x dependant

Emp_id	Name	Dep_name	Bdate
101	Ram	Meena	23-02-1988
101	Ram	Raju	23-02-1990
102	Shyam	Meena	23-02-1990
102	Shyam	Raju	23-02-1990
103	Geeta	Meena	23-02-1990
103	Geeta	Raju	23-02-1990

Join operation

Employee

emp_id	Name
101	Ram
102	Shyam
103	Geeta

Dependant

Dep_name	eid
Meena	101
Raju	102

Employee $\bowtie_{emp_id=eid}$ Dependant

emp_id	eid	Name	Dep_name
101	101	Ram	Meena
102	102	Shyam	Raju

Join operation

Employee

emp_id	Name
101	Ram
102	Shyam
103	Geeta

Dependant

Dep_name	emp_id
Meena	101
Raju	102

Employee * Dependant

emp_id	Name	Dep_name
101	Ram	Meena
102	Shyam	Raju

Division

R

ESSN	PNO
12345	1
12345	2
23456	2
23456	1
35346	3
21336	4

S

PNO
1
2

R ÷ S

ESSN
12345
23456

Schema Refinement and Normal Forms

The Evils of Redundancy

- *Redundancy* is at the root of several problems associated with relational schemas:
  - redundant storage, insert/delete/update anomalies
- Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.
- Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).
- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?

What is Normalization?

- Normalization is the process of efficiently organizing data in a database.
- There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.

- Database normalization is the process of restructuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity. It was first proposed by Edgar F. Codd as an integral part of his relational model.

Functional Dependencies (FDs)

- A <u>functional dependency</u> $X \rightarrow Y$ holds over relation R if, for every allowable instance *r* of R:
  - $t1 \in r,\ t2 \in r,\ \pi_X(t1) = \pi_X(t2)$ implies $\pi_Y(t1) = \pi_Y(t2)$
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)
- An FD is a statement about *all* allowable relations.
  - Must be identified based on semantics of application.
  - Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R!
- K is a candidate key for R means that $K \rightarrow R$
  - However, $K \rightarrow R$ does not require K to be *minimal*!

Example: Constraints on Entity Set

- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (*ssn*, *name, lot, rating, hrly_wages*, *hrs_worked*)
- *Notation*: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the *set* of attributes {S,N,L,R,W,H}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)
- Some FDs on Hourly_Emps:
  - *ssn* is the key: $S \rightarrow SNLRWH$
  - *rating* determines *hrly_wages*: $R \rightarrow W$

Example (Contd.)

- Problems due to R ⟶ W :
  - *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
  - *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
  - *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Will 2 smaller tables be better?

Wages

R	W
8	10
5	7

Hourly_Emps2

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - *ssn* $\rightarrow$ *did*, *did* $\rightarrow$ *lot* implies *ssn* $\rightarrow$ *lot*
- An FD *f* is <u>*implied by*</u> a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - F^+ = *closure of F* is the set of all FDs that are implied by *F*.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - <u>*Reflexivity*</u>: If X $\subseteq$ Y, then Y $\rightarrow$ X
  - <u>*Augmentation*</u>: If X $\rightarrow$ Y, then XZ $\rightarrow$ YZ for any Z
  - <u>*Transitivity*</u>: If X $\rightarrow$ Y and Y $\rightarrow$ Z, then X $\rightarrow$ Z
- These are *sound* and *complete* inference rules for FDs!

Reasoning About FDs (Contd.)

- Couple of additional rules (that follow from AA):
  - *Union*: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - *Decomposition*: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Example: Contracts(*cid,sid,jid,did,pid,qty,value*), and:
  - C is the key: $C \rightarrow CSJDPQV$
  - Project purchases each part using single contract: $JP \rightarrow C$
  - Dept purchases at most one part from a supplier: $SD \rightarrow P$
- $JP \rightarrow C$, $C \rightarrow CSJDPQV$ imply $JP \rightarrow CSJDPQV$
- $SD \rightarrow P$ implies $SDJ \rightarrow JP$
- $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ imply $SDJ \rightarrow CSJDPQV$

Reasoning About FDs (Contd.)

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD *X* *Y* is in the closure of a set of FDs *F*. An efficient check:
  - Compute *attribute closure* of X (denoted $\rightarrow$) wrt *F:* X^+
    - Set of all attributes A such that X A is in F^+
    - There is a linear time algorithm to compute this.
  - Check if Y is in
- Does F = {A $\rightarrow$ B, B $\rightarrow$ C, C D $\rightarrow$ E } imply A $\rightarrow$ E?
  - i.e, is A $\rightarrow$ E in the closure X^+ ? F^+ Equivalently, is E in A^+ ?

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given $A \rightarrow B$: Several tuples could have the same A value, and if so, they'll all have the same B value!

Boyce-Codd Normal Form (BCNF)

- Reln R with FDs *F* is in BCNF if, for all $X \rightarrow A$ in F^+
  - $A \in X$ (called a *trivial* FD), or
  - X contains a key for R.
- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If example relation is in BCNF, the 2 tuples must be identical (since X is a key).

X	Y	A
x	y1	a
x	y2	?

Third Normal Form (3NF)

- Reln R with FDs *F* is in 3NF if, for all X $\rightarrow$A in F^+
  - A $\in$X (called a *trivial* FD), or
  - X contains a key for R, or
  - A is part of some key for R.
- *Minimality* of a key is crucial in third condition above!
- If R is in BCNF, obviously in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
  - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

What Does 3NF Achieve?

- If 3NF violated by $X \rightarrow A$, one of the following holds:
  - X is a subset of some key K
    - We store (X, A) pairs redundantly.
  - X is not a proper subset of any key.
    - There is a chain of FDs $K \rightarrow X \rightarrow A$, which means that we cannot associate an X value with a K value unless we also associate an A value with an X value.
- But: even if reln is in 3NF, these problems could arise.
  - e.g., Reserves SBDC, $S \rightarrow C$, $C \rightarrow S$ is in 3NF, but for each reservation of sailor S, same (S, C) pair is stored.
- Thus, 3NF is indeed a compromise relative to BCNF.

Decomposition of a Relation Scheme

- Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs $S \rightarrow SNLRWH$ and $R \rightarrow W$
  - Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
    - i.e., we decompose SNLRWH into SNLRH and RW
- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

- There are three potential problems to consider:
  - Some queries become more expensive.
    - e.g., How much did sailor Joe earn? (salary = W*H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
    - Fortunately, not in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations.
    - Fortunately, not in the SNLRWH example.
- *Tradeoff*: Must consider these issues vs. redundancy.

Lossless Join Decompositions

- Decomposition of R into X and Y is *lossless-join* w.r.t. a set of FDs F if, for every instance *r* that satisfies F:
  - $\pi_X(r) \bowtie \pi_Y(r) = r$
- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem (2).)*

More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$
- In particular, the decomposition of R into UV and R - V is lossless-join if $U \rightarrow V$ holds over R.

