BACHELOR OF SCIENCE IN COMPUTER SCIENCE

YEAR 4 SEMESTER 2

CCS 418: ADVANCED DATABASE SYSTEMS

NOTES 1

Object-Oriented Databases

Need for Complex *Data Types*

- ***Traditional*** database ***applications*** in data processing had conceptually ***simple data types***
    - Relatively ***few data types***,
        - ***first normal form*** holds
- ***Complex data types*** have grown more important in ***recent years***
    - E.g. *Addresses* can be viewed as a
        - Single string, or
        - *Separate attributes* for each part, or
        - *Composite attributes* (which are not in first normal form)
    - E.g. it is often ***convenient*** to store ***multivalued attributes as-is***,
        - ***without*** creating a ***separate relation*** to store the values in 1FN
- ***Applications:***
    - computer-aided design, computer-aided software engineering
    - multimedia and image databases, and document/hypertext databases.

Object-Oriented *Data Model*

- *Loosely speaking*, an **object** *corresponds to* an ***entity*** in the ***E-R model***.
- The ***object-oriented paradigm*** is based on ***encapsulating code*** and ***data*** related to an object *into* ***single unit***.
- The ***object-oriented data model*** is a ***logical data model*** (*like* the ***E-R model***).
- ***Adaptation*** of the ***object-oriented programming paradigm***:
    - (e.g., Smalltalk, C++, Java)
    - to ***database systems***.

Object Structure

- An object has associated with it:
    - A set of **variables** that contain the ***data*** for the object. The value of each variable is itself an object.
    - A set of **messages** to which the object responds; each message may have zero, one, or more *parameters.*

- A set of **methods**, each of which is a ***body of code*** to implement a message; a method returns a value as the *response* to the message

- The physical representation of data is visible only to the implementer of the object
- ***Messages*** and responses *provide* the only *external interface* to an object.
- The term message does *not necessarily* imply physical *message passing*.
  - ***Messages*** can be *implemented* as ***procedure invocations***.

Messages and Methods

- **Methods** are programs written in ***general-purpose*** *language* with the following features:
  - only variables in the object itself may be referenced directly
  - **data** in other objects are ***referenced*** only by ***sending messages***.
- ***Methods*** can be ***read-only*** or ***update*** *methods*
  - Read-only methods do not change the value of the object
- *Strictly speaking*, ***every*** attribute of an entity must be represented by:
  - a variable and
  - two methods, one to read and the other to update the attribute
  - e.g., the attribute *address* is represented by a variable *address* and two ***messages get-address*** and ***set-address***.
  - For convenience, many object-oriented data models permit direct access to variables of other objects.

Object Classes

- ***Similar objects*** are grouped into:
  - a ***class***; each such object is called
  - an ***instance*** of its class
- ***All objects*** in a ***class*** have the ***same*:**
  - Variables, with the same types
  - message interface
  - methods

They may differ in the values assigned to variables

- *Example*: Group objects for ***people*** into a ***person*** *class*
- *Classes* are *analogous to* ***entity sets*** in the **E-R** model

Class Definition Example

class *employee* {

/*Variables */

string *name;*

string *address;*

date *start-date;*

int *salary;*

/* Messages */

int *annual-salary*();

string *get-name*();

string *get-address*();

int *set-address*(**string** *new-address);*

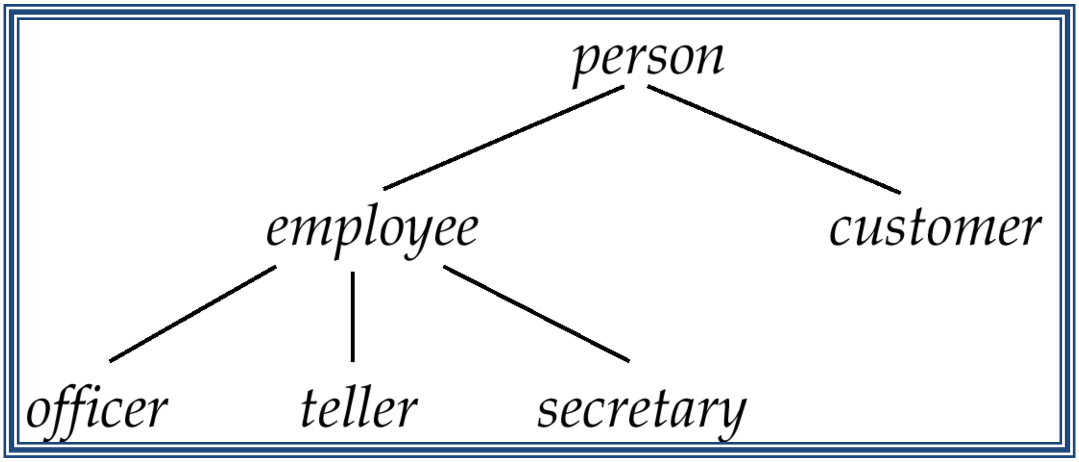
int *employment-length*();

};

- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
  - E.g. **int** *employment-length*()
    { **return** *today*() – *start-date*;}
  - **int** *set-address*(**string** *new-address*)
    { *address* = *new-address*;}

Inheritance

- E.g., class of bank customers is similar to class of bank employees, although there are differences
  - both share some variables and messages, e.g., *name* and *address.*
  - But there are variables and messages specific to each class e.g., *salary* for employees and *credit-rating* for customers.
- Every employee is a person; thus *employee* is a specialization of *person*
- *Similarly, customer* is a specialization of *person.*
- Create classes *person, employee* and *customer*
  - variables/messages applicable to all persons associated with class *person.*
  - variables/messages specific to employees associated with class *employee*; similarly for *customer*
- Place classes into a specialization/IS-A hierarchy
  - variables/messages belonging to class *person* are *inherited* by class *employee* as well as *customer*
- Result is a **class hierarchy**



Class Hierarchy Definition

class ***person***{

string *name;*

string *address:*

};

class *customer* **isa** ***person*** {

int *credit-rating;*

};

class *employee* **isa** ***person*** {

date *start-date;*

int *salary;*

};

class *officer* **isa** *employee* {

int *office-number,*

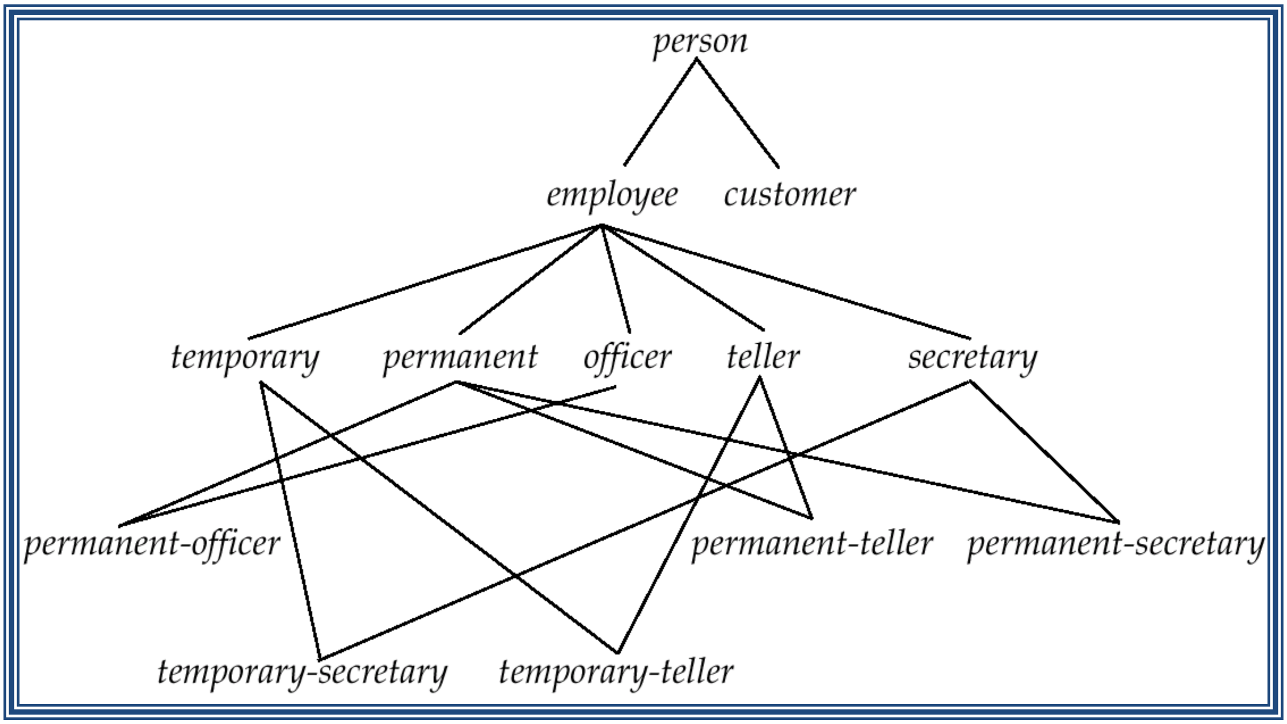
int *expense-account-number,*

};

- Full variable list for objects in the class *officer:*
  - *office-number, expense-account-number:* defined locally
  - *start-date, salary:* inherited from *employee*
  - *name, address:* inherited from *person*
- Methods inherited similar to variables.
- **Substitutability** — *any* ***method*** of a class, say *person*, ***can*** *be* ***invoked*** equally well ***with*** *any object* belonging to *any* ***subclass***, such as subclass *officer* of *person.*
- **Class extent***:* set of **all** ***objects*** in the *class*. *Two* options:

1. *Class extent* of *employee* includes ***all*** *officer*, *teller* and *secretary* objects.
2. *Class extent* of *employee* includes ***only*** employee *objects* that are ***not*** *in a subclass* such as *officer*, *teller*, or *secretary*
    - This is the ***usual choice*** in OO systems
    - Can access extents of subclasses to find all objects of subtypes of employee

Example of Multiple Inheritance



Class DAG for banking example.

Multiple **Inheritance**

- With multiple inheritance a class may **have** *more than one **superclass***.
    - The class/subclass relationship is represented by a **directed *acyclic* graph** (**DAG**)
    - Particularly useful when objects can be classified in more than one way, which are independent of each other
    - E.g. temporary/permanent is independent of Officer/secretary/teller
    - Create a subclass for each combination of subclasses
        - Need not create subclasses for combinations that are not possible in the database being modeled

- A *class* **inherits** *variables* and *methods* from all its *superclasses*
- There is potential for ***ambiguity*** when a variable/message N with the *same* ***name*** is inherited from two superclasses A and B
    - No problem if the variable/message is defined in a shared superclass
    - Otherwise, do one of the following
        - flag as an error,
        - rename variables (A.N and B.N)
        - choose one.

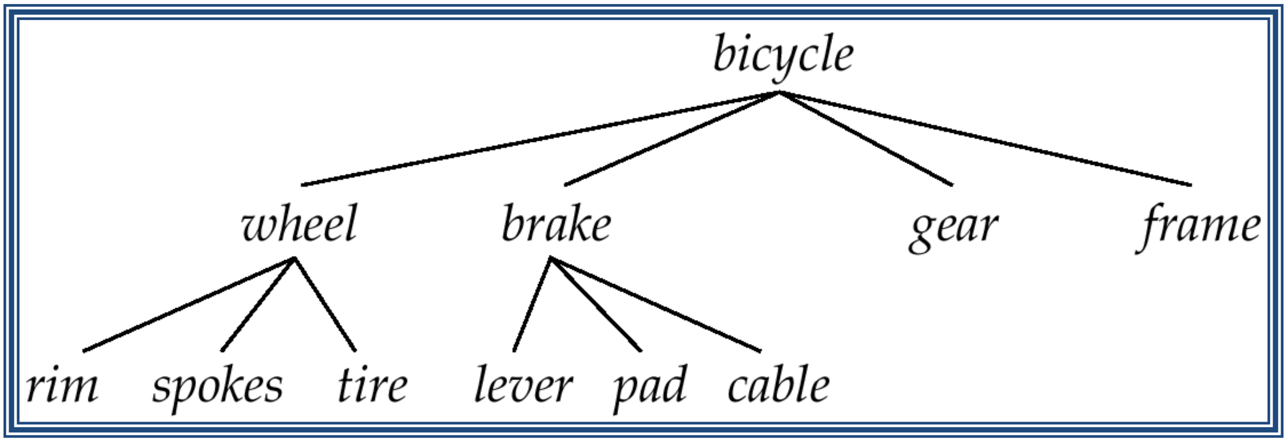
Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
    - Value – data value; e.g. primary key value used in relational systems.
    - Name – supplied by user; used for variables in procedures.
    - Built-in – identity built into data model or programming language.
        - no user-supplied identifier is required.
        - Is the form of identity used in object-oriented systems.

Object Identifiers

- **Object identifiers** used to uniquely identify objects
    - Object identifiers are unique:
        - no two objects have the same identifier
        - each object has only one object identifier
    - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
    - can be stored as a field of an object, to refer to another object.
    - Can be:
        - ***system generated*** (created by database) or
        - ***external*** (such as social-security number)
    - *System generated* identifiers:
        - Are *easier* to use, but **cannot** be used *across* database systems
        - May be ***redundant*** if unique identifier already exists

Object *Containment*



- Each component in a design may contain other components
- Can be modeled as containment of objects.
- Objects containing other objects are called **composite** objects.
- Multiple levels of containment create a **containment hierarchy**
    - links interpreted as **is-part-of,** not **is-a.**
- Allows data to be viewed at different granularities by different users.

Object-Oriented Languages

- *Object-oriented **concepts*** can be used in ***different ways***
    - Object-orientation can be used as a ***design tool***, and be encoded into,
        - for example, a ***relational*** database
    - analogous to ***modeling*** data with ***E-R*** diagram and
        - then ***converting*** to a set of ***relations***
- The ***concepts*** of object orientation can be *incorporated* into a ***programming language*** that is used to manipulate the database.
    - **Object-relational systems** – add complex types and object-orientation to relational language.
    - **Persistent programming languages** – extend object-oriented programming language to deal with databases by adding concepts such as persistence and collections.

Persistent Programming Languages

- *Persistent Programming languages* allow objects to be created and stored in a database, and used directly from a programming language
    - allow data to be manipulated directly from the programming language

- **No need** to go through **SQL**.
- **No need** for explicit format (type) changes
  - format changes are carried out ***transparently*** by system
  - Without a persistent programming language, format changes becomes a burden on the programmer
    - More code to be written
    - More chance of bugs
- **allow objects** to be ***manipulated*** in-memory
  - **no need** to explicitly **load** from or **store** to the database
    - Saved code, and saved overhead of loading/storing large amounts of data

- **Drawbacks** of *persistent programming* languages
  - Due to ***power*** of most programming ***languages***,
    - it is easy to make ***programming errors*** that **damage** the ***database***.
  - ***Complexity*** of languages makes
    - automatic high-level ***optimization*** more **difficult**.
  - Do ***not support declarative querying*** as well as relational databases

***Persistence* of Objects**

- Approaches to make transient objects persistent include establishing
  - Persistence by Class – declare all objects of a class to be persistent; simple but inflexible.
  - Persistence by Creation – extend the syntax for creating objects to specify that that an object is persistent.
  - Persistence by Marking – an object that is to persist beyond program execution is marked as persistent before program termination.
  - Persistence by Reachability - declare (root) persistent objects; objects are persistent if they are referred to (directly or indirectly) from a root object.
    - Easier for programmer, but more overhead for database system
    - Similar to garbage collection used e.g. in Java, which also performs reachability tests

Object Identity and Pointers

- A persistent object is assigned a persistent object identifier.
- Degrees of permanence of identity:
  - Intraprocedure – identity persists only during the executions of a single procedure
  - Intraprogram – identity persists only during execution of a single program or query.

