UNIT-1 Algorithms for Query Processing and Optimization

Syllabus

1.0. Introduction to Query Processing

1.1 Translating SQL Queries into Relational Algebra

1.2. Algorithms for External Sorting

1.3. Algorithms for SELECT and JOIN Operations

1.4. Algorithms for PROJECT and SET Operations

1.5. Implementing Aggregate Operations and Outer Joins

1.6. Combining Operations using Pipelining

1.7. Using Heuristics in Query Optimization

TEXT BOOKS:

1. "Fundamentals of Database Systems", Elmasri Navate, 5/e, Pearson Education.

2. Database systems, A practical approach to design, implementation & management, Thomas Connolly, Carolyn Begg, 4th edition, Pearson.

REFERENCES BOOK:

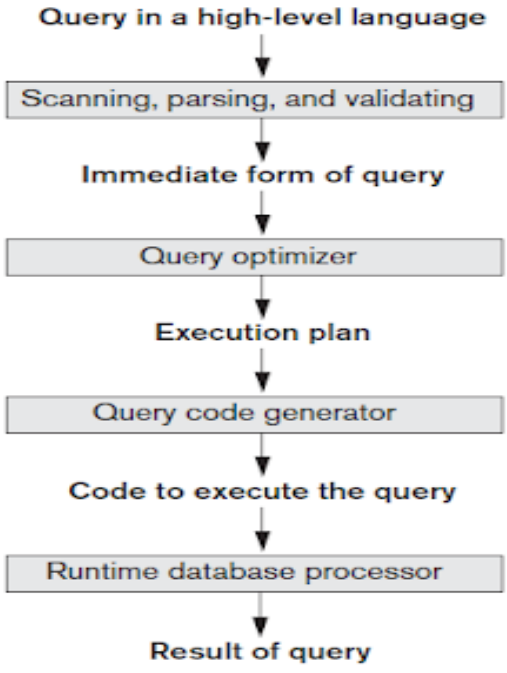
1. Getting started with No SQL Databases, Gaurav Vaish.

1.0. Introduction to Query Processing:

- Typical steps in processing a high-level query

1. Query in a high-level query language like SQL
2. Scanning, parsing, and validation
3. Intermediate-form of query like query tree

4. Query optimizer
5. Execution plan
6. Query code generator
7. Object-code for the query
8. Run-time database processor
9. Results of query



- A query expressed in a high-level query language such as SQL must be **scanned,parsed, and validate**.

- **Scanner**: identify the language tokens.
  Parser: check query syntax.

  Validate: check all attribute and relation names are valid.

- An internal representation (**query tree or query graph**) of the query is created after scanning, parsing, and validating.

- Then DBMS must devise an **execution strategy** for retrieving the result from the database files.

- How to choose a suitable (efficient) strategy for processing a query is known as **queryoptimization**.

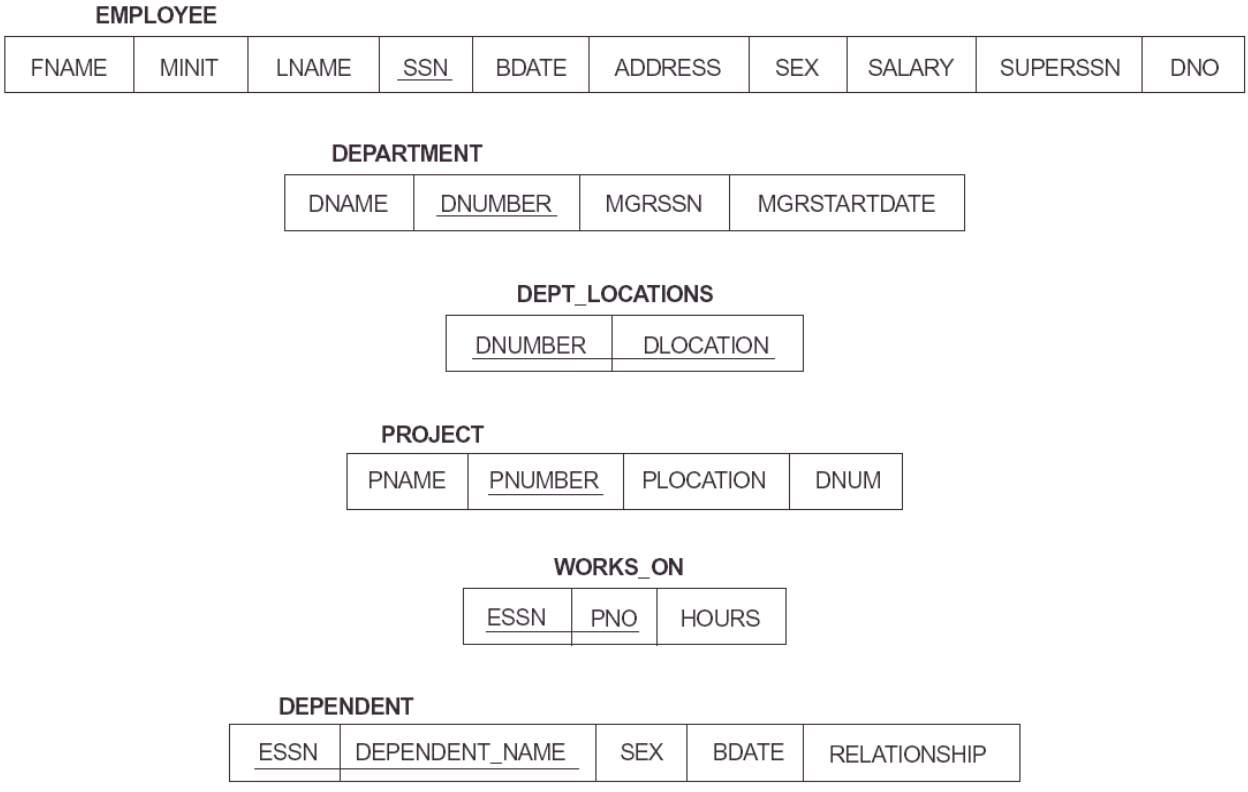
- The term optimization is actually a misnomer because in some cases the chosen **execution plan** is not the optimal strategy { it is just a reasonably e cient one.

- There are two main techniques for implementing query optimization.
- **1. Heuristic rules** for re-ordering the operations in a query.
  2. Systematically estimating the cost of different execution strategies and choosing the lowest cost estimate.

1.1. Translating SQL Queries into Relational Algebra

- **SQL query** is translated into an equivalent extended **relational algebra** expression --- represented as a **query tree**
- In order to transform a given query into a query tree, the query is decomposed into **query blocks**
- A query block contains a single **SELECT-FROM-WHERE** expression along with **GROUP-BY** and **HAVING** clauses.
- The **query optimizer** chooses an **execution plan** for each block

Figure 7.5 Schema diagram for the COMPANY relational database schema; the primary keys are underlined.



(a)

TABLE_NAME	COLUMN_NAME	NUM_DISTINCT	LOW_VALUE	HIGH_VALUE
PROJECT	PLOCATION	200	1	200
PROJECT	PNUMBER	2000	1	2000
PROJECT	DNUM	50	1	50
DEPARTMENT	DNUMBER	50	1	50
DEPARTMENT	MGRSSN	50	1	50
EMPLOYEE	SSN	10000	1	10000
EMPLOYEE	DNO	50	1	50
EMPLOYEE	SALARY	500	1	500

(b)

TABLE_NAME	NUM_ROWS	BLOCKS
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

INDEX_NAME	UNIQUENES	BLEVEL*	LEAF_ BLOCKS	DISTINCT _KEYS
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*BLEVEL is the number of levels without the leaf level.

Translating SQL Queries into Relational Algebra (1)

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > (  SELECT  MAX (SALARY)
                    FROM    EMPLOYEE
                    WHERE   DNO = 5);
```

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > C
```

$\pi_{\text{LNAME, FNAME}} (\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

```
SELECT  MAX (SALARY)
FROM    EMPLOYEE
WHERE   DNO = 5
```

$\mathcal{F}_{\text{MAX SALARY}} (\sigma_{\text{DNO=5}} (\text{EMPLOYEE}))$

SQL Queries and Relational Algebra (3)

- Example

  For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.

- SQL query:

```
SELECT   P.NUMBER,P.DNUM,E.LNAME, E.ADDRESS, E.BDATE
FROM     PROJECT AS P,DEPARTMENT AS D, EMPLOYEE AS E
WHERE    P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
         P.PLOCATION='STAFFORD';
```

- Relation algebra:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}(((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})) \underset{\text{DNUM=DNUMBER}}{} (\text{DEPARTMENT})) \underset{\text{MGRSSN=SSN}}{} (\text{EMPLOYEE}))$$

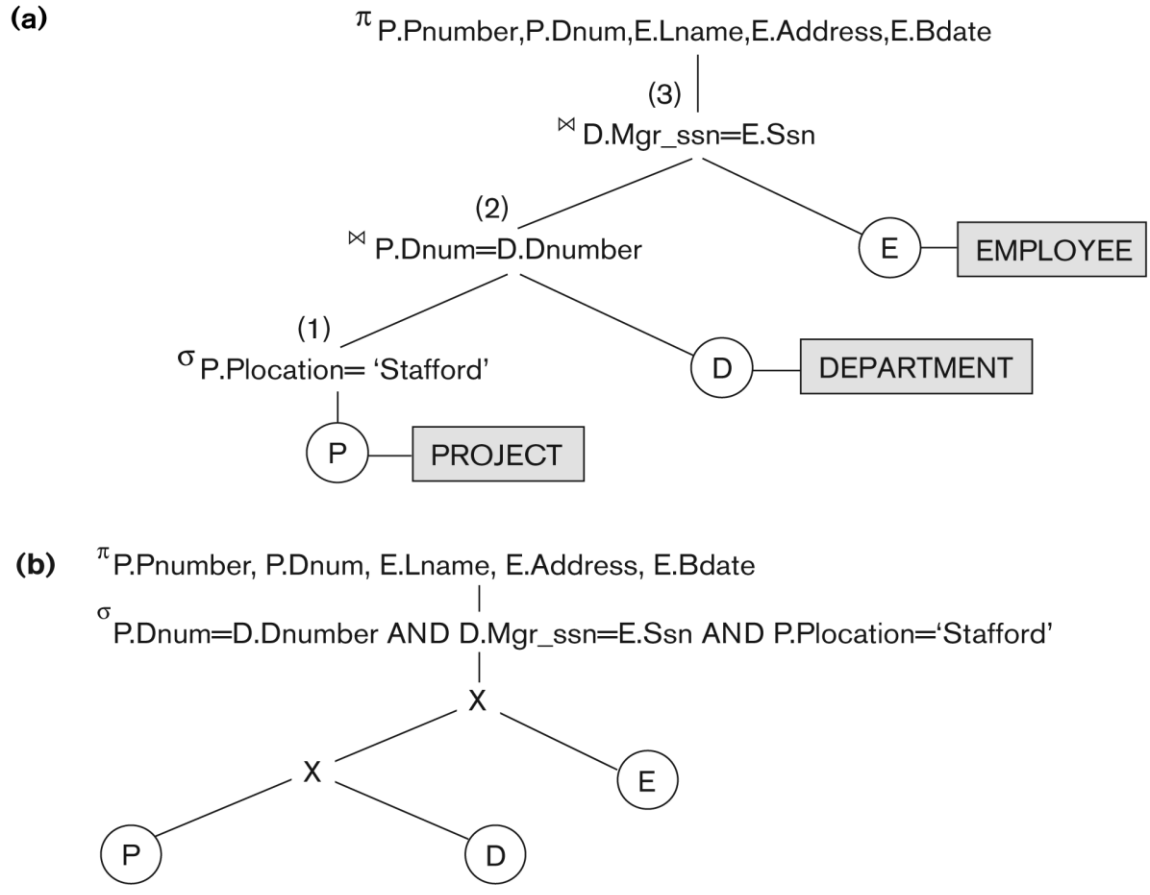


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

1.2 Algorithms for External Sorting

- An RDBMS must provide implementation(s) for all the required operations including relational operators and more

External sorting:

- Refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

Sort-Merge strategy:

- Starts by sorting small subfiles (**runs**) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.

Sorting phase

- Number of file blocks (b)
- Number of available buffers (n_B)
- Runs --- (b / n_B)

Merging phase --- passes

- Degree of merging --- the number of runs that are merged together in each pass

```
set   i ← 1;
      j ← b;              {size of the file in blocks}
      k ← nB;             {size of buffer in blocks}
      m ← ⌈(j/k)⌉;

{Sort Phase}
while (i ≤ m)
do {
      read next k blocks of the file into the buffer or if there are less than k blocks
         remaining, then read in the remaining blocks;
      sort the records in the buffer and write as a temporary subfile;
      i ← i + 1;
}

{Merge Phase: merge subfiles until only 1 remains}
set   i ← 1;
      p ← ⌈log(k−1) m⌉;   {p is the number of passes for the merging phase}
      j ← m;
while (i ≤ p)
do {
      n ← 1;
      q ← ⌈(j/(k−1))⌉;    {number of subfiles to write in this pass}
      while (n ≤ q)
      do {
            read next k−1 subfiles or remaining subfiles (from previous pass)
               one block at a time;
            merge and write as new subfile one block at a time;
            n ← n + 1;
      }
      j ← q;
      i ← i + 1;
}
```

Figure 15.2
Outline of the sort-merge algorithm for external sorting.

Analysis:

Number of file blocks = b

Number of initial runs = n_R

Available buffer space = n_B

Sorting phase: $n_R = \lceil (b/n_B) \rceil$

Degree of merging: $d_M = \text{Min}\ (n_B\text{-}1, n_R)$;

Number of passes: $n_P = \lceil (\log_{dM}(n_R)) \rceil$

Number of block accesses: $(2 * b) + (2 * b * (\log_{dM}(n_R)))$

For example:

5 initial runs [2;8;11];[4;6;7];[1;9;13];[3;12;15];[5;10;14].

The available bu ern_B= 3 blocks!d_M= 2 (two way merge)

After rst pass: 3 runs

[2;4;6; 7;8;11]; [1;3;9; 12;13;15]; [5;10;14]

After second pass: 2 runs

[1;2;3; 4;6;7; 8;9;11; 12;13;15]; [5;10;14]

After third pass:

[1;2;3; 4;5;6; 7;8;9; 10;11;12; 13;14;15]

1.3. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation:

Examples:

- (OP1): $\sigma_{SSN='123456789'}$(EMPLOYEE)
- (OP2): $\sigma_{DNUMBER>5}$(DEPARTMENT)
- (OP3): $\sigma_{DNO=5}$(EMPLOYEE)

- (OP4): $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX=F}(EMPLOYEE)$
- (OP5): $\sigma_{ESSN=123456789 \text{ AND } PNO=10}(WORKS\_ON)$

Search Methods for Simple Selection:

- S1. **Linear search** (brute force): Retrieve every *record* in the file, and test whether its attribute values satisfy the selection condition.
- S2. **Binary search**: If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used. (See OP1).

- S3. **Using a primary index or hash key** to retrieve a single record: If the selection condition involves an equality comparison on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record.
- S4. **Using a primary index** to retrieve multiple records: If the comparison condition is >, ≥ , <, or ≤ on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
- S5. **Using a clustering index** to retrieve multiple records: If the selection condition involves an equality comparison on a non-key attribute with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.
- S6. **Using a secondary (B+-tree) index** : On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key. In addition, it can be used to retrieve records on conditions involving >,>=, <, or <=. (FOR **RANGE QUERIES**)
- S7. **Conjunctive selection**: If an attribute involved in any single *simple condition* in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
- S8. **Conjunctive selection using a composite index**: If two or more attributes are involved in equality conditions in the conjunctive condition and a

composite index (or hash structure) exists on the combined field, we can use the index directly.

- S9. **Conjunctive selection by intersection of record pointers**: This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers). Each index can be used to retrieve the *record pointers* that satisfy the individual condition. The *intersection* of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

Implementing the JOIN Operation:

- Join (EQUIJOIN, NATURAL JOIN)
  - two–way join: a join on two files

    e.g. $R \rhd\lhd_{A=B} S$
  - multi-way joins: joins involving more than two files.

    e.g. $R \rhd\lhd_{A=B} S \rhd\lhd_{C=D} T$
- Examples

  (OP6): EMPLOYEE $\rhd\lhd_{DNO=DNUMBER}$ DEPARTMENT

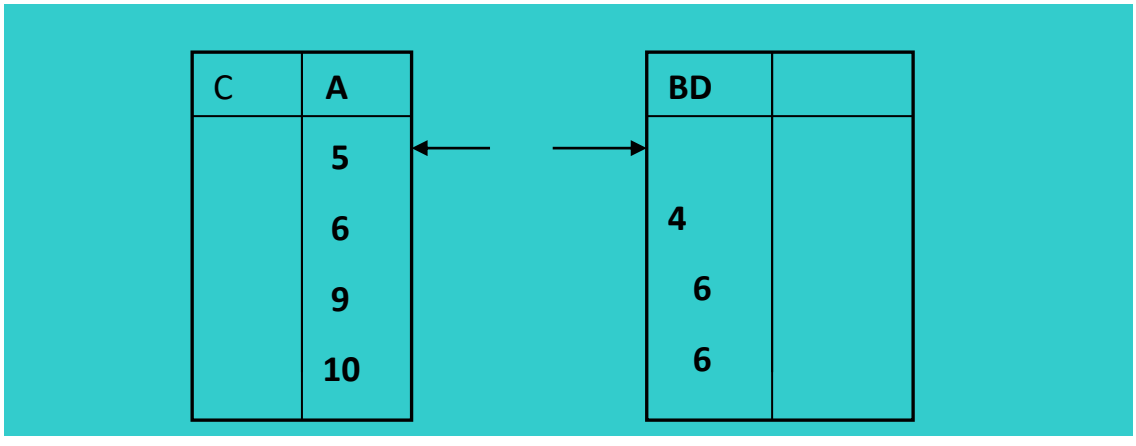
  (OP7): DEPARTMENT $\rhd\lhd_{MGRSSN=SSN}$ EMPLOYEE

Methods for implementing joins:

- J1. **Nested-loop join** (brute force): For each record *t* in *R* (outer loop), retrieve every record *s* from *S* (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.
- J2. **Single-loop join** (Using an access structure to retrieve the matching records): If an index (or hash key) exists for one of the two join attributes — say,

B of *S* — retrieve each record *t* in *R*, one at a time, and then use the access structure to retrieve directly all matching records s from *S* that satisfy *s*[*B*] = *t*[*A*].

- J3. **Sort-merge join:** If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned in order of the join attributes, matching the records that have the same values for A and B. In this method, the records of each file are scanned only once each for matching with the other file—unless both A and B are non-key attributes, in which case the method needs to be modified slightly.



Assume that A is a key of R. Initially, two pointers are used to point to the two tuples of the two relations that have the smallest values of the two joining attributes.

- J4. **Hash-join:** The records of files R and S are both hashed to the *same hash file*, using the *same hashing function* on the join attributes A of R and B of S as hash keys. A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets. A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

(a) Implementing T ←R $\triangleright\triangleleft_{A=B}$ S

- **(**a) sort the tuples in R on attribute A; /* assume R has n tuples */

```
sort the tuples in S on attribute B;  /* assumeS has m tuples  */
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i)[A] > S[j][B]
        then set j ← j + 1
        elseif R(i)[A] < S[j][B]
        then set i ← i + 1
        else { /* output a matched tuple  */
          output the combined tupe <R(i), S(j)> to T;
          /* output other tuples that match R(i), if any */
          set l ← j + 1 ;
          while ( l ≤ m) and (R(i)[A] = S[l][B])
          do { output the combined tuple <R(i), S[l]> to T;
               set l ← l + 1
              }
          /* output other tuples that match S(j), if any */
          set k ← i+1
          while ( k ≤ n) and (R(k)[A] = S[j][B])
          do { output the combined tuple <R(k), S[j]> to T;
               set k ← k + 1
              }
          set i ← i+1, j ← j+1;
          }
    }
```

Factors affecting JOIN performance

- Available buffer space
- Join selection factor
- Choice of inner VS outer relation

1.4. Algorithms for PROJECT and SET Operations

1. If <attribute list> has a key of relation R, extract all tuples from R with only the values for the attributes in <attribute list>.
2. If <attribute list> does NOT include a key of relation R, duplicated tuples must be removed from the results.

- Methods to remove duplicate tuples
  1. Sorting
  2. Hashing

- Algorithm for SET operations
- Set operations:
- UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT
- CARTESIAN PRODUCT of relations R and S include all possible combinations of records from R and S. The attribute of the result include all attributes of R and S.

- **UNION**

  1. Sort the two relations on the same attributes.

  2. Scan and merge both sorted files concurrently, whenever the same tuple exists in both relations, only one is kept in the merged results.

  Union: $T \leftarrow R \cup S$

Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (c) Implementing the operation $T \leftarrow R \cup S$. (d) Implementing the operation $T \leftarrow R \cap S$. (e) Implementing the operation $T \leftarrow R - S$.

(c)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  {  output S(j) to T;
                  set j ← j + 1
               }
      elseif R(i) < S(j)
         then  {  output R(i) to T;
                  set i ← i + 1
               }
      else set  j ← j + 1                    (* R(i)=S(j), so we skip one of the duplicate tuples *)
}
if (i ≤ n) then add tuples R(i) to R(n) to T;
if (j ≤ m) then add tuples S(j) to S(m) to T;
```

(d)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  set j ← j + 1
      elseif R(i) < S(j)
         then  set i ← i + 1
      else  {  output R(j) to T;             (* R(i)=S(j), so we output the tuple *)
               set i ← i + 1, j ← j + 1
      }
}
```

(e)
```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i) > S(j)
         then  set j ← j + 1
      elseif R(i) < S(j)
         then  {  output R(i) to T;          (* R(i) has no matching S(j), so output R(i) *)
                  set i ← i + 1
               }
      else  set i ← i + 1, j ← j + 1
}
if (i ≤ n) then add tuples R(i) to R(n) to T;
```

CARTESIAN PRODUCT

- In general, the result of R(A1, A2, ..., An) × S(B1, B2, ..., Bm) is a relation Q with degree n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm), in that order.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S.
- Hence, if R has nR tuples (denoted as |R| = nR), and S has nS tuples, then R × S will have nR * nS tuples.

1.5. Implementing Aggregate Operations and Outer Joins

- Implementing Aggregate Operations:
- **Aggregate operators**:
  - **MIN, MAX, SUM, COUNT** and **AVG**
- Options to implement aggregate operators:
  - **Table Scan**
  - **Index**
- Example
  - SELECT MAX (SALARY)
  - FROM EMPLOYEE;
- If an (ascending) index on SALARY exists for the employee relation, then the optimizer could decide on traversing the index for the largest value, which would entail following the right most pointer in each index node from the root to a leaf.
- **SUM, COUNT and AVG**
- For a **dense index** (each record has one index entry):
  - Apply the associated computation to the values in the index.

- For a **non-dense index**:
    - Actual number of records associated with each index entry must be accounted for
- With **GROUP BY**: the aggregate operator must be applied separately to each group of tuples.
    - Use sorting or hashing on the group attributes to partition the file into the appropriate groups;
    - Computes the aggregate function for the tuples in each group.
- What if we have **Clustering index** on the grouping attributes?
- Implementing Outer Join:
- **Outer Join Operators**:
    - **LEFT OUTER JOIN**
    - **RIGHT OUTER JOIN**
    - **FULL OUTER JOIN**.
- The full outer join produces a result which is equivalent to the union of the results of the left and right outer joins.
- Example:

SELECT FNAME, DNAME

FROM (EMPLOYEE **LEFT OUTER JOIN** DEPARTMENT

ON DNO = DNUMBER);

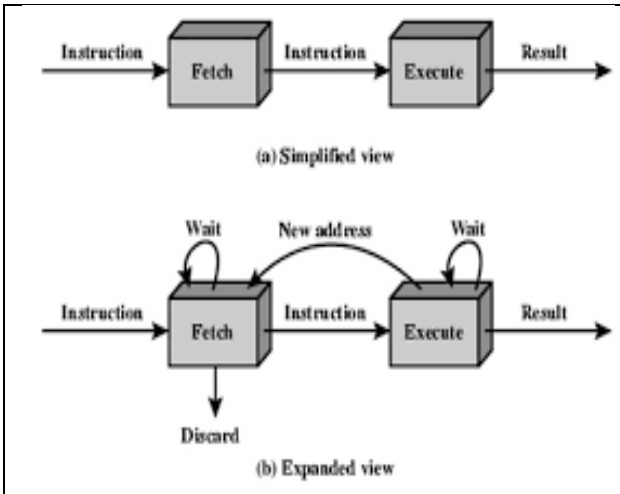
- Note: The result of this query is a table of employee names and their associated departments. It is similar to a regular join result, with the exception that if an employee does not have an associated department, the employee's name will still appear in the resulting table, although the department name would be indicated as null.
- **Modifying Join Algorithms**:
    - Nested Loop or Sort-Merge joins can be modified to implement outer join. E.g.,
        - For left outer join, use the left relation as outer relation and construct result from every tuple in the left relation.
        - If there is a match, the concatenated tuple is saved in the result.

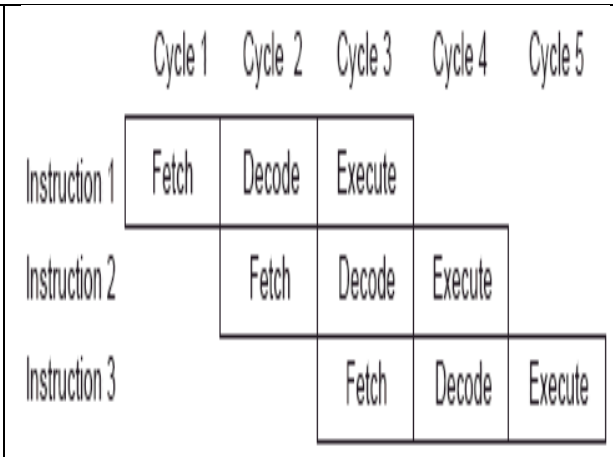
- However, if an outer tuple does not match, then the tuple is still included in the result but is padded with a null value(s).

- Executing a combination of relational algebra operators.
- Implement the previous left outer join example
  - {Compute the JOIN of the EMPLOYEE and DEPARTMENT tables}
    - TEMP1 ← $\square_{FNAME,DNAME}$ (EMPLOYEE $\underset{DNO=DNUMBER}{\bowtie}$ DEPARTMENT)
  - {Find the EMPLOYEEs that do not appear in the JOIN}
    - TEMP2 ← $\square_{FNAME}$ (EMPLOYEE) - $\square_{FNAME}$ (Temp1)
  - {Pad each tuple in TEMP2 with a null DNAME field}
    - TEMP2 ← TEMP2 x 'null'

1.6. Combining Operations using Pipelining

- Motivation
  - A query is mapped into a sequence of operations.
  - Each execution of an operation produces a temporary result.
  - Generating and saving temporary files on disk is time consuming and expensive.
- Alternative:
  - Avoid constructing temporary results as much as possible.
  - Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation.
- Example:
  - For a 2-way join, combine the 2 selections on the input and one projection on the output with the Join.
- Dynamic generation of code to allow for multiple operations to be pipelined.
- Results of a select operation are fed in a "Pipeline" to the join algorithm.
- Also known as stream-based processing.





A 4-Stage Pipeline

- S1: fetch instruction
- S2: decode and calculate effective address
- S3: fetch operand
- S4: execute instruction and store results

Time Period	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

19

6-Stage Pipelining

- Suppose a fetch-decode-execute cycle were broken into:
  1. Fetch instruction.
  2. Decode opcode.
  3. Calculate effective address of operands.
  4. Fetch operands
  5. Execute instruction
  6. Store result.
- Suppose we have a si stage pipeline.
  1. S1 fetches the instruc
  2. S2 decodes it
  3. S3 determines the ad of the operands
  4. S4 fetches them
  5. S5 executes the instr
  6. S6 stores the result.

Note: Not all instructions must go through each stage of the pipe.

1.7. Using Heuristics in Query Optimization

- Process for heuristics optimization
  - The parser of a high-level query generates an initial internal representation;
  - Apply heuristics rules to optimize the internal representation.

- A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

- The main heuristic is to apply first the operations that reduce the size of intermediate results.
    - E.g., Apply **SELECT** and **PROJECT** operations before applying the **JOIN** or other **binary operations.**

Query tree:

- A tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as **leaf nodes** of the **tree**, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.

Query graph:

- A graph data structure that corresponds to a relational calculus expression. It does *not* indicate an order on which operations to perform first. There is only a *single* graph corresponding to each query.

- Example:
  - For every project located in ‘Stafford’, retrieve the project number, the controlling department number and the department manager’s last name, address and birthdate.
- Relation algebra:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}\Big(\big(\big(\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})\big) \underset{\text{DNUM=DNUMBER}}{\bowtie} (\text{DEPARTMENT})\big) \underset{\text{MGRSSN=SSN}}{\bowtie} (\text{EMPLOYEE})\Big)$$

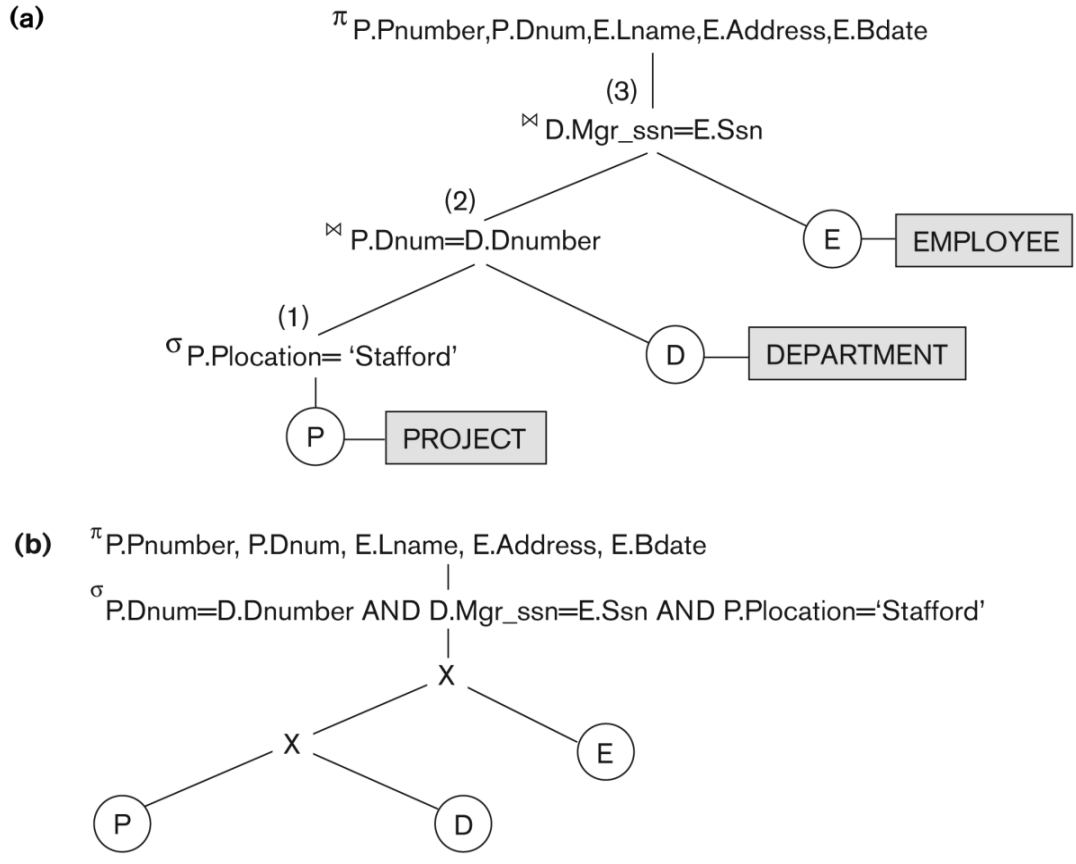


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

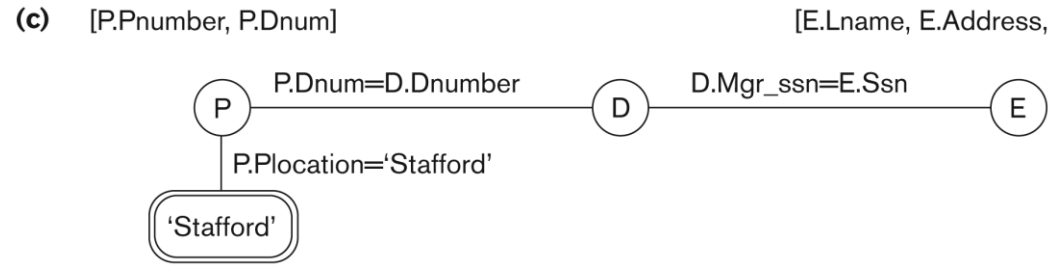


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

- Heuristic Optimization of Query Trees:
  - The same query could correspond to many different relational algebra expressions — and hence many different query trees.
  - The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.