A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP $\rightarrow$ C and SD $\rightarrow$ P.
  - BCNF decomposition: CSJDQV and SDP
  - Problem: Checking JP $\rightarrow$ C requires a join!
- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3).)*
- *Projection of set of FDs F*: If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs U $\rightarrow$ V in F^+ (*closure of F*) such that U, V are in X.

Dependency Preserving Decompositions (Contd.)

- Decomposition of R into X and Y is *dependency preserving* if $(F_X \text{ union } F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.
- Important to consider F^+, not F, in this definition:
  - ABC, $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \rightarrow A$ preserved?????
- Dependency preserving does not imply lossless join:
  - ABC, $A \rightarrow B$, decomposed into AB and BC.
- And vice-versa! (Example?)

Decomposition into BCNF

- Consider relation R with FDs F. If $X \rightarrow Y$ violates BCNF, decompose R into R - Y and XY.
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, $JP \rightarrow C$, $SD \rightarrow P$, $J \rightarrow S$
  - To deal with $SD \rightarrow P$, decompose into SDP, CSJDQV.
  - To deal with $J \rightarrow S$, decompose CSJDQV into JS and CJDQV
- In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, $CS \rightarrow Z$, $Z \rightarrow C$
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs $JP \rightarrow C$, $SD \rightarrow P$ and $J \rightarrow S$).
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
    - JPC tuples stored only for checking FD! (*Redundancy!*)

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).
- To ensure dependency preservation, one idea:
  - If $X \rightarrow Y$ is not preserved, add relation XY.
  - Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' $JP \rightarrow C$. What if we also have $J \rightarrow C$?
- Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.

Minimal Cover for a Set of FDs

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G.
  - Right hand side of each FD in G is a single attribute.
  - If we modify G by deleting an FD or by deleting attributes from an FD in G, the closure changes.
- Intuitively, every FD in G is needed, and ``*as small as possible*'' in order to get the same closure as F.
- e.g., $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$ has the following minimal cover:
  - $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$ and $EF \rightarrow H$
- M.C. $\rightarrow$ Lossless-Join, Dep. Pres. Decomp!!! (in book)

Refining an ER Diagram

- 1st diagram translated:
  Workers(S,N,L,D,S)
  Departments(D,M,B)
  - Lots associated with workers.
- Suppose all workers in a dept are assigned the same lot:
- D $\longrightarrow$ L
- Redundancy; fixed by:
  Workers2(S,N,D,S)
  Dept_Lots(D,L)
- Can fine-tune this:
  Workers2(S,N,D,S)
  Departments(D,M,B,L)

Before:

After:

Summary of Schema Refinement

- If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
  - Must consider whether all FDs are preserved. If a lossless-join, dependency preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), should consider decomposition into 3NF.
  - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.

Additional Information on Normalization with examples

Normalization
for Relational Databases

Informal Design Guidelines for Relation Schemas

- Four informal methods are considered for relation schema design.
  - Semantics of the attributes
  - Reducing the redundant values in tuples.
  - Reducing the Null values in tuples.
  - Disallowing the possibility of generating spurious tuples.

Semantics of the relation attributes

- Semantics specify how to interpret the attribute values stored in a tuple of the relation.
- For a better relation schema design, semantics of the relation should be easier to understand.

A Company Database

EMPLOYEE

SSN	ENAME	BDATE	ADDRESS	DNUMBER

DEPARTMENT

DNAME	DNUMBER	DMGRSSN

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

SSN	PNUMBER	HOURS

Guideline 1

- Do not combine attributes from multiple entity types and relationship types into a single relation.

- A relation schema corresponds to one entity type or one relationship type, it is straight forward to explain its meaning.

Poor Design

EMPL_DEPT

SSN	ENAME	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION

Redundant Information in Tuples

- Good schema design minimizes the storage space used by the base tables.

- Grouping attributes into relation schema has a significant effect on storage space.

Update anomalies

- These can be classified into following:

- Insertion Anomalies
- Deletion Anomalies
- Modification (Update) Anomalies

Insertion Anomalies

- It may not be possible to store certain information unless some other, unrelated information is stored also.
- e.g.

1. To insert a new employee tuple into EMP_DEPT, we must include attribute values for the department that the employee works for, or NULLS.
2. Attributes of a new department can not be inserted if it has no employee yet or NULLS have to be inserted for attributes of employee.

Poor Design

EMPL_DEPT

SSN	ENAME	BDATE	ADDRESS	DNUM	DNAME	DMGRSSN
1	Ram	12-01-88	Sfsdgfdhb	1	MCA	1
2	Shyam	12-02-88	Dsgnhfgfn	5	CS	2
3	Rita	22-01-88	Nhgngfngf	4	IS	4
4	Lata	12-01-89	dnhgngfn	1	MCA	1
5	Mina	17-01-82	Hvjkjskkl	1	MCA	1

Deletion Anomalies

- It may not be possible to delete certain information without losing some other, unrelated information is also deleted.

- e.g.

  If an employee tuple has to be deleted which happens to be last employee in a department, that department information will also be lost.

Modification Anomalies

- If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

- E.g.
- If we change any attribute of the department, all the employee tuples should also be modified otherwise inconsistency will occur.

Guideline 2

- Design the base relation schemas so that no insertion, deletion or modification anomalies are present in the relations.
- These guidelines sometimes have to be violated to improve the query performance.
- If any anomalies are present, it should be handled by programs.

- In general, Anomaly free base relations should be used.
- Views can be used to have such query where attributes from different relations can be displayed.

NULL values in Tuples

- Many NULLS can waste space at the storage level.
- Understanding about the attribute value and with specifying JOIN operation will be less.
- Accounting NULLS in aggregate operations such as COUNT or SUM is difficult.

- NULLS can have multiple interpretations:
  - The attribute does not apply to that particular tuple.
  - The attribute value for this tuple is unknown.
  - The value is known but not provided yet.
- But representation for all above is same i.e. NULL.

Guideline 3

- Avoid placing attributes in a base relation whose value may frequently be NULL.
- If NULL is unavoidable, they should be applied in exceptional cases.
- Using space efficiency and avoiding joins are two criteria to have such attributes which may have NULL values for many tuples.

- E.g.
- If only 10% employees have individual offices, it is not preferable to have attribute OFFICE_ADDRESS in employee relation because 90% employees will have NULL values for that attribute.

- A relation EMP_OFFICE can be created having attribute SSN & OFFICE_ADDRESS.

Generation of Spurious Tuples

- Additional tuples due to join operation will be generated which are wrong or have invalid information; these are called Spurious tuples.

Guideline 4

- Design relation schema so that they can be joined with equality condition on attributes that are either Primary keys or foreign keys and guarantee no spurious tuples are generated.

- Avoid relations that contain matching attributes which are not Primary keys or foreign keys combinations, because joining of such attributes may generate spurious tuples.