    - Interprogram – identity persists from one program execution to another, but may change if the storage organization is changed
    - Persistent – identity persists throughout program executions and structural reorganizations of data; required for object-oriented systems.
- In O-O languages such as C++, an object identifier is actually an in-memory pointer.
- Persistent pointer – persists beyond program execution
    - can be thought of as a pointer into the database
        - E.g. specify file identifier and offset into the file
    - Problems due to database reorganization have to be dealt with by keeping forwarding pointers

***Storage* and *Access* of Persistent Objects**

How to find objects in the database:

- ***Name*** objects (as you would name files)
    - Cannot scale to *large number* of objects.
    - Typically given only to class extents and other collections of objects, but not objects.
- Expose ***object identifiers*** or ***persistent pointers*** to the objects
    - Can be stored externally.
    - ***All objects have*** *object identifiers*.
- ***Store collections of objects***, and allow programs to iterate over the collections to find required objects
    - ***Model*** collections of objects as **collection types**
    - **Class extent -** the collection of all objects belonging to the class;
        - usually maintained for ***all classes*** that can have *persistent objects*.

Persistent C++ Systems

- C++ language allows support for persistence to be added without changing the language
    - Declare a class called Persistent_Object with attributes and methods to support persistence
    - **Overloading** – ability to redefine standard function names and operators (i.e., +, –, the pointer deference operator –>) when applied to new types
    - **Template classes** help to build a type-safe type system supporting collections and persistent types.
- Providing persistence without extending the C++ language is
    - relatively easy to implement
    - but more difficult to use

- Persistent C++ systems that add features to the C++ language have been built, as also systems that avoid changing the language

ODMG C++ Object Definition Language

- The ***Object Database Management Group*** is
    - ➢ an **industry consortium** aimed at
    - ➢ ***standardizing*** object-oriented databases
  - in particular *persistent programming* languages
  - Includes standards for C++, Smalltalk and Java
  - ODMG-2.0 and 3.0 (which is 2.0 plus extensions to Java)
    - ➢ The description based on ODMG-2.0
- **ODMG** C++ standard **avoids** *changes* to the C++ language
  - *provides* functionality **via** template classes and ***class libraries***

ODMG Types

- Template class d_Ref<class> used to specify references (persistent pointers)
- Template class d_Set<class> used to define sets of objects.
  - Methods include insert_element(e) and delete_element(e)
- Other collection classes such as d_Bag (set with duplicates allowed), d_List and d_Varray (variable length array) also provided.
- d_ version of many standard types provided, e.g. d_Long and d_string
    - Interpretation of these types is platform independent
    - Dynamically allocated data (e.g. for d_string) allocated in the database, not in main memory

ODMG C++ ODL: Example

```
class Branch  :  public d_Object {
   ….
        }
        class Person  :  public d_Object {
  public:
        d_String    name;       // should not use String!
                d_String    address;
};
        class Account : public d_Object {
  private:
        d_Long      balance;
```

```
   public:
           d_Long      number;
           d_Set <d_Ref<Customer>> owners;
           int     find_balance();
           int     update_balance(int delta);
};

class Customer  :  public Person {
public:
           d_Date   member_from;
           d_Long   customer_id;
           d_Ref<Branch> home_branch;
           d_Set <d_Ref<Account>> accounts; };
```

Implementing Relationships

- Relationships between classes implemented by references
- Special reference types enforces integrity by adding/removing inverse links.
  - Type d_Rel_Ref<Class, InvRef> is a reference to Class, where attribute InvRef of Class is the inverse reference.
  - Similarly, d_Rel_Set<Class, InvRef> is used for a set of references
- Assignment method (=) of class d_Rel_Ref is overloaded
  - Uses type definition to automatically find and update the inverse link
  - Frees programmer from task of updating inverse links
  - Eliminates possibility of inconsistent links
- Similarly, insert_element() and delete_element() methods of d_Rel_Set use type definition to find and update the inverse link automatically
- E.g.

```
           extern const char _owners[ ],   _accounts[ ];
class Account : public d.Object {
       ….
           d_Rel_Set <Customer, _accounts> owners;
}
  // .. Since strings can’t be used in templates …
```

const char _owners = “owners”;

const char _accounts = “accounts”;

ODMG C++ Object Manipulation Language

- Uses persistent versions of C++ operators such as new(db)

  d_Ref<Account> account = new(bank_db, “Account”) Account;

    - new allocates the object in the specified database, rather than in memory.
    - The second argument (“Account”) gives typename used in the database.
- Dereference operator -> when applied on a d_Ref<Account> reference loads the referenced object in memory (if not already present) before continuing with usual C++ dereference.
- Constructor for a class – a special method to initialize objects when they are created; called automatically on new call.
- Class extents maintained automatically on object creation and deletion
  - Only for classes for which this feature has been specified
    - Specification via user interface, not C++
  - Automatic maintenance of class extents not supported in earlier versions of ODMG

ODMG C++OML: Database and Object Functions

- Class d_Database provides methods to
    - open a database : open(databasename)
    - **give** *names* to objects: set_object_name(object, name)
    - **look up** objects *by name*: lookup_object(name)
    - **rename** objects: rename_object(oldname, newname)
    - close a database (close());
- Class d_Object is inherited by all persistent classes.
    - provides methods to allocate and delete objects
    - method mark_modified() must be called *before* an object is updated.

Is automatically called when object is created

ODMG C++ OML: Example

int create_account_owner(String name, String Address){

Database bank_db.obj;

Database * bank_db= & bank_db.obj;

```
bank_db =>open("Bank-DB");
d.Transaction Trans;
Trans.begin();

d_Ref<Account> account = new(bank_db) Account;
d_Ref<Customer> cust = new(bank_db) Customer;
cust->name - name;
cust->address = address;
cust->accounts.insert_element(account);
... Code to initialize other fields

Trans.commit();
}
```

- Class extents maintained automatically in the database.
- To access a class extent:
  d_Extent<Customer> customerExtent(bank_db);
- Class d_Extent provides method
  d_Iterator<T> create_iterator()
  to create an iterator on the class extent
- Also provides select(pred) method to return iterator on objects that satisfy selection predicate pred.
- Iterators help step through objects in a collection or class extent.
- Collections (sets, lists etc.) also provide create_iterator() method.

ODMG C++ OML: Example of Iterators

```
int print_customers() {
Database bank_db_obj;
Database * bank_db = &bank_db_obj;
bank_db->open ("Bank-DB");
d_Transaction Trans; Trans.begin ();

d_Extent<Customer> all_customers(bank_db);
d_Iterator<d_Ref<Customer>> iter;
```

```
iter = all_customers–>create_iterator();
d_Ref <Customer> p;
        while{iter.next (p))
                print_cust (p);  // Function assumed to be defined elsewhere
        Trans.commit();
}
```

ODMG C++ Binding: Other *Features*

- *Declarative query language* **OQL**, looks like **SQL**
  - Form query as a string, and execute it to get a set of results (actually a bag, since duplicates may be present)

```
        d_Set<d_Ref<Account>> result;
d_OQL_Query q1("select a
                from Customer c, c.accounts a
                where c.name='Jones'
                  and a.find_balance() > 100");
d_oql_execute(q1, result);
```

- Provides error handling mechanism based on C++ exceptions, through class d_Error
- Provides **API** for accessing the schema of a database.

Making Pointer Persistence Transparent

- **Drawback** of the ODMG C++ ***approach***:
  - ***Two types*** of **pointers**
  - ***Programmer*** has to **ensure**
    - mark_modified() is called,
    - else database can become **corrupted**
- ***ObjectStore approach***
  - Uses *exactly* the same pointer type for in-memory and database objects
  - Persistence is transparent applications
    - Except when creating objects
  - Same functions can be used on in-memory and persistent objects since pointer types are the same
  - Implemented by a technique called pointer-swizzling which is described in Chapter 11.
  - No need to call mark_modified(), modification detected automatically.

Persistent *Java* Systems

- ODMG-3.0 defines ***extensions*** to Java for persistence
  - Java does **not** *support* **templates**,
    - ➢ so language *extensions* are *required*
- *Model* for persistence: *persistence* by *reachability*
  - *Matches* Java's *garbage collection* model
  - Garbage collection needed on the database also
  - **Only** *one* pointer *type* for transient and persistent pointers
- *Class* is made ***persistence capable*** by running
  - ➢ a *post-processor* on object code generated by the Java compiler
  - *Contrast* with *pre-processor* used in C++
  - Post-processor ***adds*** mark_modified() *automatically*
- Defines collection types DSet, DBag, DList, etc.
- Uses Java iterators, no need for new iterator class

ODMG Java

- Transaction must start accessing database from one of the root object (***looked up*** *by name*)
  - finds other objects by following pointers from the root objects
- Objects ***referred*** to *from* a *fetched object* are:
  - *allocated* space in memory,
  - but *not* necessarily *fetched*
- ***Fetching*** can be done *lazily*
  - An object with space allocated but *not* yet *fetched* is called a *hollow object*
  - When a *hollow object* is accessed, its data is fetched from disk.

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Data Types

- Motivation:
    - Permit non-atomic domains (atomic ≡ indivisible)
    - Example of non-atomic domain: set of integers, or set of tuples
    - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
    - allow relations whenever we allow atomic (scalar) values — relations within relations
    - Retains mathematical foundation of relational model
    - Violates first normal form.

Example of a Nested Relation

- Example: library information system
- Each book has
    - title,
    - a set of authors,
    - Publisher, and
    - a set of keywords
- Non-1NF relation *books*

Title	Author-set	Publisher	Keyword-set
		(Name, Branch)	
Compilers	(Smith, Jones)	(McGraw-Hill, New York)	{Parsing, Analysis}
Networks	(Jones, Frick)	(Oxford, London)	{Internet, Web}

4NF Decomposition of Nested Relation

- Remove awkwardness of *flat-books* by assuming that the following multivalued dependencies hold:
    - *title* $\rightarrow\!\!\rightarrow$ *author*
    - *title* $\rightarrow\!\!\rightarrow$ *keyword*
    - *title* $\rightarrow\!\!\rightarrow$ *pub-name, pub-branch*
- Decompose *flat-doc* into 4NF using the schemas:
    - (*title, author*)
    - (*title, keyword*)
    - (*title, pub-name, pub-branch*)

4NF Decomposition of *flat–books*

Title	Author
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick
Authors	

Title	Keyword
Compilers	Parsing
Compilers	Analysis
Networks	Internet
Networks	Web
Keywords	

Title	Pub-Name	Pub-Branch
Compilers	McGraw-Hill	New York
Networks	Oxford	London
Books4		

Problems with 4NF Schema

- 4NF design requires users to include joins in their queries.
- 1NF relational view *flat-books* defined by join of 4NF relations:
    - eliminates the need for users to perform joins,
    - but loses the one-to-one correspondence between tuples and documents.
    - And has a large amount of redundancy
- Nested relations representation is much more natural here.

Complex Types and SQL

- Extensions to SQL to support complex types include:
    - Collection and large object types
        - Nested relations are an example of collection types
    - Structured types
        - Nested record structures like composite attributes
    - Inheritance
    - Object orientation
        - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard
    - Not fully implemented in any database system currently
    - But some features are present in each of the major commercial database systems
        - Read the manual of your database system to see what it supports

Structured Types and Inheritance in SQL

- Structured types can be declared and used in SQL

  create type *Name* **as**

  (first*name* **varchar**(20),

  lastname **varchar**(20))

  final

  create type *Address* **as**

  (*street* **varchar**(20),

  city **varchar**(20),

  zipcode **varchar**(20))

  not final

    - Note: **final** and **not final** indicate whether subtypes can be created

- Structured types can be used to create tables with composite attributes

  create table *customer* (

  name Name,

  address Address,

  dateOfBirth **date**)

- Dot notation used to reference components: *name.firstname*
- User-defined row types

create type *CustomerType* **as** (

name Name,

address Address,

dateOfBirth **date**)

not final

- Can then create a table whose rows are a user-defined type

create table *customer* **of** *CustomerType*

Methods

- Can add a method declaration with a structured type.
    - **method** *ageOnDate* (*onDate* **date**)

- **returns interval year**

- Method body is given separately.

create instance method *ageOnDate* (*onDate* **date**)

returns interval year

for *CustomerType*

begin

return *onDate* - **self**.*dateOfBirth*;

end

- We can now find the age of each customer:

select *name.lastname, ageOnDate* (**current_date**)

from *customer*

Inheritance

- Suppose that we have the following type definition for people:

create type *Person*

(*name* **varchar**(20),

address **varchar**(20))

- Using inheritance to define the student and teacher types

create type *Student*

under *Person*

(*degree* **varchar**(20),

department **varchar**(20))

create type *Teacher*

under *Person*

(*salary* **integer**,

department **varchar**(20))

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Inheritance

- Some SQL do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*

  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*

  under

  Student **with** (*department* **as** *student_dept*),

  Teacher **with** (*department* **as** *teacher_dept*)

Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
  - Additional constraint in SQL:
- All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
  - That is, each entity must have a most specific type
  - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*

Array and Multiset Types in SQL

- Example of array and multiset declaration:

  create type *Publisher* **as**

  (*name* **varchar**(20),

  branch **varchar**(20))

  create type *Book* **as**

  (*title* **varchar**(20),

  author-array **varchar**(20) **array** [10],

  pub-date **date**,

publisher *Publisher*,

keyword-set **varchar**(20) **multiset**)

create table *books* **of** *Book*

- Similar to the nested relation books, but with array of authors instead of set

Creation of Collection Values

- Array construction

  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets

  - **multisetset** [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the *books* relation: (‘Compilers’, **array**[`Smith’,`Jones’],

  Publisher (`McGraw-Hill’,`New York’), **multiset** [`parsing’,`analysis’])

- To insert the preceding tuple into the relation *books*

insert into *books*

values

(‘Compilers’, **array**[`Smith’,`Jones’],

Publisher (`McGraw-Hill’,`New York’), **multiset** [`parsing’,`analysis’])

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

  select *title*

  from *books*

  where ‘database’ **in** (**unnest**(*keyword-set*))

- We can access individual elements of an array by using indices

  - E.g.: If we know that a particular book has three authors, we could write:

  select *author-array*[1], *author-array*[2], *author-array*[3]

  from *books*

  where *title* = `Database System Concepts’

- To get a relation containing pairs of the form “title, author-name” for each book and each author of the book

  select *B.title*, *A.author*

  from *books* **as** *B*, **unnest** (*B.author-array*) **as** *A* (*author*)

- To retain ordering information we add a **with ordinality** clause

  select *B.title, A.author, A.position*

  from *books* **as** *B,* **unnest** (*B.author-array*) **with ordinality as**

  A (*author, position*)

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes us called **unnesting**.
- E.g.