- Example:

  Q: SELECT LNAME

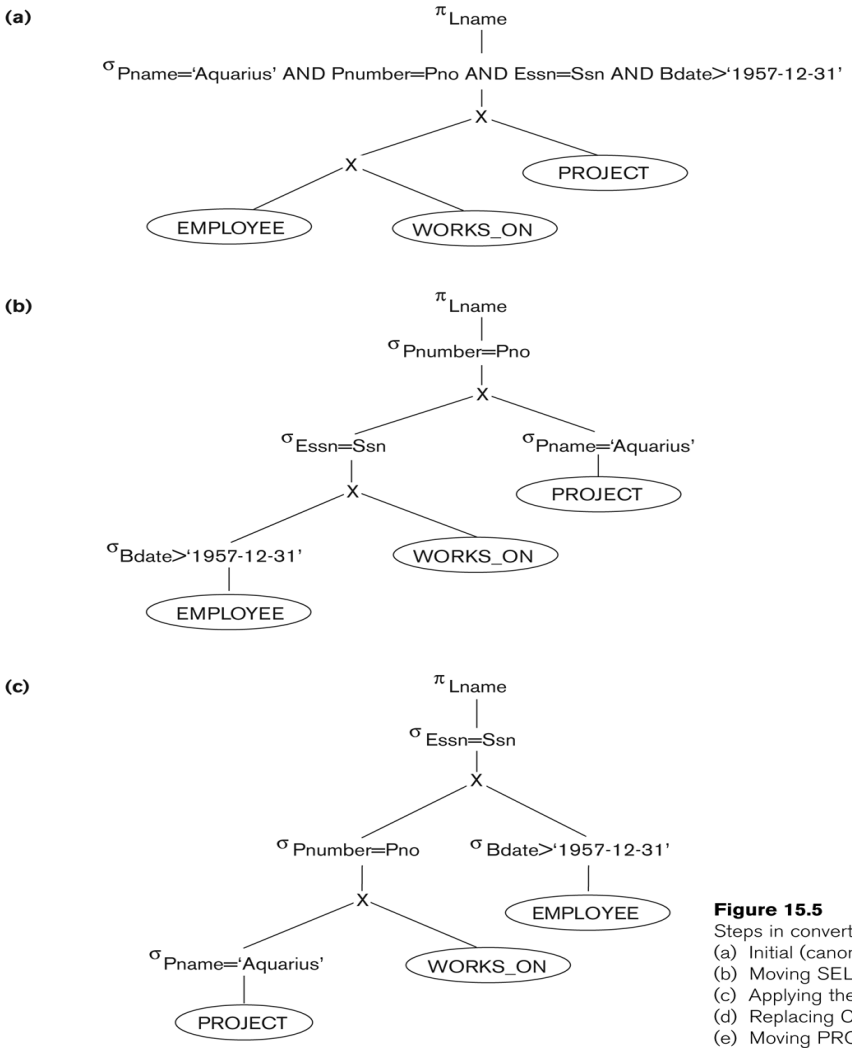
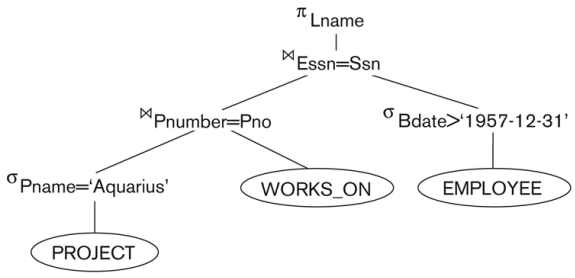


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN o
(e) Moving PROJECT operations down the query tree.

(d)



(e)

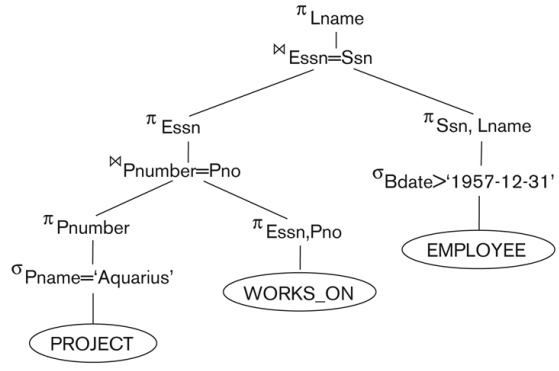


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

■ General Transformation Rules for Relational Algebra Operations:

- General Transformation Rules for Relational Algebra Operations:

1. Cascade of σ: A conjunctive selection condition can be broken up into a cascade (sequence) of individual σ operations:
   - $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$
2. Commutativity of σ: The σ operation is commutative:
   - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:
   - $\pi_{\ldots}(\pi_{\ldots}(\ldots(\pi_{\ldots}(R))\ldots)) = \pi_{\ldots}(R)$

- General Transformation Rules for Relational Algebra Operations (contd.):

5. Commutativity of (and x): The operation is commutative as is the x operation:
   - R $_{C}$ S = S $_{C}$ R; R x S = S x R
6. Commuting σ with (or x): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:
   - σ_c (R S) = (σ_c (R)) S

Alternatively, if the selection condition c can be

- General Transformation Rules for Relational Algebra Operations (contd.):

7. Commuting π with (or x): Suppose that the projection list is L = {A1, ..., An, B1, ..., Bm}, where A1, ..., An are attributes of R and B1, ..., Bm are attributes of S. If the join condition c involves only attributes in L, the two

8. Commutativity of set operations: The set operations υ and ∩ are commutative but "–" is not.

9. Associativity of , x, υ, and ∩ : These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have

- 11.The π operation commutes with υ.

$$\pi_L (R \upsilon S) = (\pi_L (R)) \upsilon (\pi_L (S))$$

- 12. Converting a (σ, x) sequence into : If the condition c of a σ that follows a x

- **Outline of a Heuristic Algebraic Optimization Algorithm:**

1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
2. Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.

3. Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
4. Using Rule 12, combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.
5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Reference Database :

```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 4.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

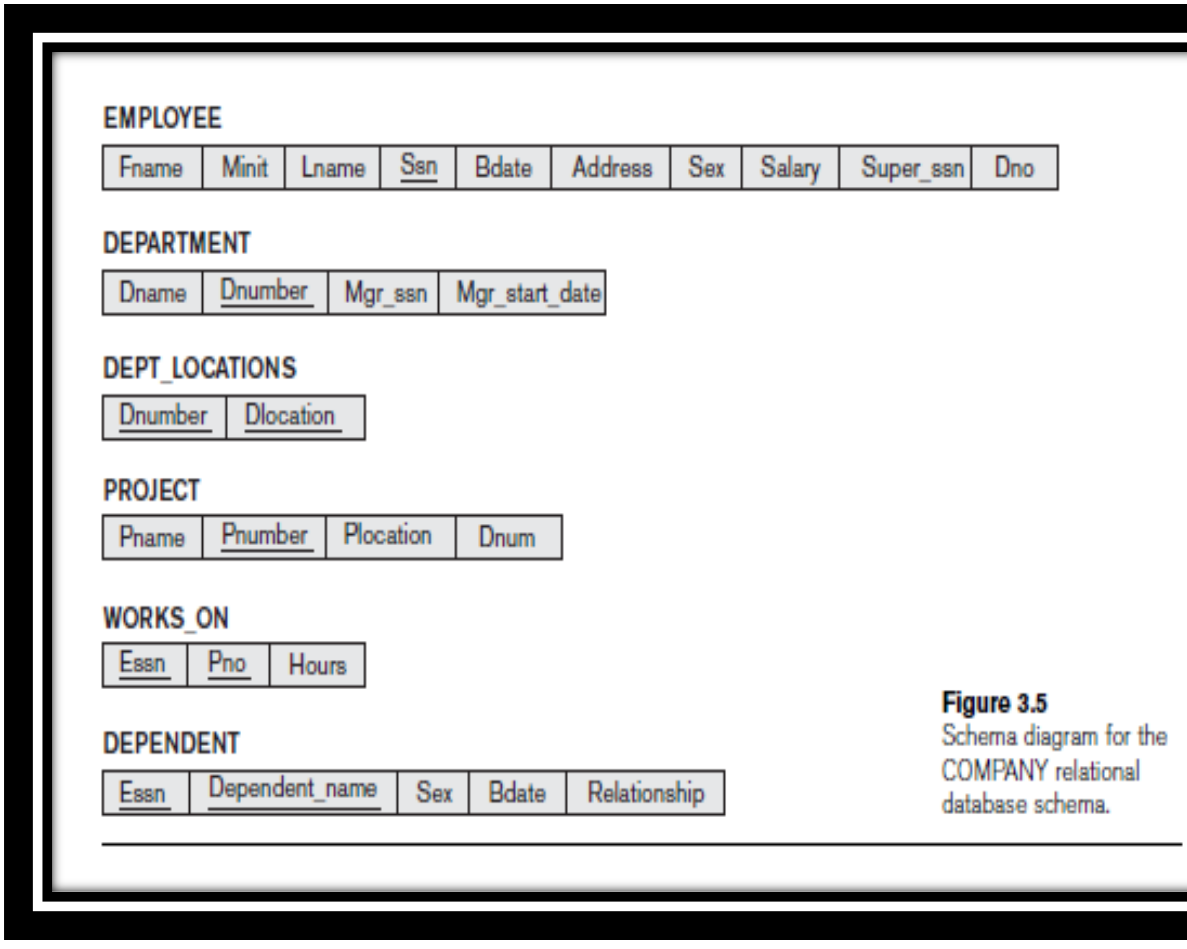


Figure 3.5
Schema diagram for the COMPANY relational database schema.

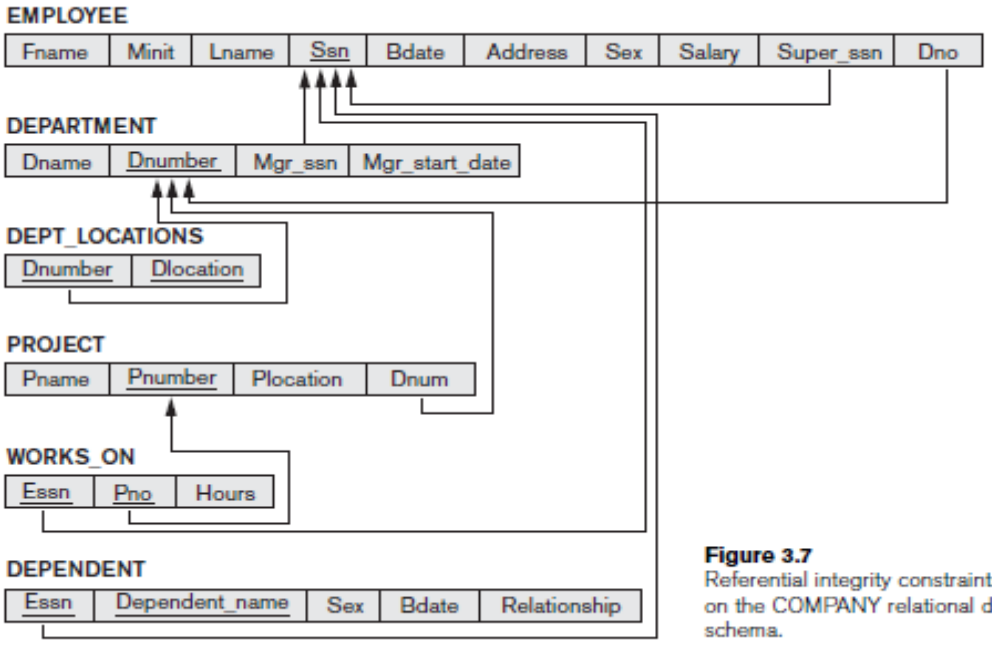


Figure 3.7
Referential integrity constraints displayed on the COMPANY relational database schema.

Figure 3.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<join condition>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$, OR $R_1 \bowtie_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Figure 6.1
Results of SELECT and PROJECT operations. (a) $\sigma_{(\text{Dno}=4 \text{ AND } \text{Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND } \text{Salary}>30000)}$ (EMPLOYEE). (b) $\pi_{\text{Lname, Fname, Salary}}$(EMPLOYEE). (c) $\pi_{\text{Sex, Salary}}$(EMPLOYEE).

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Figure 6.4
The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) **STUDENT**

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

CARTESIAN PRODUCT

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex}='\text{F}'}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

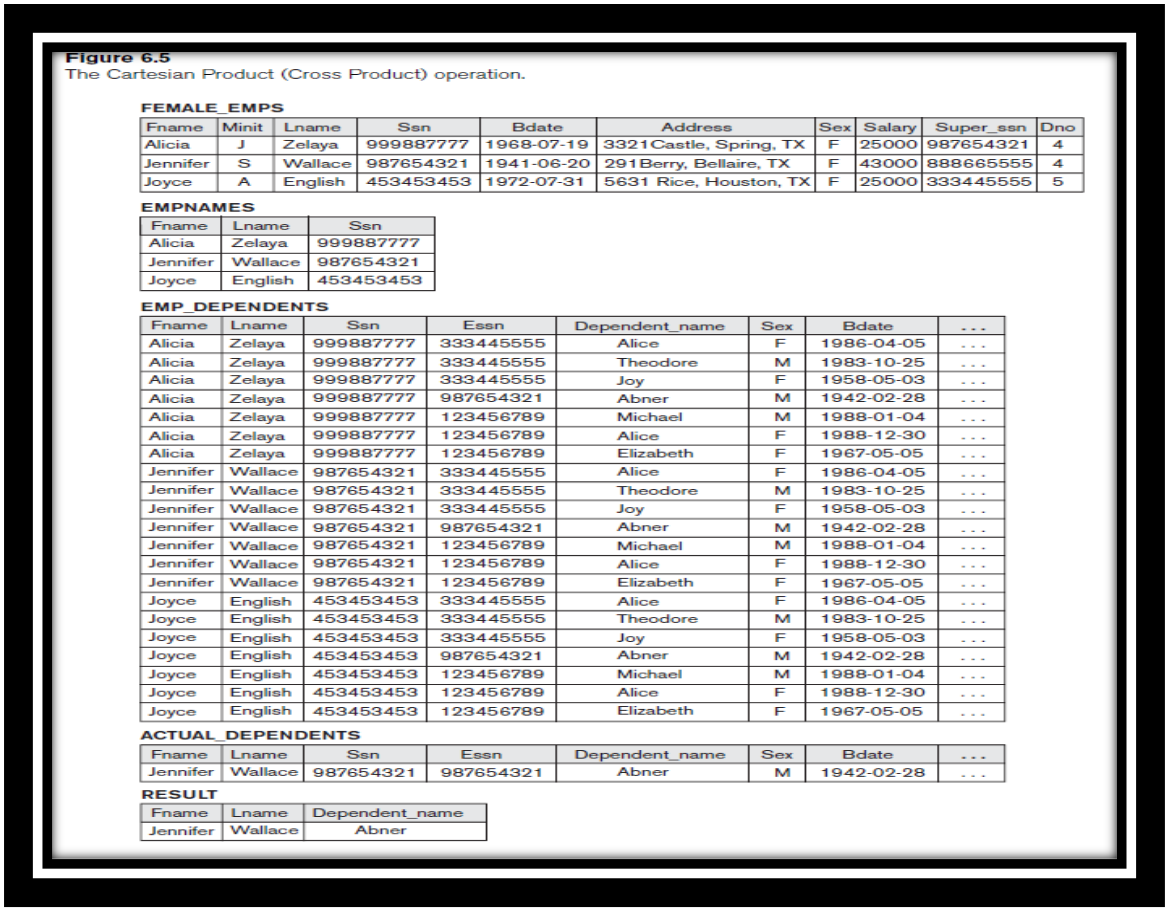


Figure 6.5
The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

UNIT - II

Syllabus:

Data base systems architecture and the system Catalog:

- System architectures for DBMSs,
- Catalogs for Relational DBMSs,
- System catalog information in oracle.
-

Practical database design and tuning:

- Physical Database Design in Relational Databases
- An overview of Database Tuning in Relational systems.

PART-I

1. System Architectures for DBMS

We distinguish between two different DBMS architectures:

- Centralized DBMS architecture –for earlier systems
- Client-Server DBMS architectures –for current systems.

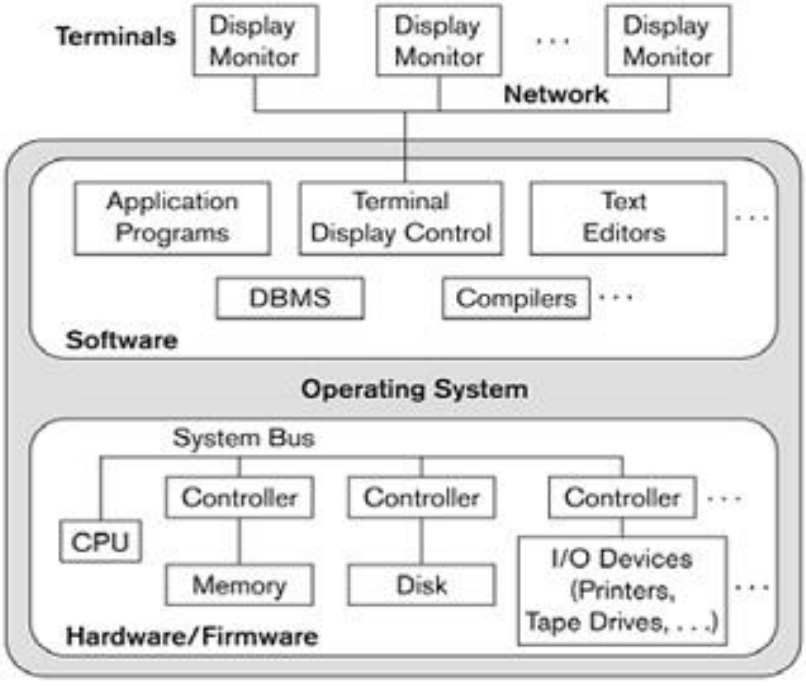
DBMS architecture followed the trends similar to those of general computer systems architectures.

Centralized DBMS Architecture

Earlier computer system architectures were based on mainframe computers.

- Mainframe computers provide the main processing of all system functions.
- Most users accessed such systems via computer terminals that provide display capabilities with no processing power.

Hence, in the centralized DBMS architecture, all functions of the system, including user application programs, user interface programs, as well as all the DBMS functionality are executed in the mainframe computer.



A physical centralized architecture.

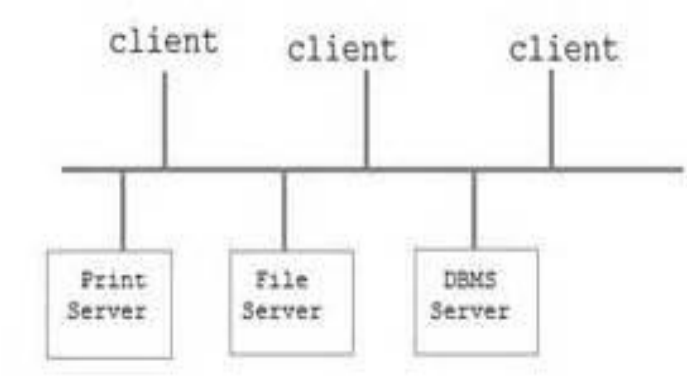
Client-Server Architecture

Client-server computer systems architectures emerged with new computing environments.

- Large number of PCs, workstations, specialized servers and another equipments are connected together via a network.

A client-Server architecture contains:

- Specialized servers with specific functionalities providing resources
  File server, printer server, web server, E-mail server, etc.
- Client machines provide users with appropriate interface to utilize these servers as well as with local processing power to run local applications.



Two- Tier Client-Server DBMS Architecture

- Client-Server architecture is increasingly being incorporated into commercial DBMSs.
- Two-tier architecture distributes the components between the client and the server to split the burden of processing.
- Two most basic approaches to this two-tiers architecture.
- **First approach**: used in RDBMS.

Query and transaction functionality remained at the server side. The server is often called query server or transaction server.

Since RDBMSsare based on the SQL languages, the servers are often called SQL servers.

The user interface programs and application programs moved to the client side. – Standards provide an Application Programming Interface (API) which allows client-side application programs to communicate with a database server.

Open Database Connectivity (ODBC) standard.

JDBC standard -for JAVA client programs.

–Basic API functions:

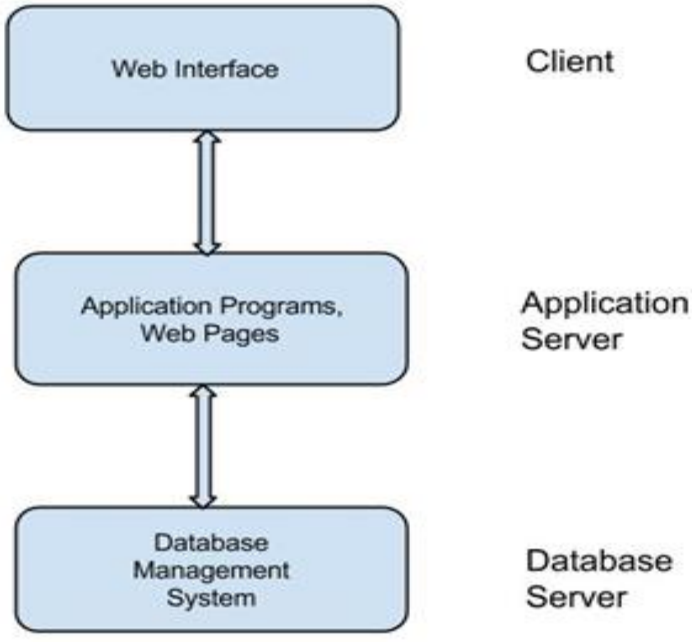
Open a connection with a database, send queries and updates, and get back results.

–When a user program requests for a DBMS access, the client establishes a connection to the DBMS then send query and transaction requests using API calls, which are processed at the server sites and the query results are sent back to the client program.
–Most DBMS vendors provide ODBC and JDBC driver libraries for their systems, hence a client program can actually connect to several RDBMS –by simply linking the corresponding library with his program.
Second approach: taken by some OODBMS.
–Divide the software modules of the DBMS between client and server in a more integrated way –exact functionality division varies from system to system.
Example:
Server side, called data server, includes these parts of the DBMS : data storage on disk, local concurrency control and recovery, buffering and caching ofdisk blocks, etc
. Client side handles: user interface, data dictionary functions, global query optimization/concurrency control/recovery, etc.

Three-Tier Client-Server DBMS Architecture:

Three-tiers architecture is an extension of the client-server model. The tiers are usually as follows:
A client, used for the presentation of the application.
An application server (AS), used for the application's business logic processing.
ADB server, used for storage and retrieval of data.

- Therefore, the application's processing requirements are moved from the client to the application tier.
- Performances may improve by reducing the amount of traffic between the DB server and the clients since more of the interactions may be performed between the application server and the DB server.
- Application server could also check for security like client's credentials
- A simplified client maintenance is achieved since application upgrades will reside only on the application server.



Catalog for Relational DBMS

- In general, a catalog is a "minidatabase"itself that stores special data often called metadata.
- This metadata contains: Information on the schemas, or descriptions, of the database that the DBMS maintains such conceptual database schema, internal database schema, etc and the mapping between schemas.

Information needed by specific DBMS modules like the query optimization module, etc.

Precisely, a relational DBMS catalog includes :

–Relation names, attribute names, attribute domains (data types), constraint descriptions (primary key, etc.).

–Description of views, storage structures and indexes.

–Security and authorization information that describes each user's privileges to access specific database relations and views and the owner of each relation.

–Etc.

- In a relational DBMS, the catalog is stored as relations which allows the DBMS software (as well as users when they are authorized to do so) to access and manipulate the catalog using a language such SQL.
- The choice of catalog relations and their schemas is not unique and vary from a DBMS to another.

Some examples of the catalog's relations:

–REL_AND_ATTR_CATALOG: possible catalog structure for base relations which stores relation names, attribute names, attribute data types, and primary key information.

–RELATION_INDEXES: possible catalog structure for indexes which stores relation names, index names, index attributes, index types, the order of attributes within indexes and the type of sort (asc/desc) in the index.

–VIEW_QUERIES & VIEW_ATTRIBUTES: two possible catalog structures for views. The first to store the text of the query corresponding to the view and the second to store the names of the attributes of the view.

REL_AND_ATTR_CATALOG

REL_NAME	ATTR_NAME	ATTR_TYPE	MEMBER_OF_PK	MEMBER_OF_FK	FK_RELATION

RELATION_INDEXES

REL_NAME	INDEX_NAME	MEMBER_ATTR	INDEX_TYPE	ATTR_NO	ASC_DESC

VIEW_QUERIES

VIEW_NAME	QUERY

VIEW_ATTRIBUTES

VIEW_NAME	ATTR_NAME	ATTR_NUM

System Catalog Information in Oracle9

Oracle catalog is called Data Dictionary.
Data dictionary contains information about schema objects such as tables, indexes, views, etc.
Access to the data dictionary done through views divided into 3 categories**:**
USER: the views with prefix USER contain schema information for objects owned by a given user.

•USER_TABLES: gives information on all base tables owned (created) by the user.

ALL: the views with prefix ALL contain information for objects owned by a user as well as objects that the user has been granted access to.

•ALL_TABLES: gives information on all base tables to which the user has access.

The views with a prefix of DBA are for the database administrator and contain information about all database objects.

•DBA_TABLES: displays the base tables information for the whole database.

To see the data dictionary views available to you, query the view DICTIONARY.

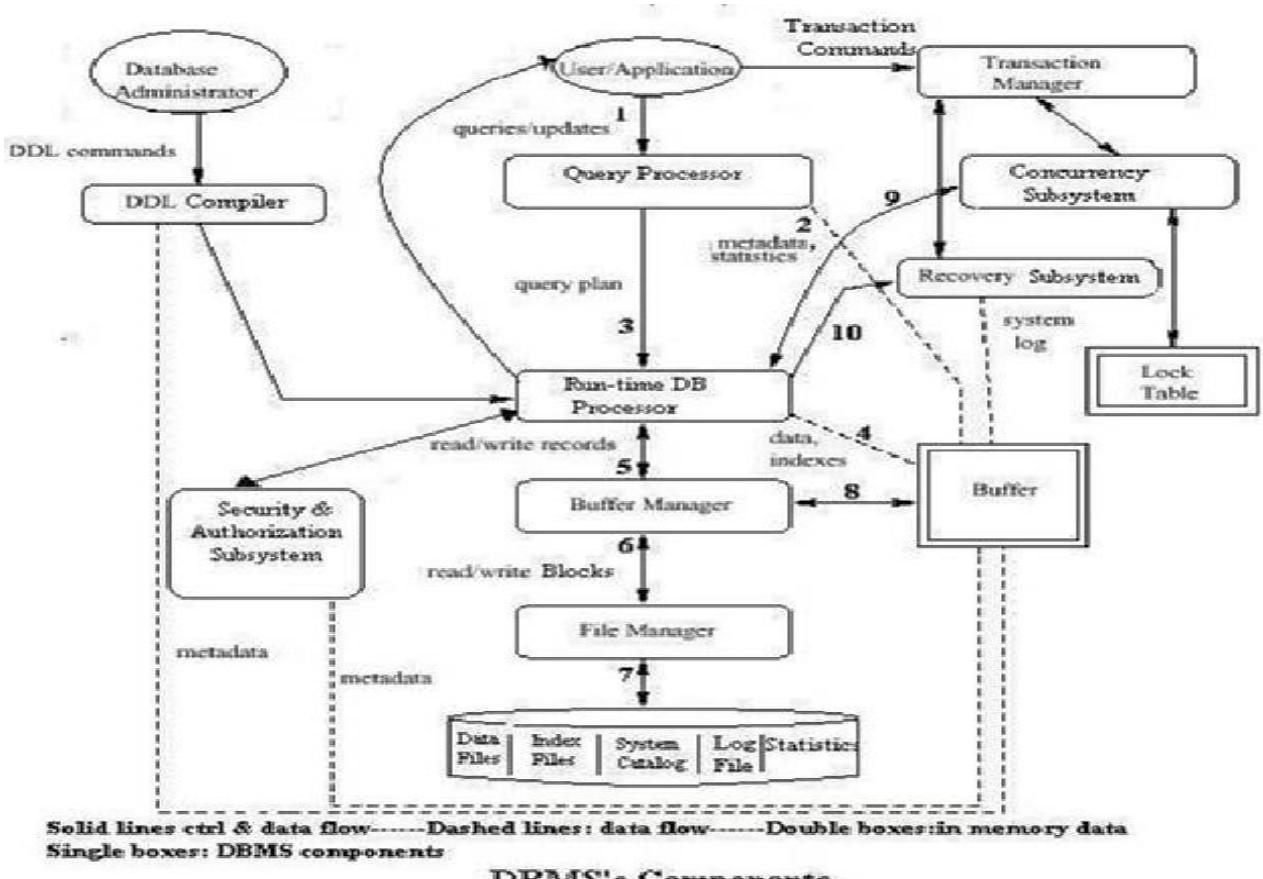
–Select table_name, comments from dictionary;

–COMMENTS field gives a little definition of the view.

Some examples:

- USER_ROLE_PRIVS: This view lists roles granted to the user.
- ALL_VIEWS: This view lists the text of views accessible to the user.
- DBA_INDEXES: This view contains descriptions for all indexes in the database
- USER_OBJECTS : This view lists objects (tables, indexes, etc.) owned by the user.
- USER_TAB_COLUMNS: This view contains information about columns of user's tables, views, and clusters
- ALL_TAB_PRIVS: This view lists the grants on objects for which the user or PUBLIC is the grantee.
- DBA_SEGMENTS: This view contains information about storage allocated for all database segments.

DBMS Component Modules



DBMS's Components

Query processor:

–handles high-level queries. It parses, validates, optimizes, and compiles or interprets a query which results in the query plan.

Run-time DB processor:

–handles database accesses at run-time by receiving retrieval or update operations and carries them out on the database.

DDL compiler:

–processes schema definitions (DDL statements) and stores descriptions of the schemas (metadata) on the DBMS catalog.

Transaction Manager:

–With the cooperation of the concurrency and recovery subsystems, this module ensures that the transactions -which are constituted of queries and other actions-are executed atomically, in consistency, in isolation and in durability.

Concurrency subsystem:

–assures that individual actions of multiple transactions are executed in such an order that the result is the same as if the transactions had executed entirelyat one time.

Recovery subsystem:

–responsible to store every change to the database separately on disk (I.e. log file) . Such information will be used to restore the system to a consistent state after a any failure has occurred.

Security & authorization subsystem:

–responsible to protect the database against unauthorized accesses.

File manager:

–File manager controls accesses to DBMS information that is stored on the disk and can use OS'sFile manager services.

Buffer manager:

–manages the main memory buffers which store data, metadata, indexes, statistics, and the log. Recall that all this information must be in main memory (i.e. in buffers) before it can be used.

•Note: Buffer Manager & File Manager together are often called Stored Data Manager.

Examples of DBMS modules interactions

–Suppose a user requests to update a record of a relation (see Figure).

- The DML statement is issued to the query processor (1) which parses it, checks if the user has the privileges to do such operation usingthe metadata provided from the catalog information found in the main memory buffers (2) and finally issues the query plan which is the optimized way to execute the statement (3).
- The run-time DB processor takes the query plan and executes it by writing the update in the buffer if the record is there (4). Otherwise, it requests the buffer manager to get the record (5). The buffer Manager requests the corresponding block from the File manager (6) who gets it from the disk (7), gives it back to the buffer manager who puts it in the buffer (8). The run-time DB processor can then execute the statement.
- Before executing the statement, the run-time DB processors informs the concurrency subsystem (9) in order to check concurrent accessesfor no inconsistency and the recovery subsystem to take the necessary actions to allow a recovery in case of failure (10)

Catalog Information accessed by DBMS Modules

DBMS modules use and access the catalog very frequently.

Query Processor Module

- Parses queries and database update statements and checks the catalog to verify whether everything is valid.

•For example, in relational system this module checks that all the relation names specified in the query exist in the catalog.

- Converts the queries and update statements into low-level file access commands by accessing the catalog to know the mapping between the conceptual and the internal schemas.-correspondence Relations-Files, etc-.
- Access the catalog information (data statistics, etc.), to know the best way to execute the query.

DDL Compiler Module.

–Parses & checks the specification of a database schema in the DDL statements and stores (populates) such description in the catalog???

Authorization and Security module

Ensures that only the DBA can update the authorization & security portion of the catalog. Stores information in the catalog that will ensure that a user can do only what he has been authorized to do.

Etc…

PART-II

Physical Database Design in Relational Databases

In this section, we begin by discussing the physical design factors that affect the performance of applications and transactions, and then we comment on the specific guidelines for RDBMSs

1. Factors That Influence Physical Database Design

Physical design is an activity where the goal is not only to create the appropriate structuring of data in storage, but also to do so in a way that guarantees good performance. For a given conceptual schema, there are many physical design alternatives in a given DBMS. It is not possible to make meaningful physical design decisions and performance analyses until the database designer knows the mix of queries, transactions, and applications that are expected to run on the database. This is called the **job mix** for the particular set of database system applications. The database administrators/designers must analyze these applications, their expected frequencies of invocation, any timing constraints on their execution speed, the expected frequency of update operations, and any unique constraints on attributes.

A. Analyzing the Database Queries and Transactions.

Before undertaking the physical database design, we must have a good idea of the intended use of the database by defining in a high-level form the queries and transactions that are expected to run on the database. For each **retrieval query**, the following information about the query would be needed:

1. The files that will be accessed by the query.1
2. The attributes on which any selection conditions for the query are specified.
3. Whether the selection condition is an equality, inequality, or a range condition.
4. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified.
5. The attributes whose values will be retrieved by the query.

The attributes listed in items 2 and 4 above are candidates for the definition of access structures, such as indexes, hash keys, or sorting of the file.

For each **update operation** or **update transaction**, the following information would be needed:

1. The files that will be updated.

2. The type of operation on each file (insert, update, or delete).
3. The attributes on which selection conditions for a delete or update are specified.
4. The attributes whose values will be changed by an update operation.

Again, the attributes listed in item 3 are candidates for access structures on the files, because they would be used to locate the records that will be updated or deleted. On the other hand, the attributes listed in item 4 are candidates for avoiding an access structure, since modifying them will require updating the access structures.

B. Analyzing the Expected Frequency of Invocation of Queries and Transactions.

Besides identifying the characteristics of expected retrieval queries and update transactions, we must consider their expected rates of invocation. This frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of the expected frequency of use for all queries and transactions. This is expressed as the expected frequency of using each attribute in each file as a selection attribute or a join attribute
over all the queries and transactions. Generally, for large volumes of processing, the informal *80–20 rule* can be used: approximately 80 percent of the processing is accounted for by only 20 percent of the queries and transactions. Therefore, in practical situations, it is rarely necessary to collect exhaustive statistics and invocation rates on all the queries and transactions; it is sufficient to determine the 20 percent or so most important ones.

C. Analyzing the Time Constraints of Queries and Transactions.

Some queries and transactions may have stringent performance constraints. For example, a transaction may have the constraint that it should terminate within 5 seconds on 95 percent of the occasions when it is invoked, and that it should never take more than 20 seconds. Such timing constraints place further priorities on the attributes that are candidates for access paths. The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files, because the primary access structures are generally the most efficient for locating records in a file.

D. Analyzing the Expected Frequencies of Update Operations.

A minimum number of access paths should be specified for a file that is frequently updated, because updating the access paths themselves slows down the update operations. For example, if a file that has frequent record insertions has 10 indexes on 10 different attributes, each of these indexes must be updated whenever a new record is inserted. The overhead for updating 10 indexes can slow down the insert operations.

E. Analyzing the Uniqueness Constraints on Attributes.

Access paths should be specified on all *candidate key* attributes—or sets of attributes—that are either the primary key of a file or unique attributes. The existence of an index (or other access path) makes it sufficient to only search the index when checking this uniqueness constraint, since all values of the attribute will exist in the leaf nodes of the index. For example, when inserting a new record, if a key attribute value of the new record *already exists in the index*, the insertion of the new record should be rejected, since it would violate the uniqueness constraint on the attribute. Once the preceding information is compiled, it is possible to address the physical database design decisions, which consist mainly of deciding on the storage structures and access paths for the database files.

2. Physical Database Design Decisions

Most relational systems represent each base relation as a physical database file. The access path options include specifying the type of primary file organization for each relation and the attributes of

which indexes that should be defined. At most, one of the indexes on each file may be a primary or a clustering index. Any number of additional secondary indexes can be created.

Design Decisions about Indexing.

The attributes whose values are required in equality or range conditions (selection operation) are those that are keys or that participate in join conditions (join operation) requiring access paths, such as indexes. The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins. On the other hand, during insert, delete, or update operations, the existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions. The physical design decisions for indexing fall into the following categories:

1. **Whether to index an attribute.**
   The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition. One reason for creating multiple indexes is that some operations can be processed by just scanning the indexes, without having to access the actual data file.

2. **What attribute or attributes to index on.**
   An index can be constructed on a single attribute, or on more than one attribute if it is a composite index. If multiple attributes from one relation are involved together in several queries, (for example, (Garment_style_#, Color) in a garment inventory database), a multiattribute (composite) index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For instance, the above index assumes that queries would be based on an ordering of colors within a Garment_style_# rather than vice versa.

3. **Whether to set up a clustered index.**
   At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. In most RDBMSs, this is specified by the keyword CLUSTER. (If the attribute is a *key*, *a primary index* is created, whereas a *clustering index* is created if the attribute is *not a key*—If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering. If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on. If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should *not* be clustered, since the main benefit of clustering is achieved when retrieving the records themselves. A clustering index may be set up as a multiattribute index if range retrieval by that composite key is useful in report creation (for example, an index on Zip_code, Store_id, and Product_id may be a clustering index for sales data).
4. **Whether to use a hash index over a tree index.**
   In general, RDBMSs use B+- trees for indexing. However, ISAM and hash indexes are also provided in some systems (see Chapter 18). B+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.

5. **Whether to use dynamic hashing for the file.**
   For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes discussed would be suitable. Currently, they are not offered by many commercial RDBMSs.

How to Create an Index.

Many RDBMSs have a similar type of command for creating an index, although it is not part of the SQL standard. The general form of this command is:

CREATE [UNIQUE] INDEX <index name>
ON <table name> (<column name> [<order>] { , <column name> [<order>] })
[CLUSTER] ;

The keywords UNIQUE and CLUSTER are optional. The keyword CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute. Thus, specifying CLUSTER on a key (unique) attribute would create some variation of a primary index, whereas specifying CLUSTER on a nonkey (nonunique) attribute would create some variation of a clustering index. The value for <order> can be either ASC (ascending) or DESC (descending), and specifies whether the data file should be ordered in ascending or descending values of the indexing attribute. The default is ASC. For example, the following would create a clustering (ascending) index on the nonkey attribute Dno of the EMPLOYEEfile:

CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;

Denormalization as a Design Decision for Speeding Up Queries.

The ultimate goal during normalization is to separate attributes into tables to minimize redundancy, and thereby avoid the update anomalies that lead to an extra processing overhead to maintain consistency in the database. The ideals that are typically followed are the third or Boyce-Codd normal forms. The above ideals are sometimes sacrificed in favor of faster execution of frequently occurring queries and transactions. This process of storing the logical database design (which may be in BCNF or 4NF) in a weaker normal form, say 2NF or 1NF, is called **denormalization**. Typically, the designer includes certain attributes from a table *S* into another table *R*. The reason is that the attributes from *S* that are included in *R* are frequently needed—along with other attributes in *R*—for answering queries or producing reports. By including these attributes, a join of *R* with *S* is avoided for these frequently occurring queries and reports. This reintroduces *redundancy* in the base tables by including the same attributes in both tables *R* and *S*. A partial functional dependency or a transitive dependency now exists in the table *R*, thereby creating the associated redundancy problems. A tradeoff exists between the additional updating needed for maintaining consistency of redundant attributes versus the effort needed to perform a join to incorporate the additional attributes needed in the result. For example, consider the following relation:

ASSIGN (Emp_id, Proj_id, Emp_name, Emp_job_title, Percent_assigned, Proj_name,

Proj_mgr_id, Proj_mgr_name),
which corresponds exactly to the headers in a report called *The Employee Assignment Roster*.
This relation is only in 1NF because of the following functional dependencies:
Proj_id → Proj_name, Proj_mgr_id
Proj_mgr_id → Proj_mgr_name
Emp_id →Emp_name, Emp_job_title
This relation may be preferred over the design in 2NF (and 3NF) consisting of the following three relations:
EMP (Emp_id, Emp_name, Emp_job_title)
PROJ (Proj_id, Proj_name, Proj_mgr_id)
EMP_PROJ (Emp_id, Proj_id, Percent_assigned)

This is because to produce the *The Employee Assignment Roster* report (with all fields shown in ASSIGN above), the latter multirelation design requires two NATURAL JOIN (indicated with *) operations (between EMP and EMP_PROJ, and between PROJ and EMP_PROJ), plus a final JOIN between PROJ and EMP to retrieve the Proj_mgr_name from the Proj_mgr_id. Thus the following JOINs would be needed (the final join would also require renaming (aliasing) of the last EMP table, which is not shown):
((EMP_PROJ * EMP) * PROJ) PROJ.Proj_mgr_id = EMP.Emp_id EMP
It is also possible to create a view for the ASSIGN table. This does not mean that the join operations will be avoided, but that the user need not specify the joins. If the view table is materialized, the joins would be avoided, but if the virtual view table is not stored as a materialized file, the join computations would still be necessary. Other forms of denormalization consist of storing extra tables to maintain original functional dependencies that are lost during BCNF decomposition. For example, Figure 15.14 shows the TEACH(Student, Course, Instructor) relation with the functional dependencies {{Student, Course}→ Instructor, Instructor → Course}.
A lossless decomposition of TEACH into T1(Student, Instructor) and T2(Instructor, Course) does *not* allow queries of the form *what course did student Smith take from instructor Navathe* to be answered without joining T1 and T2. Therefore, storing T1, T2, and TEACH may be a possible solution, which reduces the design from BCNF to 3NF. Here, TEACH is a materialized join of the other two tables, representing an extreme redundancy. Any updates to T1 and T2 would have to be applied to TEACH.An alternate strategy is to create T1 and T2 as updatable base tables, and to create TEACH as a view (virtual table) on T1 and T2 that can only be queried.

An Overview of Database Tuning in Relational Systems

After a database is deployed and is in operation, actual use of the applications, transactions, queries, and views reveals factors and problem areas that may not have been accounted for during the initial physical design. The inputs to physical design can be revised by gathering actual statistics about usage patterns. Resource utilization as well as internal DBMS processing—such as query optimization— can be monitored to reveal bottlenecks, such as contention for the same data or devices. Volumes of activity and sizes of data can be better estimated. Therefore, it is necessary to monitor and revise the physical database design constantly—an activity referred to as **database tuning**.
The goals of tuning are as follows:

- To make applications run faster.
- To improve (lower) the response time of queries and transactions.
- To improve the overall throughput of transactions.

The dividing line between physical design and tuning is very thin. The same design decisions that we discussed in Section 20.1.2 are revisited during database tuning, which is a continual adjustment of the physical design. We give a brief overview of the tuning process below.In particular, DBMSs can internally collect the following statistics:

- Sizes of individual tables.
- Number of distinct values in a column.
- The number of times a particular query or transaction is submitted and executed in an interval of time.
- The times required for different phases of query and transaction processing (for a given set of queries or transactions).

These and other statistics create a profile of the contents and use of the database.
Other information obtained from monitoring the database system activities and processes includes the following:

- **Storage statistics.** Data about allocation of storage into table spaces, index spaces, and buffer pools.
- **I/O and device performance statistics.** Total read/write activity (paging) on disk extents and disk hot spots.
- **Query/transaction processing statistics.** Execution times of queries and transactions, and optimization times during query optimization.
- **Locking/logging related statistics.** Rates of issuing different types of locks, transaction throughput rates, and log records activity.4
- **Index statistics.** Number of levels in an index, number of non contiguous leaf pages, and so on.

Tuning a database involves dealing with the following types of problems:

- How to avoid excessive lock contention, thereby increasing concurrency among transactions.
- How to minimize the overhead of logging and unnecessary dumping of data.
- How to optimize the buffer size and scheduling of processes.
- How to allocate resources such as disks, RAM, and processes for most efficient utilization.

Most of the previously mentioned problems can be solved by the DBA by setting appropriate physical DBMS parameters, changing configurations of devices, changing operating system parameters, and other similar activities. The solutions tend to be closely tied to specific systems. The DBAs are typically trained to handle these tuning problems for the specific DBMS. We briefly discuss the tuning of various physical database design decisions below.

2.1 Tuning Indexes

The initial choice of indexes may have to be revised for the following reasons:

- Certain queries may take too long to run for lack of an index.
- Certain indexes may not get utilized at all.
- Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.

Most DBMSs have a command or trace facility, which can be used by the DBA to ask the system to show how a query was executed—what operations were performed in what order and what secondary

access structures (indexes) were used. By analyzing these execution plans, it is possible to diagnose the causes of the above problems.

Some indexes may be dropped and some new indexes may be created based on the tuning analysis. The goal of tuning is to dynamically evaluate the requirements, which sometimes fluctuate seasonally or during different times of the month or week, and to reorganize the indexes and file organizations to yield the best overall performance. Dropping and building new indexes is an overhead that can be justified in terms of performance improvements. Updating of a table is generally suspended while an index is dropped or created; this loss of service must be accounted for. Besides dropping or creating indexes and changing from a non clustered to a clustered index and vice versa, **rebuilding the index** may improve performance. Most RDBMSs use B+-trees for an index. If there are many deletions on the index key, index pages may contain wasted space, which can be claimed during a rebuild operation. Similarly, too many insertions may cause overflows in a clustered index that affect performance. Rebuilding a clustered index amounts to reorganizing the entire table ordered on that key.

The available options for indexing and the way they are defined, created, and reorganized varies from system to system. As an illustration, consider the sparse and dense indexes in Chapter 18. A sparse index such as a primary index (see Section 18.1) will have one index pointer for each page (disk block) in the data file; a dense index such as a unique secondary index will have an index pointer for each record. Sybase provides clustering indexes as sparse indexes in the form of B+-trees, whereas INGRES provides sparse clustering indexes as ISAM files and dense clustering indexes as B+-trees. In some versions of Oracle and DB2, the option of setting up a

clustering index is limited to a dense index (with many more index entries), and the DBA has to work with this limitation.

2.2 Tuning the Database Design

1.2, we discussed the need for a possible denormalization, which is a departure from keeping all tables as BCNF relations. If a given physical database design does not meet the expected objectives, the DBA may revert to the logical database design, make adjustments such as denormalizations to the logical schema, and remap it to a new set of physical tables and indexes. As discussed, the entire database design has to be driven by the processing requirements as much as by data requirements. If the processing requirements are dynamically

changing, the design needs to respond by making changes to the conceptual schema if necessary and to reflect those changes into the logical schema and physical design.

These changes may be of the following nature:

- Existing tables may be joined (denormalized) because certain attributes from two or more tables are frequently needed together: This reduces the normalization level from BCNF to 3NF, 2NF, or 1NF
- For the given set of tables, there may be alternative design choices, all of which achieve 3NF or BCNF.We illustrated alternative equivalent designs. One normalized design may be replaced by another.
- A relation of the form R(K,A, B, C, D, ...)—with K as a set of key attributes— that is in BCNF can be stored in multiple tables that are also in BCNF—for example, R1(K, A, B), R2(K, C, D,), R3(K, ...)—by replicating the key K in each table. Such a process is known as **vertical partitioning**. Each table groups sets of attributes that are accessed together. For example, the table
- EMPLOYEE(Ssn, Name, Phone, Grade, Salary) may be split into two tables:
  EMP1(Ssn, Name, Phone) and EMP2(Ssn, Grade, Salary). If the original table has a large number of rows (say 100,000) and queries about phone numbers and salary information are totally

distinct and occur with very different frequencies, then this separation of tables may work better.

- Attribute(s) from one table may be repeated in another even though this creates redundancy and a potential anomaly. For example, Part_name may be replicated in tables wherever the Part# appears (as foreign key), but there may be one master table called PART_MASTER(Part#, Part_name, ...) where the Partname is guaranteed to be up-to-date.
- Just as vertical partitioning splits a table vertically into multiple tables, **horizontal partitioning** takes horizontal slices of a table and stores them as distinct tables.
- For example, product sales data may be separated into ten tables based on ten product lines. Each table has the same set of columns (attributes) but contains a distinct set of products (tuples). If a query or transaction applies to all product data, it may have to run against all the tables and the results may have to be combined. These types of adjustments designed to meet the high volume of queries or transactions, with or without sacrificing the normal forms, are commonplace in practice.

2.3 Tuning Queries

We already discussed how query performance is dependent upon the appropriate selection of indexes, and how indexes may have to be tuned after analyzing queries that give poor performance by using the commands in the RDBMS that show the execution plan of the query. There are mainly two indications that suggest that query tuning may be needed:

1. A query issues too many disk accesses (for example, an exact match query scans an entire table).
2. The query plan shows that relevant indexes are not being used.

Some typical instances of situations prompting query tuning include the following:

1. Many query optimizers do not use indexes in the presence of arithmetic expressions (such as Salary/365 > 10.50), numerical comparisons of attributes of different sizes and precision (such as Aqty = Bqty where Aqty is of type INTEGER and Bqty is of type SMALLINTEGER), NULL comparisons (such as Bdate IS NULL), and substring comparisons (such as Lname LIKE '%mann').

2. Indexes are often not used for nested queries using IN; for example, the following query:

```
SELECT Ssn FROM EMPLOYEE
WHERE Dno IN ( SELECT Dnumber FROM DEPARTMENT
WHERE Mgr_ssn = '333445555' );
```

may not use the index on Dno in EMPLOYEE, whereas using Dno = Dnumber in the WHERE-clause with a single block query may cause the index to be used.

3. Some DISTINCTs may be redundant and can be avoided without changing the result. A DISTINCT often causes a sort operation and must be avoided as much as possible.

4. Unnecessary use of temporary result tables can be avoided by collapsing multiple queries into a single query *unless* the temporary relation is needed for some intermediate processing.

5. In some situations involving the use of correlated queries, temporaries are useful. Consider the following query, which retrieves the highest paid employee in each department:

```
SELECT Ssn
FROM EMPLOYEE E
WHERE Salary = SELECT MAX (Salary)
FROM EMPLOYEE AS M
WHERE M.Dno = E.Dno;
```

This has the potential danger of searching all of the inner EMPLOYEE table M for *each* tuple from the outer EMPLOYEE table E. To make the execution more efficient, the process can be broken into two queries, where the first query just computes the maximum salary in each department as follows:

```
SELECT MAX (Salary) AS High_salary, Dno INTO TEMP
FROM EMPLOYEE
GROUP BY Dno;
SELECT EMPLOYEE.Ssn
FROM EMPLOYEE, TEMP
WHERE EMPLOYEE.Salary = TEMP.High_salary
AND EMPLOYEE.Dno = TEMP.Dno;
```

6. If multiple options for a join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons. For example, assuming that the Name attribute is a candidate key in EMPLOYEE and STUDENT, it is better to use EMPLOYEE.Ssn = STUDENT.Ssn as a join condition rather than EMPLOYEE.Name = STUDENT.Name if Ssn has a clustering index in one or both tables.

7. One idiosyncrasy with some query optimizers is that the order of tables in the FROM-clause may affect the join processing. If that is the case, one may have to switch this order so that the smaller of the two relations is scanned and the larger relation is used with an appropriate index.

8. Some query optimizers perform worse on nested queries compared to their equivalent unnested counterparts. There are four types of nested queries:

- Uncorrelated subqueries with aggregates in an inner query.
- Uncorrelated subqueries without aggregates.
- Correlated subqueries with aggregates in an inner query.
- Correlated subqueries without aggregates.

Of the four types above, the first one typically presents no problem, since most query optimizers evaluate the inner query once. However, for a query of the second type, such as the example in item 2, most query optimizers may not use an index on Dno in EMPLOYEE. However, the same optimizers may do so if the query is written as an unnested query. Transformation of correlated subqueries may involve setting temporary tables. Detailed examples are outside our scope here.

9. Finally, many applications are based on views that define the data of interest to those applications. Sometimes, these views become overkill, because a query may be posed directly against a base table, rather than going through a view that is defined by a JOIN.

2.4 Additional Query Tuning Guidelines

Additional techniques for improving queries apply in certain situations as follows:

1. A query with multiple selection conditions that are connected via OR may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used. For example,

```
SELECT Fname, Lname, Salary, Age7
FROM EMPLOYEE
WHERE Age >45 OR Salary < 50000;
```

may be executed using sequential scan giving poor performance. Splitting it up as

```
SELECT Fname, Lname, Salary, Age
FROM EMPLOYEE
WHERE Age >45
UNION
SELECT Fname, Lname, Salary, Age
FROM EMPLOYEE
WHERE Salary < 50000;
```

may utilize indexes on Age as well as on Salary.

2. To help expedite a query, the following transformations may be tried:

- NOT condition may be transformed into a positive expression.
- Embedded SELECT blocks using IN, = ALL, = ANY, and = SOME may be replaced by joins.
- If an equality join is set up between two tables, the range predicate (selection condition) on the joining attribute set up in one table may be repeated for the other table. Correlated subqueries with aggregates in an inner query.

WHERE conditions may be rewritten to utilize the indexes on multiple columns. For example,

```
SELECT Region#, Prod_type, Month, Sales
FROM SALES_STATISTICS
WHERE Region# = 3 AND ((Prod_type BETWEEN 1 AND 3) OR (Prod_type
BETWEEN 8 AND 10));
```

may use an index only on Region# and search through all leaf pages of the index for a match on Prod_type. Instead, using