Design Guidelines: Summary

1. Do not combine attributes from multiple entity types relationship type into a single relation.

2. Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation.

Design Guidelines: Summary

3. Waste of storage space due to nulls and difficulty of performing aggregation operations and joins due to null values.

4. Generation of invalid and spurious data during joins on improperly related base relations.

Functional Dependencies

- Functional Dependency is a constraint between two sets of attributes form the relation.
- **Definition:**

  A **functional dependency,** denoted by

  X $\rightarrow$ **Y,** between two sets of attributes **X** and **Y** that are subsets of **R** specifies a **constraints** on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have t1[**X**]= t2[**X**] , they must also have

  t1[**Y**]=t2[**Y**] .

- There is a functional dependency from **X** to y, i.e. the value of **y** is determined by the value of **X,** or **X** determines the value of **Y.**

- The abbreviation for functional dependency is FD or f.d.

- The set of attributes **X** is called the **left-hand side** of the FD, and y is called the **right-hand side.**

e.g.

- SSN → ENAME
- DNUMBER → DNAME
- {SSN, PNUM} → HOURS

EMPLOYEE

SSN	ENAME	BDATE	ADDRESS	DNUMBER

FD

DEPARTMENT

DNAME	DNUMBER	DMGRSSN

FD

- A functional dependency is the property of relation schema not of a particular state of R.

- Thus functional dependency is true for all the possible states of the relation.

NORMAL FORMS BASED ON PRIMARY KEYS

Most practical relational design projects take one of the following two approaches:

- First perform a conceptual schema design using a conceptual model such as ER or EER and then map the conceptual design into a set of relations.

- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports.

- Evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms using normalization theory.

Normalization of Relations

- Three normal forms were proposed by Codd (1972).
- **Normalization of data** can be looked upon as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties as follows:

1. Minimizing redundancy
2. Minimizing the insertion , deletion , and update anomalies

- If Unsatisfactory relation schemas don’t meet certain conditions i.e. the **normal form test,** these are decomposed into smaller relation schemas that meet the tests and hence posses the desirable properties .

- Thus, the normalization procedure provides database designers with the following :

1. A formal frame work for analyzing relation schemas based on their keys and on the functional dependencies among their attributes.
2. A series of normal form tests that can be carries out on individual relation schemas so that relational database san be **normalized** to any desired degree

- The process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas (together) should possess .

- These would include two properties:
- The **lossless join** or **nonadditive join property** which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
- The **dependency preservation property** : ensures that each functional dependency is represented in some individual relation resulting after decomposition.

Practical Use of Normal Forms

- The process of storing the join of higher normal form relations as a base relation – which is in a lower normal form – is known as denormalization.

- Database design in industry today pays particular attention to normalization only up to 3NF, BCNF, 4NF.

- Sometimes relations may be left in a lower normalization status, such as 2NF, for performance reasons.

Definitions of keys and Attributes Participating in keys

- Superkey:
  - A superkey of a relation schema R= {A1, A2 …. An} is a set of attributes S R with a property that no two tuples t1 and t2 in any legal relation state r of R will have t1[S] = t2[S]

- Key (minimal superkey):
  - A key K is a superkey with additional property that removal of any attribute from K will cause K not to be a superkey

- Candidate key:
  - If a relation schema has more than one key each is called a candidate key. One of the candidate key is designated as primary key, and others are called Secondary keys.

- An attribute of relation schema R is called a **prime attribute** of R if it is a member of some *candidate key* of R.

- An attribute is called **nonprime** if it is not prime attribute – that is ,if it is not a member of any candidate key.

FIRST NORMAL FORM

- It states that the domain of an attribute must include only *atomic* (simple , indivisible) *values.*

- The value of any attribute in a tuple must be a single value from the domain of that attribute.

- Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple.

A relation schema which is not in 1 NF

DNAME	DNUMBER	DMGRENO	DLOCATIONS

DNAME	DNUMBER	DMGRENO	DLOCATIONS
MCA	1	22222	Delhi, Pune, Bangalore
CS	2	11111	Bangalore
IS	3	33333	Hyderabad

- The domain of DLOCATIONS contain sets of values and hence is nonatomic, it is not functionally dependent on DNUMBER.

- In this case DNUMBER →DLOCATIONS , because each set is considered a single member of the attribute domain.

- There are three main techniques to achieve first normal form :

1. Remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation **DEPT_LOCATIONS** along with the primary key DNUMBER OF DEPARTMENT.

- The primary key of relation **DEPT_LOCATIONS** is the combination {DNUMBER, DLOCATION}.

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT for each location of a DEPARTMENT.

- Then Primary Key becomes {DNUMBER, DLOCATION} and redundancy is introduced.

1NF version of same relation with redundancy

DNAME	DNUMBER	DMGRENO	DLOCATIONS
MCA	1	22222	Delhi
CS	2	11111	Bangalore
IS	3	33333	Hyderabad
MCA	1	22222	Pune
MCA	1	22222	Bangalore

- If a maximum number of values is known for the attributes:
- for example , if it is known that at most three locations can exist for a department
- replace the DLOCATIONS attribute by two atomic attributes :
  - DLOCATION1
  - DLOCATION2
- It leads to introduction of more NULL values

DNAME	<u>DNUMBER</u>	DMGRENO	DLOCATIONS1	DLOCATIONS2

- First is generally considered best.

- First normal form also disallows multivalued attributes that are themselves composite, which are called **nested relations** because each tuple can have a relation within it.

Normalizing nested relations into 1NF

SSN	ENAME	PROJS	
		PNUMBER	HOURS

Relation with nested relation

SSN	ENAME	PNUMBER	HOURS
11111	Ram	1 2	78 87
22222	Mina	3 2	45 54

1 NF

EMPLOEE

SSN	ENAME

Primary Key: SSN

EMP_PRJ

SSN	PNUMBER	HOURS

Primary Key: {SSN, PNUMBER}

Second Normal From

- **Second normal form (2NF)** is based on the concept of *full functional dependency* .
  - full functional dependency
  - partial dependency

- **Definition :** A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key.

2NF Normalization

- A functional dependency $X \rightarrow Y$ is a full functional dependency, if removal of any attribute A from X means dependency does not hold any more.

- For any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y.

- $(X - \{A\}) \rightarrow Y$ is partial dependency

- The test for 2NF involves testing for functional dependencies those left side attributes are part of the Primary Key.

- If Primary Key contains single attribute, the test need not be applied.

Third normal form(3NF)

- **Third normal form(3NF)** is based on the concept of *transitive dependency.*

- A functional dependency X→Y in a relation schema R is **transitive dependency** if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R and both X→Z and Z→Y hold.

- **Definition:** According to Codd's original definition , a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.