select *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,

publisher.branch **as** *pub_branch*, *K.keyword*

from *books* **as** *B*, **unnest**(*B.author_array*) **as** *A* (*author*),

unnest (*B.keyword_set*) **as** *K* (*keyword*)

Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- Nesting can be done in a manner similar to aggregation, but using the function **colect**() in place of an aggregation operation, to create a multiset
- To nest the *flat-books* relation on the attribute *keyword*:

  select *title*, *author*, *Publisher* (*pub_name, pub_branch*) **as** *publisher*,

  collect (*keyword*) **as** *keyword_set*

from *flat-books*

groupby *title, author, publisher*

- To nest on both authors and keywords:

select *title,* **collect** (*author*) **as** *author_set,*

Publisher (*pub_name, pub_branch*) **as** *publisher,*

collect (*keyword*) **as** *keyword_set*

from *flat-books*

group by *title*, *publisher*

1NF Version of Nested Relation

1NF version of *books*

Title	Author	Pun-Name	Pub-Branch	Keyword
Compilers	Smith	McGraw-Hill	New York	Parsing
Compilers	Jones	McGraw-Hill	New York	Parsing
Compilers	Smith	McGraw-Hill	New York	Analysis
Compilers	Jones	McGraw-Hill	New York	Analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Flat-books

- Another approach to creating nested relations is to use subqueries in the **select** clause.

  select *title,*

  array (**select** *author*

  from *authors* **as** *A*

  where *A.title* = *B.title*

order by *A.position*) **as** *author_array,*

Publisher (*pub-name, pub-branch*) **as** *publisher,*

multiset (**select** *keyword*

from *keywords* **as** *K*

where *K.title* = *B.title*) **as** *keyword_set*

from *books4* **as** *B*

Object-Identity and Reference Types

- Define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person,* with table *people* as scope:
  - **create type** *Department* (

    name **varchar** (20),

    head **ref** (*Person*) **scope** *people*)
- We can then create a table *departments* as follows
  - **create table** *departments* **of** *Department*

- We can omit the declaration **scope** people from the type declaration and instead make an addition to the **create table** statement:

  create table *departments* **of** *Department*
  (*head* **with options scope** *people*)

Initializing Reference-Typed Values

- To create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately:

  insert into *departments*
  values (`CS', null)
  update *departments*
  set *head* = (**select** *p.person_id*
  from *people* **as** *p*
  where *name* = `John')
  where *name* = `CS'

User Generated Identifiers

- The type of the object-identifier must be specified as part of the type definition of the referenced table, and
- The table definition must specify that the reference is user generated

  create type *Person*
  (*name* **varchar**(20)
  address **varchar**(20))
  ref using varchar(20)
  create table *people* **of** *Person*
  ref is *person_id* **user generated**

- When creating a tuple, we must provide a unique value for the identifier:

  insert into *people* (*person_id, name, address*) **values**
  ('01284567', 'John', `23 Coyote Run')

- We can then use the identifier value when inserting a tuple into *departments*
  - Avoids need for a separate query to retrieve the identifier:

insert into *departments*

values(`CS’, `02184567’)

- Can use an existing primary key value as the identifier:

create type *Person*

(*name* **varchar** (20) **primary key**,

address **varchar**(20))

ref from (*name*)

create table *people* **of** *Person*

ref is *person_id* **derived**

- When inserting a tuple for *departments*, we can then use

insert into *departments*

values(`CS’,`John’)

Path Expressions

- Find the names and addresses of the heads of all departments:

select *head –>name, head –>address*

from *departments*

- An expression such as “head–>name” is called a **path expression**
- Path expressions help avoid explicit joins
  - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
  - Makes expressing the query much easier for the user

Implementing O-R Features

- Similar to how E-R features are mapped onto relation schemas
- Subtable implementation
  - Each table stores primary key and those attributes defined in that table or
  - Each table stores both locally defined and inherited attributes

Persistent Programming Languages

- Languages extended with constructs to handle persistent data
- Programmer can manipulate persistent data directly
  - no need to fetch it into memory and store it back to disk (unlike embedded SQL)

- Persistent objects:
    - by class - explicit declaration of persistence
    - by creation - special syntax to create persistent objects
    - by marking - make objects persistent after creation
    - by reachability - object is persistent if it is declared explicitly to be so or is reachable from a persistent object

Object Identity and Pointers

- Degrees of permanence of object identity
    - Intraprocedure: only during execution of a single procedure
    - Intraprogram: only during execution of a single program or query
    - Interprogram: across program executions, but not if data-storage format on disk changes
    - Persistent: interprogram, plus persistent across data reorganizations
- Persistent versions of C++ and Java have been implemented
    - C++
        - ODMG C++
        - ObjectStore
    - Java
        - Java Database Objects (JDO)

Comparison of O-O and O-R Databases

- **Relational systems**
    - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
    - complex data types, integration with programming language, high performance.
- **Object-relational systems**
    - complex data types, powerful query languages, high protection.
- Note: Many real systems blur these boundaries
    - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

ADVANCED SQL

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2005-7-27’
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
  - Example: interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
- Can extract values of individual fields from date/time/timestamp
  - Example: **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
  - Example: **cast** <string-valued-expression> **as date**
  - Example: **cast** <string-valued-expression> **as time**

User-Defined Types

- **create type** construct in SQL creates user-defined type
  - **create type** *Dollars* **as numeric (12,2) final**
- **create domain** construct in SQL-92 creates user-defined domain types
  - **create domain** *person_name* **char**(20) **not null**
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **Domain Constraints**
- **Domain constraints** are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types

- Example: **create domain** *Dollars* **numeric**(12, 2)

  create domain *Pounds* **numeric**(12,2)

- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type as below

    (**cast** *r.A* **as** *Pounds*)

    (Should also multiply by the dollar-to-pound conversion-rate)

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Constraints on a Single Relation

- not null
- primary key
- unique
- check (*P*), where *P* is a predicate
- Not Null ConstraintDeclare *branch_name* for *branch* is not null
- *branch_name* char(15) not null
- Declare the domain *Dollars* to be not null
- create domain *Dollars* numeric(12,2) not null
- The Unique Constraintunique (A_1, A_2, …, A_m)

- The unique specification states that the attributes

$A_1, A_2, \ldots A_m$

Form a candidate key.

Candidate keys are permitted to be non null (in contrast to primary keys).

The check clause

- **check** (*P*), where *P* is a predicate
  - Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.
  - **create table** *branch*

    (*branch_name* **char**(15)**,**
    branch_city **char**(30),
    assets **integer**,
    primary key (*branch_name*),
    check (*assets* >= 0))
- The **check** clause in SQL permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.
    - **create domain** *hourly_wage* **numeric(5,2)**

      constraint *value_test* **check**(*value* > = 4.00)
  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".

- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
  - The primary key clause lists attributes that comprise the primary key.
  - The unique key clause lists attributes that comprise a candidate key.
  - The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity in SQL – Example

create table *customer*
(*customer_name* **char**(20)**,**
customer_street **char**(30),
customer_city **char**(30),
primary key (*customer_name*))

create table *branch*
(branch_name **char**(15)**,**
branch_city **char**(30),
assets **numeric**(12,2),
primary key (*branch_name*))
create table *account*
(*account_number* **char**(10)**,**
branch_name **char**(15),
balance **integer**,
primary key (*account_number),*
foreign key (*branch_name*) **references** *branch*)
create table *depositor*
(*customer_name* **char**(20)**,**
account_number **char**(10)**,**
primary key (*customer_name, account_number),*

foreign key (*account_number*) **references** *account,*
foreign key (*customer_name*) **references** *customer*)

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form
  - **create assertion** <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting

  for all *X*, *P*(*X*)

  is achieved in a round-about fashion using

  not exists *X* such that not *P*(*X*)

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

```
create assertion balance_constraint check
(not exists (
    select *
        from loan
            where not exists (
        select *
                from borrower, depositor, account
                where loan.loan_number = borrower.loan_number
                    and borrower.customer_name = depositor.customer_name
                    and depositor.account_number = account.account_number
                    and account.balance >= 1000)))
```

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

create assertion *sum_constraint* **check**
(**not exists** (**select ***
from *branch*
where (**select sum**(*amount*)
from *loan*
where *loan.branch_name* =
branch.branch_name)
>= (**select sum** (*amount*)
from *account*
where *loan.branch_name* =
branch.branch_name)))

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update -** allows modification, but not deletion of data.
- **Delete** - allows deletion of data.
- Forms of authorization to modify the database schema (covered in Chapter 8):
- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop -** allows deletion of relations.
- **Authorization Specification in SQL**
- The **grant** statement is used to confer authorization
    - **grant** <privilege list>
    - **on** <relation name or view name> **to** <user list>
- <user list> is:

    - a user-id
    - **public**, which allows all valid users the privilege granted
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
    - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:
        - **grant select on** *branch* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
- **revoke** <privilege list>
- **on** <relation name or view name> **from** <user list>
- Example:
- **revoke select on** *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor
    - EXEC SQL <embedded SQL statement > END_EXEC

Note: this varies by language (for example, the Java embedding uses

SQL { …. };)

Example Query

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.
- Specify the query in SQL and declare a *cursor* for it

EXEC SQL

declare *c* **cursor for**

select *customer_name, customer_city*

from *depositor, customer, account*

where *depositor.customer_name = customer.customer_name*

and *depositor account_number = account.account_number*

and *account.balance > :amount*

END_EXEC

- The **open** statement causes the query to be evaluated

EXEC SQL **open** *c* END_EXEC

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL **fetch** *c* **into** :*cn*, *:cc* END_EXEC

Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END_EXEC

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
declare c cursor for
 select *
     from account
   where branch_name = 'Perryridge'
  for update
```

- To update tuple at the current location of cursor *c*

```
update account
    set balance = balance + 100
      where current of c
```

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char * sqlprog = "update account
                   set balance = balance * 1.05
                  where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
    - Connect with the database server
    - Send SQL commands to the database server
    - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

ODBC

- Open DataBase Connectivity(ODBC) standard
    - standard for application program to communicate with a database server.
    - application program interface (API) to
        - open a connection with a database,
        - send queries and updates,
        - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
    - connection handle,
    - the server to which to connect
    - the user identifier,
    - password
- Must also specify types of arguments:
    - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

```
int ODBCexample()
        {
RETCODE error;
HENV    env;    /* environment */
HDBC    conn;  /* database connection */
SQLAllocEnv(&env);
SQLAllocConnect(env, &conn);
SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS, "avipasswd",
SQL_NTS);
{ …. Do actual work … }
SQLDisconnect(conn);
SQLFreeConnect(conn);
SQLFreeEnv(env);
}
```

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
    - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
    - Arguments to SQLBindCol()
        - ODBC stmt variable, attribute position in query result
        - The type conversion from SQL to C.
        - The address of the variable.
        - For variable-length types like character arrays,
            - The maximum length of the variable
            - Location to store actual length when a tuple is fetched.
            - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

- Main body of program

```
    char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;
    SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                        from account
                        group by branch_name";
    error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
    if (error == SQL_SUCCESS) {
     SQLBindCol(stmt, 1, SQL_C_CHAR,   branchname , 80, &lenOut1);
     SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,        0 , &lenOut2);
         while (SQLFetch(stmt) >= SQL_SUCCESS) {
         printf (" %s  %g\n", branchname, balance);
     }
}
SQLFreeStmt(stmt, SQL_DROP);
```

More ODBC Features

- **Prepared Statement**
    - SQL statement prepared: compiled at the database
    - Can have placeholders: E.g. insert into account values(?,?,?)
    - Repeatedly executed with actual values for the placeholders
- **Metadata features**
    - finding all the relations in the database and
    - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
    - Can turn off automatic commit on a connection

    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
- transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
    - Core
    - Level 1 requires support for metadata querying
    - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
    - Open a connection
    - Create a "statement" object
    - Execute queries using the Statement object to send queries and fetch results
    - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
    {
   try {
 Class.forName ("oracle.jdbc.driver.OracleDriver");
```

```
  Connection conn = DriverManager.getConnection(   "jdbc:oracle:thin:@aura.bell-
labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
   }
   catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
   }
     }
```

- **Update to database**

```
try {
     stmt.executeUpdate(  "insert into account values
                                  ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
     System.out.println("Could not insert tuple. " + sqle);
}
```

- **Execute query and fetch and print results**

```
ResultSet rset = stmt.executeQuery( "select branch_name, avg(balance)
                                           from account
                                           group by branch_name");
while (rset.next()) {
System.out.println(
           rset.getString("branch_name") + "  " + rset.getFloat(2));
}
```

JDBC Code Details

- Getting result fields:

- rs.getString("branchname") and rs.getString(1) equivalent if branchname is the first argument of select result.

- Dealing with Null values

  int a = rs.getInt("a");

  if (rs.wasNull()) Systems.out.println("Got null value");

Procedural Extensions and Stored Procedures

- SQL provides a module language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the call statement
  - permit external applications to operate on the database without knowing about internal details

Functions and Procedures

SQL supports functions and procedures

- Functions/procedures can be written in SQL itself, or in an external programming language
- Functions are particularly useful with specialized data types such as images and geometric objects
  - Example: functions to check if polygons overlap, or to compare images for similarity
- Some database systems support **table-valued functions**, which can return a relation as a result

- SQL also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment

SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count (customer_name varchar(20))
returns integer
begin
    declare a_count integer;
    select count (* ) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Find the name and address of each customer that has more than one account.