```
SELECT Region#, Prod_type, Month, Sales
FROM SALES_STATISTICS
WHERE (Region# = 3 AND (Prod_type BETWEEN 1 AND 3))
OR (Region# = 3 AND (Prod_type BETWEEN 8 AND 10));
```

may use a composite index on (Region#, Prod_type) and work much more efficiently.

In this section, we have covered many of the common instances where the inefficiency of a query may be fixed by some simple corrective action such as using a temporary table, avoiding certain types of query constructs, or avoiding the use of views. The goal is to have the RDBMS use existing single attribute or composite attribute indexes as much as possible. This avoids full scans of data blocks or entire scanning of index leaf nodes. Redundant processes like sorting must be avoided at any cost. The problems and the remedies will depend upon the workings of a query optimizer within an RDBMS. Detailed literature exists in database tuning guidelines for database administration by the RDBMS vendors. Major relational DBMS vendors like Oracle, IBM and Microsoft encourage their large customers to share ideas of tuning at the annual expos and other forums so that the entire industry benefits by using performance enhancement techniques. These techniques are typically available in trade literature and on various Web sites.

UNIT - III

Distributed Dbms concepts and design

Distributed Database changes the way of data sharing, conceptually from centralization into decentralization.Development of computer networks promotes a decentralized mode of work. Development of distributed systems should improve the sharing ability of the data and the efficiency of data access. Distributed systems should help resolve the "Islands of information" problem

Distributed database

A logically interrelated collection of shared data and description of this data, physically distribute over a computer network.

Distributed DBMS

The software system that permits the management of the distributed databases and makes the distribution transparent to users.

Fundamental Principle: make distribution transparent to user.”**The fact that fragments are stored on different computers is hidden from the users”.**

In a distributed DBMS ,Single logical database is split into a number of fragments.

Each fragment is stored on one or more computers under the control of a separate DBMS

with the computer connected to a network. Each site is capable of independently processing user requests that require access to local data and is also capable of processing data stored on other computers in the network.

There are two applications

1) local application: Do not require data from other sites.

2) global application: do require data from other sites.

Distributed DBMS need to have at least one global application.

DDBMS have following characteristics:

- A collection of logically related shared data .
- The data is split into number of fragments.
- Fragments may be replicated.
- Fragments/replicas are allocated to sites.
- The sites are linked with computer network.
- The data at each site is under the control of a DBMS.
- The DBMS at each site can handle local applications autonomously.
- Each DBMS participates in at least one global application.

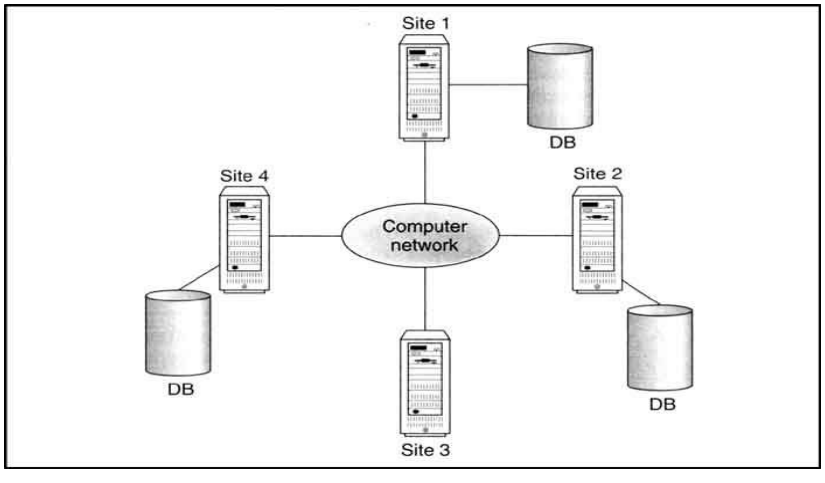


Fig: Distributeed database

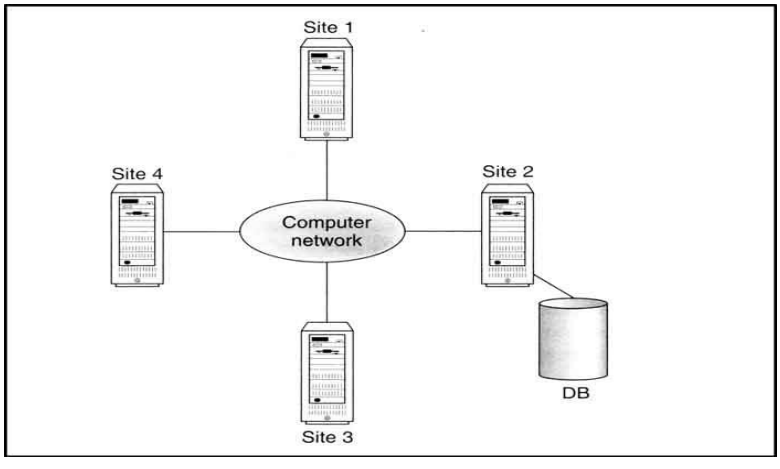
- It is not necessary for every site in the system to have its own local database as shown

- The system is expected to make the distribution transparent to the user
- Distributed database is split into fragments that can be stored on different computers and perhaps replicated .
- The objective of the transparency is to make the distributed system to appear like a centralized system.

Distributed Processing:

Distributed processing of a centralised DBMS has following characteristics :

- Much more tightly coupled than a DDBMS.
- Database design is same as for standard DBMS
- No attempt to reflect organizational structure
- Much simpler than DDBMS
- More secure than DDBMS
- No local autonomy



The system consists of data that is physically distributed across the network. If the data is centralized, even though the users may be accessing the data over the network, it is not considered as distributed DBMS, simply distributedprocessing.

Parallel DBMSs

Parallel Database Architectures:

DBMS running multiple processors and disks designed to execute operations in parallel, whenever possible, to improve performance.

Based on premise that single processor systems can no longer meet requirements for cost-effective scalability parallel DBMSs, reliability, and performance.

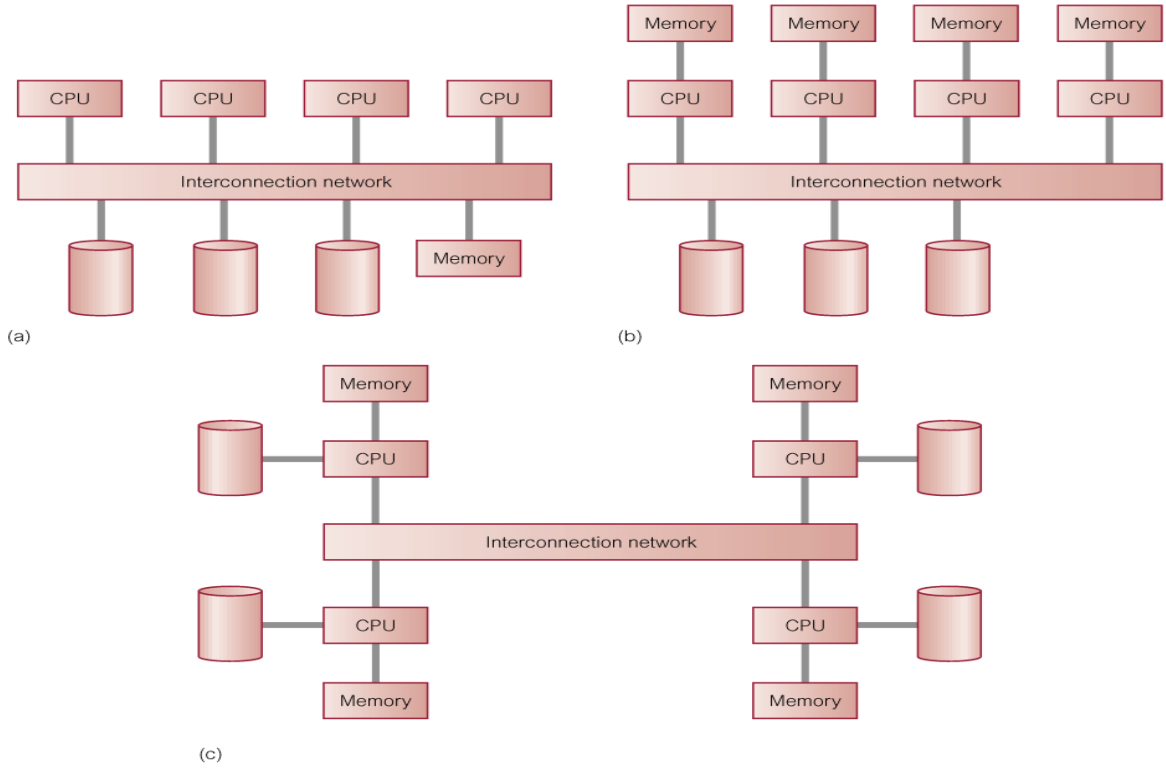
Parallel DBMSs link multiple, smaller machines to achieve same throughput as single, larger machine, with greater scalability and reliability.

Main architectures for Parallel Database are:

a. Shared memory

b. Shared disk

c. Shared nothing.



Advantages of DDBMS:

- Reflects organizational structure

- Improved shareablility and local autonomy
- Improved availability
- Improved reliability
- Improved performance
- Modular growth
- Less danger on single-point failure

Disadvantages of DDBMS:

- Complexity
- Cost
- Security
- Integrity control more difficult
- Lack of standards
- Lack of experience
- Database design more complex
- Possible slow response

Types of DDBMS:

1)Homogeneous DDBMSs

2) Heterogeneous DDBMSs

- Sites may run different DBMS products, with possibly different underlying data models.
- Occurs when sites have implemented their own databases and integration is considered later.

Homogeneous DDBMS:

- In homogeneous DDBMS, all sites use the same DBMS product.
- Much easier to design and manage.
- This design provides incremental growth by making additional new sites to DDBMS easy

- Allows increased performance by exploiting the parallel processing capability of multiple sites.

Heterogeneous DDBMSs :

- In heterogeneous DDBMS, all sites may run different DBMS products, which need not to be based on the same underlying data model and so the system may be composed of RDBMS, ORDBMS and OODBMS products.
- In heterogeneous system, communication between different DBMS are required for translations.

In order to provide DBMS transparency, users must be able to make requests in the language of the DBMS at their local site.

Data from the other sites may have

- Different hardware,
- Different DBMS products and combination of
- Different hardware and DBMS products.

The task for locating those data and performing any necessary translation are the abilities of heterogeneous DDBMS.Typical solution is to use gateways.

Open Database access and interoperability:

"The Open Group" formed Specification Working Group (SWG)

to provide specifications that create database infrastructure environment where there is:

- Common SQL API :Allows client applications to be written that do not need to know vendor of DBMS they are accessing.
- Common database protocol: Enables DBMS from one vendor to communicate directly with DBMS from another vendor without need for a gateway.
- Common network protocol: Allows communications between different DBMSs.

Multidatabase system (MDBS)

MDBS: DDBMS where each site maintains complete autonomy

Resides transparently on top of existing database and file systems presents a single database to its users.

Allows users to access and share data without requiring physical database integration.

MDBS are of 2 types:

Federated MDBS: looks like a DDBMS for global users and a centralized DBMS for local users.

Unfederated MDBS: has no “local” users

Functions and Architecture of a DDBMS:

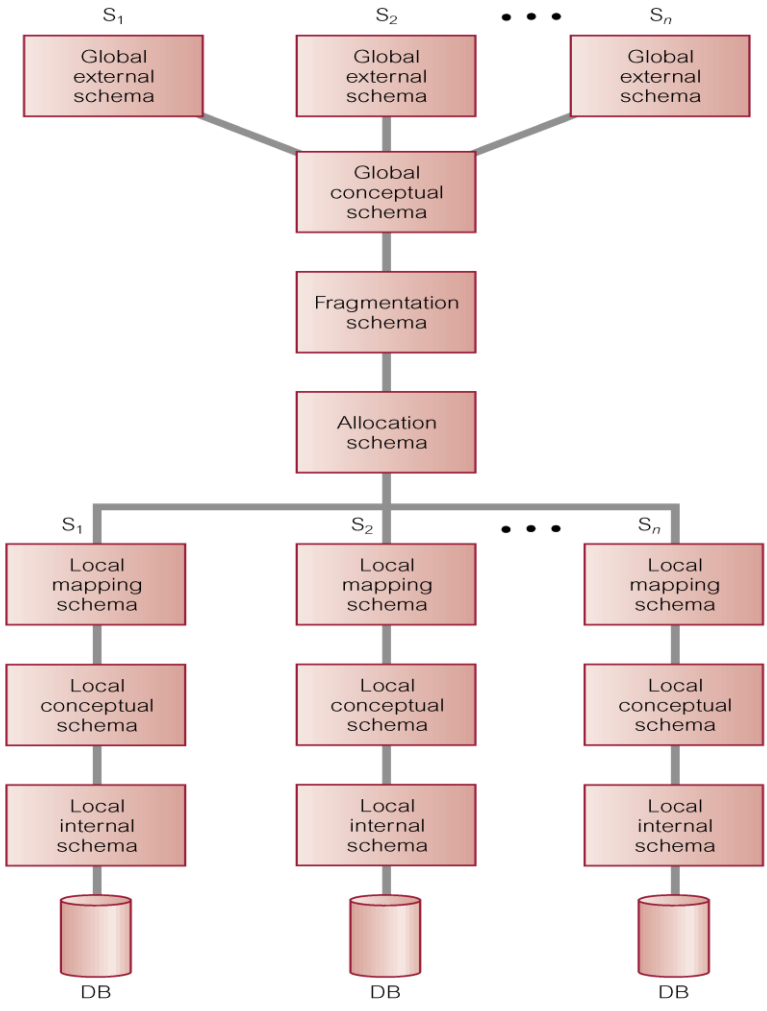
Functions of a DDBMS:

Expect DDBMS to have at least the functionality of a DBMS.

Also to have following functionality:

- Extended communication services.
- Extended Data Dictionary.
- Distributed query processing.
- Extended concurrency control.
- Extended recovery services.

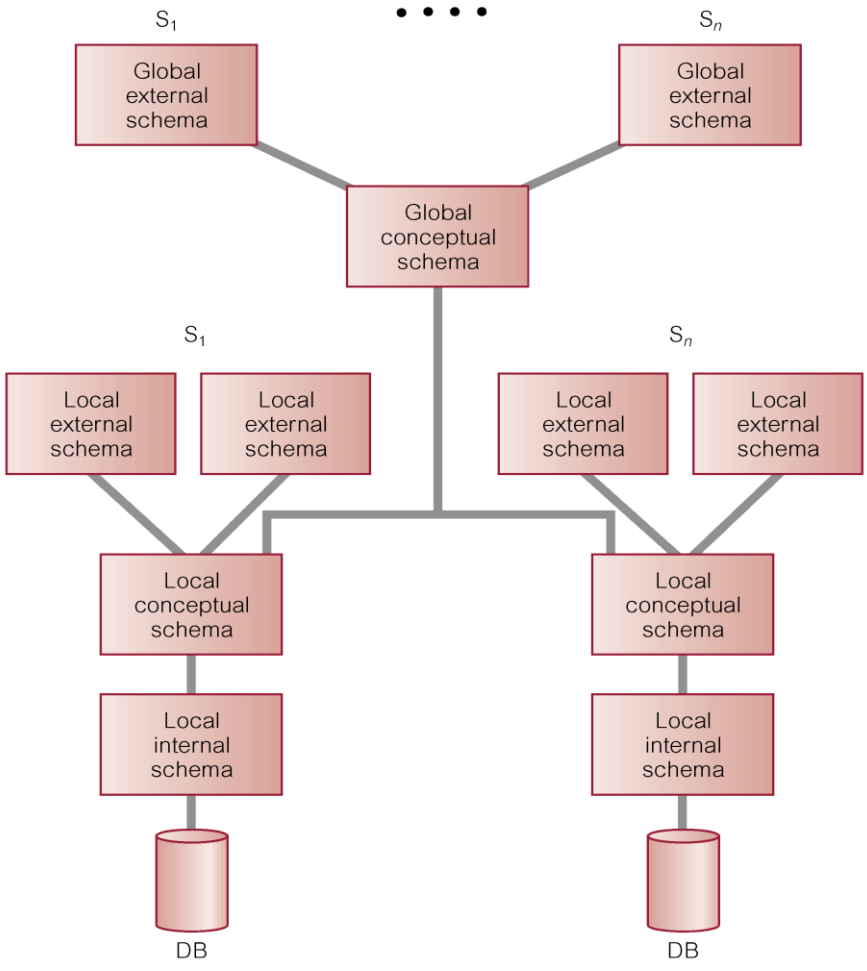
DDBMS Reference Architecture:



A reference architecture consists of:

- Set of global external schemas.
- Global conceptual schema (GCS).
- Fragmentation schema and allocation schema
- Set of schemas for each local DBMS conforming to 3-level ANSI/SPARC.

Reference Architecture for Tightly-Coupled FMDBS



Comparison with federated MDBS:

In DDBMS: GCS is union of all local conceptual schemas.

In FMDBS: GCS is subset of local conceptual schemas (LCS), consisting of data that each local system agrees to share.

GCS of tightly coupled system involves integration of either parts of LCSs or local external schemas.

FMDBS with no GCS is called loosely coupled.

Component Architecture for a DDBMS

- Local DBMS (LDBMS) component - It has its own local system catalog that stores information about the data held at that site.
- Data communications (DC) component – is the software that enables all sites to communicate with each other.
- Global System Catalog (GSC) - The GSC holds information specific to the distributed nature of the system, such as the fragmentation and allocation schemas.
- Distributed DBMS component - is the controllingunit of the entire system**.**

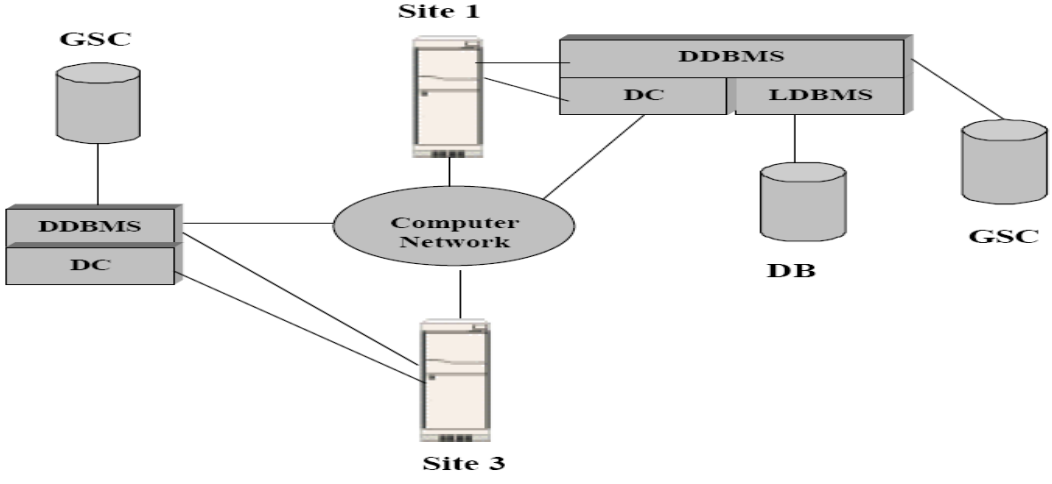


Fig 2 Components of DDBMS

.

Distributed Database Design:

Three key issues:

Fragmentation

Relation may be divided into a number of sub-relations, which are then distributed.

Allocation

Each fragment is stored at site with "optimal" distribution (see principles of distribution design).

Replication

Copy of fragment may be maintained at several sites.

Involves analyzing most important applications, based on quantitative/qualitative information.

Quantitative information (replication) used for may include:

- frequency with which an application is run;
- site from which an application is run;
- performance criteria for transactions and applications.

Qualitative information (fragmentation) may include

- transactions that are executed by application: relations, attributes and tuples.
- The type of access
- The predicates of read operation

Definition and allocation of fragments carried out strategically to achieve:

- Locality of Reference: Data should be stored close to where it is used
- Improved Reliability and Availability: Reliability and Availability are improved by replication.There is another copy of the fragment available at another site in the event of failure.
- Improved Performance: Bad allocation may result in bottlenecks,a significant degradation in performance.
- Balanced Storage Capacities and Costs: Availability and cost of storage should be taken into considerations so that cheap mass storage can be used.

– Minimal Communication Costs: Retrieval costs are minimized when locality of reference is maximized. however when replicated data item is updated the update is to be performed at all sites holding a duplicate copy, there by increasing the communication costs.

Data Allocation:

Centralized: Consists of single database and DBMS stored at one site with users distributed across the network.

Partitioned: Database partitioned into fragments, each fragment assigned to one site.

Complete Replication: Consists of maintaining complete copy of database at each site.

Selective Replication: Combination of partitioning, replication, and centralization.

Comparison of Strategies for Data Distribution:

Table 22.3 Comparison of strategies for data allocation.

	Locality of reference	Reliability and availability	Performance	Storage costs	Communication costs
Centralized	Lowest	Lowest	Unsatisfactory	Lowest	Highest
Fragmented	High[a]	Low for item; high for system	Satisfactory[a]	Lowest	Low[a]
Complete replication	Highest	Highest	Best for read	Highest	High for update; low for read
Selective replication	High[a]	Low for item; high for system	Satisfactory[a]	Average	Low[a]

[a] Indicates subject to good design.

Why Fragment?

Usage : Applications work with views rather than entire relations.

Efficiency :Data is stored close to where it is most frequently used. Data that is not needed by local applications is not stored.

Parallelism: With fragments as the unit of distribution, a transaction can be divided into several subqueries that operate on fragments.this should increase the degree of concurrency.

Security :Data not required by local applications is not stored and so not available to unauthorized users.

Disadvantages:

Performance: The performance of global applications that require data from several fragments located at different sites may be slower.

Integrity: Integrity control may be more difficult if data and functional dependencies are fragmented and located at different sites.

Correctness of Fragmentation:

Three correctness rules:

Completeness

If relation R is decomposed into fragments R_1, R_2, ... R_n, each data item that can be found in R must appear in at least one fragment.

Reconstruction

Must be possible to define a relational operation that will reconstruct R from the fragments.Reconstruction for horizontal fragmentation is Union operation and Join for vertical .

Disjointness:

If data item d_i appears in fragment R_i, then it should not appear in any other fragment.; Exception: vertical fragmentation, where primary key attributes must be repeated to allow reconstruction.

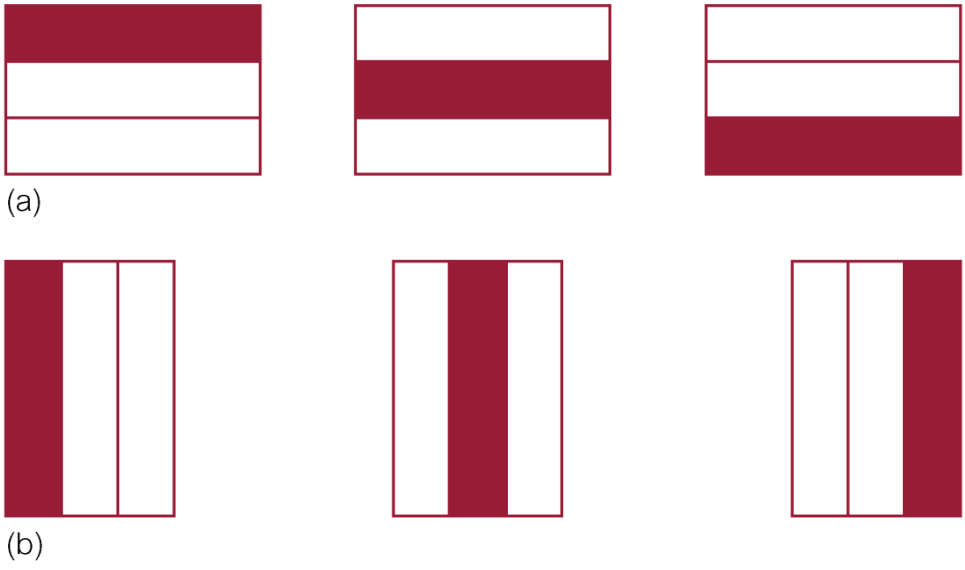
- For horizontal fragmentation, data item is a tuple
- For vertical fragmentation, data item is an attribute.

Types of Fragmentation

Four types of fragmentation:

- Horizontal
- Vertical
- Mixed

Horizontal and Vertical Fragmentation



(a) Horizontal Fragmentation (b)Vertical Fragmentation

Horizontal Fragmentation

Consists of a subset of the tuples of a relation.

Defined using Selection operation of relational algebra:

$$\sigma_p(R)$$

For example:

$$P_1 = \sigma_{type='House'}(PropertyForRent)$$

$$P_2 = \sigma_{type='Flat'}(PropertyForRent)$$

Result (PNo., St, City, postcode,type,room,rent,ownerno.,staffno., branchno.)

This strategy is determined by looking at predicates used by transactions.

Reconstruction involves using a union eg R = r1 U r2

Vertical Fragmentation

Consists of a subset of attributes of a relation.

Defined using Projection operation of relational algebra:

$$\Pi_{a1, \ldots, an}(R)$$

For example:

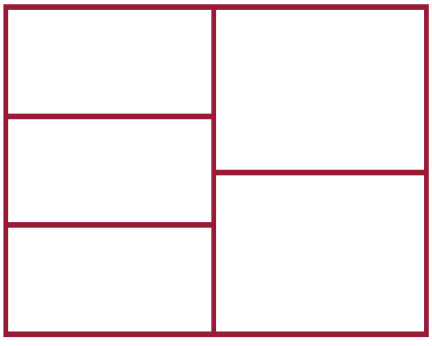
$$S_1 = \Pi_{staffNo, position, sex, DOB, salary}(Staff)$$

$$S_2 = \Pi_{staffNo, fName, lName, branchNo}(Staff)$$

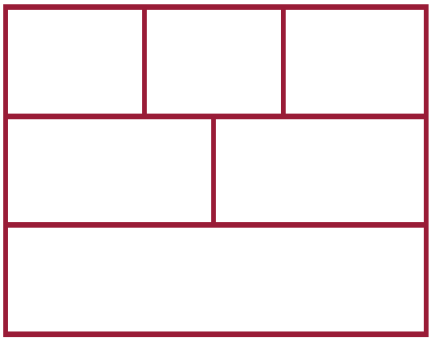
Determined by establishing *affinity* of one attribute to another.

For vertical fragements reconstruction involves the join operation; Each fragment is disjointed except for the primary key

Mixed Fragmentation



(a)



(b)

Consists of a horizontal fragment that is vertically fragmented, or a vertical fragment that is horizontally fragmented.

Defined using Selection and Projection operations of relational algebra:

$$\sigma_p(\Pi_{a1, \ldots, an}(R)) \quad \text{or}$$

$$\Pi_{a1, \ldots, an}(\sigma_p(R))$$

Eg: $S_1 = \Pi_{staffNo, position, sex, DOB, salary}(Staff)$

$S_2 = \Pi_{staffNo, fName, lName, branchNo}(Staff)$

$S_{21} = \sigma_{branchNo='B003'}(S_2)$

$S_{22} = \sigma_{branchNo='B005'}(S_2)$

$S_{23} = \sigma_{branchNo='B007'}(S_2)$

Transparencies in a DDBMS

Transparency hides implementation details from users.

Overall objective: equivalence to user of DDBMs to centralised DBMS

- FULL transparency not universally accepted objective

1)Distribution Transparency

- Fragmentation Transparency
- Location Transparency
- Replication Transparency
- **Local mapping transparency**

2)Transaction Transparency

- Concurrency Transparency
- Failure Transparency

3)Performance Transparency

4) DBMS Transparency

Distribution Transparency:

If DDBMS exhibits distribution transparency, user does not need to know:

Fragmentation transparency:

Fragmentation is the highest level of distribution transparency. If fragmentation transparency is provided by the DDBMS, then the user does not need to know that the data is fragmented, As a result, database accesses are based on the global schema,. so the user does not need to specify fragment names or data locations.

Eg: $\mathbf{S_1 = \prod_{staffNo,\ position,\ sex,\ DOB,\ salary}(Staff)}$

$\mathbf{S_2 = \prod_{staffNo,\ fName,\ lName,\ branchNo}(Staff)}$

$\mathbf{S_{21} = \sigma_{branchNo='B003'}(S_2)}$

$S_{22} = \sigma_{branchNo='B005'}(S_2)$

$S_{23} = \sigma_{branchNo='B007'}(S_2)$

Eg: Select fname,lname from staff where position=‘manager’;

Location transparency:

Location is the middle level of distribution transparency. With location transparency, the user must know how the data has been fragmented but still does not have to know the location of the data.

Eg: Select fname,lname from S21 where staffno IN(select satffno from s1 where position=‘manager’)

Replication transparency:

Closely related to location transparency is replication transparency, which means that the user is unaware of the replication of fragments. Replication transparency is implied' by location transparency.

Local mapping transparency:

This is the lowest level of distribution transparency. With local mapping transparency, user needs to specify both fragment names and the location of data items, taking into consideration any replication that may exists.

Eg:Select fname,lname from S21 AT SITE3 where staffno IN(select staffno from s1 AT SITE5 where(position=‘manager’)

Clearly, this is a more complex and time-consuming query for the user to enter than the first. It is unlikely that a system that provides only this level of transparency would be acceptable to end-users.

Naming transparency: Each item in a DDB must have a unique name.

One solution: create central name server. But the drawbacks are

- loss of some local autonomy.
- central site may become a bottleneck.
- low availability: if the central site fails.

Alternative solution: prefix object with identifier of creator site, each fragment and its copies. Then each site uses alias. However, this results in:

For example, Branch created at site S1 might be named S1.BRANCH.

Also need to identify each fragment and its copies.

Thus, copy 2 of fragment 3 of Branch created at site S1 might be referred to as S1.BRANCH.F3.C2.

However, this results in loss of distribution transparency

An approach that resolves these problems uses aliases for each database object.

Thus, S1.BRANCH.F3.C2 might be known as LocalBranch by user at site S1.

DDBMS has task of mapping an alias to appropriate database object.

Transaction Transparency:

Ensures all distributed Ts maintain distributed database's integrity and consistency.

- Distributed Transaction(T) accesses data stored at more than one location.
- Each T is divided into no. of subTs, one for each site that has to be accessed.
- DDBMS must ensure the indivisibility of both the global T and each of the subTs.
- Ensures that all distributed transactions maintain distributed database's integrity and consistency.

Distributed Transaction

- T prints out names of all staff, using schema defined above as S1, S2, S21, S22, and S23. Define three subtransactions TS3, TS5, and TS7 to represent agents at sites 3, 5, and 7.

Time	T_{S_3}	T_{S_5}	T_{S_7}
t_1	begin_transaction	begin_transaction	begin_transaction
t_2	read(fName, lName)	read(fName, lName)	read(fName, lName)
t_3	print(fName, lName)	print(fName, lName)	print(fName, lName)
t_4	end_transaction	end_transaction	end_transaction

Concurrency transparency: All Ts must execute independently and be logically consistent with results obtained if Ts executed in some arbitrary serial order.

Replication makes concurrency more complex.

If a copy of a replicated data item is updated, update must be propagated to all copies.

Could propagate changes as part of original transaction, making it an atomic operation.

However, if one site holding copy is not reachable, then transaction is delayed until site is reachable.

Could limit update propagation to only those sites currently available. Remaining sites updated when they become available again.

Could allow updates to copies to happen asynchronously, sometime after the original update. Delay in regaining consistency may range from a few seconds to several hours.

Failure transparency: must ensure atomicity and durability of global T.

DDBMS must ensure atomicity and durability of global transaction.

Means ensuring that subtransactions of global transaction either all commit or all abort.

Thus, DDBMS must synchronize global transaction to ensure that all subtransactions have completed successfully before recording a final COMMIT for global transaction.

Must do this in presence of site and network failures.

Means ensuring that subTs of global T either all commit or all abort.

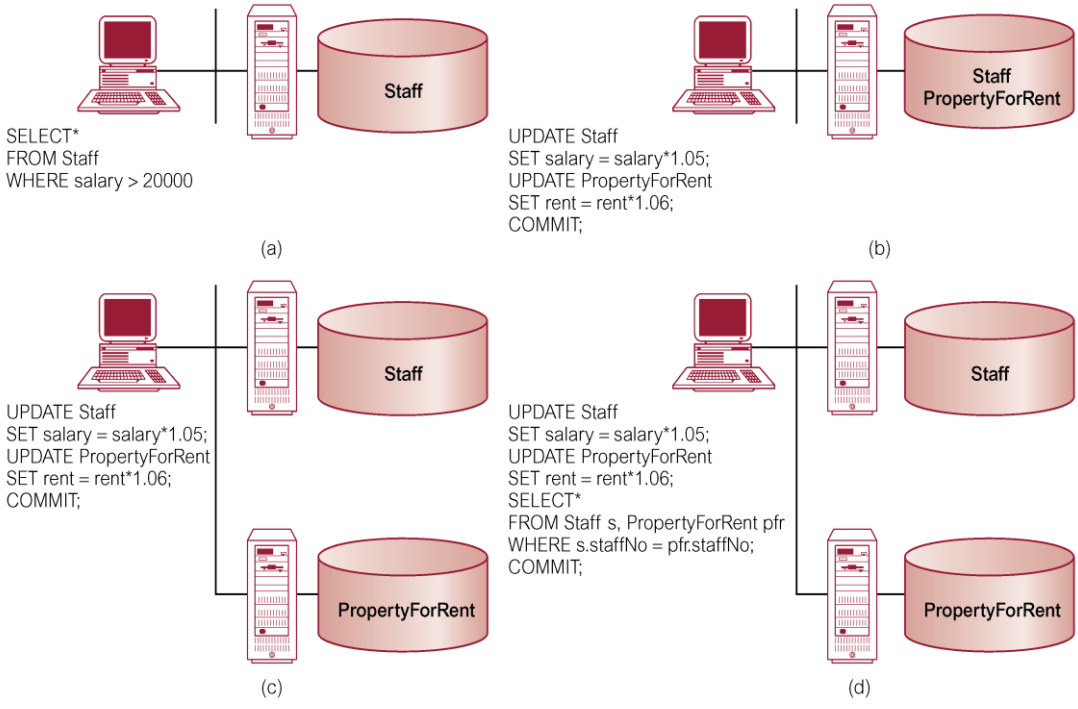
Classification transactons: In IBM's Distributed Relational Database Architecture (DRDA), four types of Ts:

a)Remote request

b)Remote unit of work

c)Distributed unit of work

d)Distributed request.



Performance Transparency

Performance transparency requires a DDBMS to perform as if it were a centralized DBMS. In a distributed environment, the system should suffer any performance degradation due to the distributed architecture, for example the presence of the network Performance transparency also requires the DDBMS to determine the most cost-effective strategy to execute a request.

- In a centralized DBMS, the query processor (QP) must evaluate every data request and find an optimal execution strategy, consisting of an ordered sequence

of operations on the database. In a distributed environment, the distributed query processor (DQP) maps a data request into an ordered sequence of operations on the local databases. It has the added complexity of taking into account the fragmentation, replication and allocation schemas. The DQP has to decide:

- Which fragment to access?
- Which copy of fragment to use, if the fragment is replicated?
- Which location to use.

The DQP produces an execution strategy that is optimized with respect to some cost function. Typically, the costs associated with a distributed request include:

- The access time (I/O) cost involved in accessing the physical data on disk;
- The CPU time cost incurred when performing operations on data in main memory;
- The communication cost associated with the transmission of data across the network.

The first two factors are the only ones considered in a centralized system. In· a distributes environment, the DDBMS must take account of the communication cost, which may be the most dominant factor in WANs with a bandwidth of a few kilobytes per second. In such cases, optimization may ignore I/O and CPU costs. However, LANs have a bandwidth comparable to that of disks, so in such cases optimization should not ignore I/O and CPU costs entirely.

- Example:

Property(propNo, city) 10000 records in London

Client(clientNo,maxPrice) 100000 records in Glasgow

Viewing(propNo, clientNo) 1000000 records in London

SELECT p.propNo

FROM Property p INNER JOIN

Client c INNER JOIN Viewing v ON c.clientNo = v.clientNo)

ON p.propNo = v.propNo

WHERE p.city='Aberdeen' AND c.maxPrice > 200000;

This query selects properties that viewed in aberdeen that have a price greater than £200, 000

Assume:

Each tuple in each relation is 100 characters long.

10 renters with maximum price greater than £200,000.

100 000 viewings for properties in Aberdeen.

In addition the data transmission rate is 10,000 characters per sec and there is a 1 sec access delay to send a message.

Table 22.4 Comparison of distributed query processing strategies.

	Strategy	Time
(1)	Move Client relation to London and process query there	16.7 minutes
(2)	Move Property and Viewing relations to Glasgow and process query there	28 hours
(3)	Join Property and Viewing relations at London, select tuples for Aberdeen properties, and for each of these in turn, check at Glasgow to determine if associated maxPrice > £200,000	2.3 days
(4)	Select clients with maxPrice > £200,000 at Glasgow and for each one found, check at London for a viewing involving that client and an Aberdeen property	20 seconds
(5)	Join Property and Viewing relations at London, select Aberdeen properties, and project result over propertyNo and clientNo and move this result to Glasgow for matching with maxPrice > £200,000	16.7 minutes
(6)	Select clients with maxPrice > £200,000 at Glasgow and move the result to London for matching with Aberdeen properties	1 second

DBMS Transparency

DBMS transparency hides the knowledge that the local DBMSs may be different, and is therefore only applicable to heterogeneous DDBMSs. It is one of the most difficult transparencies to provide as a generalization.

Date's 12 Rules for DDBMS

Fundamental Principle

To the user, a distributed system should look exactly like a nondistributed system.

1.Local Autonomy
2.No Reliance on a Central Site
3.Continuous Operation
4.Location Independence
5.Fragmentation Independence
6.Replication Independence
7.Distributed Query Processing
8.Distributed Transaction Processing
9.Hardware Independence
10.Operating System Independence
11.Network Independence
12.Database Independence

Last four rules are ideals.

Distributed DBMSs - Advanced Concepts

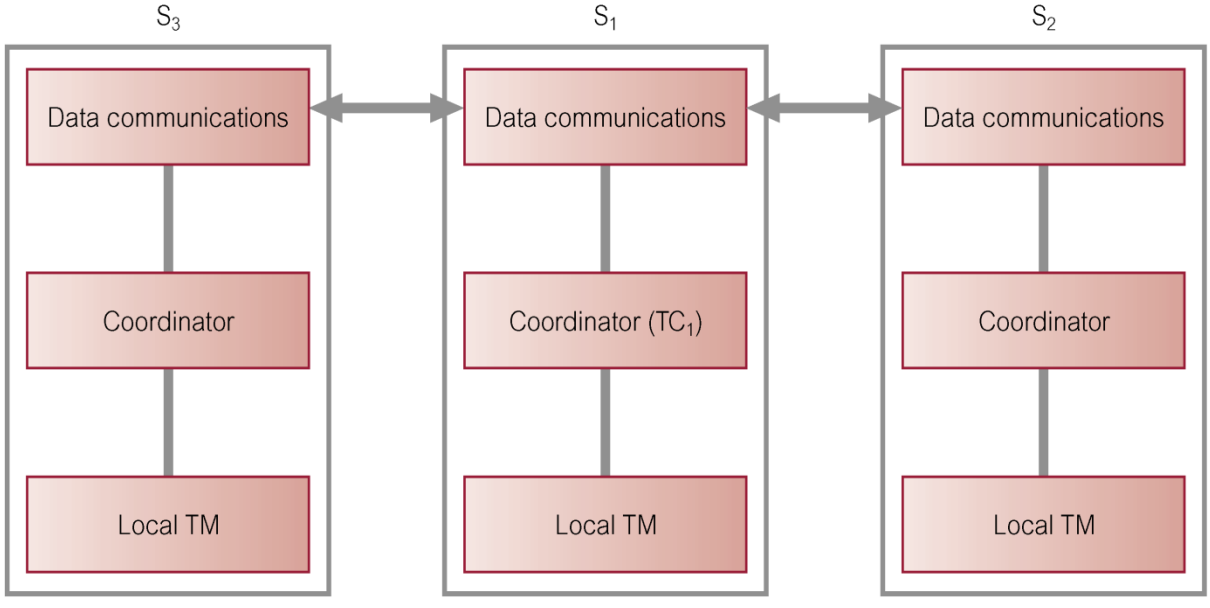
Distributed Transaction Management

- Distributed transaction accesses data stored at more than one location.

- Divided into a number of sub-transactions, one for each site that has to be accessed, represented by an agent.
- Indivisibility of distributed transaction is still fundamental to transaction concept.
- DDBMS must also ensure indivisibility of each sub-transaction.

Thus, DDBMS must ensure:

- synchronization of subtransactions with other local transactions executing concurrently at a site;
- synchronization of subtransactions with global transactions running simultaneously at same or different sites.
- Global transaction manager (transaction coordinator) at each site, to coordinate global and local transactions initiated at that site.



Distributed Locking Protocols:

Four schemes:

- Centralized 2PL .
- Primary Copy 2PL.

- Distributed 2PL.
- Majority Locking.

Centralized 2PL

- Single site that maintains all locking information.
- One lock manager for whole of DDBMS.
- Local transaction managers involved in global transaction request and release locks from lock manager.
- Or transaction coordinator can make all locking requests on behalf of local transaction managers.

  Advantage - easy to implement.

  Disadvantages - bottlenecks and lower reliability.

Primary Copy 2PL :

- Lock managers distributed to a number of sites.
- Each lock manager responsible for managing locks for set of data items.
- For replicated data item, one copy is chosen as primary copy, others are slave copies
- Only need to write-lock primary copy of data item that is to be updated.
- Once primary copy has been updated, change can be propagated to slaves.

Disadvantages - deadlock handling is more complex; still a degree of centralization in system.

Advantages - lower communication costs and better performance than centralized 2PL.

Distributed 2PL :

- Lock managers distributed to every site.
- Each lock manager responsible for locks for data at that site.
- If data not replicated, equivalent to primary copy 2PL.
- Otherwise, implements a Read-One-Write-All (ROWA) replica control protocol.
- Using ROWA protocol:
  - Any copy of replicated item can be used for read.
  - All copies must be write-locked before item can be updated.

Disadvantages - deadlock handling more complex; communication costs higher than primary copy 2PL.

Majority Locking:

- Extension of distributed 2PL.
- To read or write data item replicated at n sites, sends a lock request to more than half the n sites where item is stored.
- Transaction cannot proceed until majority of locks obtained.
- Overly strong in case of read locks.

Distributed Timestamping:

- Objective is to order transactions globally so older transactions (smaller timestamps) get priority in event of conflict.
- In distributed environment, need to generate unique timestamps both locally and globally.
- System clock or incremental event counter at each site is unsuitable.
- Concatenate local timestamp with a unique site identifier: <local timestamp, site identifier>.
- Site identifier placed in least significant position to ensure events ordered according to their occurrence as opposed to their location.
- To prevent a busy site generating larger timestamps than slower sites:
- Each site includes their timestamps in messages.
- Site compares its timestamp with timestamp in message and, if its timestamp is smaller, sets it to some value greater than message timestamp

Distributed Deadlock Management:

More complicated if lock management is not centralized.

Local Wait-for-Graph (LWFG) may not show existence of deadlock.

May need to create GWFG, union of all LWFGs.

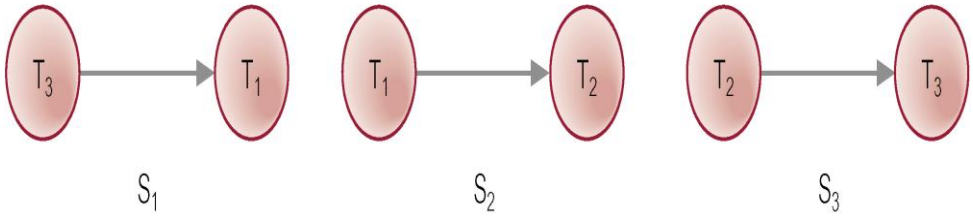
Look at three schemes:

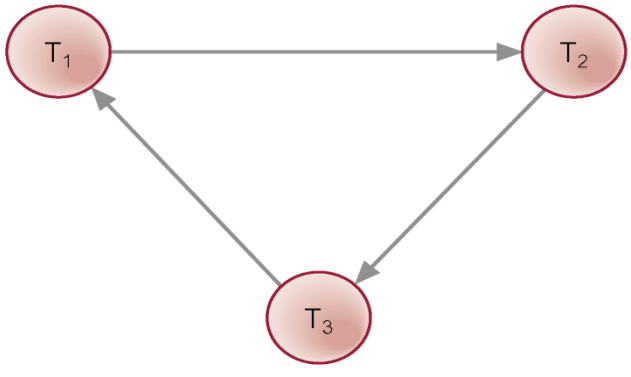
- Centralized Deadlock Detection.
- Hierarchical Deadlock Detection.
- Distributed Deadlock Detection.

Example - Distributed Deadlock

- T_1 initiated at site S_1 and creating agent at S_2,
- T_2 initiated at site S_2 and creating agent at S_3,
- T_3 initiated at site S_3 and creating agent at S_1.

Time	S_1	S_2	S_3
t_1	read_lock(T_1, x_1)	write_lock(T_2, y_2)	read_lock(T_3, z_3)
t_2	write_lock(T_1, y_1)	write_lock(T_2, z_2)	
t_3	write_lock(T_3, x_1)	write_lock(T_1, y_2)	write_lock(T_2, z_3)





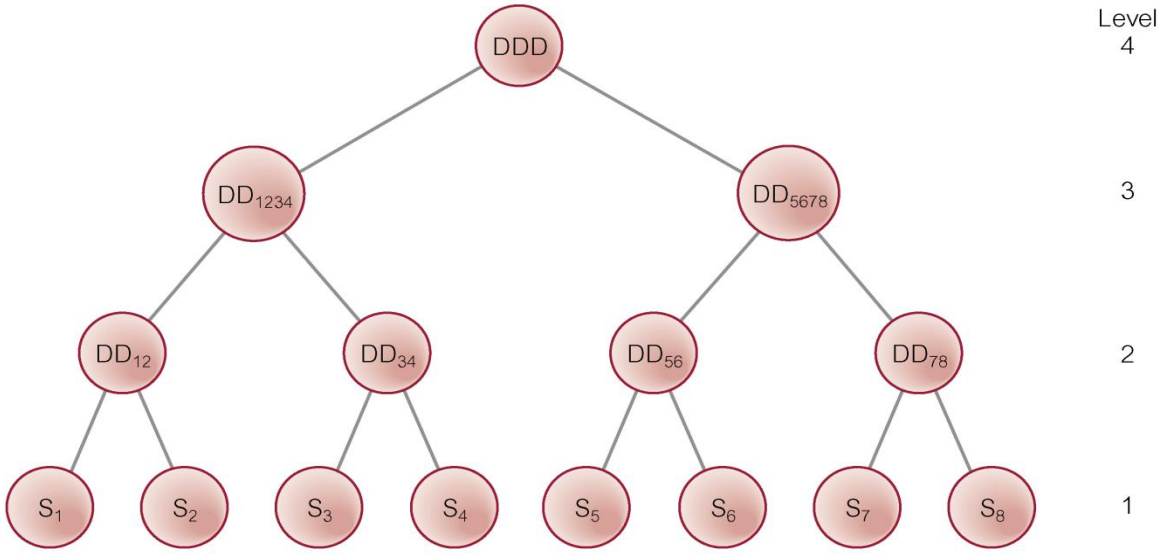
Centralized Deadlock Detection

- Single site appointed deadlock detection coordinator (DDC).
- DDC has responsibility for constructing and maintaining GWFG.

- If one or more cycles exist, DDC must break each cycle by selecting transactions to be rolled back and restarted.

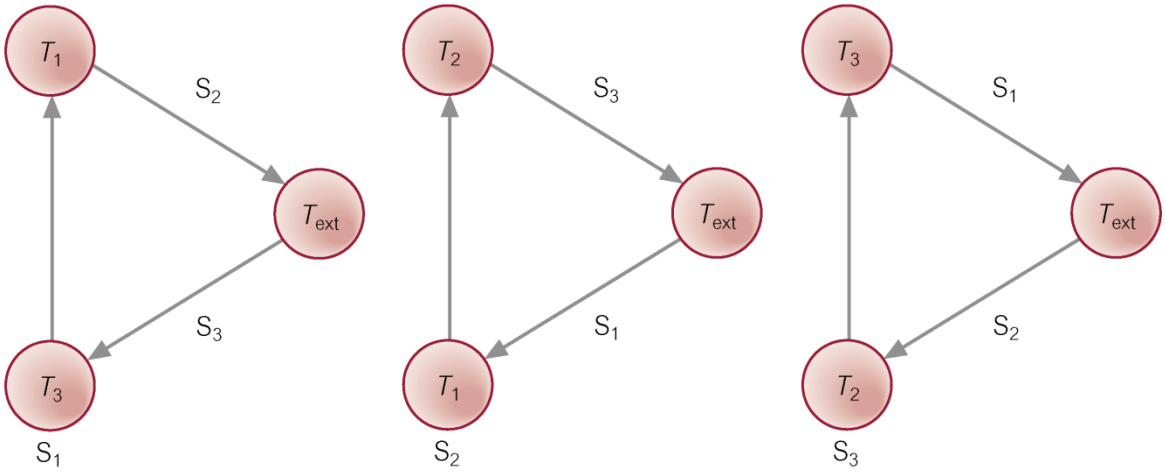
Hierarchical Deadlock Detection

- Sites are organized into a hierarchy.
- Each site sends its LWFG to detection site above it in hierarchy.
- Reduces dependence on centralized detection site.



Distributed Deadlock Detection

- Most well-known method developed by Obermarck (1982).
- An external node, T_{ext}, is added to LWFG to indicate remote agent.
- If a LWFG contains a cycle that does not involve T_{ext}, then site and DDBMS are in deadlock.
- Global deadlock may exist if LWFG contains a cycle involving T_{ext}.
- To determine if there is deadlock, the graphs have to be merged.
- Potentially more robust than other methods.



S_1: $T_{ext} \rightarrow T_3 \rightarrow T_1 \rightarrow T_{ext}$

S_2: $T_{ext} \rightarrow T_1 \rightarrow T_2 \rightarrow T_{ext}$

S_3: $T_{ext} \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ext}$

Transmit LWFG for S_1 to the site for which transaction T_1 is waiting, site S_2.

LWFG at S_2 is extended and becomes:

S_2: $T_{ext} \rightarrow T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_{ext}$

Still contains potential deadlock, so transmit this WFG to S_3:

S_3: $T_{ext} \rightarrow T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_{ext}$

GWFG contains cycle not involving T_{ext}, so deadlock exists.

Distributed Database Recovery

Failures in Distributed Environment:

Four types of failure particular to distributed systems:

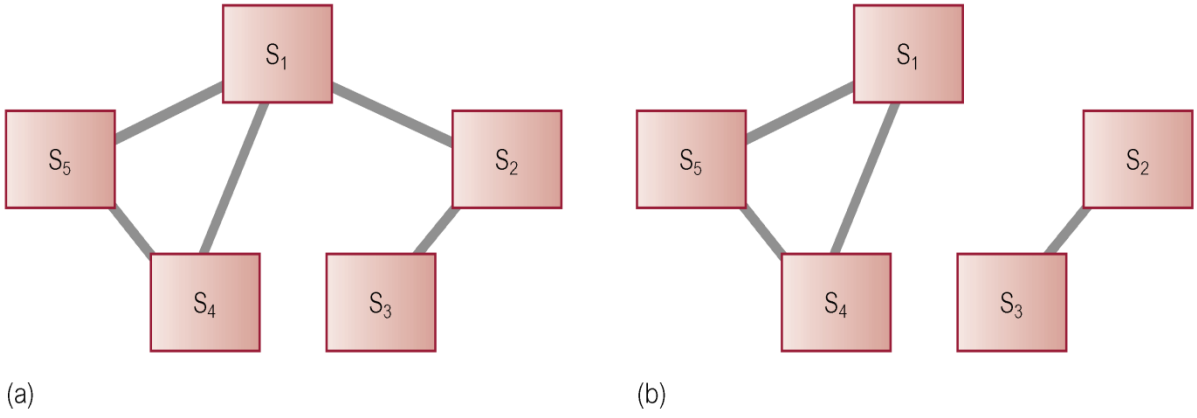
- Loss of a message.
- Failure of a communication link.
- Failure of a site.
- Network partitioning..

DDBMS is highly dependent on ability of all sites to be able to communicate reliably with one another.

Communication failures can result in network becoming split into two or more partitions.

May be difficult to distinguish whether communication link or site has failed.

Partitioning of a network



a)Before failure b)After Failure

Distributed Recovery Protocols

Two-Phase Commit (2PC)

Two phases: a voting phase and a decision phase.

Coordinator asks all participants whether they are prepared to commit transaction.

- If one participant votes abort, or fails to respond within a timeout period, coordinator instructs all participants to abort transaction.
- If all vote commit, coordinator instructs all participants to commit.

All participants must adopt global decision.

If participant votes abort, free to abort transaction immediately

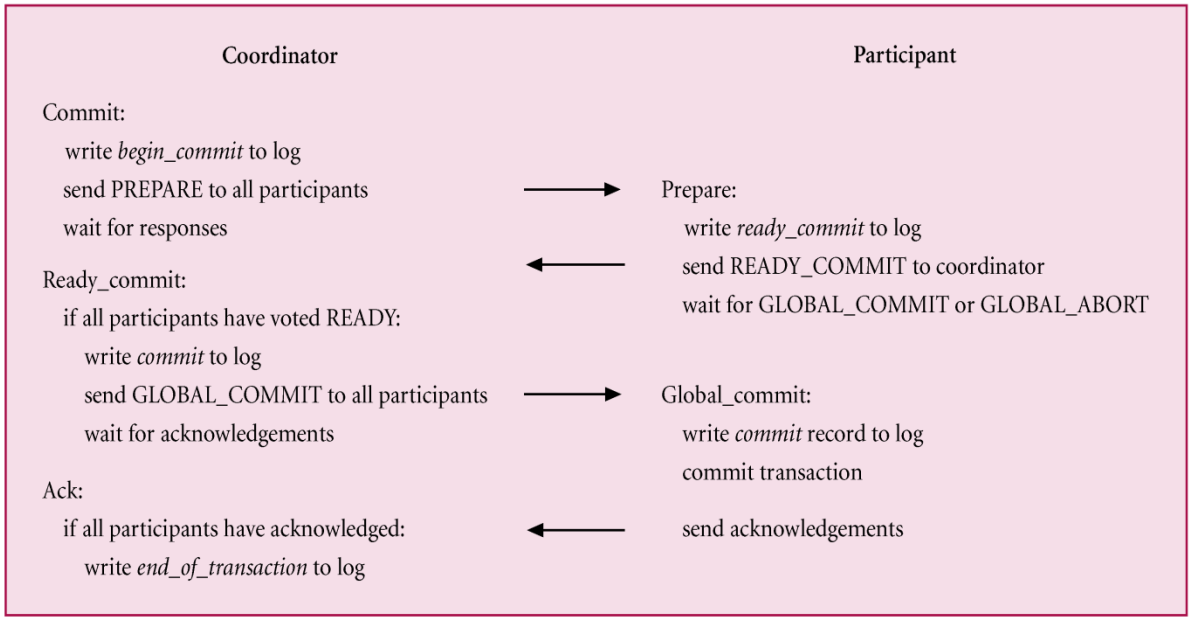
If participant votes commit, must wait for coordinator to broadcast global-commit or global-abort message.

Protocol assumes each site has its own local log and can rollback or commit transaction reliably.

If participant fails to vote, abort is assumed.

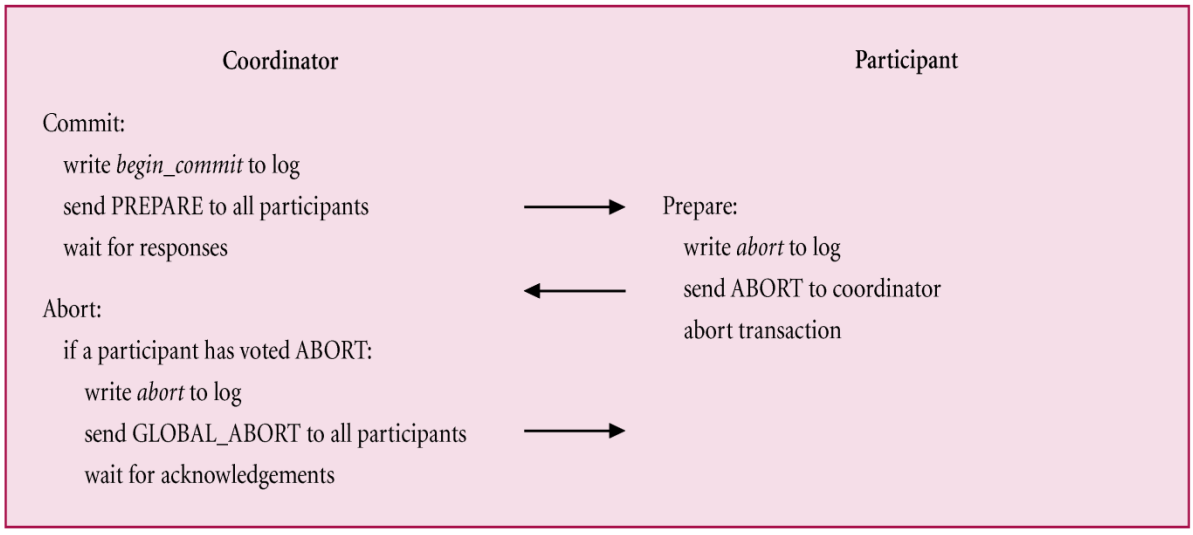
If participant gets no vote instruction from coordinator, can abort.

2PC Protocol for Participant Voting Commit



(a)

2PC Protocol for Participant Voting Abort



(b)

Termination Protocols

Invoked whenever a coordinator or participant fails to receive an expected message and times out.

Coordinator

Timeout in WAITING state

- Globally abort the transaction.

Timeout in DECIDED state

- Send global decision again to sites that have not acknowledged.

Termination Protocols – Participant

Simplest termination protocol is to leave participant blocked until communication with the coordinator is re-established. Alternatively:

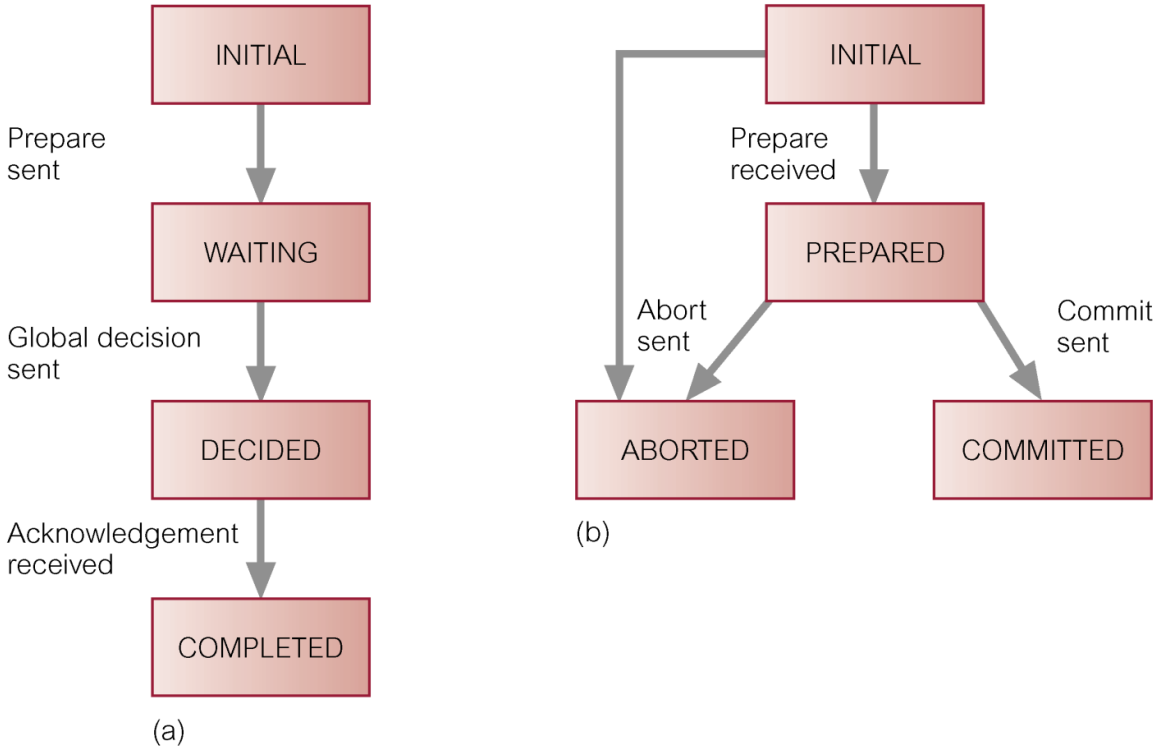
Timeout in INITIAL state

- Unilaterally abort the transaction.

Timeout in the PREPARED state

- Without more information, participant blocked.
- Could get decision from another participant .

State Transition Diagram for 2PC



(a) coordinator (b) participant

Recovery Protocols

Action to be taken by operational site in event of failure. Depends on what stage coordinator or participant had reached.

Coordinator Failure

Failure in INITIAL state

- Recovery starts the commit procedure.

Failure in WAITING state

- Recovery restarts the commit procedure.

2PC - Coordinator Failure

Failure in DECIDED state

- On restart, if coordinator has received all acknowledgements, it can complete successfully. Otherwise, has to initiate termination protocol discussed above.

Objective to ensure that participant on restart performs same action as all other participants and that this restart can be performed independently.

Failure in INITIAL state

- Unilaterally abort the transaction.

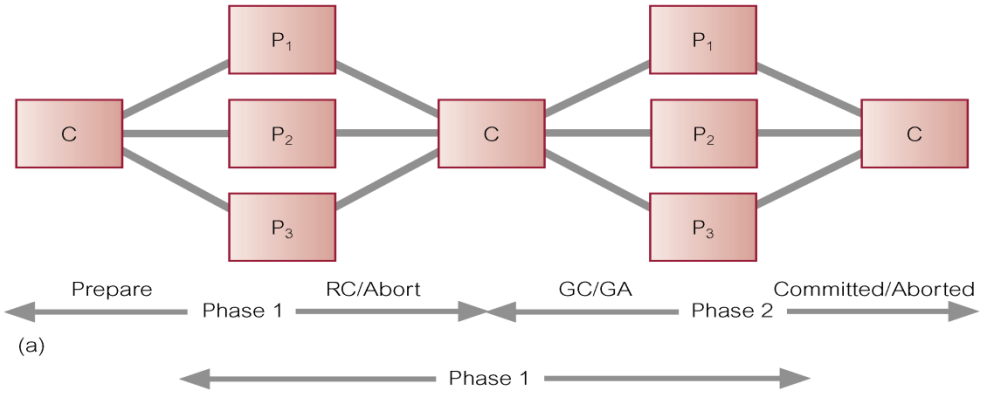
Failure in PREPARED state

- Recovery via termination protocol above.

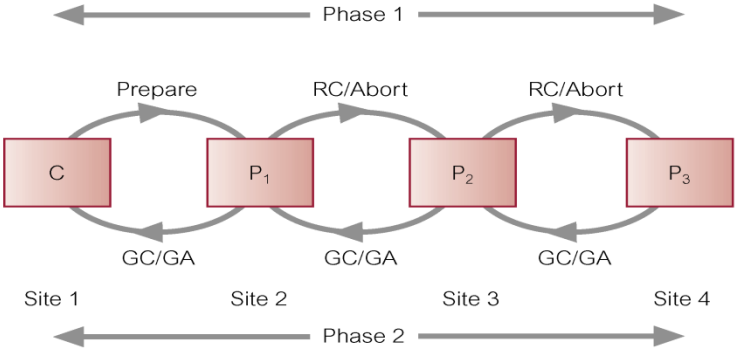
Failure in ABORTED/COMMITTED states

- On restart, no further action is necessary.

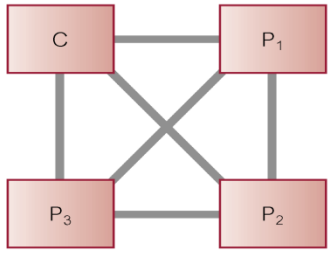
2PC Communication Topologies



(a)



(b)



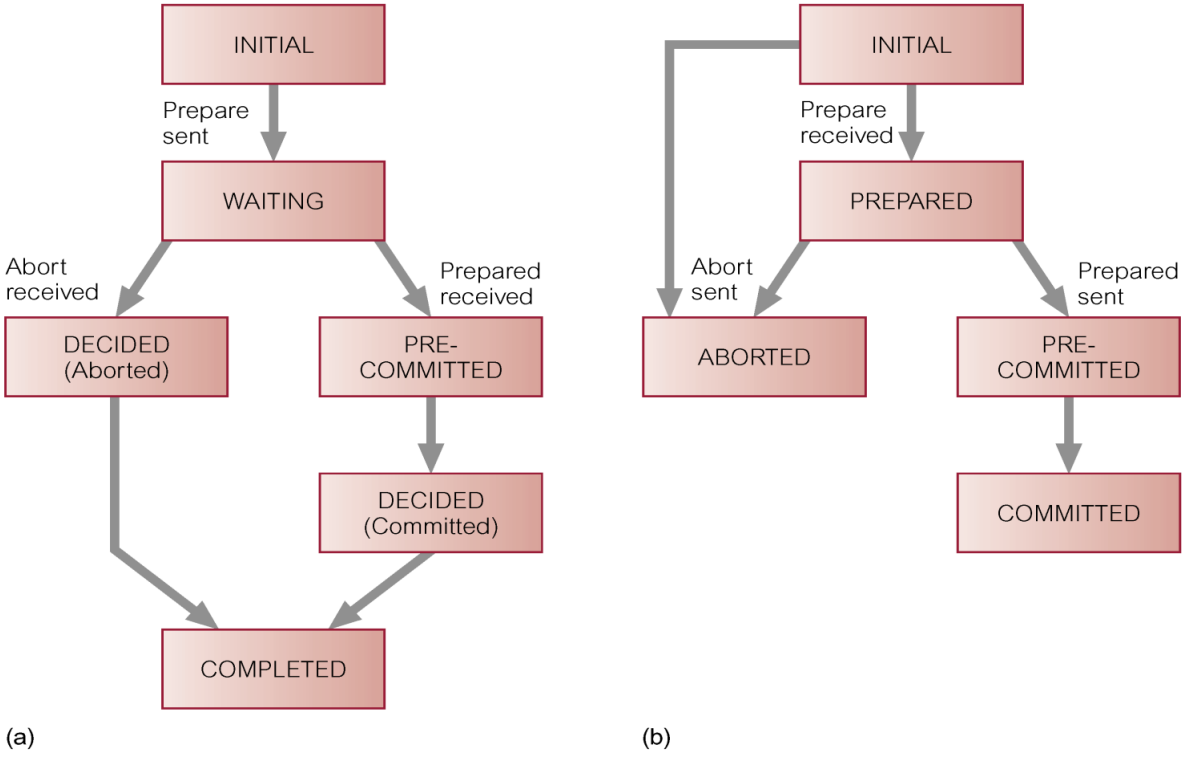
(c)

Three-Phase Commit (3PC)

- 2PC is not a non-blocking protocol.
- For example, a process that times out after voting commit, but before receiving global instruction, is blocked if it can communicate only with sites that do not know global decision.
- Probability of blocking occurring in practice is sufficiently rare that most existing systems use 2PC.
- Alternative non-blocking protocol, called three-phase commit (3PC) protocol.

- Non-blocking for site failures, except in event of failure of all sites.
- Communication failures can result in different sites reaching different decisions, thereby violating atomicity of global transactions.
- 3PC removes uncertainty period for participants who have voted commit and await global decision.
- Introduces third phase, called pre-commit, between voting and global decision.
- On receiving all votes from participants, coordinator sends global pre-commit message.
- Participant who receives global pre-commit, knows all other participants have voted commit and that, in time, participant itself will definitely commit.

State Transition Diagram for 3PC



(a) Coordinator (b) participant

Network Partitioning

If data is not replicated, can allow transaction to proceed if it does not require any data from site outside partition in which it is initiated.

Otherwise, transaction must wait until sites it needs access to are available.

If data is replicated, procedure is much more complicated.

Identifying Updates

Time	P_1	P_2
t_1	begin_transaction	begin_transaction
t_2	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	$\mathbf{bal_x} = \mathbf{bal_x} - 5$
t_3	write($\mathbf{bal_x}$)	
t_4	commit	commit
t_5		begin_transaction
t_6		$\mathbf{bal_x} = \mathbf{bal_x} - 5$
t_7		write($\mathbf{bal_x}$)
t_8		commit

Identifying Updates

- Successfully completed update operations by users in different partitions can be difficult to observe.
- In P_1, transaction withdrawn £10 from account and in P_2, two transactions have each withdrawn £5 from same account.
- At start, both partitions have £100 in bal_x, and on completion both have £90 in bal_x.
- On recovery, not sufficient to check value in bal_x and assume consistency if values same.

Maintaining Integrity

Time	P_1	P_2
t_1	begin_transaction	begin_transaction
t_2	$bal_x = bal_x - 60$	$bal_x = bal_x - 50$
t_3	write(bal_x)	write(bal_x)
t_4	commit	commit

- Successfully completed update operations by users in different partitions can violate constraints.
- Have constraint that account cannot go below £0.
- In P_1, withdrawn £60 from account and in P_2, withdrawn £50.
- At start, both partitions have £100 in bal_x, then on completion one has £40 in bal_x and other has £50.
- Importantly, neither has violated constraint.
- On recovery, bal_x is –£10, and constraint violated.

Network Partitioning

- Processing in partitioned network involves trade-off in availability and correctness.

- Correctness easiest to provide if no processing of replicated data allowed during partitioning.
- Availability maximized if no restrictions placed on processing of replicated data.
- In general, not possible to design non-blocking commit protocol for arbitrarily partitioned networks.

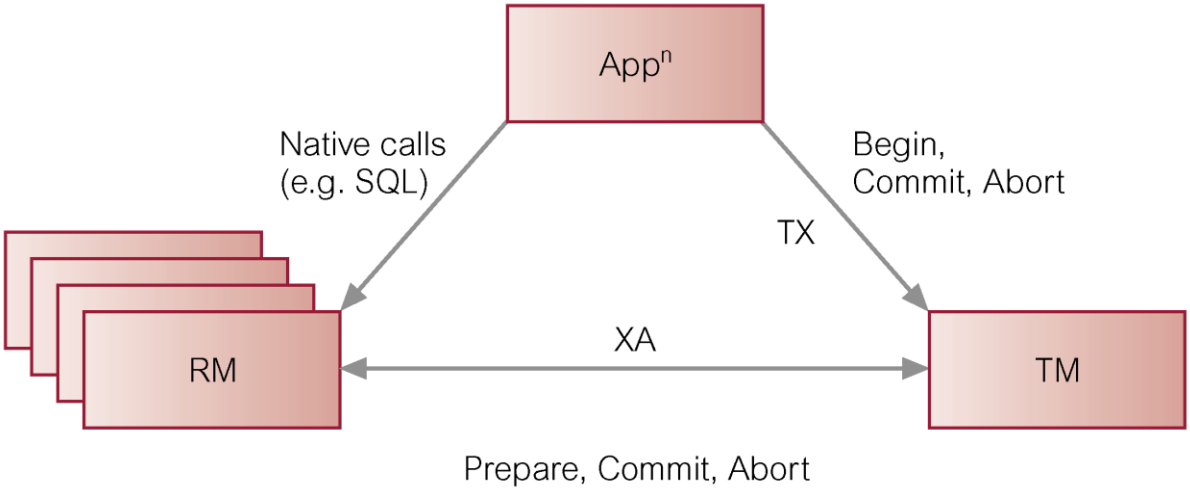
X/OPEN DTP Model

- Open Group is vendor-neutral consortium whose mission is to cause creation of viable, global information infrastructure.
- Formed by merge of X/Open and Open Software Foundation.
- X/Open established DTP Working Group with objective of specifying and fostering appropriate APIs for TP.
- Group concentrated on elements of TP system that provided the ACID properties.
- X/Open DTP standard that emerged specified three interacting components:
    - an application,
    - a transaction manager (TM),
    - a resource manager (RM).
- Any subsystem that implements transactional data can be a RM, such as DBMS, transactional file system or session manager.
- TM responsible for defining scope of transaction, and for assigning unique ID to it.
- Application calls TM to start transaction, calls RMs to manipulate data, and calls TM to terminate transaction.
- TM communicates with RMs to coordinate transaction, and TMs to coordinate distributed transactions.

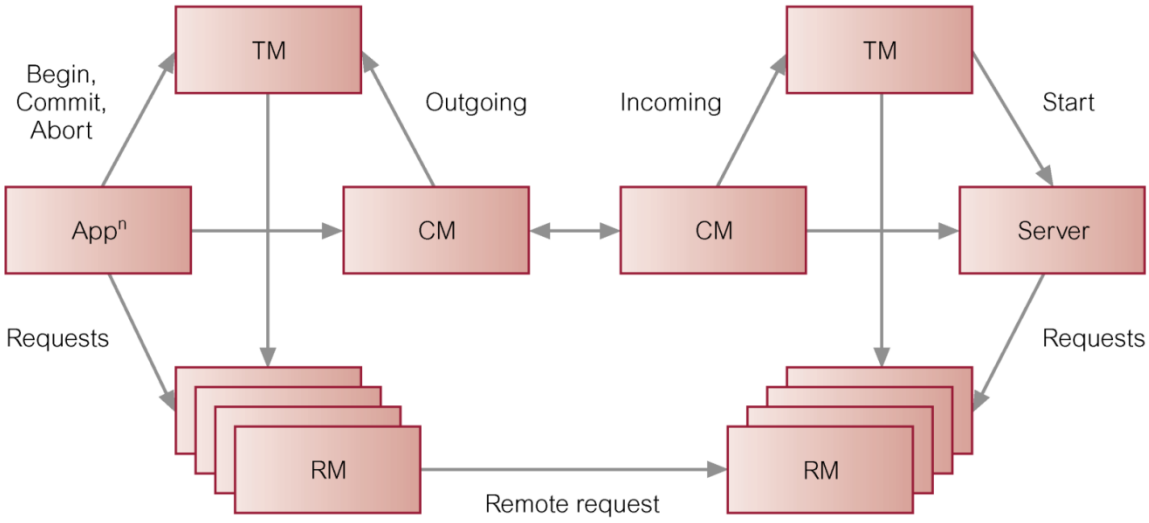
X/OPEN DTP Model – Interfaces

- Application may use TX interface to communicate with a TM.

- TX provides calls that define transaction scope, and whether to commit/abort transaction.
- TM communicates transactional information with RMs through XA interface.
- Finally, application can communicate directly with RMs through a native API, such as SQL or ISAM.



X/OPEN Interfaces in Distributed Environment



Replication Servers

- Currently some prototype and special-purpose DDBMSs, and many of the protocols and problems are well understood.
- However, to date, general purpose DDBMSs have not been widely accepted.
- Instead, database replication, the copying and maintenance of data on multiple servers, may be more preferred solution.
- Every major database vendor has replication solution.

Synchronous versus Asynchronous Replication

Synchronous – updates to replicated data are part of enclosing transaction.

- If one or more sites that hold replicas are unavailable transaction cannot complete.
- Large number of messages required to coordinate synchronization.

Asynchronous - target database updated after source database modified.

Delay in regaining consistency may range from few seconds to several hours or even days.

Functionality

At basic level, has to be able to copy data from one database to another (synch. or asynch.).

Other functions include:

- Scalability.
- Mapping and Transformation.
- Object Replication.
- Specification of Replication Schema.
- Subscription mechanism.
- Initialization mechanism.

Data Ownership

Ownership relates to which site has privilege to update the data.

Main types of ownership are:

- Master/slave (or asymmetric replication),
- Workflow,
- Update-anywhere (or peer-to-peer or symmetric replication).

Master/Slave Ownership

- Asynchronously replicated data is owned by one (master) site, and can be updated by only that site.
- Using 'publish-and-subscribe' metaphor, master site makes data available.
- Other sites 'subscribe' to data owned by master site, receiving read-only copies.
- Potentially, each site can be master site for non-overlapping data sets, but update conflicts cannot occur.

UNIT – IV

Object-orientation is an approach to software construction that has shown considerable promise for solving some of the classic

problems of software development. The underly- ing concept behind object technology is that all software should be constructed out of standard, reusable components wherever possible. Traditionally, software engineering and database management have existed as separate disciplines. Database technology has concentrated on the static aspects of information storage, while software engineering has modeled the dynamic aspects of software. With the arrival of the third generation of Data- base Management Systems, namely **Object-Oriented Database Management Systems** (OODBMSs) and **Object-Relational Database Management Systems** (ORDBMSs), the two disciplines have been combined to allow the concurrent modeling of both data and the processes acting upon the data.

However, there is currently significant dispute regarding this next generation of DBMSs. The success of relational systems in the past two decades is evident, and the

traditionalists believe that it is sufficient to extend the relational model with additional (object-oriented) capabilities. Others believe that an underlying relational model is inadequate to handle complex applications, such as computer-aided design, computer-aided software engineering, and geographic information systems. To help understand these new types of DBMS, and the arguments on both sides, we devote four chapters to discussing the technology and issues behind them.

In Chapter 26 we consider the emergence of OODBMSs and examine some of the issues underlying these systems. In Chapter 27 we examine the object model proposed by the Object Data Management Group (ODMG), which has become a *de facto* standard for OODBMSs, and ObjectStore, a commercial OODBMS. In Chapter 28 we consider the emergence of ORDBMSs and examine some of the issues underlying these systems. In particular, we will examine SQL:2003, the latest release of the ANSI/ ISO standard for SQL, and examine some of the object-oriented features of Oracle. In this chapter we dis- cuss concepts that are common to both OODBMSs and ORDBMSs.

25.1 Advanced Database Applications

The computer industry has seen significant changes in the last decade. In database systems, we have seen the widespread acceptance of RDBMSs for traditional business applica- tions, such as order processing, inventory control, banking, and airline reservations. However, existing RDBMSs have proven inadequate for applications whose needs are quite different from those of traditional business database applications. These applications include:

- computer-aided design (CAD);
- computer-aided manufacturing (CAM);
- computer-aided software engineering (CASE);
- network management systems;
- office information systems (OIS) and multimedia systems;

- digital publishing;
- geographic information systems (GIS);
- interactive and dynamic Web sites.

Computer-aided design (CAD)

A CAD database stores data relating to mechanical and electrical design covering, for example, buildings, aircraft, and integrated circuit chips. Designs of this type have some common characteristics:

- Design data is characterized by a large number of types, each with a small number of instances. Conventional databases are typically the opposite. For example, the *DreamHome* database consists of only a dozen or so relations, although relations such as PropertyForRent, Client, and Viewing may contain thousands of tuples.
- Designs may be very large, perhaps consisting of millions of parts, often with many interdependent subsystem designs.
- The design is not static but evolves through time. When a design change occurs, its implications must be propagated through all design representations. The dynamic nature of design may mean that some actions cannot be foreseen at the beginning.
- Updates are far-reaching because of topological or functional relationships, tolerances, and so on. One change is likely to affect a large number of design objects.
- Often, many design alternatives are being considered for each component, and the correct version for each part must be maintained. This involves some form of version control and configuration management.
- There may be hundreds of staff involved with the design, and they may work in parallel on multiple versions of a large design. Even so, the end-product must be consistent and coordinated. This is sometimes referred to as *cooperative engineering*.

Computer-aided manufacturing (CAM)

A CAM database stores similar data to a CAD system, in addition to data relating to discrete production (such as cars on an assembly line) and continuous

production (such as chemical synthesis). For example, in chemical manufacturing there will be applications that monitor information about the state of the system, such as reactor vessel temperatures, flow rates, and yields. There will also be applications that control various physical pro- cesses, such as opening valves, applying more heat to reactor vessels, and increasing the flow of cooling systems. These applications are often organized in a hierarchy, with a top-level application monitoring the entire factory and lower-level applications mon- itoring individual manufacturing processes. These applications must respond in real time and be capable of adjusting processes to maintain optimum performance within tight tolerances. The applications use a combination of standard algorithms and custom rules to respond to different conditions. Operators may modify these rules occasionally to optimize performance based on complex historical data that the system has to maintain. In this example, the system has to maintain large volumes of data that is hierarchical in nature and maintain complex relationships between the data. It must also be able to rapidly navigate the data to review and respond to changes.

Computer-aided software engineering (CASE)

A CASE database stores data relating to the stages of the software development lifecycle: planning, requirements collection and analysis, design, implementation, testing, maintenance, and documentation. As with CAD, designs may be extremely large, and cooperative engineering is the norm. For example, software configuration management tools allow concurrent sharing of project design, code, and documentation. They also track the dependencies between these components and assist with change management. Project management tools facilitate the coordination of various project management activities, such as the scheduling of potentially highly complex interdependent tasks, cost estimation, and progress monitoring.

Network management systems

Network management systems coordinate the delivery of communication services across a computer network. These systems perform such tasks as network path management, problem management, and network planning. As with the chemical manufacturing example we discussed earlier, these systems also handle complex data and require real-time performance and continuous operation. For example, a telephone call might involve a chain of network switching devices that route a message from sender to receiver, such as:

Node ⇔ Link ⇔ Node ⇔ Link ⇔ Node ⇔ Link ⇔ Node

where each Node represents a port on a network device and each Link represents a slice of bandwidth reserved for that connection. However, a node may participate in several different connections and any database that is created has to manage a complex graph of relationships. To route connections, diagnose problems, and balance loadings, the network management systems have to be capable of moving through this complex graph in real time.

Office information systems (OIS) and multimedia systems

An OIS database stores data relating to the computer control of information in a business, including electronic mail, documents, invoices, and so on. To provide better support for this area, we need to handle a wider range of data types other than names, addresses, dates, and money. Modern systems now handle free-form text, photographs, diagrams, and audio and video sequences. For example, a multimedia document may handle text, photographs, spreadsheets, and voice commentary. The documents may have a specific structure imposed on them, perhaps described using a mark-up language such as SGML (Standardized Generalized Markup Language), HTML (HyperText Markup Language), or XML (eXtended Markup Language), as we discuss in Chapter 30.

Documents may be shared among many users using systems such as electronic mail and bulletin-boards based on Internet technology.[†] Again, such applications need to store data that has a much richer structure than tuples consisting of numbers and text strings. There is also an increasing need to capture handwritten notes using electronic devices. Although

[†] A potentially damaging criticism of database systems, as noted by a number of observers, is that the largest 'database' in the world – the World Wide Web – initially developed with little or no use of database technology. We discuss the integration of the World Wide Web and DBMSs in Chapter 29.

many notes can be transcribed into ASCII text using handwriting analysis techniques, most such data cannot. In addition to words, handwritten data can include sketches, diagrams, and so on.

In the *DreamHome* case study, we may find the following requirements for handling multimedia.

- *Image data* A client may query an image database of properties for rent. Some queries may simply use a textual description to identify images of desirable properties. In other cases it may be useful for the client to query using graphical images of the features that may be found in desirable properties (such as bay windows, internal cornicing, or roof gardens).

- *Video data* A client may query a video database of properties for rent. Again, some queries may simply use a textual description to identify the video images of desirable properties. In other cases it may be useful for the client to query using video features of the desired properties (such as views of the sea or surrounding hills).

- *Audio data* A client may query an audio database that describes the features of properties for rent. Once again, some queries may simply use a textual description to identify the desired property. In other cases it may be useful for the client to use audio features of the desired properties (such as the noise level from nearby traffic).

- *Handwritten data* A member of staff may create notes while carrying out inspections of properties for rent. At a later date, he or she may wish to query such data to find all notes made about a flat in Novar Drive with dry rot.

Digital publishing

The publishing industry is likely to undergo profound changes in business practices over the next decade. It is becoming possible to store books, journals, papers, and articles electronically and deliver them over high-speed networks to consumers. As with office information systems, digital publishing is being extended to handle multimedia documents consisting of text, audio, image, and video data and animation. In some cases, the amount of information available to be put online is enormous, in the order of petabytes (10^{15} bytes), which would make them the largest databases that a DBMS has ever had to manage.

Geographic information systems (GIS)

A GIS database stores various types of spatial and temporal information, such as that used in land management and underwater exploration. Much of the data in these systems is derived from survey and satellite photographs, and tends to be very large. Searches may involve identifying features based, for example, on shape, color, or texture, using advanced pattern-recognition techniques.

For example, EOS (Earth Observing System) is a collection of satellites launched by NASA in the 1990s to gather information that will support scientists concerned with long-term trends regarding the earth's atmosphere, oceans, and land. It is anticipated that these satellites will return over one-third of a petabyte of information per year. This data will be integrated with other data sources and will be stored in EOSDIS (EOS Data and Information System). EOSDIS will supply the information needs of both scientists and

non-scientists. For example, schoolchildren will be able to access EOSDIS to see a simulation of world weather patterns. The immense size of this database and the need to support thousands of users with very heavy volumes of information requests will provide many challenges for DBMSs.

Interactive and dynamic Web sites

Consider a Web site that has an online catalog for selling clothes. The Web site maintains a set of preferences for previous visitors to the site and allows a visitor to:

- browse through thumbnail images of the items in the catalog and select one to obtain a full-size image with supporting details;
- search for items that match a user-defined set of criteria;
- obtain a 3D rendering of any item of clothing based on a customized specification (for example, color, size, fabric);
- modify the rendering to account for movement, illumination, backdrop, occasion, and so on;
- select accessories to go with the outfit, from items presented in a sidebar;
- select a voiceover commentary giving additional details of the item;
- view a running total of the bill, with appropriate discounts;
- conclude the purchase through a secure online transaction.

The requirements for this type of application are not that different from some of the above advanced applications: there is a need to handle multimedia content (text, audio, image, video data, and animation) and to interactively modify the display based on user preferences and user selections. As well as handling complex data, the site also has the added complexity of providing 3D rendering. It is argued that in such a situation the database is not just presenting information to the visitor but is *actively* engaged in selling, dynamically providing customized information and atmosphere to the visitor (King, 1997).

As we discuss in Chapters 29 and 30, the Web now provides a relatively new paradigm for data management, and languages such as XML hold significant promise, particularly for the e-Commerce market. The Forrester Research Group is predicting that business-to- business transactions will reach US$2.1 trillion in Europe and US$7 trillion in the US by 2006. Overall, e-Commerce is expected to account for US$12.8 trillion in worldwide cor- porate revenue by 2006 and potentially represent 18% of sales in the global economy. As the use of the Internet increases and the technology becomes more sophisticated, then we will see Web

sites and business-to-business transactions handle much more complex and interrelated data.

Other advanced database applications include:

- *Scientific and medical applications*, which may store complex data representing systems such as molecular models for synthetic chemical compounds and genetic material.
- *Expert systems*, which may store knowledge and rule bases for artificial intelligence (AI) applications.
- Other applications with complex and interrelated objects and procedural data.

25.2 Weaknesses of RDBMSs

In Chapter 3 we discussed how the relational model has a strong theoretical foundation, based on first-order predicate logic. This theory supported the development of SQL, a declar- ative language that has now become the standard language for defining and manipulating relational databases. Other strengths of the relational model are its simplicity, its suitability for Online Transaction Processing (OLTP), and its support for data independence. How- ever, the relational data model, and relational DBMSs in particular, are not without their disadvantages. Table 25.1 lists some of the more significant weaknesses often cited by the proponents of the object-oriented approach. We discuss these weaknesses in this section and leave readers to judge for themselves the applicability of these weaknesses.

Poor representation of 'real world' entities

The process of normalization generally leads to the creation of relations that do not cor- respond to entities in the 'real world'. The fragmentation of a 'real world' entity into many relations, with a physical representation that reflects this structure, is inefficient leading to many joins during query processing. As we have already seen in Chapter 21, the join is one of the most costly operations to perform.

Semantic overloading

The relational model has only one construct for representing data and relationships between data, namely the *relation*. For example, to represent a many-to-many (*:*) relationship between two entities A and B, we create three relations, one to represent each of the entities A and B, and one to represent the relationship. There is no mechanism to distinguish between entities and relationships, or to distinguish between different kinds of relationship that exist between entities. For example, a 1:* relationship might be *Has*, *Owns*, *Manages*, and so on. If such distinctions could be made, then it might be possible to

Table 25.1 Summary of weaknesses of relational DBMSs.

Weakness
Poor representation of ‘real world’ entities Semantic overloading
Poor support for integrity and enterprise constraints Homogeneous data structure
Limited operations
Difficulty handling recursive queries Impedance mismatch
Other problems with RDBMSs associated with concurrency, schema changes, and poor navigational access

build the semantics into the operations. It is said that the relational model is **semantically overloaded**.

There have been many attempts to overcome this problem using **semantic data models**, that is, models that represent more of the meaning of data. The interested reader is referred to the survey papers by Hull and King (1987) and Peckham and Maryanski (1988). How- ever, the relational model is not completely without semantic features. For example, it has domains and keys (see Section 3.2), and functional, multi-valued, and join dependencies (see Chapters 13 and 14).

Poor support for integrity and general constraints

Integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. In Section 3.3 we introduced the concepts of entity and referential integrity, and in Section 3.2.1 we introduced domains, which are also a type of constraint. Unfortunately, many commercial systems do not fully support these constraints and it is necessary to build them into the applications. This, of course, is dangerous and can lead to duplication of effort and, worse still, inconsistencies. Furthermore, there is no support for general constraints in the relational model, which again means they have to be built into the DBMS or the application.

As we have seen in Chapters 5 and 6, the SQL standard helps partially resolve this claimed deficiency by allowing some types of constraints to be specified as part of the Data Definition Language (DDL).

Homogeneous data structure

The relational model assumes both horizontal and vertical homogeneity. Horizontal homogeneity means that each tuple of a relation must be composed of the same attributes. Vertical homogeneity means that the values in a particular column of a relation must all come from the same domain. Further, the intersection of a row and column must be an atomic value. This fixed structure is too restrictive for many ‘real world’ objects that have a complex structure, and it leads to unnatural joins, which are inefficient as mentioned above. In defense of the relational data model, it could equally be argued that its symmetric structure is one of the model’s strengths.

Among the classic examples of complex data and interrelated relationships is a parts explosion where we wish to represent some object, such as an aircraft, as being composed of parts and composite parts, which in turn are composed of other parts and composite parts, and so on. This weakness has led to research in complex object or non-first normal form (NF^2) database systems, addressed in the papers by, for example, Jaeschke and Schek (1982)

and Bancilhon and Khoshafian (1989). In the latter paper, objects are defined recurs- ively as follows:

(1) Every atomic value (such as integer, float, string) is an object.

(2) If $a_1, a_2, \ldots, a_n$ are distinct attribute names and $o_1, o_2, \ldots, o_n$ are objects, then $[a_1:o_1, a_2:o_2, \ldots, a_n:o_n]$ is a tuple object.

(3) If $o_1, o_2, \ldots, o_n$ are objects, then $S = \{o_1, o_2, \ldots, o_n\}$ is a set object.

In this model, the following would be valid objects:

Atomic objects	B003, John,
Glasgow Set	{SG37, SG14, SG5}
Tuple	[branchNo: B003, street: 163 Main St, city: Glasgow]
Hierarchical tuple	[branchNo: B003, street: 163 Main St, city: Glasgow, staff: {SG37, SG14, SG5}]
Set of tuples	{[branchNo: B003, street: 163 Main St, city: Glasgow], [branchNo: B005, street: 22 Deer Rd, city: London]}
Nested relation	{[branchNo: B003, street: 163 Main St, city: Glasgow, staff: {SG37, SG14, SG5}], [branchNo: B005, street: 22 Deer Rd, city: London, staff: {SL21, SL41}]}

Many RDBMSs now allow the storage of **Binary Large Objects** (BLOBs). A BLOB is a data value that contains binary information representing an image, a digitized video or audio sequence, a procedure, or any large unstructured object. The DBMS does not have any knowledge concerning the content of the BLOB or its internal structure. This prevents the DBMS from performing queries and operations on inherently rich and structured data types. Typically, the database does not manage this information directly, but simply contains a reference to a file. The use of BLOBs is not an elegant solution and storing this information in external files denies it many of the protections naturally afforded by the DBMS. More importantly, BLOBs cannot contain other BLOBs, so they cannot take the form of composite objects. Further, BLOBs generally ignore the behavioral aspects of objects. For example, a picture can be stored as a BLOB in some relational DBMSs. However, the picture can only be stored and displayed. It is not possible to manipulate the internal structure of the picture, nor is it possible to display or manipulate parts of the picture. An example of the use of BLOBs is given in Figure 18.12.

Limited operations

The relational model has only a fixed set of operations, such as set and tuple-oriented operations, operations that are provided in the SQL specification. However, SQL does not allow new operations to be specified. Again, this is too

restrictive to model the behavior of many 'real world' objects. For example, a GIS application typically uses points, lines, line groups, and polygons, and needs operations for distance, intersection, and containment.

Difficulty handling recursive queries

Atomicity of data means that repeating groups are not allowed in the relational model. As a result, it is extremely difficult to handle recursive queries, that is, queries about relationships that a relation has with itself (directly or indirectly). Consider the simplified Staff relation shown in Figure 25.1(a), which stores staff numbers and the corresponding manager's staff number. How do we find all the managers who, directly or indirectly, manage staff member S005? To find the first two levels of the hierarchy, we use:

Figure 25.1

(a) Simplified Staff relation; (b) transitive closure of Staff relation.

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL

(a)

staffNo	managerstaffNo
S005	S004
S004	S003
S003	S002
S002	S001
S001	NULL
S005	S003
S005	S002
S005	S001
S004	S002
S004	S001
S003	S001

(b)