ENAME
SSN
ADDRESS
DNUMBER
DNAME
DMGRSSN
3NF Normalization
ENAME
SSN
ADDRESS
DNUMBER
DNUMBER
DNAME
DMGRSSN

GENERAL DEFINITIONS OF SECOND AND THIRD NORMAL FORMS

NORMAL FORM	TEST	REMEDY(NORMALIZATION)
First (1NF)	Relation should have no nonatomic attributes or nested relations	Form new relations for each nonatomic attribute of nested relation
Second (2NF)	For relations where primary key contains multiple attributes , no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute (s). make sure to keep the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute(or by a set of nonkey attributes .) that is , there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that include the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

General Definition of second Normal Form

- A relation schema R is in **second normal form(2NF)** if every nonprime attribute A in R is not partially dependent on any key of R.

General definition of Third Normal Form

- **Definition :**A relation schema R is in **third normal form(3NF)** if , whenever a nontrivial functional dependency X→A holds in R , either
  - X is a superkey of R , or
  - A is a prime attribute of R

PROPERTY_ID
COUNTY_NAME
LOT#
AREA
PRICE
TAX_RATE
FD1
FD2
FD3
FD4

PROPERTY_ID
COUNTY_NAME
LOT#
AREA
PRICE

COUNTY_NAME
TAX_RATE

PROPERTY_ID	COUNTY_NAME	LOT#	AREA	PRICE	TAX_RATE

PL/SQL

Overview
Data Types, Constant, Variables
Operators
Conditions & Loops
Cursors & Records
Arrays
Procedures & Functions
Triggers
Exception Handling
Packages

PL/SQL-Overview

What is PL/SQL?

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

PL/SQL is one of three key programming languages embedded in the Oracle Database,along with SQL itself and Java.

PL/SQL-Overview

PL/SQL stands for Procedural Language Extension to SQL.

PL/SQL extends SQL by adding programming structures and subroutines available in any high-level language.

PL/SQL is used for both server-side and client-side development.

Features

PL/SQL is language. A completely portable, high-performance Transaction-processing.

PL/SQL provides a built-in, interpreted and OS independent programming environment.

PL/SQL can also directly be called from the command-line SQL*Plus interface. Direct call can also be made from external programming language calls to database.

PL/SQL's general syntax is based on that of ADA and Pascal programming language.

Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

PL/SQL-Overview

PL/SQL is there any Advantage?

Each SELECT statement is a request against the database and is sent to the Oracle server

.

The results of each SELECT statement are sent back to the client.

Each time a SELECT statement is executed, network traffic is generated.

Hence, multiple SELECT statements result in multiple round-trip transmissions, adding significantly to the network traffic.

When these SELECT statements are combined into a PL/SQL program, they are sent to the server as a single unit.

The SELECT statements in this PL/SQL program are executed at the server.

The server sends the results of these SELECT statements back to the client, also as a single unit.

Therefore, a PL/SQL program encompassing multiple SELECT statements can be executed at the server and have the results returned to the client in one round trip.

This is a more efficient process than having each SELECT statement executed independently.

Application
SQL
SQL
SQL
SQL
Other DBMSs
Application
SQL
IF ... THEN
SQL
ELSE
SQL
END IF;
SQL
Oracle Database with PL/SQL
Application
RPC
Oracle Database with PL/SQL and Stored Procedures

There are a number of places where we can write code that our applications can use:

- As part of applications
- PL/SQL code embedded in views
- Batch routines

As part of applications

- PL/SQL program units can return a set of values (functions), or PL/SQL routines can perform database operations (procedures).
- These functions and procedures may be called by other functions and procedures.

PL/SQL code embedded in views

- Oracle allows us to embed code in database views.
- We can also embed PL/SQL in INSTEAD OF triggers on a view and allow us to perform INSERT, UPDATE, and DELETE operations on complex views, with PL/SQL programmatically handling how these operations should be handled.

Batch routines

- Batch routines run code that processes a large number of records at the same time.
- These routines are usually large, complex, and database intensive. This type of routine should assuredly be written in PL/SQL.
- Examples
  - Generating invoices for every customer in a system
  - Processing payroll for an entire organization

Features of PL/SQL

It is tightly integrated with SQL. It provides access to predefined SQL packages.
It provides support for developing Web Applications and Server Pages.

It offers extensive error checking.
It offers numerous data types.
It offers a variety of programming structures.
It supports structured programming through functions and procedures.
It supports object-oriented programming.
PL/SQL supports both static and dynamic SQL.
Static SQL supports DML operations and transaction control from PL/SQL block.
Dynamic SQL is SQL allows embedding DDL statements in PL/SQL blocks

How to Start with PL/SQL Programming???

Save, Edit and Execute PL/SQL program

Type your program in SQL *plus

To save : **save <FileName>**

Program is saved in the home folder, to save in other folder give path.

save \test\firstprg.sql

To make changes: **edit <FileName>**

To edit program saved in folder other then bin

edit \test\firstprg.sql

To Execute: @ **<FileName>**

To execute program saved in folder other then bin.

\test\firstprg.sql

PL/SQL block structure

PL/SQL blocks can be divided into two groups:

- Anonymous
- Named

Named PL/SQL blocks are used when creating subroutines like procedures, functions, and packages. The subroutines then can be stored in the database and referenced by their names later.

In addition, subroutines such as procedures and functions can be defined within the anonymous PL/SQL block.

Block Structure

Declarations

This section starts with the keyword DECLARE.

It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

Executable Commands

This section is enclosed between the keywords BEGIN and END and it is a mandatory section.

It consists of the executable PL/SQL statements of the program.

It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.

Exception Handling

This section starts with the keyword EXCEPTION.

This section is also optional and contains exceptions that handle errors in the program.

Declare - example

```
declare
    v_sal_nr NUMBER;
    v_name_tx VARCHAR2(10) DEFAULT ‘KING’;
    v_height_nr NUMBER := 4;
    v_start_dt DATE := SYSDATE; -- same as DEFAULT
begin
...
```

Declare - example

```
declare
    variable_name table.column%TYPE;
    variable_name2 variable_name%TYPE;
    variable_row table%ROWTYPE;
begin
```

To see the output on sqlplus execute following commands

```
set serveroutput on
set serveroutput on size 5000;
```

- Example-1
  - begin
    - dbms_ouput.put_line(“hello”);
    - end
    -
    - Example 2
      - Declare
      - Message varchar2(20):=”Hello, World”;
      - Begin
      - dbms_output.put_line(message);
      - End;

PL/SQL code blocks are comprised of statements.

Each statement ends with a semi-colon.

PL/SQL code blocks are followed by a slash (/) in the first position of the following line. This causes the code block statements to be executed.

The only PL/SQL code block keyword that is followed by a semi-colon is the End keyword.