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```

Table Functions

- SQL added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function accounts_of (customer_name char(20)
    returns table (     account_number char(10),
                        branch_name char(15),
                        balance numeric(12,2))
return table
    (select account_number, branch_name, balance
     from account
     where exists (
      select *
      from depositor
```

where *depositor.customer_name* = *accounts_of.customer_name*

and *depositor.account_number* = *account.account_number*))

- Usage

  select *

  from table (*accounts_of* (‘Smith’))

SQL Procedures

- The *author_count* function could instead be written as procedure:

  create procedure *account_count_proc* (**in** *title* **varchar**(20), **out** *a_count* **integer)**

  begin

  select count(*author*) **into** *a_count*

  from *depositor*

  where *depositor.customer_name* = *account_count_proc.customer_name*

  end

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

  declare *a_count* **integer**;

  call *account_count_proc*(‘Smith’, *a_count*);

  Procedures and functions can be invoked also from dynamic SQL

- SQL allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Procedural Constructs

- Compound statement: **begin … end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- **While** and **repeat** statements:

  declare *n* **integer default** 0;

  while *n* < 10 **do**

```
        set n = n + 1
    end while
    repeat
set n = n  – 1
    until n = 0
    end repeat
```

- **For** loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```
declare n  integer default 0;
for r  as
     select balance from account
      where branch_name = 'Perryridge'
 do
           set n = n + r.balance
 end for
```

- Conditional statements (**if-then-else**)