```
SELECT managerStaffNo
FROM Staff
WHERE staffNo = 'S005'
UNION
SELECT managerStaffNo
FROM Staff
WHERE staffNo =
(SELECT managerStaffNo
FROM Staff
WHERE staffNo = 'S005');
```

We can easily extend this approach to find the complete answer to this query. For this particular example, this approach works because we know how many levels in the hierarchy have to be processed. However, if we were to ask a more general query, such as 'For each

member of staff, find all the managers who directly or indirectly manage the individual', this approach would be impossible to implement using interactive SQL. To overcome this problem, SQL can be embedded in a high-level programming lan- guage, which provides constructs to facilitate iteration (see Appendix E). Additionally, many RDBMSs provide a report writer with similar constructs. In either case, it is the application rather than the inherent capabilities of the system that provides the required functionality.

An extension to relational algebra that has been proposed to handle this type of query is the unary **transitive closure**, or **recursive closure**, operation (Merrett, 1984):

Transitive closure	The transitive closure of a relation R with attributes (A_1, A_2) defined on the same domain is the relation R augmented with all tuples successively deduced by transitivity; that is, if (a, b) and (b, c) are tuples of R, the tuple (a, c) is also added to the result.

This operation cannot be performed with just a fixed number of relational algebra operations, but requires a loop along with the Join, Projection, and Union operations. The result of this operation on our simplified Staff relation is shown in Figure 25.1(b).

Impedance mismatch

In Section 5.1 we noted that SQL-92 lacked *computational completeness*. This is true with most Data Manipulation Languages (DMLs) for RDBMSs. To overcome this prob- lem and to provide additional flexibility, the SQL standard provides embedded SQL to help develop more complex database applications (see Appendix E). However, this approach produces an **impedance mismatch** because we are mixing different program- ming paradigms:

- SQL is a declarative language that handles rows of data, whereas a high-level language such as 'C' is a procedural language that can handle only one row of data at a time.

- SQL and 3GLs use different models to represent data. For example, SQL provides the built-in data types Date and Interval, which are not available in traditional programming languages. Thus, it is necessary for the application program to convert between the two representations, which is inefficient both in programming effort and in the use of runtime resources. It has been estimated that as much as 30% of programming effort and code space is expended on this type of conversion (Atkinson *et al.*, 1983). Furthermore, since we are using two different type systems, it is not possible to automatically type check the application as a whole.

It is argued that the solution to these problems is not to replace relational languages by row-level object-oriented languages, but to introduce set-level facilities into programming languages (Date, 2000). However, the basis of OODBMSs is to provide a much more seamless integration between the DBMS's data model and the host programming language. We return to this issue in the next chapter.

Other problems with RDBMSs

- Transactions in business processing are generally short-lived and the concurrency control primitives and protocols such as two-phase locking are not particularly suited for long-duration transactions, which are more common for

complex design objects (see Section 20.4).

- Schema changes are difficult. Database administrators must intervene to change data- base structures and, typically, programs that access these structures must be modified to adjust to the new structures. These are slow and cumbersome processes even with current technologies. As a result, most organizations are locked into their existing database structures. Even if they are willing and able to change the way they do busi- ness to meet new requirements, they are unable to make these changes because they cannot afford the time and expense required to modify their information systems (Taylor, 1992). To meet the requirement for increased flexibility, we need a system that caters for natural schema evolution.

- RDBMSs were designed to use content-based *associative access* (that is, declarative statements with selection based on one or more predicates) and are poor at *navigational access* (that is, access based on movement between individual records). Navigational access is important for many of the complex applications we discussed in the previous section.

Of these three problems, the first two are applicable to many DBMSs, not just relational systems. In fact, there is no underlying problem with the relational model that would prevent such mechanisms being implemented.

The latest release of the SQL standard, SQL:2003, addresses some of the above deficiencies with the introduction of many new features, such as the ability to define new data types and operations as part of the data definition language, and the addition of new constructs to make the language computationally complete. We discuss SQL:2003 in detail in Section 28.4.

25.3 Object-Oriented Concepts

In this section we discuss the main concepts that occur in object-orientation. We start with a brief review of the underlying themes of abstraction, encapsulation, and information hiding.

25.3.1 Abstraction, Encapsulation, and Information Hiding

Abstraction is the process of identifying the essential aspects of an entity and ignoring the unimportant properties. In software engineering this means that we concentrate on what an object is and what it does before we decide how it should be implemented. In this way we delay implementation details for as long as possible, thereby avoiding commitments that we may find restrictive at a later stage. There are two fundamental aspects of abstraction: encapsulation and information hiding.

The concept of **encapsulation** means that an object contains both the data structure and the set of operations that can be used to manipulate it. The concept of **information hiding** means that we separate the external aspects of an object from its internal details, which are hidden from the outside world. In this way the internal details of an object can be changed without affecting the applications that use it, provided the external details remain the same. This prevents an application becoming so interdependent that a small change has enormous ripple effects. In other words information hiding provides a form of *data independence*.

These concepts simplify the construction and maintenance of applications through **modularization**. An object is a 'black box' that can be constructed and modified independently of the rest of the system, provided the external interface is not changed. In some systems, for example Smalltalk, the ideas of encapsulation and information hiding are brought together. In Smalltalk the object structure is always hidden and only the operation interface can ever be visible. In this way the object structure can be changed without affecting any applications that use the object.

There are two views of encapsulation: the object-oriented programming language (OOPL) view and the database adaptation of that view. In some OOPLs encapsulation is achieved through **Abstract Data Types** (ADTs). In this view an object has an interface part and an implementation part. The interface provides a specification of the operations that can be performed on the object; the implementation part consists of the data structure for the ADT and the functions that realize the interface. Only the interface part is visible to other objects or users. In the database view, proper encapsulation is achieved by ensuring that programmers have access only to the interface part. In this way encapsulation provides a form of *logical data independence*: we can change the internal implementation of an ADT without changing any of the applications using that ADT (Atkinson *et al*., 1989).

Objects and Attributes 25.3.2

Many of the important object-oriented concepts stem from the Simula programming language developed in Norway in the mid-1960s to support simulation of 'real world' processes (Dahl and Nygaard, 1966), although object-oriented programming did not emerge as a new programming paradigm until the development of the Smalltalk language (Goldberg and Robson, 1983). Modules in Simula are not based on procedures as they are in conventional programming languages, but on the physical objects being modeled in the simulation. This seemed a sensible approach as the objects are the key to the simulation: each object has to maintain some information about its current **state**, and additionally has actions (**behavior**) that have to be modeled. From Simula, we have the definition of an object.

Object A uniquely identifiable entity that contains both the attributes that describe the state of a 'real world' object and the actions that are associated with it.

In the *DreamHome* case study, a branch office, a member of staff, and a property are examples of objects that we wish to model. The concept of an object is simple but, at the same time, very powerful: each object can be defined and maintained independently of the others. This definition of an object is very similar to the definition of an entity given in Section 11.1.1. However, an object encapsulates both state and behavior; an entity models only state.

The current state of an object is described by one or more **attributes** (**instance**

variables). For example, the branch office at 163 Main St may have the attributes shown in Table 25.2. Attributes can be classified as simple or complex. A **simple attribute** can be a primitive type such as integer, string, real, and so on, which takes on literal values; for example, branchNo in Table 25.2 is a simple attribute with the literal value 'B003'. A **complex attribute** can contain collections and/or references. For example, the attribute SalesStaff is a **collection** of Staff objects. A **reference attribute** represents a relationship between objects and contains a value, or collection of values, which are themselves objects (for example, SalesStaff is, more precisely, a collection of references to Staff objects). A reference attribute is conceptually similar to a foreign key in the relational data model or a pointer in a programming language. An object that contains one or more complex attributes is called a **complex object** (see Section 25.3.9).

Table 25.2 Object attributes for branch instance.

Attribute	Value
branchNo	B003
street	163 Main St
city	Glasgow
postcode	G11 9QX
SalesStaff	Ann Beech; David Ford
Manager	Susan Brand

Attributes are generally referenced using the 'dot' notation. For example, the street attribute of a branch object is referenced as:

branchObject.street

Object Identity

A key part of the definition of an object is unique identity. In an object-oriented system, each object is assigned an **Object Identifier** (OID) when it is created that is:

- system-generated;
- unique to that object;
- invariant, in the sense that it cannot be altered during its lifetime. Once the object is created, this OID will not be reused for any other object, even after the object has been deleted;
- independent of the values of its attributes (that is, its state). Two objects could have the same state but would have different identities;
- invisible to the user (ideally).

Thus, object identity ensures that an object can always be uniquely identified, thereby automatically providing entity integrity (see Section 3.3.2). In fact, as object identity

ensures uniqueness system-wide, it provides a stronger constraint than the relational data model's entity integrity, which requires only uniqueness within a relation. In addition, objects can contain, or refer to, other objects using object identity. However, for each referenced OID in the system there should always be an object present that corresponds to the OID, that is, there should be no **dangling references**. For example, in the *DreamHome* case study, we have the relationship Branch *Has* Staff. If we embed each branch object in the related staff object, then we encounter the problems of information redundancy and update anomalies discussed in Section 13.2. However, if we instead embed the OID of the branch object in the related staff object, then there continues to be only one instance of each branch object in the system and consistency can be maintained more easily. In this way, objects can be *shared* and OIDs can be used to maintain referential integrity (see Section 3.3.3). We discuss referential integrity in OODBMSs in Section 25.6.2.

There are several ways in which object identity can be implemented. In an RDBMS, object identity is *value-based*: the primary key is used to provide uniqueness of each tuple in a relation. Primary keys do not provide the type of object identity that is required in object-oriented systems. First, as already noted, the primary key is only unique within a relation, not across the entire system. Second, the primary key is generally chosen from the attributes of the relation, making it dependent on object state. If a potential key is subject to change, identity has to be simulated by unique identifiers, such as the branch number branchNo, but as these are not under system control there is no guarantee of protection against violations of identity. Furthermore, simulated keys such as B001, B002, B003, have little semantic meaning to the user.

Other techniques that are frequently used in programming languages to support identity are variable names and pointers (or virtual memory addresses), but these approaches also compromise object identity (Khoshafian and Abnous, 1990). For example, in 'C' and C++ an OID is a physical address in the process memory space. For most database purposes this address space is too small: scalability requires that OIDs be valid across storage volumes, possibly across different computers for distributed DBMSs. Further, when an object is deleted, the memory formerly occupied by it should be reused, and so a new object may be created and allocated to the same space as the deleted object occupied. All references to the old object, which became invalid after the deletion, now become valid again, but unfortunately referencing the wrong object. In a similar way moving an object from one address to another invalidates the object's identity. What is required is a *logical object identifier* that is independent of both state and location. We discuss logical and physical OIDs in Section 26.2.

There are several advantages to using OIDs as the mechanism for object identity:

- *They are efficient* OIDs require minimal storage within a complex object. Typically, they are smaller than textual names, foreign keys, or other semantic-based references.

- *They are fast* OIDs point to an actual address or to a location within a table that gives the address of the referenced object. This means that objects can be located quickly whether they are currently stored in local memory or on disk.

- *They cannot be modified by the user* If the OIDs are system-generated and kept invisible, or at least read-only, the system can ensure entity and referential integrity more easily. Further, this avoids the user having to maintain integrity.

- *They are independent of content* OIDs do not depend upon the data contained in the object in any way. This allows the value of every attribute of an object to change, but for the object to remain the same object with the same OID.

Note the potential for ambiguity that can arise from this last property: two objects can appear to be the same to the user (all attribute values are the same), yet have different OIDs and so be different objects. If the OIDs are invisible, how does the user distinguish between these two objects? From this we may conclude that primary keys are still required to allow users to distinguish objects. With this approach to designating an object, we can distinguish between object identity (sometimes called object equivalence) and object equality. Two objects are **identical** (equivalent) if and only if they are the same object (denoted by '='), that is their OIDs are the same. Two objects are **equal** if their states are the same (denoted by '= ='). We can also distinguish between shallow and deep equality:

objects have **shallow equality** if their states contain the same values when we exclude references to other objects; objects have **deep equality** if their states contain the same values and if related objects also contain the same values.

Methods and Messages

An object encapsulates both data and functions into a self-contained package. In object technology, functions are usually called **methods**. Figure 25.2 provides a conceptual representation of an object, with the attributes on the inside protected from the outside by the methods. Methods define the **behavior** of the object. They can be used to change the object's state by modifying its attribute values, or to query the values of selected attributes. For example, we may have methods to add a new property for rent at a branch, to update a member of staff 's salary, or to print out a member of staff 's details.

A method consists of a name and a body that performs the behavior associated with the method name. In an object-oriented language, the body consists of a block of code that carries out the required functionality. For example, Figure 25.3 represents the method to update a member of staff 's salary. The name of the method is updateSalary, with an input parameter *increment*, which is added to the **instance variable** salary to produce a new salary.

Messages are the means by which objects communicate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object. Again, the dot notation is generally used to access a method. For example, to execute the updateSalary

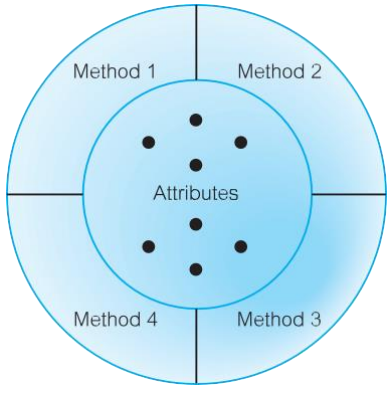


Figure 25.2
Object showing attributes and methods.