Fundamentals of PL/SQL

- Full-featured programming language
- An interpreted language
- Type in editor, execute in SQL*Plus

Item Type	Capitalization	Example
Reserved word	Uppercase	BEGIN, DECLARE
Built-in function	Uppercase	COUNT, TO_DATE
Predefined data type	Uppercase	VARCHAR2, NUMBER
SQL command	Uppercase	SELECT, INSERT
Database object	Lowercase	student, f_id
Variable name	Lowercase	current_s_id, current_f_last

Table 4-1 PL/SQL command capitalization styles

Variables and Data Types

- Variables
  - Used to store numbers, character strings, dates, and other data values
  - Avoid using keywords, table names and column names as variable names
  - Must be declared with data type before use: *variable_name data_type_declaration*;

Scalar Data Types

- Represent a single value

Data Type	Description	Sample Declaration
VARCHAR2	Variable-length character string	`current_s_last VARCHAR2(30);`
CHAR	Fixed-length character string	`student_gender CHAR(1);`
DATE	Date and time	`todays_date DATE;`
INTERVAL	Time interval	`curr_time_enrolled INTERVAL YEAR TO MONTH;` `curr_elapsed_time INTERVAL DAY TO SECOND;`
NUMBER	Floating-point, fixed-point, or integer number	`current_price NUMBER(5,2);`

Table 4-2 Scalar database data types

Scalar Data Types

Data Type	Description	Sample Declaration
Integer number subtypes (BINARY_INTEGER, INTEGER, INT, SMALLINT)	Integer	`counter BINARY_INTEGER;`
Decimal number subtypes (DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL)	Numeric value with varying precision and scale	`student_gpa REAL;`
BOOLEAN	True/False value	`order_flag BOOLEAN;`

Table 4-3 General scalar data types

Composite and Reference Variables

- Composite variables
  - RECORD: contains multiple scalar values, similar to a table record
  - TABLE: tabular structure with multiple columns and rows
  - VARRAY: variable-sized array
- Reference variables
  - Directly reference a specific database field or record and assume the data type of the associated field or record
  - %TYPE: same data type as a database field
  - %ROWTYPE: same data type as a database record

PL/SQL Program Blocks

Figure 4-1 Structure of a PL/SQL program block

- Comments:
  - Not executed by interpreter
  - Enclosed between /* and */
  - On one line beginning with --

Arithmetic Operators

Operator	Description	Example	Result
**	Exponentiation	2 ** 3	8
* /	Multiplication Division	2 * 3 9/2	6 4.5
+ -	Addition Subtraction	3 + 2 3 - 2	5 1
-	Negation	-5	-5

Table 4-5 PL/SQL arithmetic operators in describing order of precedence

Assignment Statements

- Assigns a value to a variable
- *variable_name* := *value*;
- Value can be a literal:
  - current_s_first_name := 'John';
- Value can be another variable:
  - current_s_first_name := s_first_name;

Executing a PL/SQL Program in SQL*Plus

```
--PL/SQL program to display the current date
DECLARE
   todays_date DATE;
BEGIN
   todays_date := SYSDATE;
   DBMS_OUTPUT.PUT_LINE('Today''s date is ');
   DBMS_OUTPUT.PUT_LINE(todays_date);
END;
```

Figure 4-2 PL/SQL program commands

- Create program in text editor
- Paste into SQL*Plus window
- Press Enter, type / then enter to execute

PL/SQL Data Conversion Functions

Data Conversion Function	Description	Example
TO_CHAR	Converts either a number or a date value to a string using a specific format model	`TO_CHAR(2.98, '$999.99');` `TO_CHAR(SYSDATE, 'MM/DD/YYYY');`
TO_DATE	Converts a string to a date using a specific format model	`TO_DATE('07/14/2003', 'MM/DD/YYYY');`
TO_NUMBER	Converts a string to a number	`TO_NUMBER('2');`

Table 4-6 PL/SQL data conversion functions

Manipulating Character Strings with PL/SQL

- To concatenate two strings in PL/SQL, you use the double bar (||) operator:
  - *new_string* := *string1* || *string2*;
- To remove blank leading spaces use the LTRIM function:
  - *string := LTRIM(string_variable_name);*
- To remove blank trailing spaces use the RTRIM function:
  - *string := RTRIM(string_variable_name);*
- To find the number of characters in a character string use the LENGTH function:
  - *string_length := LENGTH(string_variable_name);*

Manipulating Character Strings with PL/SQL

- To change case, use UPPER, LOWER, INITCAP
- INSTR function searches a string for a specific substring:
  - *start_position := INSTR(original_string, substring);*
- SUBSTR function extracts a specific number of characters from a character string, starting at a given point:
  - *extracted_string := SUBSTR(string_variable, starting_point, number_of_characters);*

Debugging PL/SQL Programs

- Syntax error:
  - Command does not follow the guidelines of the programming language
  - Generates compiler or interpreter error messages
- Logic error:
  - Program runs but results in an incorrect result
  - Caused by mistake in program

Finding and Fixing Syntax Errors

- Interpreter flags the line number and character location of syntax errors
- If error message appears and the flagged line appears correct, the error usually occurs on program lines *preceding* the flagged line
- Comment out program lines to look for hidden errors
- One error (such as missing semicolon) may cause more – fix one error at a time

Finding and Fixing Logic Errors

- Locate logic errors by viewing variable values during program execution
- There is no SQL*Plus debugger
- Use DBMS_OUTPUT statements to print variable values

Lesson B

- Create PL/SQL decision control structures
- Use SQL queries in PL/SQL programs
- Create loops in PL/SQL programs
- Create PL/SQL tables and tables of records
- Use cursors to retrieve database data into PL/SQL programs
- Use the exception section to handle errors in PL/SQL programs

PL/SQL Decision Control Structures

- Use IF/THEN structure to execute code if condition is true
  - IF *condition* THEN

    commands that execute if condition is TRUE;

    END IF;
- If condition evaluates to NULL it is considered false
- Use IF/THEN/ELSE to execute code if condition is true or false
  - IF *condition* THEN

    commands that execute if condition is TRUE;

    ELSE

    commands that execute if condition is FALSE;

    END IF;
- Can be nested – be sure to end nested statements

PL/SQL Decision Control Structures

- Use IF/ELSIF to evaluate many conditions:
  - IF *condition1* THEN

    commands that execute if condition1 is TRUE;

    ELSIF *condition2* THEN

    commands that execute if condition2 is TRUE;

    ELSIF *condition3* THEN

    commands that execute if condition3 is TRUE;

    ...