  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
        if r.balance < 1000
   then set l = l + r.balance
elseif r.balance < 5000
   then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL also supports a **case** statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
        declare out_of_stock condition
declare exit handler for out_of_stock
```

begin

…

.. **signal** out-of-stock

end

- The handler here is **exit** -- causes enclosing **begin..end** to be exited
- Other actions possible on exception

External Language Functions/Procedures

- SQL permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

create procedure account_count_proc(**in** *customer_name* **varchar**(20), **out** count **integer**)

language C
external name ' /usr/avi/bin/account_count_proc'

create function account_count(*customer_name* **varchar**(20))
returns integer
language C
external name '/usr/avi/bin/author_count'

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance

- Direct execution in the database system's space is used when efficiency is more important than security

Security with External Language Routines

- To deal with security problems
  - Use **sandbox** techniques
    - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Or, run external language functions/procedures in a separate process, with no access to the database process' memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space

Recursion in SQL

- SQL permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

This example view, *empl,* is called the *transitive closure* of the *manager* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *manager* with itself
    - This can give only a fixed number of levels of managers
    - Given a program we can construct a database with a greater number of levels of managers on which the program will not work
  - The next slide shows a *manager* relation and each step of the iterative process that constructs *empl* from its recursive definition. The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be *monotonic*. That is, if we add tuples to *manger* the view contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	*Tuples in empl*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Advanced SQL Features

- Create a table with the same schema as an existing table:

  create table *temp_account* **like** *account*

- SQL allows subqueries to occur *anywhere* a value is required provided the subquery returns only one value. This applies to updates as well
- SQL:2003 allows subqueries in the **from** clause to access attributes of other relations in the **from** clause using the **lateral** construct:

  select *customer_name, num_accounts*

  from *customer*, **lateral** (

          select count(*)

          from *account*

          where *account.customer_name* = *customer.customer_name*)

          as *this_customer* (*num_accounts*)
- Merge construct allows batch processing of updates.
- Example: relation *funds_received* (*account_number, amount*) has batch of deposits to be added to the proper account in the *account* relation

  merge into *account* **as** *A*

  using (**select** *

           from *funds_received* **as** *F*)

  on (A.*account_number* = *F.account_number*)

  when matched then

          update set *balance* = *balance* + *F.amount*

DATABASE SYSTEM ARCHITECTURES

- **Centralized** and **Client-Server** Systems
- **Server** System Architectures
- **Parallel** Systems
- **Distributed** Systems
- Network Types

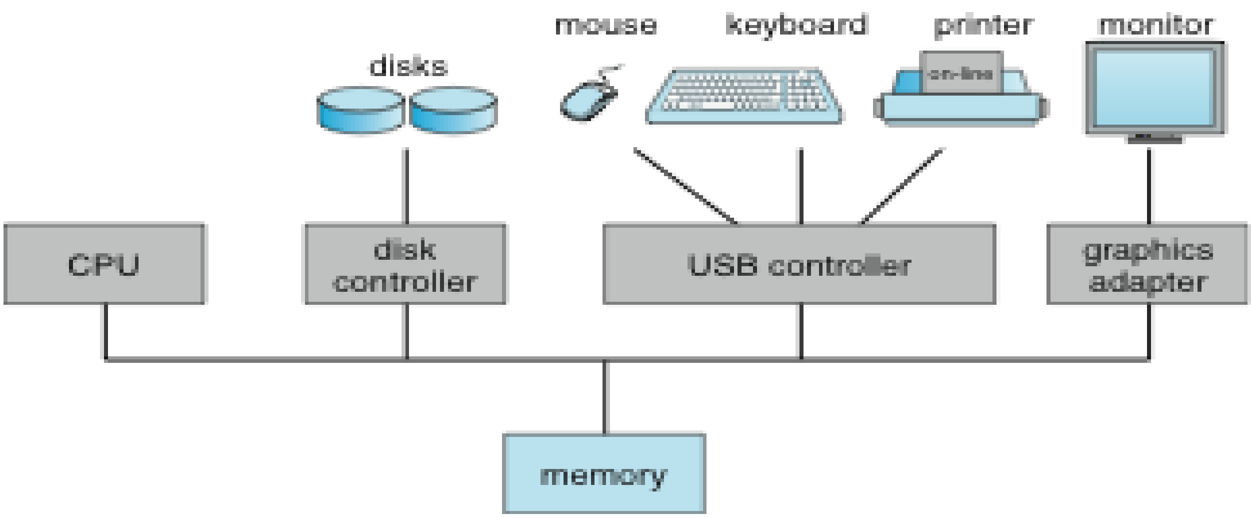
Centralized Systems

- Run on a **single** computer system and
  - do not interact with other computer systems.
- General-purpose computer system:

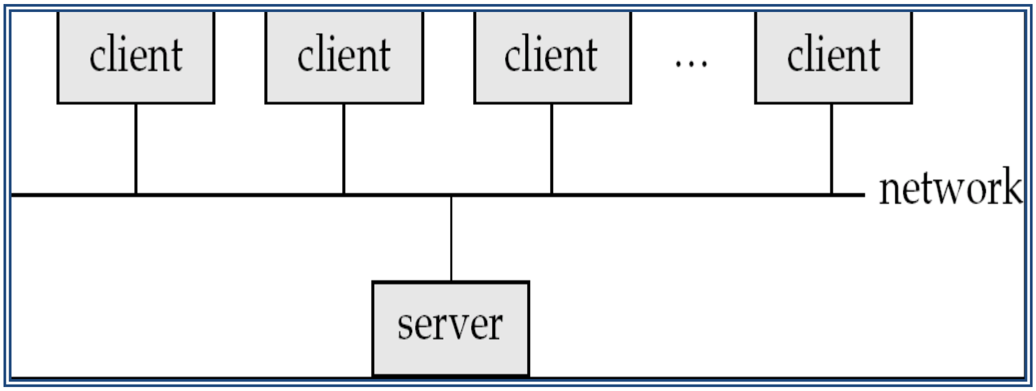
- one to a few **CPU**s and a number of device controllers that are
  - connected through a common **bus** that provides access to **shared** memory.

- Single-user system (e.g., personal computer or workstation):
  - desk-top unit, single user, usually has only one **CPU** and one or two hard **disks**;
    - the **OS** may support only **one** user.
- Multi-user system:
  - more **disks**, more **memory**, multiple **CPUs**, and a multi-user **OS**.
    - Serve a **large number** of users who are connected to the system via terminals
    - Often called ***server*** systems.

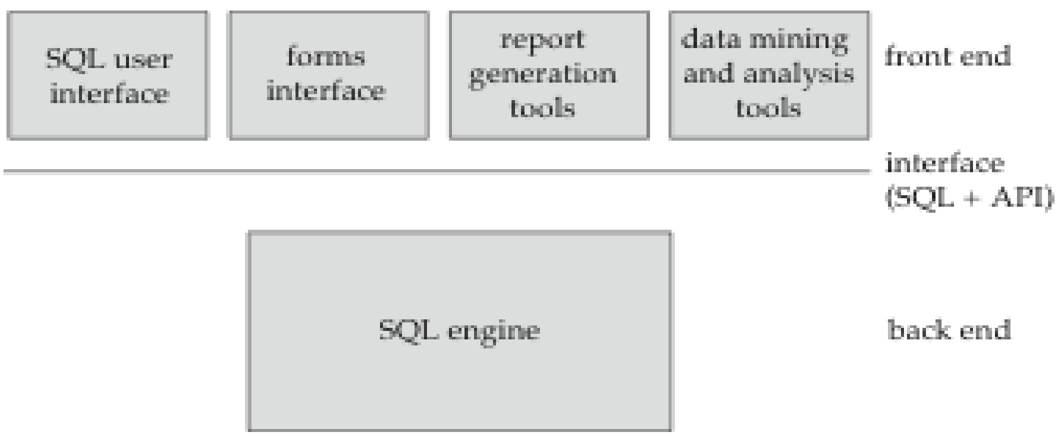
A Centralized Computer System



- **Client-Server Systems**
  - Server systems satisfy requests generated at *m* client systems,
    - whose general structure is shown below:



- Database *functionality* can be divided into:
    - Back-end: manages access structures, query evaluation and optimization, concurrency control and recovery.
    - Front-end: consists of tools such as *forms*, *report-writers*, and graphical user interface facilities.
- The interface between the front-end and the back-end is:
    - through SQL or through an Application Program Interface (API).



- **Advantages of replacing mainframes with:**
    - networks of workstations
    - or personal computers
    - connected to back-end server machines:
        - better functionality for the cost
        - flexibility in
            - locating resources and

- expanding facilities
- better user interfaces
- easier maintenance

Server System Architecture

- **Server** systems can be broadly categorized into **two** kinds:
  - **transaction servers** which are :
    - widely used in **relational** database systems,
  - **data servers**, used in **object-oriented** database systems

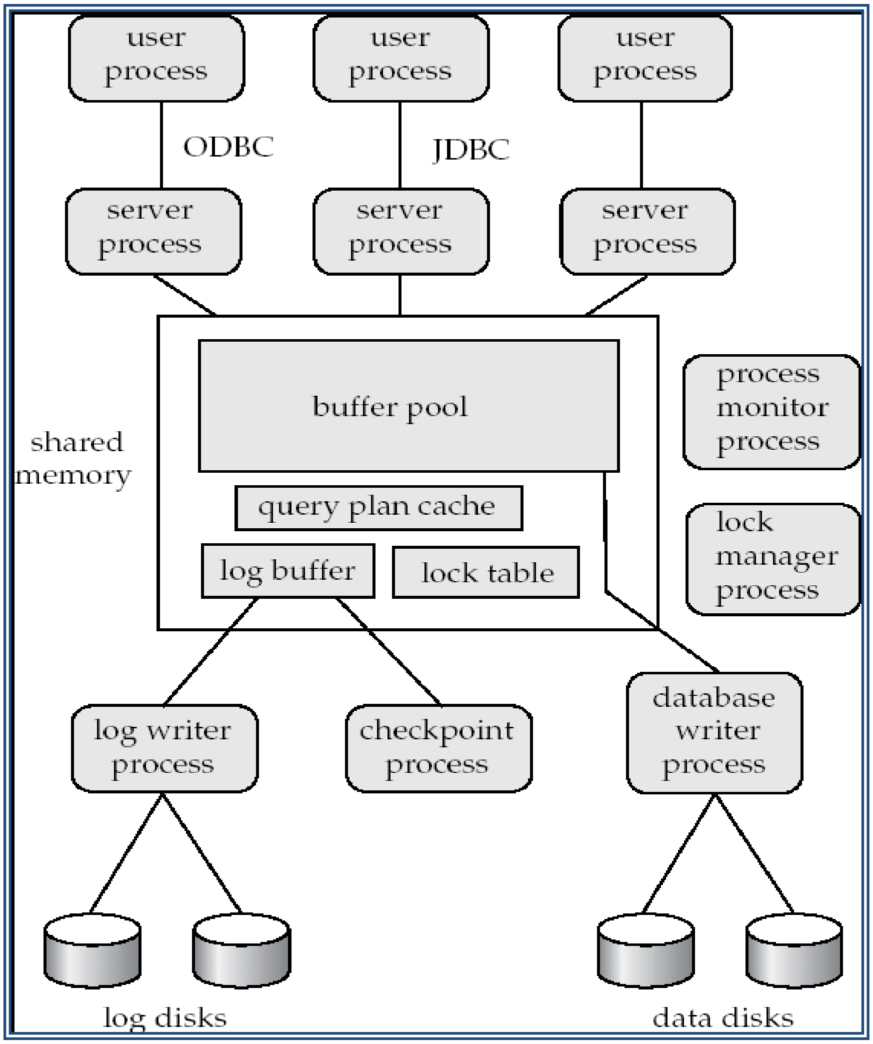
Transaction Servers

- Also called **query server** systems or **SQL *server*** systems
  - Clients send requests to the server
  - Transactions are executed at the server
  - **Results** are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a *remote procedure call* (**RPC**) mechanism.
- Transactional RPC **allows** many RPC calls to form a transaction.
- *Open Database Connectivity* (**ODBC**) is a **C** language application program interface standard from Microsoft for connecting to a server,
    - sending **SQL** requests, and receiving **results**.
    - **JDBC** standard is similar to ODBC, for Java

Transaction Server Process Structure

- A typical transaction **server** consists of :
  - multiple processes accessing data in ***shared memory***.
- **Server** processes
  - These receive user queries (transactions),
    - execute them and **send** results back
  - Processes may be **multithreaded**,
    - allowing a single process to execute **several** user queries concurrently

- Typically multiple multithreaded server processes
- Lock manager process
  - More on this later
- Database writer process
  - **Output** modified buffer blocks to disks continually
- Log writer process
  - Server processes :
    - simply **add** log records to log record **buffer**
  - Log writer process:
    - **outputs** log records to stable storage.
- Checkpoint process
  - Performs periodic **checkpoints**
- Process monitor process
  - **Monitors** other processes, and :
    - takes **recovery** actions if any of the other processes **fail**
    - E.g. aborting any transactions being executed by a server process and restarting it



- ***Shared memory*** contains shared data
  - Buffer pool
  - Lock table
  - Log buffer
  - **Cached** query plans (reused if same query submitted again)
  - **All** database processes can access ***shared memory***
- To ensure that **no** two processes are accessing :
  - the same data structure
  - at the same time,
  - databases systems implement:
    - **mutual exclusion** using either:

        - Operating system semaphores
        - Atomic instructions such as test-and-set

- To **avoid** overhead of
    - interprocess communication for **lock** request/grant,
        - each database process operates **directly** on the **lock table**
        - instead of sending requests to lock manager process
- **Lock manager** process still used for:
    - **deadlock** detection

Data Servers

- Used in **high-speed** LANs, in cases where:
    - The clients are comparable in processing power to the server
    - The tasks to be executed are **compute** intensive.
- Data are shipped to clients where:
    - processing is performed, and
    - then shipped results back to the server.
- This architecture requires **full** back-end functionality at the clients.
- Used in many **object-oriented** database systems
- Issues:
    - Page-Shipping *versus* Item-Shipping
    - Locking
    - Data Caching
    - Lock Caching
- **Page-shipping** versus **item-shipping**
    - Smaller unit of shipping ⇒ more messages
    - Worth **prefetching** related items along with requested item
    - Page shipping can be thought of as a form of **prefetching**
- Locking
    - Overhead of **requesting** and **getting locks** from server is high due to message delays
    - Can grant locks on requested and prefetched items;

- with page shipping, transaction is granted lock on whole page.
  - **Locks** on a prefetched item can be **called back** by the **server**,
    - and returned by **client transaction**:
      - if the prefetched item has **not** been used.
  - Locks on the page can be **deescalated** to locks on items in the page :
    - when there are lock **conflicts**.
    - Locks on ***unused*** *items* can then be **returned** to **server**.
- **Data Caching**
  - **Data** can be **cached** at client even in between transactions
  - But **check** that **data** is up-to-date before it is used (**cache coherency**)
  - Check can be done when **requesting** lock on data item
- **Lock Caching**
  - **Locks** can be **retained** by client system even in between transactions
  - Transactions can **acquire** cached locks locally,
    - without **contacting server**
  - **Server calls back** locks from clients:
    - when it **receives conflicting** lock request.
    - Client **returns** lock once **no** local transaction is using it.
  - Similar to **deescalation**, but **across** transactions

Parallel Systems

- **Parallel** database systems consist of:
    - multiple **processors** and
    - multiple **disks**
  - connected by a **fast** interconnection network.
- A **coarse-grain parallel** machine consists of:
  - a small number of ***powerful processors***
- A **massively parallel** or **fine grain parallel** machine utilizes:
  - **thousands** of ***smaller processors***.
- **Two** main performance measures:
  - **Throughput**:

- the **number of tasks** that can be **completed** in a given **time interval**

- **response time**:
  - the **amount of time** it takes to **complete** a **single task**
    - from the time it is submitted

Speed-Up **and** ***Scale-Up***

- **Speedup**:
  - a **fixed-sized** problem executing on a small system is:
    - given to a system which is ***N*-times** larger.
  - Measured by:

speedup = *small system elapsed time*

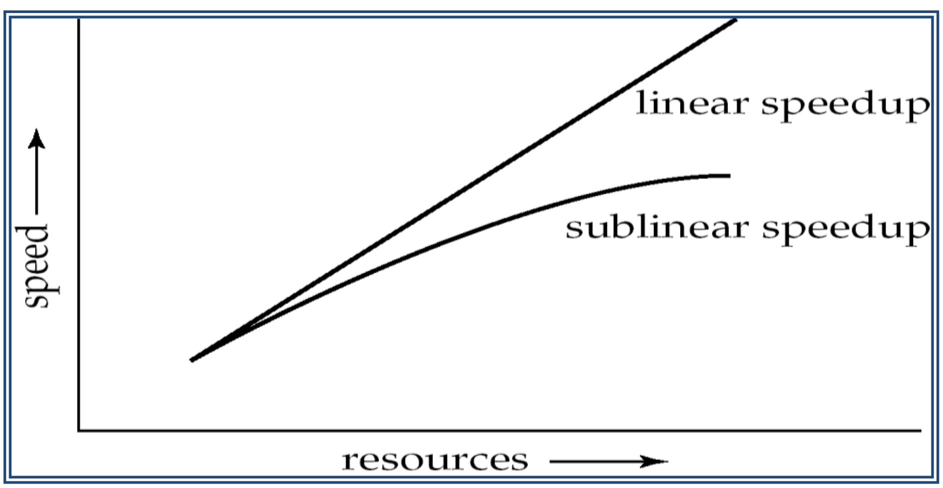
large system elapsed time

- Speedup is **linear** if **equation** equals **N**.
- **Scaleup**:
- **increase** the size of **both** the problem and the system
  - ***N*-times** larger system used to perform ***N*-times** larger job
  - Measured by:

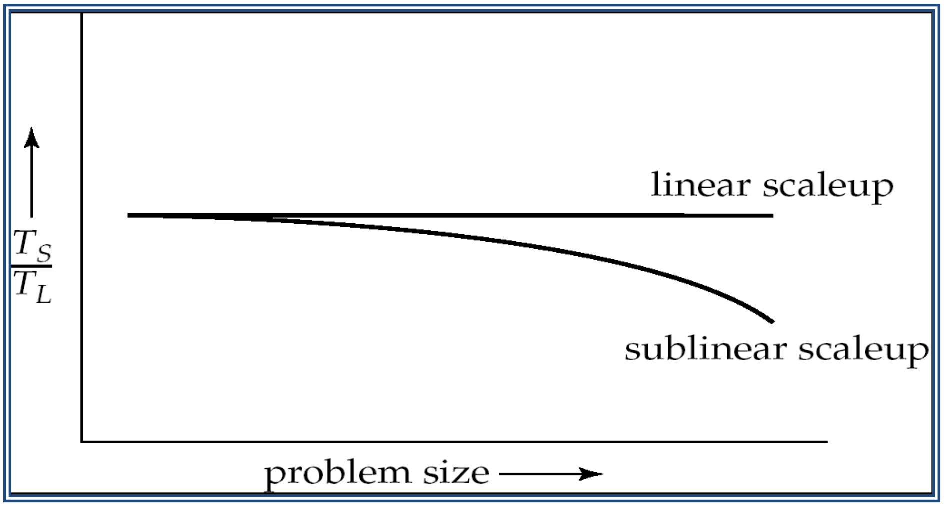
scaleup = *small system small problem elapsed time*

big *system* ***big*** *problem elapsed time*

- Scale up is **linear** if **equation** equals **1**.



Scaleup



***Batch* and *Transaction* Scaleup**

- **Batch scaleup:**
  - A **single** large job;
    - typical of *most* database queries and **scientific** simulation.
  - Use an *N*-times **larger computer**
    - on *N*-times **larger problem**.
- **Transaction scaleup**:
  - **Numerous** small queries submitted by independent users to a shared database;
    - typical transaction processing and timesharing systems.
  - ***N*-times** as many users submitting requests (hence, *N*-times as many requests)
    - to an ***N*-times** larger database,
    - on an ***N*-times** larger computer.
  - Well-suited to **parallel** execution.

Factors Limiting *Speedup* and *Scaleup*

Speedup and scaleup are often sublinear due to:

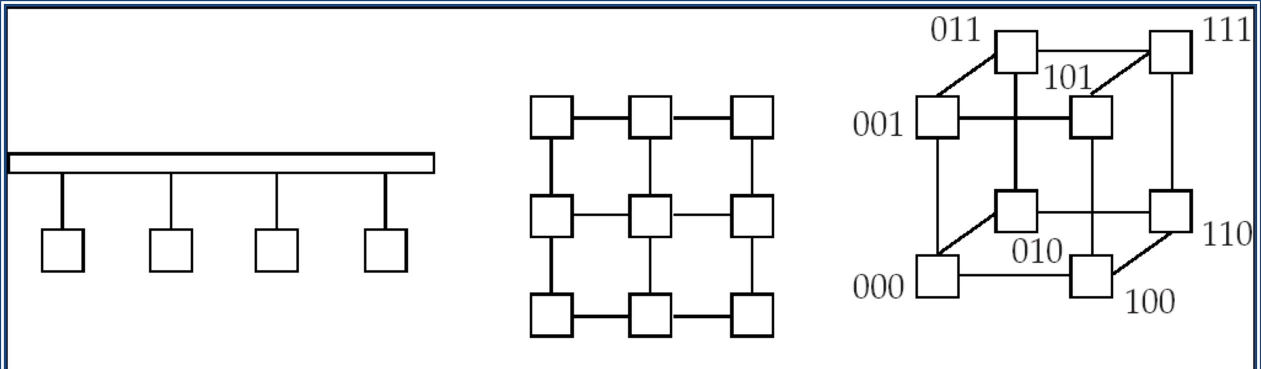
- **Startup** costs:
  - Cost of **starting up** multiple processes may **dominate** computation time,
    - **if** the **degree** of parallelism is **high**.
- **Interference**:
  - Processes accessing shared resources (e.g.,system bus, disks, or locks)

        - **compete** with each other,
            - thus **spending time** waiting on other processes,
            - rather than **performing** useful work.
- **Skew**:
    - **Increasing** the degree of parallelism
        - **increases** the variance in service times of parallely executing **tasks**.
        - Overall execution time determined by **slowest** parallel **tasks**.

Interconnection Network Architectures

- **Bus**.
    - System components ***send data*** on and ***receive data*** from
        - a **single** communication bus;
    - Does **not** scale **well** with *increasing parallelism*.
- **Mesh**.
    - Components are arranged as **nodes** in a **grid**, and
        - each component is **connected** to all adjacent components
    - Communication **links grow** with *growing number of components*, and
        - so scales **better**.
    - But may require **$2\sqrt{n}$ hops** to send message to a node
    - (or $\sqrt{n}$ with wraparound connections at edge of **grid**).
- **Hypercube**.
    - Components are numbered in binary;
        - components are connected to one another :
            - **if** their binary representations differ in exactly one bit.
    - ***n*** components are ***connected*** to ***log(n)*** other components and
        - can **reach** each other via at most ***log(n)*** links;
        - **reduces** communication delays.

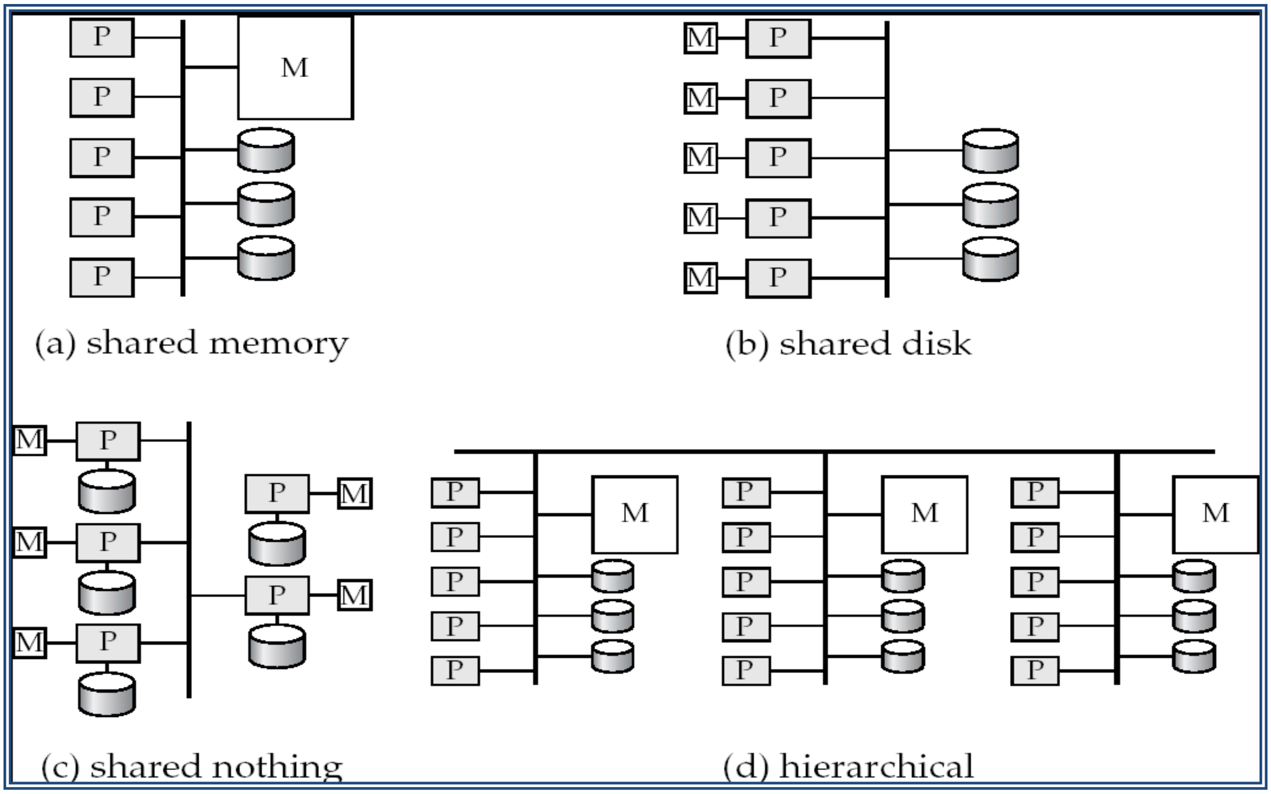
Interconnection Architectures



(a) bus (b) mesh (c) hypercube

Parallel Database Architectures

- **Shared memory** –
  - processors share a common memory
- **Shared disk** –
  - processors share a common disk
- **Shared nothing** –
  - processors share neither a common memory nor common disk
- **Hierarchical** –
  - **hybrid** of the above architectures



Shared *Memory*

- Processors and disks have access to a common memory,
    - typically via a bus
    - or through an interconnection network.
- Extremely **efficient** communication between processors —
    - **data** in shared memory can be **accessed** by any processor
        - without **having to** move it using software.
- Downside – architecture is **not** scalable beyond 32 or 64 processors
    - since the bus or the interconnection network becomes a **bottleneck**
- Widely used for **lower** degrees of parallelism (**4** to **8**).

Shared *Disk*

- **All** processors can directly **access all** disks
        - via an interconnection network,
        - but the processors **have *private memories***.

    - The memory bus is **not** a bottleneck
    - Architecture provides a degree of **fault-tolerance** —
        - if **a** processor **fails**, the other processors can **take over** its tasks
        - since the database is resident on disks
        - that are accessible from all processors.
- Ex: IBM Sysplex and DEC clusters (now part of Compaq)
        - running Rdb (now Oracle Rdb) were early commercial users
- Downside:
    - **bottleneck** now occurs at:
        - interconnection to the disk subsystem.
- Shared-disk systems **can scale** to a somewhat **larger number** of processors,
    - but communication between processors is **slower**.

Shared *Nothing*

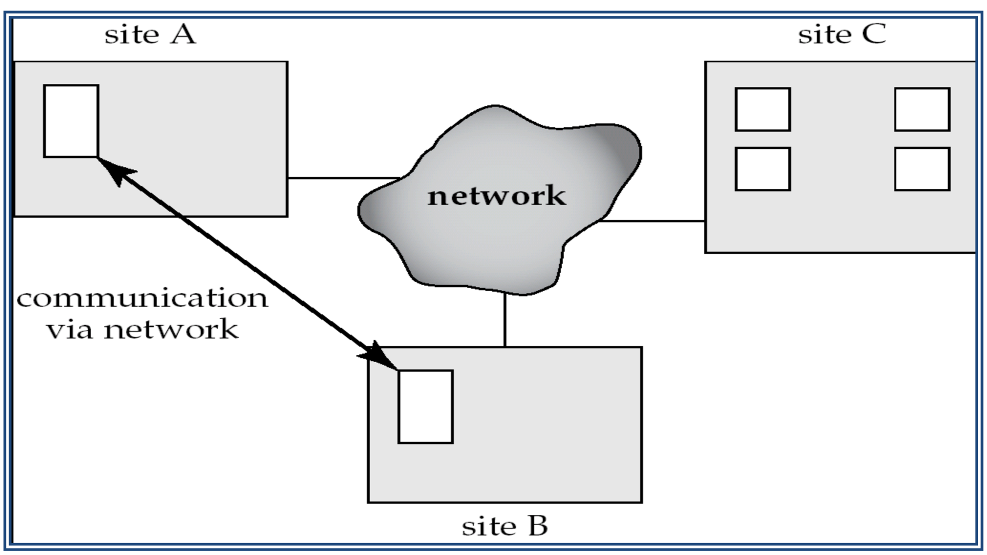
- Node consists of a processor, memory, and one or more disks.
- Processors at one **node** communicate with another processor at another **node**
    - using an interconnection network.
    - A **node** functions as the **server** for:
        - the **data** on the **disk** or disks the node **owns**.
- Ex: Teradata, Tandem, Oracle-n CUBE
- **Data** accessed from local disks (and local memory accesses)
    - do not pass through interconnection network,
    - thereby **minimizing** the interference of resource sharing.
- Shared-nothing multiprocessors :
    - can be **scaled up** to **thousands** of processors
        - without **interference**.
- Main **drawback**: cost of communication and non-local disk access;
    - sending data involves software interaction at both ends.

Hierarchical

- **Combines** characteristics of :
  - shared-memory, shared-disk, and shared-nothing architectures.
- **Top level** is a shared-nothing architecture –