```
method void updateSalary(float increment)
{
        salary = salary + increment;
}
```

Figure 25.3 Example of a method.

method on a Staff object, staffObject, and pass the method an increment value of 1000, we write:

```
staffObject.updateSalary(1000)
```

In a traditional programming language, a message would be written as a function call:

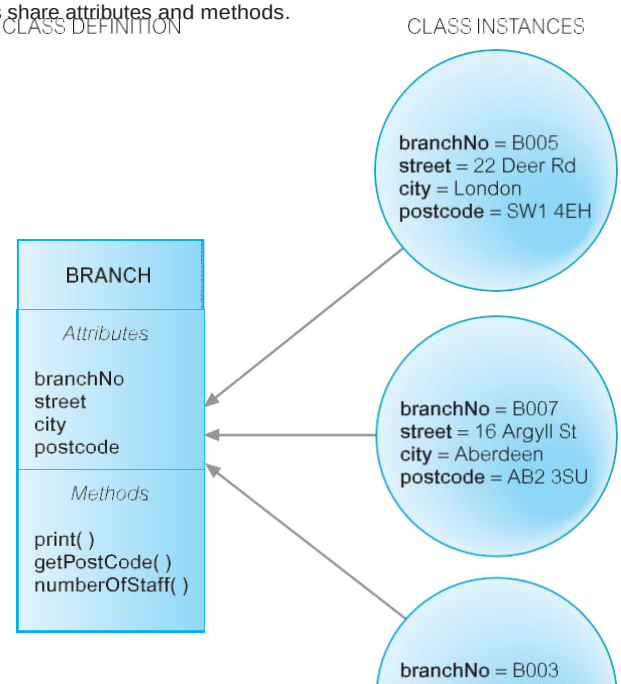
```
updateSalary(staffObject, 1000)
```

Classes

In Simula, classes are blueprints for defining a set of similar objects. Thus, objects that have the same attributes and respond to the same messages can be grouped together to form a **class**. The attributes and associated methods are defined once for the class rather than separately for each object. For example, all branch objects would be described by a single Branch class. The objects in a class are called **instances** of the class. Each instance has its own value(s) for each attribute, but shares the same attribute names and methods with other instances of the class, as illustrated in Figure 25.4.

In the literature, the terms 'class' and 'type' are often used synonymously, although some authors make a distinction between the two terms as we now describe. A *type*

Figure 25.4 Class instances share attributes and methods.



corresponds to the notion of an abstract data type (Atkinson and Buneman, 1989). In programming languages, a variable is declared to be of a particular type. The compiler can use this type to check that the operations performed on the variable are compatible with its type, thus helping to ensure the correctness of the software. On the other hand, a *class* is a blueprint for creating objects and provides methods that can be applied on the objects. Thus, a class is referred to at runtime rather than compile time.

In some object-oriented systems, a class is also an object and has its own attributes and methods, referred to as **class attributes** and **class methods**, respectively. Class attributes describe the general characteristics of the class, such as totals or averages; for example, in the class Branch we may have a class attribute for the total number of branches. Class methods are used to change or query the state of class attributes. There are also special class methods to create new instances of the class and to destroy those that are no longer required. In an object-oriented language, a new instance is normally created by a method called new. Such methods are usually called **constructors**. Methods for destroying objects and reclaiming the space occupied are typically called **destructors**. Messages sent to a class method are sent to the class rather than an instance of a class, which implies that the class is an instance of a higher-level class, called a **metaclass**.

Subclasses, Superclasses, and Inheritance

Some objects may have similar but not identical attributes and methods. If there is a large degree of similarity, it would be useful to be able to share the common properties (attributes and methods). **Inheritance** allows one class to be defined as a special case of a more general class. These special cases are known as **subclasses** and the more general cases are known as **superclasses**. The process of forming a superclass is referred to as **generalization** and the process of forming a subclass is **specialization**. By default, a subclass inherits all the properties of its superclass(es) and, additionally, defines its own unique properties. However, as we see shortly, a subclass can also redefine inherited properties. All instances of the subclass are also instances of the superclass. Further, the *principle of substitutability* states that we can use an instance of the subclass whenever a method or a construct expects an instance of the superclass.

The concepts of superclass, subclass, and inheritance are similar to those discussed for the Enhanced Entity–Relationship (EER) model in Chapter 12, except that in the object-oriented paradigm inheritance covers both state and behavior. The relationship between the subclass and superclass is sometimes referred to as **A KIND OF** (AKO) relationship, for example a Manager is AKO Staff. The relationship between an instance and its class is sometimes referred to as **IS-A**; for example, Susan Brand IS-A Manager.

There are several forms of inheritance: single inheritance, multiple inheritance, repeated

inheritance, and selective inheritance. Figure 25.5 shows an example of **single inheritance**, where the subclasses Manager and SalesStaff inherit the properties of the superclass Staff. The term 'single inheritance' refers to the fact that the subclasses inherit from no more than one superclass. The superclass Staff could itself be a subclass of a superclass, Person, thus forming a **class hierarchy**.

Figure 25.6 shows an example of **multiple inheritance** where the subclass SalesManager inherits properties from both the superclasses Manager and SalesStaff. The provision of a

Figure 25.5

Single inheritance.

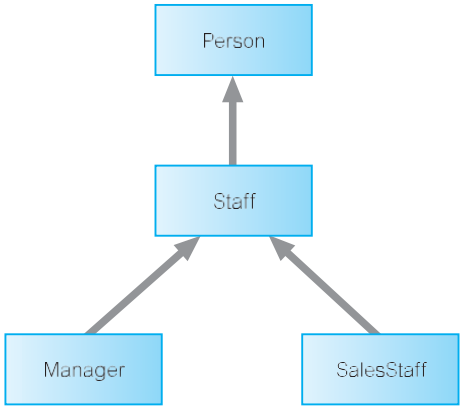
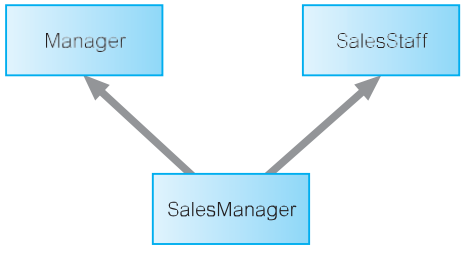


Figure 25.6

Multiple inheritance.



mechanism for multiple inheritance can be quite problematic as it has to provide a way of dealing with conflicts that arise when the superclasses contain the same attributes or methods. Not all object-oriented languages and DBMSs support multiple inheritance as a matter of principle. Some authors claim that multiple inheritance introduces a level of complexity that is hard to manage safely and consistently. Others argue that it is required to model the 'real world', as in this example. Those languages that do support it, handle conflict in a variety of ways, such as:

- Include both attribute/method names and use the name of the superclass as a qualifier. For example, if bonus is an attribute of both Manager and SalesStaff, the subclass SalesManager could inherit bonus from both superclasses and qualify the instance of bonus in SalesManager as either Manager.bonus or SalesStaff.bonus.

- Linearize the inheritance hierarchy and use single inheritance to avoid conflicts. With this approach, the inheritance hierarchy of Figure 25.6 would be interpreted as:

SalesManager → Manager → SalesStaff

or

SalesManager → SalesStaff → Manager

With the previous example, SalesManager would inherit one instance of the attribute bonus, which would be from Manager in the first case, and SalesStaff in the second case.

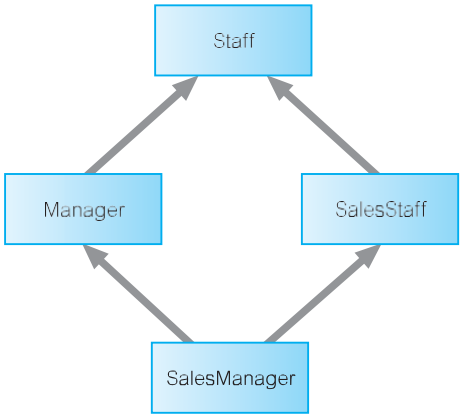


Figure 25.7
Repeated inheritance.

- Require the user to redefine the conflicting attributes or methods.
- Raise an error and prohibit the definition until the conflict is resolved.

Repeated inheritance is a special case of multiple inheritance where the superclasses inherit from a common superclass. Extending the previous example, the classes Manager and SalesStaff may both inherit properties from a common superclass Staff, as illustrated in Figure 25.7. In this case, the inheritance mechanism must ensure that the SalesManager class does not inherit properties from the Staff class twice. Conflicts can be handled as dis- cussed for multiple inheritance.

Selective inheritance allows a subclass to inherit a limited number of properties from the superclass. This feature may provide similar functionality to the view mechanism discussed in Section 6.4 by restricting access to some details but not others.

Overriding and Overloading

As we have just mentioned, properties (attributes and methods) are automatically inherited by

subclasses from their superclasses. However, it is possible to redefine a property in the subclass. In this case, the definition of the property in the subclass is the one used. This process is called **overriding**. For example, we might define a method in the Staff class to increment salary based on a commission:

```
method void giveCommission(float branchProfit) {
salary = salary + 0.02 * branchProfit;
}
```

However, we may wish to perform a different calculation for commission in the Manager subclass. We can do this by redefining, or *overriding*, the method giveCommission in the Manager subclass:

```
method void giveCommission(float branchProfit) {
salary = salary + 0.05 * branchProfit;
}
```

Figure 25.8 Overloading print method: (a) for Branch object; (b) for Staff object.