    ELSE

    commands that execute if none of the

    conditions are TRUE;

    END IF;

IF/ELSIF Example

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2     todays_date DATE;
  3     current_day VARCHAR2(9);
  4  BEGIN
  5     todays_date := SYSDATE;
  6     -- extract day portion from current date, and trim trailing blank spaces
  7     current_day := TO_CHAR(todays_date, 'DAY');
  8     current_day := INITCAP(current_day);
  9     current_day := RTRIM(current_day);
 10     -- IF/ELSIF condition to determine current day
 11     IF current_day = 'Friday' THEN
 12       DBMS_OUTPUT.PUT_LINE('Today is Friday!');
 13     ELSIF current_day = 'Saturday' THEN
 14       DBMS_OUTPUT.PUT_LINE('Today is Saturday!');
 15     ELSIF current_day = 'Sunday' THEN
 16       DBMS_OUTPUT.PUT_LINE('Today is Sunday!');
 17     ELSIF current_day = 'Monday' THEN
 18       DBMS_OUTPUT.PUT_LINE('Today is Monday!');
 19     ELSIF current_day = 'Tuesday' THEN
 20       DBMS_OUTPUT.PUT_LINE('Today is Tuesday!');
 21     ELSIF current_day = 'Wednesday' THEN
 22       DBMS_OUTPUT.PUT_LINE('Today is Wednesday!');
 23     ELSIF current_day = 'Thursday' THEN
 24       DBMS_OUTPUT.PUT_LINE('Today is Thursday!');
 25     ELSE
 26       DBMS_OUTPUT.PUT_LINE('Current day not found.');
 27     END IF;
 28  END;
 29  /
Today is Tuesday!

PL/SQL procedure successfully completed.
```

Add/modify these commands

Figure 4-17 Using an IF/ELSIF structure

Complex Conditions

- Created with logical operators AND, OR and NOT
- AND is evaluated before OR
- Use () to set precedence

Figure 4-19 Evaluating AND and OR in an expression

Using SQL Queries in PL/SQL Programs

- Action queries can be used as in SQL*Plus
- May use variables in action queries
- DDL commands may not be used in PL/SQL

Loops

- Program structure that executes a series of program statements, and periodically evaluates an exit condition to determine if the loop should repeat or exit
- Pretest loop: evaluates the exit condition before any program commands execute
- Posttest loop: executes one or more program commands before the loop evaluates the exit condition for the first time
- PL/SQL has 5 loop structures

The LOOP...EXIT Loop

```
LOOP
  [program statements]
  IF condition THEN
    EXIT;
  END IF;
  [additional program statements]
END LOOP
```

The LOOP...EXIT WHEN Loop

```
LOOP
    program statements
    EXIT WHEN condition;
END LOOP;
```

The WHILE...LOOP

WHILE *condition* LOOP

program statements

END LOOP;

The Numeric FOR Loop

FOR *counter_variable* IN *start_value* .. *end_value*
LOOP
    program statements
END LOOP;

Stored Procedures and Functions Procedures

The procedures and functions that were part of an anonymous block and were called within the executable section.

It is also possible to store the procedure or function definition in the database and have it invoked from various environments that have access to the database.

Stored Procedures and Functions Procedures

A Procedure or function is a schema object that logically groups a set of SQL and other PL/SQL programming language statements together to perform a specific task.

Procedures and Functions are created in user’s schema and stored in a database for continued use.

The general syntax for creating procedure is

```
Create [or replace] procedure <proc-name>
[(<parameter-list>)] as
  <declarations>
  Begin
  <executable-section>
[exception
   <exception-section>]
End;
```

Create a table called employee having following columns

EMPCODE	NUMBER(5)
EMPNAME	VARCHAR2(20)
DOB	DATE
DEPT	VARCHAR2(20)
SALARY	NUMBER(6)

Insert 5 or 6 records in the table.

Declare a procedure to increase the salary of an employee by 2000 by specifying the employee code.

```
create or replace procedure sal_hike  (eno in
employee1.empcode%type, increase in employee1.salary%type)
as  sal employee1.salary%type;
new_sal employee1.salary%type;
begin
select salary into sal from employee1 where empcode=eno;
new_sal:=sal+increase;
update employee1 set salary=new_sal where empcode=eno;
dbms_output.put_line('Table updated with new salary' ||new_sal);
end;
/
```

The PL/SQL code can be stored in a file with extension .sql and can be executed by the command

@ filename.sql

exec proc-name(parameters);

Functions

The general syntax for creating function is

```
Create [or replace] function <func-name>
[(<parameter-list>)] return <datatype> as
<declarations>
Begin
   <executable-section>
[exception
    <exception-section>
End;
```

When stored function is invoked from within an SQL statement there are three restrictions:
There should not be no ‘out’ parameters.
The function should be applicable to a row in the table.
Return data type should be compatible with an SQL data type.

Create a table called STUDENT having the following columns

REGNO NOT NULL	VARCHAR2(10)
NAME	VARCHAR2(20)
MAJOR	VARCHAR2(20)
BDATE	DATE

Insert 5 or 6 records in the table.

Write a function to get the major subject offered by a student with a specific register number

```
create or replace function get_major(rno in
student1.regno%type)
return student1.major%type as smajor
student1.major%type;
begin
select major into smajor from student1 where
student1.regno=rno;
  return(smajor);
end;
/
```

The function should be written in a .sql file and will be executed as @filename.sql

Records – PL/SQL

Records

A record is a composite data structure composed of one or more elements.

Records are very much like a row of a database table, but each element of the record does not stand on its own.

PL/SQL supports three kinds of records:

table-based

cursor-based

programmer-defined

Table based Records

The %ROWTYPE attribute enables a programmer to create table-based and cursor-based records

```
DECLARE
        emp_rec employee1%rowtype;
BEGIN
        SELECT * into emp_rec FROM employee1 WHERE
empcode = 111;
        dbms_output.put_line('Employee - code ' ||
emp_rec.empcode);
        dbms_output.put_line('Employee - name ' ||
emp_rec.empname);

        dbms_output.put_line('Employee - DOB ' ||
emp_rec.dob);
        dbms_output.put_line('Employee - salary ' ||
emp_rec.salary);
END;
/
```

User-Defined Records

```
DECLARE
        type books is record (title varchar(50),author varchar(50),book_id
number);
        book1 books;
        book2 books;
BEGIN
        book1.title := 'DBMS'; book1.author := 'ABCD’; book1.book_id :=
123;
        book2.title := 'NoSQL'; book2.author := 'PAL'; book2.book_id :=
456;
--Print book 1 record
        dbms_output.put_line('Book 1 title : '|| book1.title);
        dbms_output.put_line('Book 1 author : '|| book1.author);
        dbms_output.put_line('Book book_id : ' || book1.book_id);
-- Print book 2record
        dbms_output.put_line('Book 2 title : '|| book2.title);
        dbms_output.put_line('Book 2 author : '|| book2.author);
        dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

Cursors

- Pointer to a memory location that the DBMS uses to process a SQL query
- Use to retrieve and manipulate database data

Implicit Cursor

Figure 4-26 Implicit cursor

Using an Implicit Cursor

- Executing a SELECT query creates an implicit cursor
- To retrieve it into a variable use INTO:
  - SELECT *field1*, *field2*, ...

  INTO *variable1*, *variable2*, ...

  FROM *table1*, *table2*, ...

  WHERE *join_ conditions*

  AND *search_condition_to_retrieve_1_record*;
- Can only be used with queries that return exactly one record

Implicit cursors are created in PL/SQL by any DML statement that do not have explicit cursor defined within the statement.