  - **nodes** connected by an interconnection network, and
  - do **not** share disks or memory *with each other*.
- Each node of the system could be:
  - a shared-memory system with a few processors.
- Alternatively, each node could be:
  - a shared-disk system, and each of the systems sharing a set of disks
  - could be a shared-memory system.
- **Reduce** the **complexity** of programming such systems by:
  - **distributed virtual-memory** architectures
  - Also called **non-uniform** memory architecture **(NUMA)**

Distributed Systems

- **Data spread** over multiple machines
  - (also referred to as **sites** or **nodes)**.
- Network interconnects the machines
- **Data shared** by users on multiple machines



Distributed Databases

- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- Heterogeneous distributed databases
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality
- Differentiate between *local* and *global* transactions
  - A local transaction accesses data in the *single* site at which the transaction was initiated.
  - A global transaction either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

Trade-offs in Distributed Systems

- Sharing data – users at one site able to access the data residing at some other sites.
- Autonomy – each site is able to retain a degree of control over data stored locally.
- Higher system availability through redundancy — data can be replicated at remote sites, and system can function even if a site fails.
- Disadvantage: added complexity required to ensure proper coordination among sites.
  - Software development cost.
  - Greater potential for bugs.
  - Increased processing overhead.

Implementation Issues for Distributed Databases

- Atomicity needed even for transactions that update data at multiple sites
- The two-phase commit protocol (2PC) is used to **ensure** atomicity
  - Basic idea: each site **executes** transaction until just before commit, and
    - then **leaves** final decision to a **coordinator**
  - Each site must **follow** decision of coordinator,
    - even if there is a **failure** while waiting for coordinators decision
- **2PC** is not always appropriate:

- other transaction models based on:
  - persistent messaging, and workflows, are **also used**.
- Distributed concurrency control (and deadlock detection) **required**
- Data items may be **replicated** to **improve** data availability

Network Types

- **Local-area networks (**LANs) –
  - composed of processors that are distributed over **small** geographical areas,
    - such as a **single building**
    - or a **few** adjacent **buildings**.
- **Wide-area networks (**WANs) –
  - composed of processors distributed over a **large** geographical area.
- WANs with **continuous** connection (e.g. the Internet) are:
  - **needed** for implementing **distributed** database systems
- **Groupware** applications such as *Lotus notes*:
    - can **work** on WANs with **discontinuous** connection:
  - Data is replicated.
  - Updates are propagated to replicas periodically.
  - Copies of data may be updated independently.
  - Non-serializable executions can thus result. Resolution is application dependent.

DISTRIBUTED DATABASES

- **Heterogeneous** and **Homogeneous** Databases
- Distributed **Data** Storage
- Distributed **Transactions**
- Commit **Protocols**
- **Concurrency** Control in Distributed Databases
- **Availability**
- Distributed **Query** Processing

- **Heterogeneous** Distributed Databases
- **Directory** Systems

Distributed Database System

- A **distributed database system** consists of **loosely coupled sites** that **share no** physical **component**
- **Database** systems that **run** on **each site** are **independent** of each other
- **Transactions** may **access data** at one or **more sites**

Homogeneous Distributed Databases

- In a **homogeneous** distributed database:
    - **All sites** have **identical software**
    - **Are aware** of **each other** and agree to cooperate in processing user requests.
    - **Each site surrenders** part of **its autonomy** in terms of right to change schemas or software
    - **Appears** to user as a **single system**
- In a **heterogeneous** distributed database
    - **Different sites** may use **different schemas** and software
        - Difference in schema is a **major problem** for **query processing**
        - Difference in software is a major problem for **transaction processing**
    - **Sites** may **not** be **aware** of each other and may **provide** only limited facilities for cooperation in transaction processing

Distributed Data Storage

- Assume **relational** data **model**
- **Replication**:
    - System maintains **multiple copies** of data, stored in different sites, for **faster retrieval** and **fault tolerance**.
- **Fragmentation**:
    - Relation is partitioned into **several fragments** stored in distinct sites
- Replication and fragmentation **can be combined:**

- Relation is partitioned into several fragments: system maintains **several** identical **replicas** of each such **fragment**.

Data Replication

- A relation or fragment of a relation is **replicated** if it is **stored redundantly** in two or more sites.
- **Full replication** of a relation is the case where **the relation** is stored at **all sites**.
- **Fully redundant** databases are those in which **every site** contains a **copy** of the **entire database**.
- **Advantages** of **Replication**
    - **Availability**: failure of site containing relation ***r*** does not result in unavailability of ***r*** is replicas exist.
    - **Parallelism**: queries on ***r*** may be processed by several nodes in parallel.
    - **Reduced data transfer**: relation ***r*** is available locally at each site containing a replica of ***r***.
- **Disadvantages** of **Replication**
    - **Increased cost of updates**: each replica of relation ***r*** must be updated.
    - **Increased complexity** of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless **special** concurrency control **mechanisms** are implemented.
        - One **solution**: **choose** one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- **Division** of relation r into fragments $r_1, r_2, \ldots, r_n$ which contain sufficient information to reconstruct relation **r**.
- **Horizontal fragmentation**: each tuple of ***r*** is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation ***r*** is split into several smaller schemas
    - All schemas must contain a common candidate key (or **superkey**) to ensure **lossless join** property.

- A special attribute, the **tuple-id** attribute may be added to each schema to serve as a candidate key.

- **Example** : relation **account** with following schema
- *Account* = (*branch_name*, *account_number*, *balance*)

Horizontal Fragmentation of *account* Relation

branch_name	account_number	balance
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1 = \sigma_{branch\_name=\text{"Hillside"}}(account)$

branch_name	account_number	balance
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2 = \sigma_{branch\_name=\text{"Valleyview"}}(account)$

Vertical Fragmentation of *employee_info* Relation

branch_name	customer_name	tuple_id
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}(employee\_info)$

account_number	balance	tuple_id
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}(employee\_info)$

Advantages of Fragmentation

- **Horizontal**:
  - **allows parallel processing** on fragments of a relation
  - **allows** a **relation** to be **split** so that **tuples** are **located** where they are most frequently accessed
- **Vertical**:
  - **allows tuples** to be **split** so that each part of the tuple is stored where it is most frequently accessed
  - **tuple-id** attribute allows **efficient joining** of vertical fragments
  - **allows parallel processing** on a relation
- Vertical and horizontal fragmentation can be **mixed**.
  - Fragments may be successively fragmented to an arbitrary depth.

Data Transparency

- **Data transparency**: Degree to which system **user** may remain **unaware** of the details of how and where the **data** items are **stored** in a distributed system
- Consider transparency **issues** in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

Naming of Data Items – Criteria

1. Every data item must have a **system-wide unique name**.
2. It should be possible to **find** the location of **data items efficiently**.
3. It should be possible to **change** the location of **data items transparently**.
4. Each site should be **able** to **create** new **data items autonomously**.

Centralized Scheme - Name Server

- **Structure**:
  - name **server assigns all** names
  - each **site maintains** a record of **local** data items
  - **sites ask** name **server** to **locate non-local** data items
- **Advantages**:
  - **satisfies** naming criteria 1-3
- **Disadvantages**:
  - does **not** satisfy naming criterion 4
  - name **server** is a potential performance **bottleneck**
  - name **server** is a single point of **failure**

Use of Aliases

- **Alternative** to centralized scheme: each site **prefixes** its own site identifier to any name that it generates **i.e.,** ***site* 17.*account*.**
  - **Fulfills** having a **unique** identifier, and **avoids** problems associated with central control.
  - However, **fails** to achieve network **transparency**.

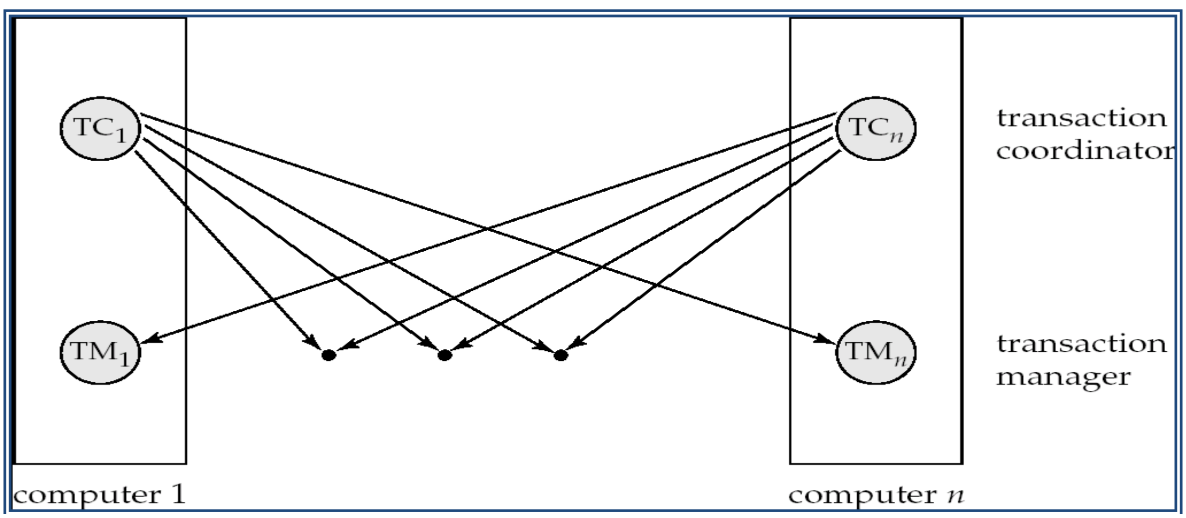
- **Solution**: Create a set of **aliases** for data items; Store the mapping of aliases to the **real names** at each site.

The **user** can be **unaware** of the physical location of a data item, and is unaffected if the data item is moved from one site to another.

Distributed Transactions

- **Transaction** may **access data** at several sites.
- Each site has a **local** transaction **manager** responsible for:
    - **Maintaining** a **log** for recovery purposes
    - **Participating** in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction **coordinator**, which is responsible for:
    - **Starting** the **execution** of transactions that originate at the site.
    - **Distributing subtransactions** at appropriate sites for execution.
    - **Coordinating** the **termination** of each transaction that originates at the site, which may result in the transaction being committed at **all** sites or aborted at **all** sites.

Transaction System Architecture



System Failure Modes

- Failures unique to distributed systems:
    - Failure of a site.
    - Loss of massages
        - Handled by network transmission control protocols such as TCP-IP
    - Failure of a communication link
        - Handled by network protocols, by routing messages via alternative links
    - Network partition
        - A network is said to be partitioned when it has been split into two or more subsystems that lack any connection between them
            - Note: a subsystem may consist of a single node

- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to **ensure atomicity** across sites
  - a transaction which executes at multiple sites **must** either be **committed** at **all** the **sites**, or **aborted** at **all** the **sites**.
  - **not acceptable** to have a transaction committed at one site and aborted at another
- The ***two-phase commit*** (2PC) protocol is **widely** used
- The ***three-phase commit*** (3PC) protocol is more **complicated** and more **expensive**, but avoids some drawbacks of two-phase commit protocol. This protocol is **not used** in practice.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop model** – failed sites simply stop working, and do **not cause any** other **harm**, such as sending incorrect messages to other sites.
- Execution of the protocol is **initiated** by the **coordinator** after the last step of the transaction has been reached.
- The protocol **involves all** the local **sites** at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction **coordinator** at S_i be C_i

Phase 1: Obtaining a Decision

- Coordinator C_i **asks** all participants to ***prepare*** to **commit** transaction T_i.
  - C_i **adds** the records <**prepare** *T*> to the log and **forces** log to stable storage
  - **sends prepare** *T* messages to **all** sites at which T executed
- Upon receiving message, **transaction manager** at each site **determines** if it can **commit** the transaction
  - **if not**, **adds** a record **<no *T*>** to the log and **sends abort *T*** message to C_i
  - **if** the transaction **can** be **committed**, then:
  - **adds** the record **<ready *T*>** to the log
  - **forces** *all records* for T to stable storage
  - **sends ready** *T* message to C_i

Phase 2: Recording the Decision

- T can be committed **if** C_i received a **ready *T*** message from **all** the participating **sites**: otherwise T must be aborted.
- Coordinator C_i **adds** a decision record, **<commit *T*>** or **<abort *T*>**, to the log and **forces** record onto stable storage. Once the record on stable storage it is **irrevocable** (even if failures occur)
- Coordinator C_i **sends** a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate **action** locally.

Handling of Failures - Site Failure

When site S_k recovers, it **examines** its log to determine the **fate** of transactions active at the time of the failure.

- Log contain **<commit *T*>** record: site executes **redo** (*T*)

- Log contains **<abort *T*>** record: site executes **undo** (*T*)
- Log contains **<ready *T*>** record: site **must consult C_i** to determine the **fate** of ***T***.
  - If ***T*** committed, **redo** (*T*)
  - If ***T*** aborted, **undo** (*T*)
- The log contains **no control records** concerning ***T*** replies that S_k failed before responding to the **prepare *T*** message from C_i
  - since the failure of S_k precludes the sending of such a response C_i **must abort *T***
  - S_k **must** execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator C_i **fails** while the commit protocol for ***T*** is executing then **participating sites must** decide on ***T***'s **fate**:
  1. If an active site **contains** a <**commit** *T*> record in its log, then ***T* must** be committed.
  2. If an active site **contains** an <**abort** *T*> record in its log, then ***T* must** be aborted.
  3. If some active participating site does **not contain** a **<ready *T*>** record in its log, then the failed coordinator C_i **cannot have decided** to commit ***T***. Can therefore **abort *T***.
  4. If none of the above cases holds, then **all** active sites must **have** a **<ready *T*>** record in their logs, but **no additional control records** (such as <**abort** *T*> of <**commit** *T*>). In this case active sites **must** wait for C_i to recover, to find decision.
- **Blocking problem** : active sites may have to wait for failed coordinator C_i to recover.

Handling of Failures - Network Partition

- If the **coordinator** and all its **participants** remain **in one partition**, the failure has no effect on the commit protocol.
- If the **coordinator** and its participants belong to **several partitions**:
  - **Sites** that are **not in** the partition containing the **coordinator think** the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - **No harm results**, but sites may still **have to wait** for decision from **coordinator**.
- The **coordinator** (and the sites that are in the **same partition** as the coordinator) **think** that the **sites** in the **other partition have failed**, and follow the usual commit protocol.
    - Again, **no harm results**

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
  - The **recovering site must determine** the **commit-abort status** of such transactions by contacting other sites; this **can slow** and potentially **block recovery**.
- **Recovery algorithms** can **note** lock information in the log.

- Instead of **<ready *T*>**, write out **<ready *T, L*>** *L* = **list of locks** held by ***T*** when the log is written (read locks can be omitted).
- For every in-doubt transaction ***T***, **all** the **locks noted** in the **<ready *T, L*>** log record **are reacquired**.

- After **lock reacquisition**, transaction processing can resume; the commit or rollback of in-doubt transactions is performed **concurrently** with the execution of **new transactions**.

Alternative Models of Transaction Processing

- Notion of **a single transaction** spanning multiple sites **is inappropriate** for many applications
  - E.g. transaction **crossing** an **organizational boundary**