During the processing of an implicit cursor, Oracle automatically performs the Open, Fetch and Close operations for the cursor.

Implicit Cursors are created for the following statements of SQL:

Select…into, Insert, Update and Delete.

Example: Implicit cursor (Update Command)

```
Declare
total_rows number(2);
    Begin
      update employee1 set salary=50000+5000 where empcode=111;
    If sql%isopen then
        dbms_output.put_line('cursor is open');
    else
        dbms_output.put_line('cursor is not open');
    End if;
    If sql%notfound then
        dbms_output.put_line('No Employees selected');
    Elsif sql%found then
         total_rows:=sql%rowcount;
        dbms_output.put_line('total_rows || 'Employees Selected');
    End if;
    End;
    /
```

Implicit cursors uses the following constructs to check whether the SQL query/statement has been found or executed.

1. **%found:** returns true if an insert, update or delete statement affected one or more rows or a select..into statement returned one or more rows

2. **%notfound:** The logical opposite of %found.

3. **%isopen:** always returns false for implicit cursors, because oracle closes the SQL cursor automatically after executing its associated SQL statement.

4**.%Rowcount:** returns the number of rows affected by an Insert, update, or delete statement or returned by Select..into statement.

Explicit Cursor

- Use for queries that return multiple records or no records
- Must be explicitly declared and used
- An explicit cursor can be generated in the Declare section of the PL/SQL block.
- The advantage of explicit cursor over implicit cursor is, explicit cursor gives more programmatic control to programmer.
- Implicit cursors are less efficient than explicit cursors.

Using an Explicit Cursor

- Declare the cursor
  - *CURSOR cursor_name IS select_query;*
- Open the cursor
  - *OPEN cursor_name;*
- Fetch the data rows
  - *LOOP*

    FETCH cursor_name INTO variable_name(s);

    EXIT WHEN cursor_name%NOTFOUND;
- Close the cursor
  - *CLOSE cursor_name;*

Explicit Cursor – Fetching the rows

The rows from a table can be fetched into explicit cursor as

fetch c_customers into c_id, c_name, c_addr;

where c_customers is table name

c_id, c_name, c_addr are variables created in PL/SQL, like column types of c_customer table.

The declaration of cursor allows the application to sequentially process each row of data as the cursor returns it.

An explicit cursor can be generated in the Declare section of the PL/SQL block.

Explicit Cursor with %ROWTYPE

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2    current_bldg_code VARCHAR2(5);
  3    CURSOR location_cursor IS
  4      SELECT room, capacity
  5      FROM location
  6      WHERE bldg_code = current_bldg_code;
  7    location_row location_cursor%ROWTYPE;
  8  BEGIN
  9    current_bldg_code := 'LIB';
 10    OPEN location_cursor;
 11    LOOP
 12      FETCH location_cursor INTO location_row;
 13      EXIT WHEN location_cursor%NOTFOUND;
 14      DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 15      location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 16    END LOOP;
 17    CLOSE location_cursor;
 18  END;
 19  /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

Modify these commands

Program output

Figure 4-31 Processing an explicit cursor using a %ROWTYPE variable

Cursor FOR Loop

- Automatically opens the cursor, fetches the records, then closes the cursor
- *FOR variable_name(s) IN cursor_name LOOP*

  processing commands

  END LOOP;
- Cursor variables cannot be used outside loop

Using Cursor FOR Loop

Oracle SQL*Plus

File Edit Search Options Help

```
SQL> DECLARE
  2     current_bldg_code VARCHAR2(5);
  3     CURSOR location_cursor IS
  4       SELECT room, capacity
  5       FROM location
  6       WHERE bldg_code = current_bldg_code;
  7     location_row location_cursor%ROWTYPE;
  8   BEGIN
  9     current_bldg_code := 'LIB';
 10     FOR location_row IN location_cursor LOOP
 11       DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 12       location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 13     END LOOP;
 14   END;
 15   /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

Add/modify these commands

Figure 4-32 Processing an explicit cursor using a cursor FOR loop

Handling Runtime Errors in PL/SQL Programs

- Runtime errors cause exceptions
- Exception handlers exist to deal with different error situations
- Exceptions cause program control to fall to exception section where exception is handled

```
EXCEPTION
  WHEN exception1_name THEN
     exception1 handler commands;
  WHEN exception2_name THEN
     exception2 handler commands;
  …
  WHEN OTHERS THEN
     other handler commands;
END;
```

Figure 4-34 Exception handler syntax

Exception Handling

An error condition during a program execution is called an exception in PL/SQL.

PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling goes here >
WHEN exception1 THEN
exception1-handling-statements
WHEN exception2 THEN
exception2-handling-statements
........
WHEN others THEN
exception3-handling-statements
END;
/
```

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax for raising an exception:

```
DECLARE
exception_name EXCEPTION;
BEGIN
IF condition THEN
RAISE exception_name;
END IF;
EXCEPTION
WHEN exception_name THEN
statement;
END;
/
```

Predefined Exceptions

Oracle Error Code	Exception Name	Description
ORA-00001	DUP_VAL_ON_INDEX	Command violates primary key unique constraint
ORA-01403	NO_DATA_FOUND	Query retrieves no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion (such as trying to convert "2B" to a number)
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or data conversion operation

Table 4-10 Common PL/SQL predefined exceptions

Undefined Exceptions

- Less common errors
- Do not have predefined names
- Must declare your own name for the exception code in the declaration section
  - *DECLARE*

    e_exception_name EXCEPTION;

    PRAGMA EXCEPTION_INIT(e_exception_name,

    -Oracle_error_code);

User-Defined Exceptions

- Not a real Oracle error
- Use to enforce business rules

```
DECLARE
   e_exception_name EXCEPTION;
   other variable declarations;
BEGIN
   other program commands
   IF exception_condition THEN
      RAISE e_exception_name;
   END IF;
   other program commands
EXCEPTION
   WHEN e_exception_name THEN
      exception handler commands;
END;
```

Defining the exception

Raising the exception

Handling the exception

Figure 4-40 General syntax for declaring, raising, and handling a user-defined exception

Summary

- PL/SQL is a programming language for working with an Oracle database
- Scalar, composite and reference variables can be used
- The IF/THEN/ELSE decision control structure allows branching logic
- Five loop constructs allow repeating code
- Cursors are returned from queries and can be explicitly iterated over
- Exception handling is performed in the exception section. User defined exceptions help to enforce business logic

Intuition

- ``*Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search on disk can be high.
- 2 Simple ideas:
  - Create an `index' file.
  - Use a large fanout F since each dereference is $$$$$.

✉ *Can do binary search on (smaller) index file!*

ISAM

- Index file may still be quite large. But we can apply the idea repeatedly!

✉ *Leaf pages contain data entries.*

Review of Indexes

- *As for any index, 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries **k***.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Comments on ISAM

File creation: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.

- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
  - Perhaps in an overflow page
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.
  - Thus data pages remain sequential/continguous

Static tree structure: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)

After Inserting 23, 48*, 41*, 42* …*

… *Then Deleting 42*, 51*, 97**

✉ *Note that 51* appears in index levels, but not in leaf!*

Pros and Cons of ISAM

- Cons
  - After many inserts and deletes, long overflow chains can develop
  - Overflow records may not be sorted
- Pros
  - Inserts and deletes are fast since there's no need to balance the tree
  - No need to lock nodes of the original tree for concurrent access
  - If the tree has had few updates, then interval queries are fast.

B+ Trees

Table of Contents

- History of B+ Trees
- Definition of B+ Trees
- Searching in B+ Trees
- Inserting into B+ Trees
- Deleting from B+ Trees
- Works Cited

History of B+ Trees

- First described in paper by Rudolf Bayer and Edward M. McCreight in 1972
- "Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189 (1972)“ -- Wikipedia

Description of B+ Trees

- A variation of B-Trees
- B-Trees commonly found in databases and file systems
- Sacrifices space for efficiency (not as much re-balancing required compared to other balanced trees)

Description of B+ Trees (cont.)

- Main difference of B-Trees and B+ Trees
  - B-Trees: data stored at every level (in every node)
  - B+ Trees: data stored only in leaves
    - Internal nodes only contain keys and pointers
    - All leaves are at the same level (the lowest one)

Description of B+ Trees (cont.)

- B+ trees have an order n
- An internal node can have up to n-1 keys and n pointers
- Built from the bottom up

Description of B+ Trees (cont.)

- All nodes must have between ceil(n/2) and n keys (except for the root)

- For a B+ tree of order n and height h, it can hold up to n^h keys

Searching in B+ Trees

- Searching just like in a binary search tree
- Starts at the root, works down to the leaf level
- Does a comparison of the search value and the current “separation value”, goes left or right

Inserting into a B+ Tree

- A search is first performed, using the value to be added
- After the search is completed, the location for the new value is known

Inserting into a B+ Tree (cont.)

- If the tree is empty, add to the root
- Once the root is full, split the data into 2 leaves, using the root to hold keys and pointers
- If adding an element will overload a leaf, take the median and split it

Inserting into a B+ Tree (cont.)

- Example:
  - Suppose we had a B+ tree with n = 3
    - 2 keys max. at each internal node
    - 3 pointers max. at each internal node
    - Internal nodes include the root

Inserting Into B+ Trees (cont.)

- Case 1: Empty root
- Insert 6

Inserting into B+ Trees (cont.)

- Case 2: Full root
- Suppose we have this root:

- In order to insert another number, like 5, we must split the root and create a new level.

Inserting into B+ Trees (cont.)

- After splitting the root, we would end up with this:

Inserting into B+ Trees (cont.)

- Case 3: Adding to a full node
- Suppose we wanted to insert 7 into our tree:

Inserting into B+ Trees (cont.)

- 7 goes with 5 and 6. However, since each node can only hold a maximum of 2 keys, we can take the median of all 3 (which would be 6), keep it with the left (5), and create a new leaf for 7.

- An alternative way is to keep the median with the right and create a new leaf for 5.

Inserting into B+ Trees (cont.)

Inserting into B+ Trees (cont.)

- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
- Using the last tree, suppose we were to insert 4.

Inserting into B+ Trees (cont.)

- We would need to split the leftmost leaf, which would require another split of the root.
- A new root is created with the pointers referencing the old split root.

Inserting into B+ Trees (cont.)

Deleting from B+ Trees

- Deletion, like insertion, begins with a search.
- When the item to be deleted is located and removed, the tree must be checked to make sure no rules are violated.
- The rule to focus on is to ensure that each node has at least ceil(n/2) pointers.

Deleting from B+ Trees (cont.)

- Take the previous example:

Deleting from B+ Trees (cont.)

- Suppose we want to delete 5.

- This would not require any rebalancing since the leaf that 5 was in still has 1 element in it.

Deleting from B+ Trees (cont.)

Deleting from B+ Trees (cont.)

- Suppose we want to remove 6.
- This would require rebalancing, since removing the element 6 would require removal of the entire leaf (since 6 is the only element in that leaf).

Deleting from B+ Trees (cont.)

- Once we remove the leaf node, the parent of that leaf node no longer follows the rule.
- It has only 1 child, which is less than the 2 required (ceil(3/2) = 2).
- Then, the tree must be compacted in order to enforce this rule.

Deleting from B+ Trees (cont.)

- The end product would look something like this:

Adjusting the data in leaf nodes of B+ - Trees

Rotation

B+ trees can incorporate rotation to reduce the number of page splits.

A rotation occurs when a leaf page is full, but one of its sibling pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting the indices as necessary. Typically, the left sibling is checked first (if it exists) and then the right sibling.

Example :

As an example, consider the B+ tree before the addition of the record containing a key of 70.This record belongs in the leaf node containing 50 55 60 65. Notice that this node is full, but its left sibling is not.

25
50
75
5
10
15
20
25
28
30
50
55
60
65
75
80
85
90

25
55
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90

Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires ≥ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node
-
-

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

(a)

P_1	B_1	K_1	P_2	B_2	K_2	$\cdots$	P_{m-1}	B_{m-1}	K_{m-1}	P_m

(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.
-

B-Tree Index File Example

B-tree (above) and B+-tree (below) on same data

B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages.

Hash-Based Indexes

Introduction

- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.
- A good hash function is:
  - Random
  - Uniform
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)

- The number of buckets provided remains unchanged at all times.

Bucket Overflow

- The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.
- **Overflow Chaining** − When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.

Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r.* Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

Example

- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

- **<u>Insert</u>**: If bucket is full, *<u>split</u>* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert **h**(r)=20 (Causes Doubling)

Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **<u>3</u>** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where *di* = *d0* + *i*.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry r, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', r belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Overview of LH File

- In the middle of a round.

Linear Hashing (Contd.)

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- Can choose any criterion to `trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries.

Example: End of a Round

LH Described as a Variant of EH

- The two schemes are actually quite similar:
  - Begin with an EH index where directory has *N* elements.
  - Use overflow pages, split buckets round-robin.
  - First split is at bucket 0. (Imagine directory being doubled at this point.) But elements <1,*N*+1>, <2,*N*+2>, ... are the same. So, need only create directory element *N*, which differs from 0, now.
    - When bucket 1 splits, create directory element *N*+1, etc.
- So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding i'th is easy), we actually don't need a directory! Voila, LH.

Example of Dynamic Hashing

branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Insert

- Brighton
- Downtown
- Downtown
- Mianus
- Perryridge
- Perryridge
- Perryridge
- Round Hill

Example of Dynamic Hashing

Example of Dynamic Hashing

Example of Dynamic Hashing

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!