  - No organization would **like to permit** an **externally initiated transaction** to **block local transactions** for an indeterminate period
- **Alternative models carry out** transactions by **sending messages**
  - Code to **handle messages** must be carefully designed to **ensure atomicity** and **durability** properties for updates
    - **Isolation cannot** be guaranteed, in that **intermediate stages** are **visible**, but code **must ensure no inconsistent states** result due to concurrency
  - **Persistent messaging systems** are systems that **provide transactional properties** to messages
    - **Messages** are guaranteed to be **delivered** exactly **once**
    - Will discuss implementation **techniques** later
- Motivating example: **funds transfer** between two banks
  - **Two phase commit** would have the potential to **block updates** on the accounts involved in funds transfer
  - Alternative solution:
    - **Debit** money from source account **and send a message** to other site
    - Site **receives message and** credits destination account
  - **Messaging** has long been used for **distributed transactions** (even before computers were invented!)
- **Atomicity** issue:
  - **Once transaction** sending a message is **committed**, **message must** be guaranteed to **be delivered**
    - Guarantee as long as **destination site** is **up and reachable**, code to **handle undeliverable messages** must also be available
      - e.g. **credit money back** to source account.
  - If sending **transaction aborts**, **message** must **not** be sent

Error Conditions with Persistent Messaging

- Code to handle messages has to **take care** of **variety of failure** situations (even **assuming** guaranteed message delivery)
  - E.g. if destination account does **not exist**, failure message **must** be **sent back** to source site
  - When failure message is received from destination site, **or** destination site itself does **not exist**, **money must** be **deposited back** in source account
    - **Problem** if source account has been **closed**

- – **get humans** to **take care** of problem
- **User** code executing transaction processing using **2PC** does **not have** to **deal** with such failures
- There are many situations where **extra effort** of **error handling is worth** the benefit of **absence of blocking**
  - E.g. pretty much all **transactions across** organizations

Persistent Messaging and Workflows

- **Workflows** provide a **general model** of transactional processing involving **multiple sites** and possibly **human** processing of certain steps
  - E.g. when a bank receives a **loan** application, it may need to
    - **Contact** external **credit-checking** agencies
    - Get **approvals** of one or more **managers** and then **respond** to the **loan** application
  - **Persistent messaging** forms the **underlying** infrastructure for **workflows** in a distributed environment

Concurrency Control

- Modify **concurrency control schemes** for use in **distributed environment**.
- We **assume** that **each site** participates in the execution of a **commit** protocol to **ensure global** transaction **automicity**.
- We **assume** all **replicas** of any item are updated
  - Will see how to **relax** this in case of site failures later

Single Lock-Manager Approach

- System maintains a ***single*** **lock manager** that resides in a ***single*** chosen **site**, say S_i
- When a transaction needs to **lock** a **data item**, it **sends** a **lock request** to S_i and **lock manager determines** whether the lock can be **granted** immediately
  - If **yes**, lock manager **sends** a **message** to the site which initiated the request
  - If **no**, **request** is **delayed** until **it can be** granted, at which time a message is sent to the initiating site
- The transaction **can read** the data item from ***any*** one of the **sites** at which a replica of the data item resides.
- **Writes must** be performed on **all replicas** of a data item
- **Advantages** of scheme:
  - Simple **implementation**
  - Simple **deadlock** handling
- **Disadvantages** of scheme are:
  - **Bottleneck**: lock manager **site** becomes a **bottleneck**
  - **Vulnerability**: system is vulnerable to lock manager **site failure**.

Distributed Lock Manager

- In this approach, functionality of **locking** is implemented by **lock managers** at **each site**
  - Lock managers **control access** to **local data** items
    - But **special protocols** may be used for **replicas**
- **Advantage**: **work** is **distributed** and can be made **robust** to failures

- **Disadvantage**: **deadlock detection** is more complicated
  - Lock managers **cooperate** for deadlock detection
    - More on this later
- Several **variants** of this approach
  - **Primary** copy
  - **Majority** protocol
  - **Biased** protocol
  - **Quorum** consensus

Primary Copy

- **Choose one replica** of data item to be the **primary copy**.
  - **Site containing** the replica is called the **primary site** for that data item
  - **Different** data **items** can have **different primary sites**
- When a transaction needs to **lock** a data item ***Q***, it requests a lock at the **primary site** of ***Q***.
  - Implicitly gets lock on **all replicas** of the data item
- **Benefit:**
  - **Concurrency control** for replicated data handled **similarly** to unreplicated data - **simple** implementation.
- **Drawback:**
  - **If** the **primary site** of ***Q*** **fails**, ***Q*** is **inaccessible** even though **other** sites containing a **replica** may be accessible.

Majority Protocol

- **Local lock manager** at each site administers **lock** and **unlock** requests for data items stored at **that site**.
- When a **transaction** wishes to lock an **unreplicated** data item ***Q*** residing at site $\mathbf{S}_i$, a message is sent to $\mathbf{S}_i$'s **lock manager**.
  - **If *Q*** is **locked** in an **incompatible** mode, then the **request** is **delayed** until it can be granted.
  - **When** the **lock** request **can be granted**, the lock manager **sends** a message back to the initiator indicating that the lock request has been granted.
- In case of **replicated data**
  - If ***Q*** is replicated at **n sites**, then a lock request **message** must be **sent** to **more than half** of the **n sites** in which ***Q*** is stored.
  - The transaction does **not operate** on ***Q*** **until** it has **obtained** a lock on a **majority** of the replicas of ***Q***.
  - When **writing** the data item, transaction performs **writes** on ***all*** **replicas**.
- **Benefit**
  - **Can** be used even when some **sites** are **unavailable**
    - n details on how handle **writes** in the presence of **site failure** later
- **Drawback**
  - **Requires** $\mathbf{2(}n\mathbf{/2 + 1)}$ messages for handling **lock** requests, and $\mathbf{(}n\mathbf{/2 + 1)}$ messages for handling **unlock** requests.
  - **Potential** for **deadlock** even with **single item** - e.g., each of **3** transactions may have **locks** on **1/3rd of** the **replicas** of a data.

Biased Protocol

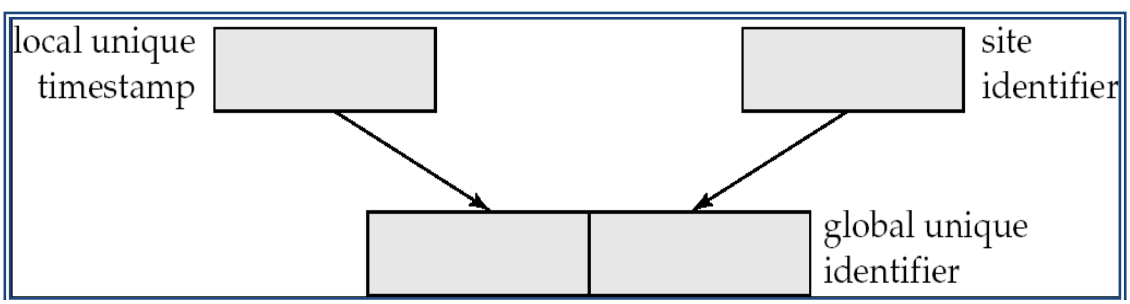
- **Local lock manager** at each site as in majority protocol, however, **requests** for **shared locks** are handled differently than requests for **exclusive locks**.
- **Shared locks**. When a transaction needs to lock data item ***Q***, it simply requests a lock on ***Q*** from the lock manager at **one site** containing a replica of *Q*.
- **Exclusive locks**. When transaction needs to lock data item ***Q***, it requests a lock on ***Q*** from the lock manager at **all sites** containing a replica of *Q*.
- Advantage - imposes **less overhead** on **read** operations.
- Disadvantage - **additional overhead** on **writes**

Quorum Consensus Protocol

- A **generalization** of both **majority** and **biased** protocols
- **Each site** is assigned a **weight**.
  - Let **S** be the **total** of **all** site **weights**
- Choose two values **read quorum Q_r** and **write quorum Q_w**
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
  - **Quorums** can be chosen (and **S** computed) separately for **each item**
- **Each read** must lock enough replicas that the **sum** of the site weights is **>= Q_r**
- **Each write** must lock enough replicas that the **sum** of the site weights is **>= Q_w**
- For now we assume **all replicas** are **written**
  - Extensions to allow some **sites** to be **unavailable** described later

Timestamping

- **Timestamp based** concurrency-control protocols can be used in **distributed** systems
- Each **transaction** must be given a **unique** timestamp
- Main **problem**: how to **generate** a **timestamp** in a distributed fashion:
  - **Each site** generates a **unique local** timestamp using either a logical **counter** or the local **clock**.
  - **Global unique** timestamp is **obtained** by **concatenating** the unique **local** timestamp with the unique **identifier**.



- A site S_i with a **slow clock** will assign smaller timestamps
  - Still logically correct: **serializability not affected**
  - **But**: "**disadvantages**" transactions of the site S_i
- To **fix** this problem:

- Define within each site S_i a **logical clock** (LC_i), which generates the **unique local** timestamp
- Require that S_i **advance** its logical clock whenever a request is received from a transaction T_i with timestamp $< x,y>$ and **x** is greater that the current value of LC_i.
- In this case, site S_i **advances** its logical clock to the value $x + 1$.

Replication with Weak Consistency

- Many **commercial** databases **support** replication of data with **weak** degrees of **consistency** (I.e., **without** a guarantee of **serializabiliy**)
- **E.g.: master-slave replication**: **updates** are performed at a **single "master"** site, and propagated to **"slave" sites**.
  - **Propagation** is not part of the **update transaction**: it is decoupled
    - May be **immediately** after transaction **commits**
    - May be **periodic**
  - Data may **only** be **read** at **slave sites**, not updated
    - **No need** to obtain **locks** at any remote site
  - Particularly **useful** for **distributing information**
    - E.g. from central office to branch-office
  - Also **useful** for running **read-only** queries **offline** from the main database
- Replicas **should see** a **transaction-consistent snapshot** of the database
  - That is, a **state of the database** reflecting **all** effects of **all** transactions up to **some point** in the serialization order, and no effects of any later transactions.
- E.g. **Oracle** provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a **remote** site:
  - **Snapshot** refresh either by **recomputation** or by **incremental** update
  - **Automatic** refresh (**continuous** or **periodic**) or **manual** refresh

Multimaster and Lazy Replication

- With **multimaster replication** (also called **update-anywhere** replication) **updates** are **permitted at any replica**, and are automatically **propagated** to **all** replicas
  - **Basic model:** in distributed databases, where:
    - **transactions** are **unaware** of the details of replication, and
    - **database** system **propagates** updates **as part of** the same transaction
    - Coupled with 2 phase commit
- Many systems **support lazy propagation** where:
  - **Updates** are **transmitted** after transaction commits
  - **Allows** updates to occur even if some **sites** are **disconnected** from the network,
  - **But** at the cost of **consistency!**

Deadlock Handling

Consider the following **two transactions** and history, with item **X** and transaction T_1 at site 1, and item Y and transaction T_2 at site 2:

T_1: write (X)
write (Y)

T_2: write (Y)
write (X)

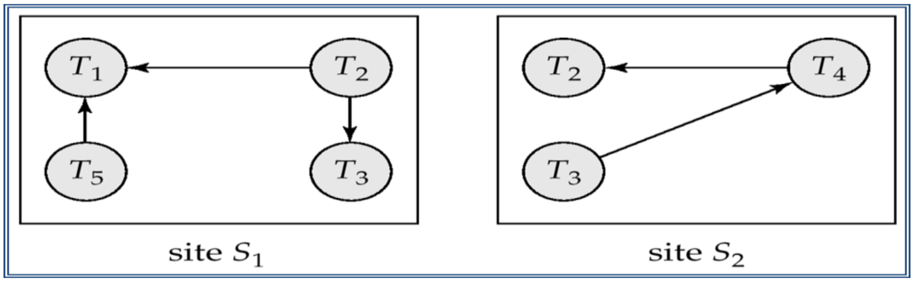
X-lock on X write (X)	X-lock on Y write (Y) wait for X-lock on X

Result: **deadlock** which **cannot** be detected **locally** at either site

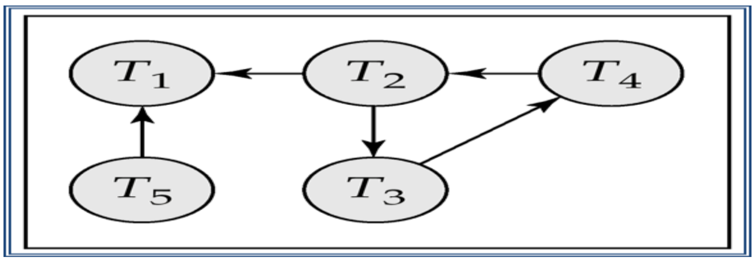
Centralized Approach

- A **global wait-for graph** is constructed and maintained in a ***single*** **site**; the deadlock-detection **coordinator**
    - ***Real graph***: Real, but **unknown**, state of the system.
    - ***Constructed graph***: **Approximation generated** by the controller during the execution of its algorithm .
- the **global wait-for graph** can be **constructed** when:
    - a **new edge** is **inserted** in or **removed** from one of the **local wait-for graphs**.
    - a number of **changes** have **occurred** in a **local wait-for graph**.
    - the **coordinator** needs to **invoke cycle-detection**.
- If the **coordinator** finds a cycle, it selects a **victim** and **notifies** all sites. The sites **roll back** the **victim** transaction.

Local and Global Wait-For Graphs



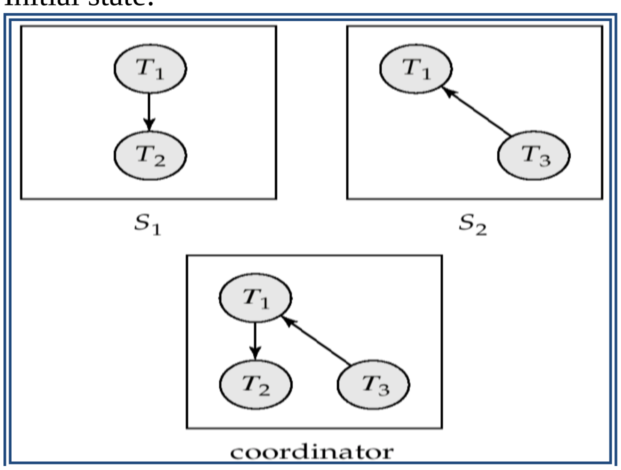
Local



Global

Example Wait-For Graph for False Cycles

Initial state:



- Suppose that starting from the state shown in figure,
  1. ***T*$_2$ releases resources** at ***S*$_1$**
     - resulting in a message **remove *T*$_1$ → *T*$_2$** message from the Transaction Manager at site ***S*$_1$** to the coordinator)
  2. And then ***T*$_2$ requests a resource** held by ***T*$_3$** at site ***S*$_2$**
     - resulting in a message **insert *T*$_2$ → *T*$_3$** from ***S*$_2$** to the coordinator
- Suppose further that the **insert** message **reaches before** the **delete** message
  - this can **happen** due to **network delays**

- The **coordinator** would then **find** a **false cycle**

  $$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$
- The **false cycle** above **never existed** in reality.
- **False cycles cannot occur** if **two-phase locking** is **used**.

Unnecessary Rollbacks

- **Unnecessary rollbacks** may result when deadlock has indeed occurred and a **victim** has been **picked**, and **meanwhile another** of the **transactions** was **aborted** for reasons unrelated to the deadlock.
- Unnecessary rollbacks **can result from false cycles** in the global wait-for graph;
  - however, **likelihood** of **false cycles** is **low**.

Availability

- **High availability**: time for which system is not fully usable should be extremely low (e.g. **99.99% availability**)
- **Robustness**: ability of system to **function** in spite of **failures** of **components**
- **Failures** are **more** likely in **large** distributed systems
- To **be robust**, a distributed system **must:**
  - **Detect** failures
  - **Reconfigure** the system so computation may continue
  - **Recover/reintegrate** when a site or link is repaired
- **Failure detection**: distinguishing **link failure** from **site failure** is **hard**
  - (partial) **solution**: have **multiple links**, multiple link failure is likely a site failure

Reconfiguration

- **Reconfiguration**:
  - **Abort** all **transactions** that were active at a **failed site**
    - **Making** them **wait** could **interfere with other** transactions since they may **hold locks** on other sites
    - However, in case **only some replicas** of a data item failed, it may be **possible** to continue transactions that **had accessed** data at a failed site (more on this later)
  - If **replicated data** items were **at failed site**, update system catalog to **remove** them from the **list** of replicas.
    - This should be **reversed** when failed site **recovers**, but additional **care** needs to be taken to bring values **up to date**
  - If a **failed site** was a **central server** for some subsystem, an **election** must be **held** to determine the **new server**
    - E.g. **name server**, concurrency **coordinator**, global **deadlock detector**
- Since **network partition** may **not** be **distinguishable** from **site failure**, the following situations **must** be **avoided:**
  - **Two** or more **central servers** elected in distinct partitions
  - **More** than one **partition** updates a replicated data item
- **Updates** must be able to **continue** even if some **sites** are **down**
- **Solution**: **majority based approach**
  - Alternative of "read **one** write **all available**" is tantalizing but **causes problems**

Majority-Based Approach

- The **majority protocol** for distributed concurrency control can be modified to **work** even if some **sites** are **unavailable**
  - **Each replica** of each item has a **version number** which is **updated** when the replica is updated, as outlined below:
  - A **lock request** is sent to at least ½ the **sites** at which item replicas are stored and **operation continues only** when a **lock** is **obtained** on a **majority** of the sites
  - **Read** operations **look** at **all replicas** locked, and **read** the value from the replica with **largest** version number
    - **May write** this **value** and version number **back** to replicas with lower version numbers (**no need** to obtain **locks** on **all replicas** for this task)
  - **Write** operations
    - **find** highest version number like reads, and set new version number to old highest version **+ 1**
    - **Writes** are then **performed** on all locked replicas and version number on these replicas is set to new version number
  - **Failures** (network and site) cause **no problems** as long as:
    - Sites at commit contain a **majority** of replicas of any updated data items
    - During reads a **majority** of replicas are **available** to find version numbers
    - Subject to above, 2 phase commit can be used to update replicas
  - Note: **reads** are **guaranteed** to see latest version of data item
  - **Reintegration is trivial**: **nothing** needs to be done
- **Quorum consensus** algorithm can be similarly extended

Read One Write All (Available?)

- **Biased protocol** is a **special case** of **quorum consensus**
  - Allows **reads** to read **any** one **replica** but **updates require all replicas** to **be available** at commit time (called read **one** write **all**)
- Read **one** write **all available** (**ignoring failed sites**) is **attractive**, **but incorrect:**
  - If **failed link** may **come** back **up**, **without** a disconnected **site** ever **being aware** that it was disconnected
  - The **site** then **has old values**, and a **read** from that site **would return** an **incorrect value**
  - If **site** was **aware** of failure **reintegration could** have been **performed**, **but no** way to **guarantee** this
  - With **network partitioning**, sites in **each partition may update** same item **concurrently**
    - **believing sites** in **other partition** have **all failed**

Site Reintegration

- **When** failed **site recovers**, it **must catch up** with **all updates** that it **missed** while it was down
  - Problem: **updates** may be **happening** to items whose replica is stored at the site **while** the **site** is **recovering**
  - Solution 1: **halt** all **updates** on system while **reintegrating** a **site**

        - **Unacceptable** disruption
    - Solution 2: **lock** all **replicas** of all data items **at** the **site**, update to latest version, then release locks
        - Other **solutions** with **better concurrency** also available

Comparison with Remote Backup

- **Remote backup systems** (**hot spare**) are also designed to **provide high availability**
- **Remote backup** systems are **simpler** and have **lower overhead**
    - **All actions** performed at a **single site**, and only **log** records **shipped**
    - **No need** for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can provide higher availability by having **multiple (> 2) replicas** and using the majority protocol
    - Also **avoid** failure detection and **switchover** time associated with **remote backup** systems

Coordinator Selection

- Backup **coordinators**
    - **site** which **maintains** enough information locally to **assume** the **role** of coordinator **if** the **actual coordinator fails**
    - **executes** the same algorithms and **maintains** the same internal state information
    - **executes** state information **as** the **actual** coordinator, **if** the **actual** coordinator **fails**
    - **allows fast recovery** from coordinator failure **but involves overhead** during normal processing.
- **Election** algorithms
    - used to **elect** a **new coordinator** in case of failures
    - Example: **Bully Algorithm** - applicable to systems where **every site** can send a message to every other site.

Bully Algorithm

- If site S_i sends a **request** that is **not answered** by the coordinator within a time interval T, **assumes** that the **coordinator** has **failed,** S_i tries to **elect itself** as the new coordinator:
    - S_i **sends** an election message to **every site** with a higher identification number,
    - S_i then **waits** for any of these **processes** to **answer** within T,
    - If **no response** within T, **assumes** that **all sites** with number **greater** than i have **failed**, S_i **elects itself** the new coordinator,
    - If **answer** is **received,** S_i **begins** time interval T', **waiting** to **receive** a message that **a site** with a **higher** identification number has been **elected**.
    - If **no message** is **received** within T', **assumes** the **site** with a **higher** number has **failed**; S_i **restarts** the algorithm.
- After a failed **site recovers**, it immediately **begins** execution of the same algorithm:
    - If there are **no active sites** with higher numbers, the **recovered site forces** all processes with lower numbers to **let** it **become** the **coordinator** site,
    - even if there is **a currently active coordinator** with a lower number.

Distributed Query Processing

- For **centralized systems**, the primary criterion for measuring the **cost** of a particular strategy is the **number of disk accesses**.
- In a **distributed system**, other issues must be taken into account:
  - The **cost** of a **data transmission** over the **network**.
  - The **potential gain** in performance from having **several sites** process **parts of the query** in **parallel**.

Query Transformation

- **Translating** algebraic **queries** on **fragments**.
  - It **must** be possible to **construct** relation ***r*** from its fragments
  - Replace relation ***r*** by the **expression** to **construct** relation ***r*** from its fragments
- Consider the **horizontal** fragmentation of the *account* relation into

$account_1 = \sigma_{branch\_name = \text{“Hillside”}} (account)$

$account_2 = \sigma_{branch\_name = \text{“Valleyview”}} (account)$

- The **query** $\sigma_{branch\_name = \text{“Hillside”}} (account)$ becomes

$\sigma_{branch\_name = \text{“Hillside”}} (account_1 \cup account_2)$

which is **optimized into**

$\sigma_{branch\_name = \text{“Hillside”}} (account_1) \cup \sigma_{branch\_name = \text{“Hillside”}} (account_2)$

- Since $account_1$ has **only** tuples **pertaining** to the Hillside branch, we can eliminate the selection operation.
- Apply the definition of $account_2$ to obtain

$$\sigma_{branch\_name = \text{“Hillside”}} (\sigma_{branch\_name = \text{“Valleyview”}} (account))$$

- This **expression** is the **empty** set regardless of the contents of the *account* relation.
- Final strategy is for the Hillside site to return $account_1$ as the result of the query.

Simple Join Processing

- Consider the following relational algebra expression in which the **three relations** are **neither** replicated **nor** fragmented

$$account \bowtie depositor \bowtie branch$$

- ***account*** is stored at site S_1
- ***depositor*** at S_2
- ***branch*** at S_3
- For a **query** issued **at site** S_I, the system needs to produce the **result at site** S_I

Possible Query Processing Strategies

- **Ship** copies of **all three** relations to **site** S_I and choose a strategy for processing the entire locally at site S_I.
- **Ship** a copy of the **account** relation to **site** S_2 and compute $temp_1 = account\ depositor$ *at* S_2. **Ship** $temp_1$ f
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following **factors**:
  - **amount of data** being **shipped**
  - **cost of transmitting** a **data block** between sites
  - relative **processing speed** at **each site**

Heterogeneous Distributed Databases

- Many database applications **require data** from a variety of **preexisting databases** located in a heterogeneous collection of hardware and software platforms
- **Data models** may differ (**hierarchical**, relational , etc.)
- **Transaction** commit **protocols** may be **incompatible**
- **Concurrency control** may be based on **different** techniques (locking, **timestamping**, etc.)
- **System-level** details almost certainly are totally **incompatible**.
- A **multidatabase system** is a **software layer** on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - Creates an **illusion** of logical **database integration** without any physical database integration

Advantages

- **Preservation** of **investment** in existing
  - hardware
  - system **software**
  - **Applications**
- **Local autonomy** and administrative control
- **Allows** use of **special-purpose DBMSs**
- **Step** towards a **unified homogeneous** DBMS
  - **Full integration** into a homogeneous DBMS faces:
    - Technical difficulties and **cost of conversion**
    - Organizational/political difficulties
      - Organizations do not want to **give up control** on their data
      - Local databases wish to retain a **great** deal of **autonomy**

Unified View of Data

- **Agreement** on a common **data model**
  - Typically the **relational** model
- **Agreement** on a common **conceptual schema**
  - **Different names** for **same** relation/attribute
  - **Same** relation/attribute **name** means **different things**
- **Agreement** on a single **representation** of shared data
  - E.g. data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- **Agreement** on **units of measure**
- Variations in **names**
  - E.g. Köln vs Cologne, Mumbai vs Bombay

Query Processing

- Several issues in **query** processing in a **heterogeneous** database
- **Schema translation**
  - Write a **wrapper** for each **data source** to translate data to a global schema
  - **Wrappers** must also **translate updates** on global schema to updates on local schema
- **Limited** query **capabilities**:
  - Some data sources allow only **restricted** forms of **selections**
    - E.g. web forms, **flat file** data sources
  - **Queries** have to be **broken** up and processed partly at the source and partly at a different site
- Removal of **duplicate information** when sites have overlapping information
  - **Decide which** sites to execute query
- **Global** query **optimization**

Mediator Systems

- **Mediator** systems are systems that **integrate** multiple heterogeneous data sources by providing an **integrated global view**, and providing **query facilities** on global view
  - Unlike full fledged multidatabase systems, **mediators** generally **do not** bother about **transaction processing**
  - But the terms **mediator** and **multidatabase** are sometimes used interchangeably
  - The term **virtual database** is also used to refer to mediator/multidatabase systems

Directory Systems

- Typical kinds of **directory information**
  - **Employee** information such as **name**, **id**, **email**, **phone**, office addr, ..
  - Even **personal** information to be accessed from multiple places
    - e.g. Web browser bookmarks
- **White** pages
  - Entries organized by **name** or **identifier**
    - Meant for forward lookup to find more about an entry
- **Yellow** pages
  - Entries organized by **properties**
  - For reverse lookup to find entries matching specific requirements
- When **directories** are to be **accessed across** an organization
  - Alternative 1: Web interface. Not great for programs
  - **Alternative 2**: Specialized **directory access protocols**
    - Coupled with specialized **user interfaces**

Directory Access Protocols

- Most commonly used directory access protocol:
  - **LDAP** (**L**ightweight **D**irectory **A**ccess **P**rotocol)
  - Simplified from earlier **X.500 protocol**
- Question: **Why not** use database protocols like **ODBC/JDBC**?
- **Answer**:

    - **Simplified protocols** for a **limited** type of **data access**, **evolved** parallel to ODBC/JDBC
    - Provide a **nice hierarchical naming** mechanism similar to file system directories
        - **Data** can be **partitioned** amongst **multiple servers** for different parts of the hierarchy, yet give a single view to user
            - E.g. different servers for **Bell Labs** Murray Hill and **Bell Labs** Bangalore
    - Directories **may use** databases as storage mechanism

LDAP: Lightweight Directory Access Protocol

- LDAP Data Model
- Data Manipulation
- Distributed Directory Trees

LDAP Data Model

- LDAP directories store **entries**
    - Entries are similar to objects
- Each entry must have unique **distinguished name (DN)**
- DN made up of a sequence of **relative distinguished names (RDNs)**
- E.g. of a DN
    - cn=Silberschatz, ou-Bell Labs, o=Lucent, c=USA
    - Standard RDNs (can be specified as part of schema)
        - cn: common name ou: organizational unit
        - o: organization   c: country
    - Similar to paths in a file system but written in reverse direction
- Entries can have attributes
    - Attributes are multi-valued by default
    - LDAP has several built-in types
        - Binary, string, time types
        - Tel: telephone number   PostalAddress: postal address
- LDAP allows definition of **object classes**
    - Object classes specify attribute names and types
    - Can use inheritance to define object classes
    - Entry can be specified to be of one or more object classes
        - No need to have single most-specific type
- Entries organized into a **directory information tree** according to their DNs
    - Leaf level usually represent specific objects
    - Internal node entries represent objects such as organizational units, organizations or countries
    - Children of a node inherit the DN of the parent, and add on RDNs
        - E.g. internal node with DN c = USA
            - Children nodes have DN starting with c=USA and further RDNs such as o or ou
        - DN of an entry can be generated by traversing path from root
    - Leaf level can be an alias pointing to another entry
        - Entries can thus have more than one DN

- E.g. person in more than one organizational unit

LDAP Data Manipulation

- Unlike SQL, LDAP does not define DDL or DML
- Instead, it defines a network protocol for DDL and DML
  - Users use an API or vendor specific front ends
  - LDAP also defines a file format
    - LDAP Data Interchange Format (LDIF)
- Querying mechanism is very simple: only selection & projection

LDAP Queries

- LDAP query must specify
  - Base: a node in the DIT from where search is to start
  - A search condition
    - Boolean combination of conditions on attributes of entries
      - Equality, wild-cards and approximate equality supported
  - A scope
    - Just the base, the base and its children, or the entire subtree from the base
  - Attributes to be returned
  - Limits on number of results and on resource consumption
  - May also specify whether to automatically dereference aliases
- LDAP URLs are one way of specifying query
- LDAP API is another alternative

LDAP URLs

- First part of URL specifis server and DN of base
  - ldap::/aura.research.bell-labs.com/o=Lucent,c=USA
- Optional further parts separated by ? symbol
  - ldap::/aura.research.bell-labs.com/o=Lucent,c=USA??sub?cn=Korth
  - Optional parts specify
    1. attributes to return (empty means all)
    2. Scope (sub indicates entire subtree)
    3. Search condition (cn=Korth)

Distributed Directory Trees

- Organizational information may be split into multiple directory information trees
  - Suffix of a DIT gives RDN to be tagged onto to all entries to get an overall DN
    - E.g. two DITs, one with suffix o = Lucent, c = USA
      and another with suffix o = Lucent, c = India
  - Organizations often split up DITs based on geographical location or by organizational structure
  - Many LDAP implementations support replication (master-slave or multi-master replication) of DITs (not part of LDAP 3 standard)
- A node in a DIT may be a **referral** to a node in another DIT

- E.g. Ou= Bell Labs may have a separate DIT, and DIT for o=Lucent may have a leaf with ou=Bell Labs containing a referral to the Bell Labs DIT
- Referalls are the key to integrating a distributed collection of directories
- When a server gets a query reaching a referral node, it may either
    - Forward query to referred DIT and return answer to client, or
    - Give referral back to client, which transparently sends query to referred DIT (without user intervention)