```
method void print( )  {
    printf("Branch number: %s\n", branchNo);
    printf("Street: %s\n", street);
    printf("City: %s\n", city);
    printf("Postcode: %s\n", postcode);
}
```

(a)

```
method void print( )  {
    printf("Staff number: %s\n", staffNo);
    printf("First name: %s\n", fName);
    printf("Last name: %s\n", lName);
    printf("Position: %s\n", position);
    printf("Sex: %c\n", sex);
    printf("Date of birth: %s\n", DOB);
    printf("Salary: %f\n", salary);
}
```

(b)

The ability to factor out common properties of several classes and form them into a superclass that can be shared with subclasses can greatly reduce redundancy within sys- tems and is regarded as one of the main advantages of object-orientation. Overriding is an important feature of inheritance as it allows special cases to be handled easily with minimal impact on the rest of the system.

Overriding is a special case of the more general concept of **overloading**. Overloading allows the name of a method to be reused within a class definition or across class definitions. This means that a single message can perform different functions depending on which object receives it and, if appropriate, what parameters are passed to the method. For example, many classes will have a print method to print out the relevant details for an object, as shown in Figure 25.8.

Overloading can greatly simplify applications, since it allows the same name to be used for the same operation irrespective of what class it appears in, thereby allowing context to determine which meaning is appropriate at any given moment. This saves having to provide unique names for methods such as printBranchDetails or printStaffDetails for what is in essence the same functional operation.

Polymorphism and Dynamic Binding

Overloading is a special case of the more general concept of **polymorphism**, from the Greek meaning 'having many forms'. There are three types of polymorphism: operation, inclusion, and parametric (Cardelli and Wegner, 1985). Overloading,

as in the previous example, is a type of **operation** (or ***ad hoc***) **polymorphism**. A method defined in a superclass and inherited in its subclasses is an example of **inclusion polymorphism**. **Parametric polymorphism**, or **genericity** as it is sometimes called, uses types as param- eters in generic type, or class, declarations. For example, the following template definition:

```
template <type T> T max(x:T,
y:T) {
if (x > y)          return x;
else                return y;
}
```

defines a generic function max that takes two parameters of type T and returns the maximum of the two values. This piece of code does not actually establish any methods. Rather, the generic description acts as a template for the later establishment of one or more different methods of different types. Actual methods are instantiated as:

```
int max(int, int);                          // instantiate max function for two
integer types real max(real, real);         // instantiate max function for two real
types
```

The process of selecting the appropriate method based on an object's type is called **binding**. If the determination of an object's type can be deferred until runtime (rather than compile time), the selection is called **dynamic (late) binding**. For example, consider the class hierarchy of Staff with subclasses Manager and SalesStaff shown in Figure 25.5, and assume that each class has its own print method to print out relevant details. Further assume that we have a list consisting of an arbitrary number of objects, *n* say, from this hierarchy. In a conventional programming language, we would need a CASE statement or a nested IF statement to print out the corresponding details:

```
FOR i = 1 TO n DO

SWITCH (list[i]. type) {

CASE staff:                 printStaffDetails(list[i].object); break;
CASE manager:               printManagerDetails(list[i].object); break;
CASE salesPerson:           printSalesStaffDetails(list[i].object); break;

}
```

If a new type is added to the list, we have to extend the CASE statement to handle the new type, forcing recompilation of this piece of software. If the language supports dynamic binding and overloading, we can overload the print methods with the single name print and replace the CASE statement with the line:

```
list[i].print()
```

Furthermore, with this approach we can add any number of new types to the list and, provided we continue to overload the print method, no recompilation of this code is required. Thus, the concept of polymorphism is orthogonal to (that is, independent of) inheritance.

Complex Objects

There are many situations where an object consists of subobjects or components. A com-

plex object is an item that is viewed as a single object in the 'real world' but combines with other objects in a set of complex **A-PART-OF** relationships (APO). The objects contained may themselves be complex objects, resulting in an **A-PART-OF hierarchy**. In an object-oriented system, a contained object can be handled in one of two ways. First, it can be encapsulated within the complex object and thus form part of the complex object. In this case, the structure of the contained object is part of the structure of the complex object and can be accessed only by the complex object's methods. On the other hand, a contained object can be considered to have an independent existence from the complex object. In this

case, the object is not stored directly in the parent object but only its OID. This is known as **referential sharing** (Khoshafian and Valduriez, 1987). The contained object has its own structure and methods, and can be owned by several parent objects.

These types of complex object are sometimes referred to as **structured complex** objects, since the system knows the composition. The term **unstructured complex object** is used to refer to a complex object whose structure can be interpreted only by the application program. In the database context, unstructured complex objects are sometimes known as Binary Large Objects (BLOBs), which we discussed in Section 25.2.

25.4 Storing Objects in a Relational Database

One approach to achieving persistence with an object-oriented programming language, such as C++ or Java, is to use an RDBMS as the underlying storage engine. This requires mapping class instances (that is, objects) to one or more tuples distributed over one or more relations. This can be problematic as we discuss in this section. For the purposes of discussion, consider the inheritance hierarchy shown in Figure 25.9, which has a Staff superclass and three subclasses: Manager, SalesPersonnel, and Secretary.

To handle this type of class hierarchy, we have two basics tasks to perform:

- Design the relations to represent the class hierarchy.
- Design how objects will be accessed, which means:
  - writing code to decompose the objects into tuples and store the decomposed objects in relations;
  - writing code to read tuples from the relations and reconstruct the objects.

We now describe these two tasks in more detail.

Figure 25.9 Sample inheritance hierarchy for Staff.

Mapping Classes to Relations

There are a number of strategies for mapping classes to relations, although each results in a loss of semantic information. The code to make objects persistent and to read the objects back from the database is dependent on the strategy chosen. We consider three alternatives:

(1) Map each class or subclass to a relation.
(2) Map each subclass to a relation.
(3) Map the hierarchy to a single relation.

Map each class or subclass to a relation

One approach is to map each class or subclass to a relation. For the hierarchy given in Figure 25.9, this would give the following four relations (with the primary key underlined):

Staff (staffNo, fName, lName, position, sex, DOB, salary) Manager (staffNo, bonus, mgrStartDate)

SalesPersonnel (staffNo, salesArea, carAllowance) Secretary (staffNo, typingSpeed)

We assume that the underlying data type of each attribute is supported by the RDBMS, although this may not be the case – in which case we would need to write additional code to handle the transformation of one data type to another.

Unfortunately, with this relational schema we have lost semantic information: it is no longer clear which relation represents the superclass and which relations represent the subclasses. We would therefore have to build this knowledge into each application, which as we have said on other occasions can lead to duplication of code and potential for inconsistencies to arise.

Map each subclass to a relation

A second approach is to map each subclass to a relation. For the hierarchy given in Figure 25.9, this would give the following three relations:

Manager (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate) SalesPersonnel (staffNo, fName, lName, position, sex, DOB, salary, salesArea, carAllowance) Secretary (staffNo, fName, lName, position, sex, DOB, salary, typingSpeed)

Again, we have lost semantic information in this mapping: it is no longer clear that these relations are subclasses of a single generic class. In this case, to produce a list of all staff we would have to select the tuples from each relation and then union the results together.

Map the hierarchy to a single relation

A third approach is to map the entire inheritance hierarchy to a single relation, giving in this case:

Staff (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate, salesArea, carAllowance, typingSpeed, typeFlag)

The attribute typeFlag is a discriminator to distinguish which type each tuple is (for example, it may contain the value 1 for a Manager tuple, 2 for a SalesPersonnel tuple, and 3 for a Secretary tuple). Again, we have lost semantic information in this mapping. Further, this mapping will produce an unwanted number of nulls for attributes that do not apply to that tuple. For example, for a Manager tuple, the attributes salesArea, carAllowance, and typingSpeed will be null.

Accessing Objects in the Relational Database

Having designed the structure of the relational database, we now need to insert objects into the database and then provide a mechanism to read, update, and delete the objects. For example, to insert an object into the first relational schema in the previous section (that is, where we have created a relation for each class), the code may look something like the following using programmatic SQL (see Appendix E):

```
Manager* pManager = new Manager;                    // create a new Manager object
. . . code to set up the object . . .
EXEC SQL INSERT INTO Staff VALUES (:pManager->staffNo, :pManager->fName,
:pManager->lName, :pManager->position, :pManager->sex, :pManager->DOB,
:pManager->salary);
EXEC SQL INSERT INTO Manager VALUES (:pManager->bonus,
:pManager->mgrStartDate);
```

On the other hand, if Manager had been declared as a persistent class then the following (indicative) statement would make the object persistent in an OODBMS:

```
Manager* pManager = new Manager;
```

In Section 26.3, we examine different approaches for declaring persistent classes. If we now wished to retrieve some data from the relational database, say the details for managers with a bonus in excess of £1000, the code may look something like the following:

```
Manager* pManager =
new Manager;
CURSOR FOR
SELECT staffNo, fName,
lName, salary, bonus
FROM Staff s, Manager m
WHERE s.staffNo =
m.staffNo AND bonus >
1000;
EXEC SQL OPEN managerCurso
```

```
r; for ( ; ; ) {
EXEC SQL FETCH managerCursor                    // fetch the next record in the result
INTO :staffNo, :fName, :lName, :salary, :bonus;
pManager->staffNo = :staffNo;                   // transfer the data to the Manager
object
```

```
pManager->fName = :fName; pManager->lName = :lName;
pManager->salary = :salary; pManager->bonus = :bonus;

strcpy(pManager->position, "Manager");

}

EXEC SQL CLOSE managerCursor;                    // close the cursor before completing
```

On the other hand, to retrieve the same set of data in an OODBMS, we may write the following code:

```
os_Set<Manager*> &highBonus
                = managerExtent->query("Manager*", "bonus > 1000", db1);
```

This statement queries the extent of the Manager class (managerExtent) to find the required instances (bonus > 1000) from the database (in this example, db1). The commercial OODBMS ObjectStore has a collection template class os_Set, which has been instantiated in this example to contain pointers to Manager objects <Manager*>. In Section 27.3 we provide additional details of object persistence and object retrieval with ObjectStore.

The above examples have been given to illustrate the complexities involved in mapping an object-oriented language to a relational database. The OODBMS approach that we discuss in the next two chapters attempts to provide a more seamless integration of the programming language data model and the database data model thereby removing the need for complex transformations, which, as we discussed earlier, could account for as much as 30% of programming effort.

25.5 Next-Generation Database Systems

In the late 1960s and early 1970s, there were two mainstream approaches to constructing DBMSs. The first approach was based on the hierarchical data model, typified by IMS (Information Management System) from IBM, in response to the enormous information storage requirements generated by the Apollo space program. The second approach was based on the network data model, which attempted to create a database standard and resolve some of the difficulties of the hierarchical model, such as its inability to represent complex relationships effectively. Together, these approaches represented the **first genera- tion** of DBMSs. However, these two models had some fundamental disadvantages:

- complex programs had to be written to answer even simple queries based on naviga- tional record-oriented access;

- there was minimal data independence;
- there was no widely accepted theoretical foundation.

In 1970, Codd produced his seminal paper on the relational data model. This paper was very timely and addressed the disadvantages of the former approaches, in particular their lack of data independence. Many experimental relational DBMSs were implemented thereafter, with the first commercial products appearing in the late 1970s and early 1980s.

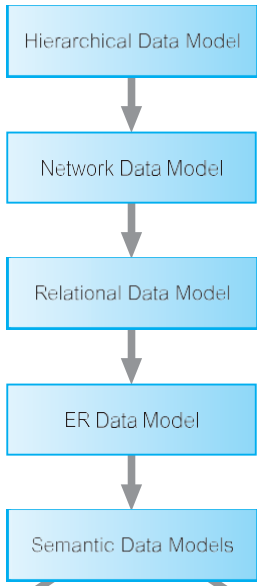
Now there are over a hundred relational DBMSs for both mainframe and PC environments, though some are stretching the definition of the relational model. Relational DBMSs are referred to as **second-generation** DBMSs.

However, as we discussed in Section 25.2, RDBMSs have their failings, particularly their limited modeling capabilities. There has been much research attempting to address this problem. In 1976, Chen presented the Entity–Relationship model that is now a widely accepted technique for database design, and the basis for the methodology presented in Chapters 15 and 16 of this book (Chen, 1976). In 1979, Codd himself attempted to address some of the failings in his original work with an extended version of the relational model called RM/T (Codd, 1979), and thereafter RM/ V2 (Codd, 1990). The attempts to provide a data model that represents the 'real world' more closely have been loosely classified as **semantic data modeling**. Some of the more famous models are:

- the Semantic Data Model (Hammer and McLeod, 1981);
- the Functional Data Model (Shipman, 1981), which we examine in Section 26.1.2;
- the Semantic Association Model (Su, 1983).

In response to the increasing complexity of database applications, two 'new' data models have emerged: the **Object-Oriented Data Model** (OODM) and the **Object-Relational Data Model** (ORDM), previously referred to as the **Extended Relational Data Model** (ERDM). However, unlike previous models, the actual composition of these models is not clear. This evolution represents **third-generation** DBMSs, as illustrated in Figure 25.10.

Figure 25.10 History of data models.



There is currently considerable debate between the OODBMS proponents and the relational supporters, which resembles the network/relational debate of the 1970s. Both sides agree that traditional RDBMSs are inadequate for certain types of application. However, the two sides differ on the best solution. The OODBMS proponents claim that RDBMSs are satisfactory for standard business applications but lack the capability to support more complex applications. The relational supporters claim that relational technology is a necessary part of any real DBMS and that complex applications can be handled by extensions to the relational model.

At present, relational/object-relational DBMSs form the dominant system and object-oriented DBMSs have their own particular niche in the marketplace. If OODBMSs are to become dominant they must change their image from being systems solely for complex applications to being systems that can also accommodate standard business applications with the same tools and the same ease of use as their relational counterparts. In particular, they must support a declarative query language compatible with SQL. We devote Chap- ters 26 and 27 to a discussion of OODBMSs and Chapter 28 to ORDBMSs.

25.6 Oriented Database Design

In this section we discuss how to adapt the methodology presented in Chapters 15 and 16 for an OODBMS. We start the discussion with a comparison of the basis for our methodology, the Enhanced Entity–Relationship model, and the main object-oriented concepts. In Section 25.6.2 we examine the relationships that can exist between objects and how referential integrity can be handled. We conclude this section with some guidelines for identifying methods.

Comparison of Object-Oriented Data Modeling and Conceptual Data Modeling

The methodology for conceptual and logical database design presented in Chapters 15 and 16, which was based on the Enhanced Entity–Relationship (EER) model, has similarities with Object-Oriented Data Modeling (OODM). Table 25.3 compares OODM with Con- ceptual Data Modeling (CDM). The main difference is the encapsulation of both state and behavior in an object, whereas CDM captures only state and has no knowledge of behavior. Thus,

CDM has no concept of messages and consequently no provision for encapsulation.

The similarity between the two approaches makes the conceptual and logical data modeling methodology presented in Chapters 15 and 16 a reasonable basis for a method- ology for object-oriented database design. Although this methodology is aimed primarily at relational database design, the model can be mapped with relative simplicity to the network and hierarchical models. The logical data model produced had many-to-many relationships and recursive relationships removed (Step 2.1). These are unnecessary changes for object-oriented modeling and can be omitted, as they were introduced because

Table 25.3 Comparison of OODM and CDM.

OODM	CDM	Difference
Object	Entity	Object includes behavior
Attribute	Attribute	None
Association	Relationship	Associations are the same but inheritance in OODM includes both state and behavior
Message		No corresponding concept in CDM
Class	Entity type/Supertype	None
Instance	Entity	None
Encapsulation		No corresponding concept in CDM

of the limited modeling power of the traditional data models. The use of normalization in the methodology is still important and should not be omitted for object-oriented database design. Normalization is used to improve the model so that it satisfies various constraints that avoid unnecessary duplication of data. The fact that we are dealing with objects does not mean that redundancy is acceptable. In object-oriented terms, second and third normal form should be interpreted as:

'Every attribute in an object is dependent on the object identity.'

Object-oriented database design requires the database schema to include both a descrip- tion of the object data structure and constraints, and the object behavior. We discuss behavior modeling in Section 25.6.3.

Relationships and Referential Integrity

Relationships are represented in an object-oriented data model using **reference attributes** (see Section 25.3.2), typically implemented using OIDs. In the methodology presented in Chapters 15 and 16, we decomposed all non-binary relationships (for example, ternary relationships) into binary relationships. In this section we discuss how to represent binary relationships based on their cardinality: one-to-one (1:1), one-to-many (1:*), and many-to- many (*:*).

1:1 relationships

A 1:1 relationship between objects A and B is represented by adding a reference attribute to object A

and, to maintain referential integrity, a reference attribute to object B. For example, there is a 1:1 relationship between Manager and Branch, as represented in Figure 25.11.

25.6.2

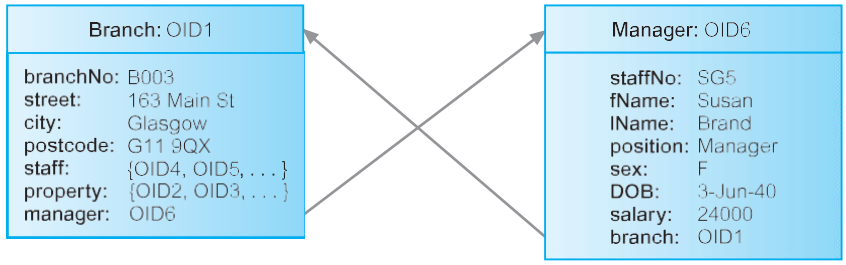


Figure 25.11
A 1:1 relationship between Manager and Branch.

1:* relationships

A 1:* relationship between objects A and B is represented by adding a reference attribute to object B and an attribute containing a set of references to object A. For example, there are 1:* relationships represented in Figure 25.12, one between Branch and SalesStaff, and the other between SalesStaff and PropertyForRent.

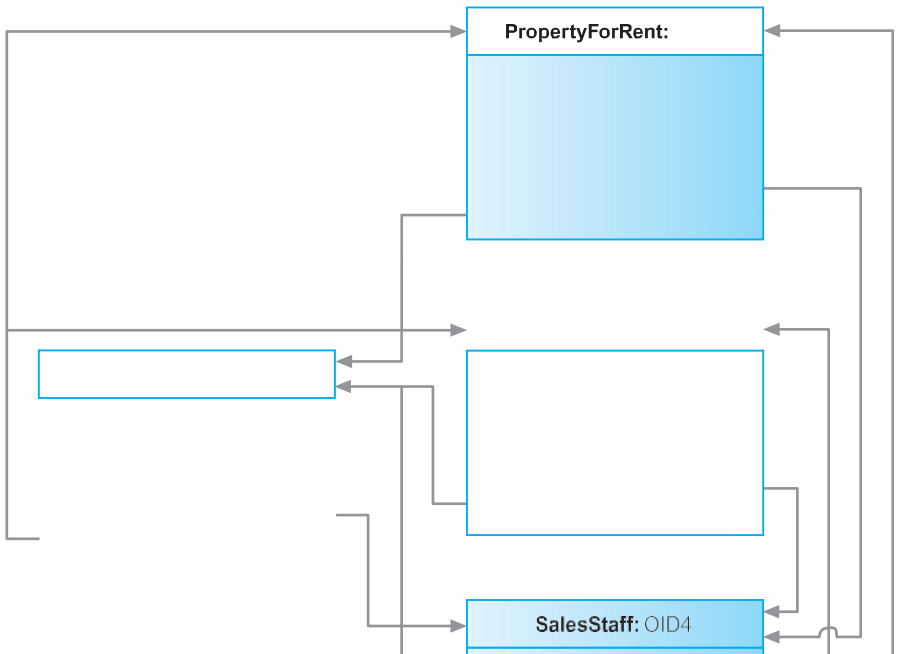
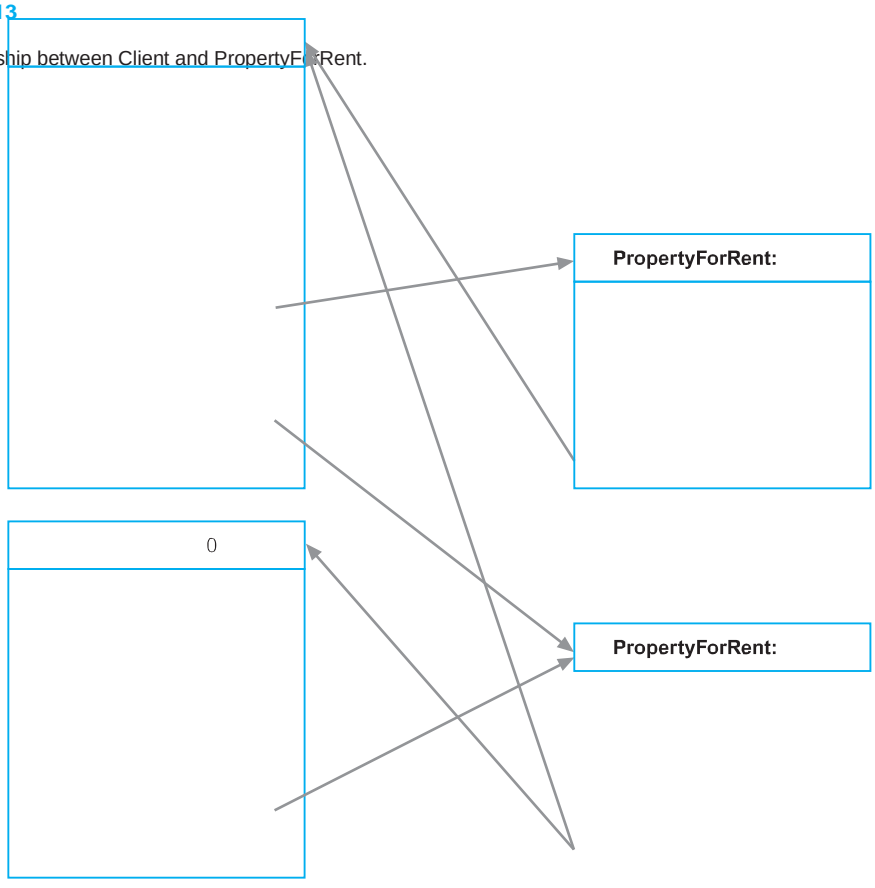


Figure 25.12
1:* relationships between Branch, SalesStaff, and PropertyForRent.

Figure 25.13

A *:* relationship between Client and PropertyForRent.



: relationships

A *:* relationship between objects A and B is represented by adding an attribute con- taining a set of references to each object. For example, there is a *:* relationship between Client and PropertyForRent, as represented in Figure 25.13. For relational database design, we would decompose the *:* relationship into two 1:* relationships linked by an inter- mediate entity. It is also possible to represent this model in an OODBMS, as shown in Figure 25.14.

Referential integrity

In Section 3.3.3 we discussed referential integrity in terms of primary and foreign keys. Referential integrity requires that any referenced object must exist. For example, consider the 1:1 relationship between Manager and Branch in Figure 25.11. The Branch instance, OID1, references a Manager instance, OID6. If the user deletes this Manager instance with- out updating the Branch instance accordingly, referential integrity is lost. There are several techniques that can be used to handle referential integrity:

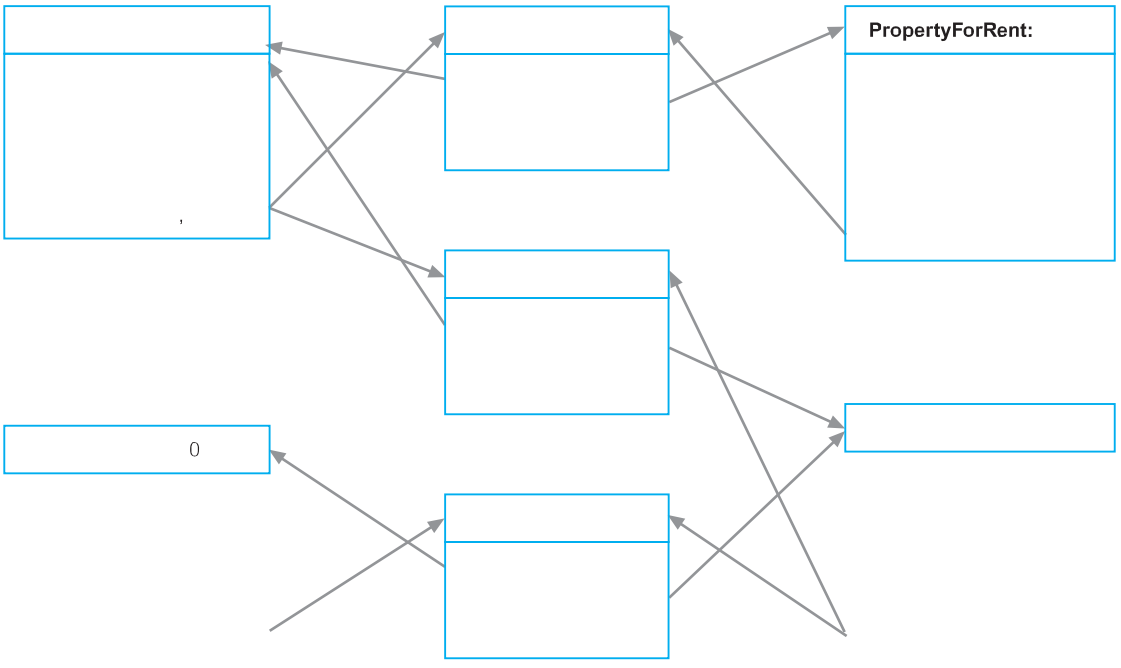


Figure 25.14

Alternative design of

: relationship with intermediate class.

25.6.3

ects when they are no longer accessible by the user. This is the approach taken by GemStone.

- *Allow the user to delete objects when they are no longer required* In this case the system may detect an invalid reference automatically and set the reference to NULL (the null pointer) or disallow the deletion. The Versant OODBMS uses this approach to enforce referential integrity.

- *Allow the user to modify and delete objects and relationships when they are no longer required* In this case the system automatically maintains the integrity of objects, possibly using inverse attributes. For example, in Figure 25.11 we have a relationship from Branch to Manager and an inverse relationship from Manager to Branch. When a Manager object is deleted, it is easy for the system to use this inverse relationship to adjust the reference in the Branch object accordingly. The Ontos, Objectivity/DB, and ObjectStore OODBMSs provide this form of integrity, as does the ODMG Object Model (see Section 27.2).

- *Do not allow the user to explicitly delete objects* In this case the system is responsible for 'garbage collection'; in other words, the system automatically deletes obj

Behavioral Design

The EER approach by itself is insufficient to complete the design of an object-oriented database. The EER approach must be supported with a technique that identifies and documents the behavior of each class of object. This involves a detailed analysis of the

processing requirements of the enterprise. In a conventional data flow approach using Data Flow Diagrams (DFDs), for example, the processing requirements of the system are analyzed separately from the data model. In object-oriented analysis, the processing re- quirements are mapped on to a set of methods that are unique for each class. The methods that are visible to the user or to other objects (**public methods**) must be distinguished from methods that are purely internal to a class (**private methods**). We can identify three types of public and private method:

- constructors and destructors;
- access;
- transform.

Constructors and destructors

Constructor methods generate new instances of a class and each new instance is given a unique OID. Destructor methods delete class instances that are no longer required. In some systems, destruction is an automatic process: whenever an object becomes inaccessible from other objects, it is automatically deleted. We referred to this previously as garbage collection.

Access methods

Access methods return the value of an attribute or set of attributes of a class instance. It may return a single attribute value, multiple attribute values, or a collection of values. For example, we may have a method getSalary for a class SalesStaff that returns a member of staff's salary, or we may have a method getContactDetails for a class Person that returns a person's address and telephone number. An access method may also return data relating to the class. For example, we may have a method getAverageSalary for a class SalesStaff that calculates the average salary of all sales staff. An access method may also derive data from an attribute. For example, we may have a method getAge for Person that calculates a person's age from the date of birth. Some systems automatically generate a method to access each attribute. This is the approach taken in the SQL:2003 standard, which provides an auto- matic *observer* (get) method for each attribute of each new data type (see Section 28.4).

Transform methods

Transform methods change (transform) the state of a class instance. For example, we may have a method incrementSalary for the SalesStaff class that increases a member of staff's salary by a specified amount. Some systems automatically generate a method to update each attribute. Again, this is the approach taken in the SQL:2003 standard, which provides an automatic *mutator* (put) method for each attribute of each new data type (see Section 28.4).

Identifying methods

There are several methodologies for identifying methods, which typically combine the following approaches:

- identify the classes and determine the methods that may be usefully provided for each class;
- decompose the application in a top-down fashion and determine the methods that are required to provide the required functionality.

For example, in the *DreamHome* case study we identified the operations that are to be undertaken at each branch office. These operations ensure that the appropriate information is available to manage the office efficiently and effectively, and to support the services provided to owners and clients (see Appendix A). This is a top-down approach: we inter- viewed the relevant users and, from that, determined the operations that are required. Using the knowledge of these required operations and using the EER model, which has identified the classes that were required, we can now start to determine what methods are required and to which class each method should belong.

A more complete description of identifying methods is outside the scope of this book. There are several methodologies for object-oriented analysis and design, and the interested reader is referred to Rumbaugh *et al*. (1991), Coad and Yourdon (1991), Graham (1993), Blaha and Premerlani (1997), and Jacobson *et al.* (1999).

Oriented Analysis and Design with UML

In this book we have promoted the use of the UML (Unified Modeling Language) for ER modeling and conceptual database design. As we noted at the start of Chapter 11, UML represents a unification and evolution of several object-oriented analysis and design methods that appeared in the late 1980s and early 1990s, particularly the Booch method from Grady Booch, the Object Modeling Technique (OMT) from James Rumbaugh *et al*., and Object-Oriented Software Engineering (OOSE) from Ivar Jacobson *et al*. The UML has been adopted as a standard by the Object Management Group (OMG) and has been accepted by the software community as the primary notation for modeling objects and components.

The UML is commonly defined as ‘a standard language for specifying, constructing, visualizing, and documenting the artifacts of a software system’. Analogous to the use of architectural blueprints in the construction industry, the UML provides a common language for describing software models. The UML does not prescribe any particular methodology, but instead is flexible and customizable to fit any approach and it can be used in conjunction with a wide range of software lifecycles and development processes.

The primary goals in the design of the UML were to:

- Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.

- Provide extensibility and specialization mechanisms to extend the core concepts. For example, the UML provides *stereotypes*, which allow new elements to be defined by extending and refining the semantics of existing elements. A stereotype is enclosed in double chevrons (<< … >>).

n Be independent of particular programming languages and development processes.

n Provide a formal basis for understanding the modeling language.

n Encourage the growth of the object-oriented tools market.

n Support higher-level development concepts such as collaborations, frameworks, patterns, and components.

n Integrate best practices.

In this section we briefly examine some of the components of the UML.

UML Diagrams

UML defines a number of diagrams, of which the main ones can be divided into the fol- lowing two categories:

n *Structural diagrams,* which describe the static relationships between components.

These include:

- class diagrams,
- object diagrams,
- component diagrams,
- deployment diagrams.

n *Behavioral diagrams,* which describe the dynamic relationships between components.

These include:

- use case diagrams,
- sequence diagrams,
- collaboration diagrams,
- statechart diagrams,
- activity diagrams.

We have already used the class diagram notation for ER modeling earlier in the book. In the remainder of this section we briefly discuss the remaining types of diagrams and pro- vide examples of their use.

Object diagrams

Object diagrams model instances of classes and are used to describe the system at a par- ticular point in time. Just as an object is an instance of a class we can view an object dia- gram as an instance of a class diagram. We referred to this type of diagram as a semantic net diagram in Chapter 11. Using this technique, we can validate the class diagram (ER diagram in our case) with 'real world' data and record test cases. Many object diagrams are depicted using only entities and relationships (*objects* and *associations* in the UML terminology). Figure 25.15 shows an example of an object diagram for the Staff *Manages* PropertyForRent relationship.

25.7.1

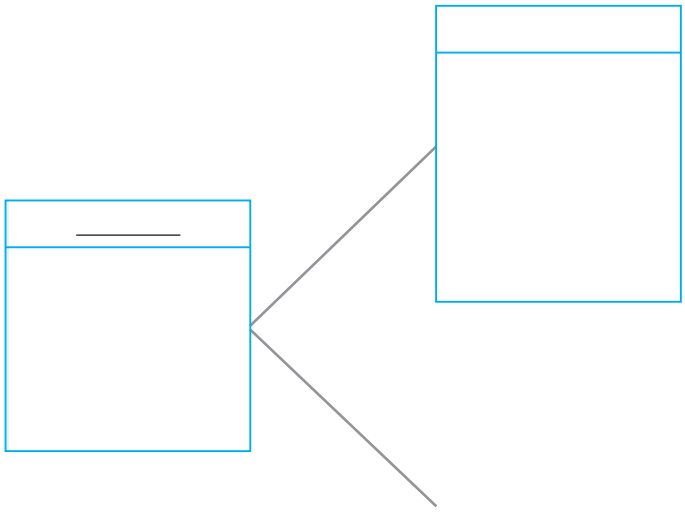


Figure 25.15
Example object diagram showing instances of the Staff *Manages* PropertyForRent relationship.

Component diagrams

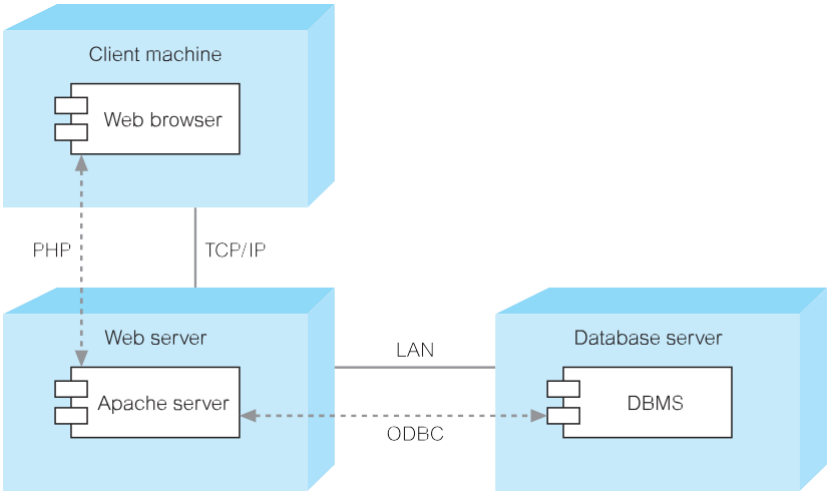
Component diagrams describe the organization and dependencies among physical soft- ware components, such as source code, runtime (binary) code, and executables. For example, a component diagram can illustrate the dependency between source files and executable files, similar to the information within makefiles, which describe source code dependencies and can be used to compile and link an application. A component is repre- sented by a rectangle with two tabs overlapping the left edge. A dependency is denoted by a dotted arrow going from a component to the component it depends on.

Deployment diagrams

Deployment diagrams depict the configuration of the runtime system, showing the hardware nodes, the components that run on these nodes, and the connections between nodes. A node is represented by a three-dimensional cube. Component and deployment diagrams can be combined as illustrated in Figure 25.16.

Use case diagrams

The UML enables and promotes (although does not mandate or even require) a use-case driven approach for modeling objects and components. Use case diagrams model the functionality provided by the system (*use cases*), the users who interact with the system (*actors*), and the association between the users and the functionality. Use cases are used in the requirements collection and analysis phase of the software development lifecycle to represent the high-level requirements of the system. More specifically, a use case specifies



a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor (Jacobson *et al.*, 1999).

An individual use case is represented by an ellipse, an actor by a stick figure, and an association by a line between the actor and the use case. The role of the actor is written beneath the icon. Actors are not limited to humans. If a system communicates with another application, and expects input or delivers output, then that application can also be con- sidered an actor. A use case is typically represented by a verb followed by an object, such as View property, Lease property. An example use case diagram for Client with four use cases is shown in Figure 25.17(a) and a use case diagram for Staff in Figure 25.17(b). The use case notation is simple and therefore is a very good vehicle for communication.

Sequence diagrams

A sequence diagram models the interactions between objects over time, capturing the behavior of an individual use case. It shows the objects and the *messages* that are passed between these objects in the use case. In a sequence diagram, objects and actors are shown as columns, with vertical *lifelines* indicating the lifetime of the object over time. An activation/focus of control, which indicates when the object is performing an action, is modeled as a rectangular box on the lifeline; a lifeline is represented by a vertical dotted line extending from the object. The destruction of an object is indicated by an X at the appropriate point on its lifeline. Figure 25.18 provides an example of a sequence diagram for the Search properties use case that may have been produced during design (an earlier sequence diagram may have been produced without parameters to the messages).

Collaboration diagrams

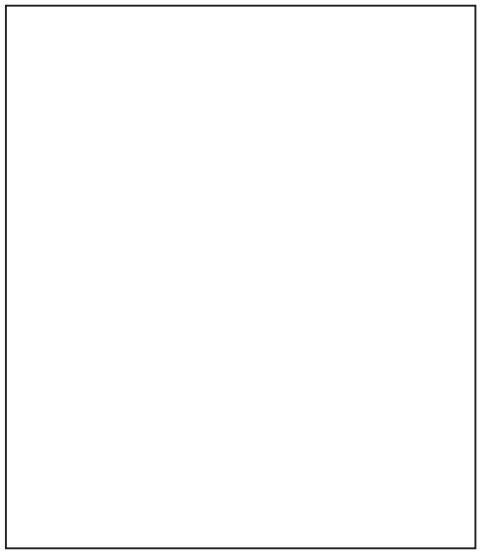
A collaboration diagram is another type of interaction diagram, in this case showing the interactions between objects as a series of sequenced messages. This type of diagram is a cross between an object diagram and a sequence diagram. Unlike the sequence diagram,

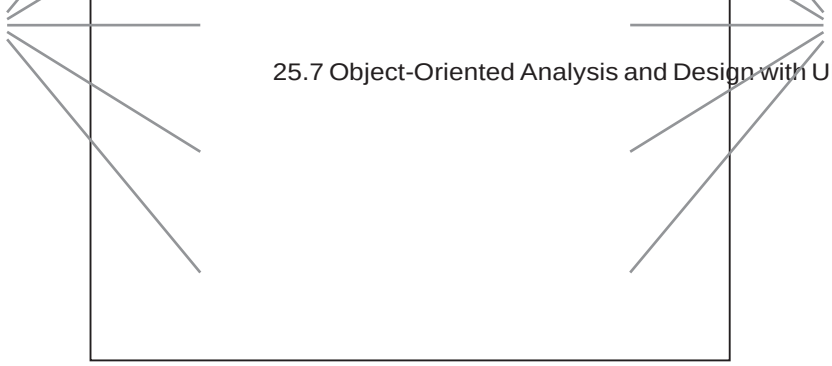
Figure 25.16 Combined component and deployment diagram.

Figure 25.17

(a) Use case diagram with an actor (Client) and four use cases; (b) use case diagram for Staff.

case





which models the interaction in a column and row type format, the collaboration diagram

(b)

uses the free-form arrangement of objects, which makes it easier to see all interactions involving a particular object. Messages are labeled with a chronological number to maintain ordering information. Figure 25.19 provides an example of a collaboration diagram for the Search properties use case.

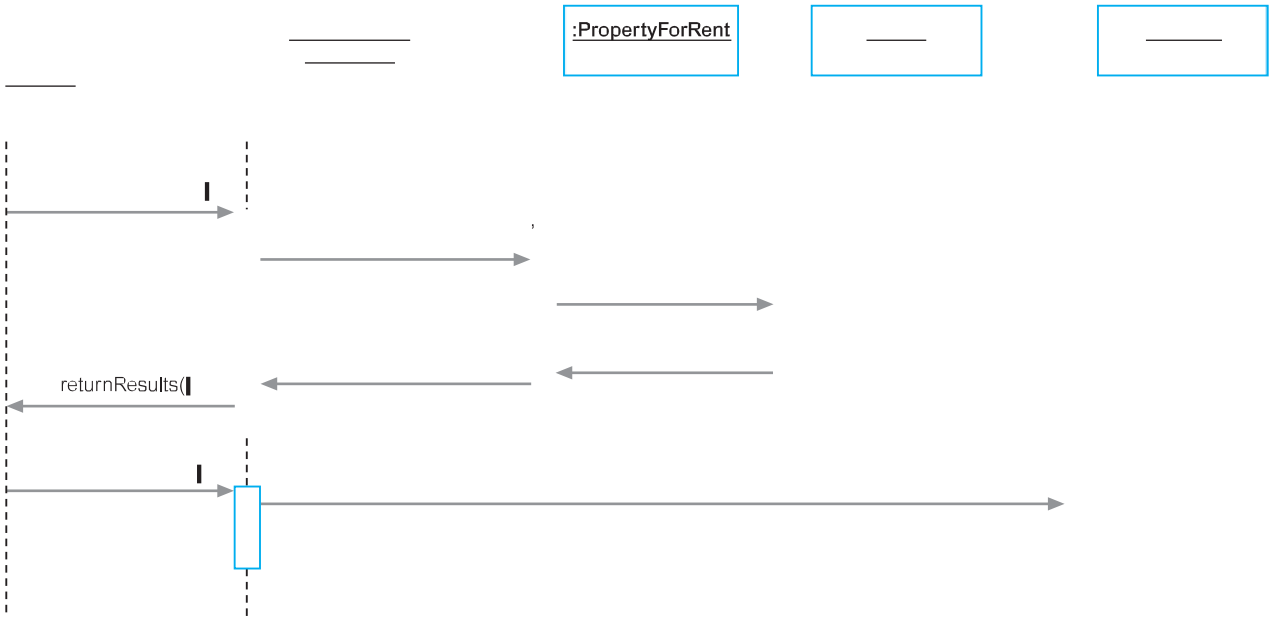


Figure 25.18 Sequence diagram for Search properties use case.

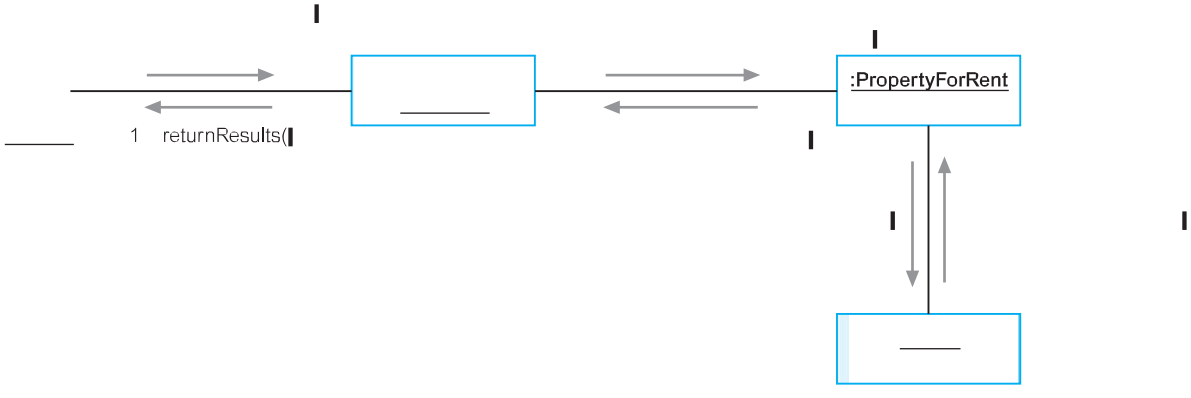
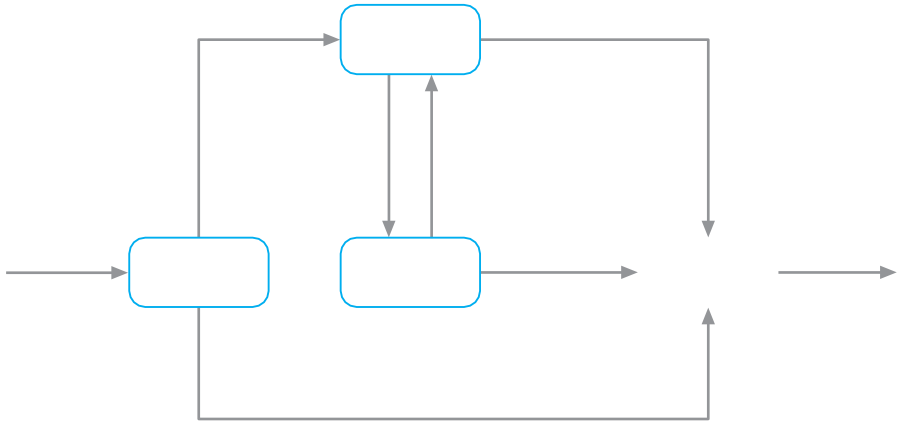


Figure 25.19 Collaboration diagram for Search properties use case.

Statechart diagrams

Statechart diagrams, sometimes referred to as state diagrams, show how objects can change in response to external events. While other behavioral diagrams typically model the inter- action between multiple objects, statechart diagrams usually model the transitions of a specific object. Figure 25.20 provides an example of a statechart diagram for PropertyForRent. Again, the notation is simple consisting of a few symbols:

Figure 25.20
Statechart diagram for PropertyForRent.



rejectProperty

- *States* are represented by boxes with rounded corners.
- *Transitions* are represented by solid arrows between states labeled with the 'event-name/action' (the *event* triggers the transition and *action* is the result of the transition). For example, in Figure 25.20, the transition from state Pending to Available is triggered by an approveProperty event and gives rise to the action called makeAvailable().
- *Initial state* (the state of the object before any transitions) is represented by a solid circle with an arrow to the initial state.
- *Final state* (the state that marks the destruction of the object) is represented by a solid circle with a surrounding circle and an arrow coming from a preceding state.

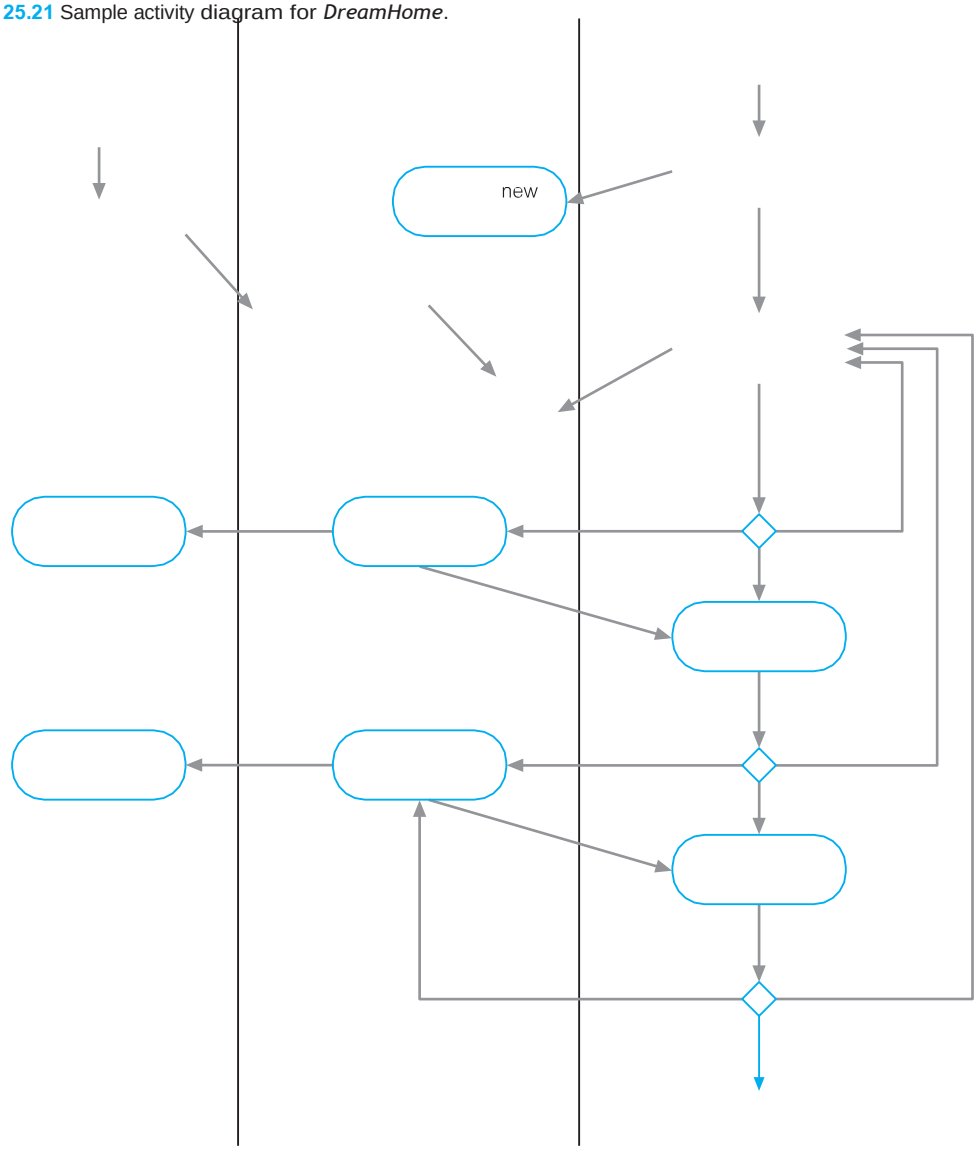
Activity diagrams

Activity diagrams model the flow of control from one activity to another. An activity diagram typically represents the invocation of an operation, a step in a business process, or an entire business process. It consists of activity states and transitions between them. The diagram shows flow of control and branches (small diamonds) can be used to specify alternative paths of transitions. Parallel flows of execution are represented by fork and join constructs (solid rectangles). *Swimlanes* can be used to separate independent areas. Figure 25.21 shows a first-cut activity diagram for *DreamHome*.

25.7.2 Usage of UML in the Methodology for Database Design

Many of the diagram types we have described above are useful during the database sys- tem development lifecycle, particularly during requirements collection and analysis, and database and application design. The following guidelines may prove helpful (McCready, 2003):

Figure 25.21 Sample activity diagram for *DreamHome*.



- Produce use case diagrams from the requirements specification or while producing the requirements specification to depict the main functions required of the system. The use cases can be augmented with use case descriptions, textual descriptions of each use case.

- Produce the first-cut class diagram (ER model).

- Produce a sequence diagram for each use case or group of related use cases. This will show the interaction between classes (entities) necessary to support the functionality defined in each use case. Collaboration diagrams can easily be produced from the

sequence diagrams (for example, the CASE tool Rational Rose can automatically pro- duce a collaboration diagram from the corresponding sequence diagram).

- It may be useful to add a *control class* to the class diagram to represent the interface between the actors and the system (control class operations are derived from the use cases).
- Update the class diagram to show the required methods in each class.
- Create a state diagram for each class to show how the class changes state in response to messages it receives. The appropriate messages are identified from the sequence diagrams.
- Revise earlier diagrams based on new knowledge gained during this process (for example, the creation of state diagrams may identify additional methods for the class diagram).

Chapter Summary

- Advanced database applications include computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided software engineering (CASE), network management systems, office information systems (OIS) and multimedia systems, digital publishing, geographic information systems (GIS), and inter- active and dynamic Web sites, as well as applications with complex and interrelated objects and procedural data.
- The relational model, and relational systems in particular, have weaknesses such as poor representation of 'real world' entities, semantic overloading, poor support for integrity and enterprise constraints, limited operations, and impedance mismatch. The limited modeling capabilities of relational DBMSs have made them unsuitable for advanced database applications.
- The concept of **encapsulation** means that an object contains both a data structure and the set of operations that can be used to manipulate it. The concept of **information hiding** means that the external aspects of an object are separated from its internal details, which are hidden from the outside world.
- An **object** is a uniquely identifiable entity that contains both the attributes that describe the **state** of a 'real world' object and the actions (**behavior**) that are associated with it. Objects can contain other objects. A key part of the definition of an object is unique identity. In an object-oriented system, each object has a unique system-wide identifier (the **OID**) that is independent of the values of its attributes and, ideally, invisible to the user.
- **Methods** define the behavior of the object. They can be used to change the object's state by modifying its attribute values or to query the value of selected attributes. **Messages** are the means by which objects com- municate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object.

- Objects that have the same attributes and respond to the same messages can be grouped together to form a **class**. The attributes and associated methods can then be defined once for the class rather than separately for each object. A class is also an object and has its own attributes and methods, referred to as **class attributes** and **class methods**, respectively. Class attributes describe the general characteristics of the class, such as totals or averages.

- **Inheritance** allows one class to be defined as a special case of a more general class. These special cases are known as **subclasses** and the more general cases are known as **superclasses**. The process of forming a superclass is referred to as **generalization**; forming a subclass is **specialization**. A subclass inherits all the properties of its superclass and additionally defines its own unique properties (attributes and methods). All instances of the subclass are also instances of the superclass. The *principle of substitutability* states that an instance of the subclass can be used whenever a method or a construct expects an instance of the superclass.
- **Overloading** allows the name of a method to be reused within a class definition or across definitions. **Overriding**, a special case of overloading, allows the name of a property to be redefined in a subclass. **Dynamic binding** allows the determination of an object's type and methods to be deferred until runtime.
- In response to the increasing complexity of database applications, two 'new' data models have emerged: the **Object-Oriented Data Model** (OODM) and the **Object-Relational Data Model** (ORDM). However, unlike

Review Questions

25.1 Discuss the general characteristics of advanced database applications.

25.2 Discuss why the weaknesses of the relational data model and relational DBMSs may make them unsuitable for advanced database applications.

25.3 Define each of the following concepts in the relational context of an object-oriented data model:

(a) abstraction, encapsulation, and information hiding;
(b) objects and attributes;
(c) object identity;
(d) methods and messages;
(e) classes, subclasses, superclasses,
(f) overriding and overloading;
(g) polymorphism and dynamic binding.

Give examples using the *DreamHome* sample data shown in Figure 3.3.

25.4 Discuss the difficulties involved in mapping objects created in an object-oriented programming language to a database.

25.5 Describe the three generations of DBMSs.

25.6 Describe how relationships can be modeled in an OODBMS.

25.7 Describe the different modeling notations in the UML.

Exercises

Investigate one of the advanced database applications discussed in Section 25.1, or a similar one that handles complex, interrelated data. In particular, examine its functionality and the data types and operations it uses. Map the data types and operations to the object-oriented concepts discussed in Section 25.3.

Analyze one of the RDBMSs that you currently use. Discuss the object-oriented features provided by the system. What additional functionality do these features provide?

For the *DreamHome* case study documented in Appendix A, suggest attributes and methods that would be appropriate for Branch, Staff, and PropertyForRent classes.

Produce use case diagrams and a set of associated sequence diagrams for the *DreamHome* case study docu- mented in Appendix A.

Produce use case diagrams and a set of associated sequence diagrams for the *University Accommodation Office* case study documented in Appendix B.1.

Produce use case diagrams and a set of associated sequence diagrams for the *Easy Drive School of Motoring* case study documented in Appendix B.2.

Produce use case diagrams and a set of associated sequence diagrams for the *Wellmeadows Hospital* case study documented in Appendix B.3.

Chapter

6

Object-Oriented DBMSs – Concepts

Chapter Objectives

In this chapter you will learn:

- The framework for an object-oriented data model.
- The basics of the functional data model.
- The basics of persistent programming languages.
- The main points of the OODBMS Manifesto.
- The main strategies for developing an OODBMS.
- The difference between the two-level storage model used by conventional DBMSs and the single-level model used by OODBMSs.
- How pointer swizzling techniques work.
- The difference between how a conventional DBMS accesses a record and how an OODBMS accesses an object on secondary storage.
- The different schemes for providing persistence in programming languages.
- The advantages and disadvantages of orthogonal persistence.

In the previous chapter we reviewed the weaknesses of the relational data model against the requirements for the types of advanced database applications that are emerging. We also introduced the concepts of object-orientation, which solve some of the classic problems of software development. Some of the advantages often cited in favor of object- orientation are:

- The definition of a system in terms of objects facilitates the construction of software components that closely resemble the application domain, thus assisting in the design and understandability of systems.

- Owing to encapsulation and information hiding, the use of objects and messages encourages modular design – the implementation of one object does not depend on the

internals of another, only on how it responds to messages. Further, modularity is re- inforced and software can be made more reliable.

- The use of classes and inheritance promotes the development of reusable and exten- sible components in the construction of new or upgraded systems.

In this chapter we consider the issues associated with one approach to integrating object-oriented concepts with database systems, namely the **Object-Oriented Database Management System** (OODBMS). The OODBMS started in the engineering and design domains and has recently also become the favored system for financial and telecommunications applications. The OODBMS market is small in comparison to the relational DBMS market and while it had an estimated growth rate of 50% at the end of the 1990s, the market has not maintained this growth.

In the next chapter we examine the object model proposed by the Object Data Management Group, which has become a *de facto* standard for OODBMSs. We also look at ObjectStore, a commercial OODBMS.

Moving away from the traditional relational data model is sometimes referred to as a *revolutionary approach* to integrating object-oriented concepts with database systems. In contrast, in Chapter 28 we examine a more *evolutionary approach* to integrating object-oriented concepts with database systems that extends the relational model. These evolutionary systems are referred to now as **Object-Relational DBMSs** (ORDBMSs), although an earlier term used was *Extended-Relational DBMSs*.

Structure of this Chapter

In Section 26.1 we provide an introduction to object-oriented data models and persistent languages, and discuss how, unlike the relational data model, there is no universally agreed object-oriented data model. We also briefly review the *Object-Oriented Database System Manifesto*, which proposed thirteen mandatory features for an OODBMS, and examine the different approaches that can be taken to develop an OODBMS. In Section 26.2 we examine the difference between the two-level storage model used by conventional DBMSs and the single-level model used by OODBMSs, and how this affects data access. In Section 26.3 we discuss the various approaches to providing persistence in programming languages and the different techniques for pointer swizzling. In Section 26.4 we examine some other issues

associated with OODBMSs, namely extended transaction models, version management, schema evolution, OODBMS architectures, and benchmarking. In Section 26.6 we review the advantages and disadvantages of OODBMSs.

To gain full benefit from this chapter, the reader needs to be familiar with the contents of Chapter 25. The examples in this chapter are once again drawn from the *DreamHome* case study documented in Section 10.4 and Appendix A.

Introduction to Object-Oriented Data Models and OODBMSs

In this section we discuss some background concepts to the OODBMS including the functional data model and persistent programming languages. We start by looking at the definition of an OODBMS.

Definition of Object-Oriented DBMSs 26.1.1

In this section we examine some of the different definitions that have been proposed for an object-oriented DBMS. Kim (1991) defines an Object-Oriented Data Model (OODM), Object-Oriented Database (OODB), and an Object-Oriented DBMS (OODBMS) as:

OODM	A (logical) data model that captures the semantics of objects supported in object-oriented programming.
OODB	A persistent and sharable collection of objects defined by an OODM.
OODBMS	The manager of an OODB

These definitions are very non-descriptive and tend to reflect the fact that there is no one object-oriented data model equivalent to the underlying data model of relational systems. Each system provides its own interpretation of base functionality. For example, Zdonik and Maier (1990) present a threshold model that an OODBMS must, at a minimum, satisfy:

(1) it must provide database functionality;

(2) it must support object identity;

(3) it must provide encapsulation;

(4) it must support objects with complex state.

The authors argue that although inheritance may be useful, it is not essential to the defini- tion, and an OODBMS could exist without it. On the other hand, Khoshafian and Abnous (1990) define an OODBMS as:

(1) object-orientation = abstract data types + inheritance + object identity;

(2) OODBMS = object-orientation + database capabilities.

Yet another definition of an OODBMS is given by Parsaye *et al*. (1989):

(1) high-level query language with query optimization capabilities in the underlying system;

(2) support for persistence, atomic transactions, and concurrency and recovery control;

(3) support for complex object storage, indexes, and access methods for fast and efficient retrieval;

(4) OODBMS = object-oriented system + (1) + (2) + (3).

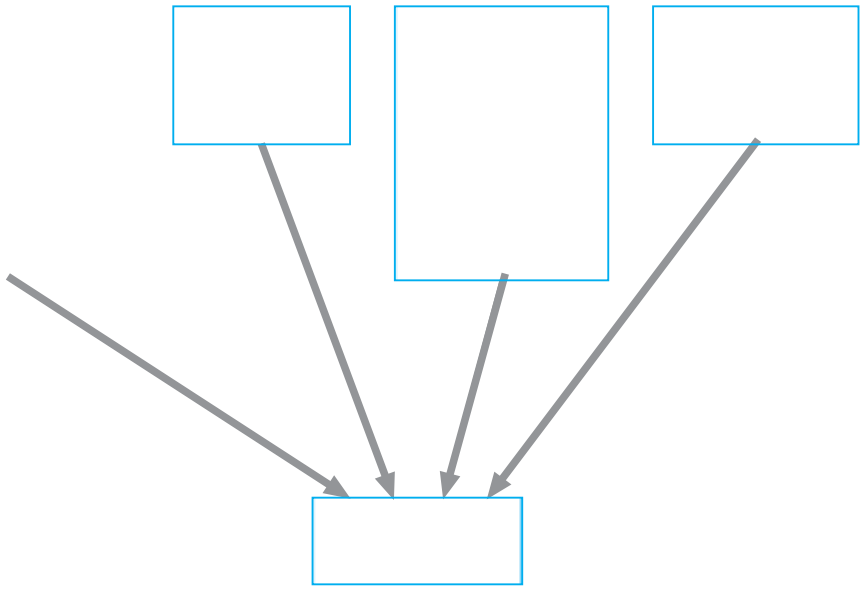


Figure 26.1
Origins of object-oriented data model.

Studying some of the current commercial OODBMSs, such as GemStone from Gemstone Systems Inc. (previously Servio Logic Corporation), Objectivity/DB from Objectivity Inc., ObjectStore from Progress Software Corporation (previously Object Design Inc.), 'FastObjects by Poet' from Poet Software Corporation, Jasmine Object Database from Computer Associates/ Fujitsu Limited, and Versant (VDS) from Versant Corporation, we can see that the concepts of object-oriented data models are drawn from different areas, as shown in Figure 26.1.

In Section 27.2 we examine the object model proposed by the Object Data Management

Group (ODMG), which many of these vendors intend to support. The ODMG object model is important because it specifies a standard model for the semantics of database objects and supports interoperability between compliant OODBMSs. For surveys of the basic concepts of Object-Oriented Data Models the interested reader is referred to Dittrich (1986) and Zaniola *et al*. (1986).

Functional Data Models

In this section we introduce the functional data model (FDM), which is one of the simplest in the family of semantic data models (Kerschberg and Pacheco, 1976; Sibley and Kerschberg, 1977). This model is interesting because it shares certain ideas with the object approach including object identity, inheritance, overloading, and navigational access. In the FDM, any data retrieval task can be viewed as the process of evaluating and return- ing the result of a function with zero, one, or more arguments. The resulting data model is conceptually simple while at the same time is very expressive. In the FDM, the main modeling primitives are **entities** and **functional relationships**.

Entities

Entities are decomposed into (abstract) entity types and printable entity types. **Entity types** correspond to classes of 'real world' objects and are declared as functions with zero arguments that return the type ENTITY. For example, we could declare the Staff and PropertyForRent entity types as follows:

```
Staff() > ENTITY

PropertyForRent() > ENTITY
```

Printable entity types are analogous to the base types in a programming language and include: INTEGER, CHARACTER, STRING, REAL, and DATE. An attribute is defined as a *functional relationship*, taking the entity type as an argument and returning a printable entity type. Some of the attributes of the Staff entity type could be declared as follows:

```
staffNo(Staff) › STRING sex(Staff) >
CHAR salary(Staff) > REAL
```

Thus, applying the function staffNo to an entity of type Staff returns that entity's staff number, which is a printable value of type STRING. We can declare a composite attribute by first declaring the attribute to be an entity type and then declaring its components as functional relationships of the entity type. For example, we can declare the composite attribute Name of Staff as follows:

```
Name() > ENTITY

Name(Staff) > NAME fName(Name)
› STRING lName(Name) › STRING
```

Relationships

Functions with arguments model not only the properties (attributes) of entity types but also relationships between entity types. Thus, the FDM makes no distinction between attributes and relationships. Each relationship may have an inverse relationship defined. For exam- ple, we may model the one-to-many relationship Staff *Manages* PropertyForRent as follows:

```
Manages(Staff ) > PropertyForRent ManagedBy(PropertyForRent) › Staff
INVERSE OF Manages
```

In this example, the double-headed arrow is used to represent a one-to-many relationship. This notation can also be used to represent multi-valued attributes. Many-to-many relation- ships can be modeled by using the double-headed arrow in both directions. For example, we may model the *.* relationship Client *Views*

PropertyForRent as follows:

```
Views(Client) > PropertyForRent ViewedBy(PropertyForRent) › Client
INVERSE OF Views
```

Note, an entity (instance) is some form of token identifying a unique object in the database and typically representing a unique object in the 'real world'. In addition, a function maps a given entity to one or more target entities (for example, the function Manages maps a

particular Staff entity to a set of PropertyForRent entities). Thus, all inter-object relationships are modeled by associating the corresponding entity instances and not their names or keys. Thus, referential integrity is an implicit part of the functional data model and requires no explicit enforcement, unlike the relational data model.

The FDM also supports multi-valued functions. For example, we can model the attribute viewDate of the previous relationship Views as follows:

viewDate(Client, PropertyForRent) › DATE

Inheritance and path expressions

The FDM supports inheritance through entity types. For example, the function Staff() returns a set of staff entities formed as a subset of the ENTITY type. Thus, the entity type Staff is a subtype of the entity type ENTITY. This subtype/supertype relationship can be extended to any level. As would be expected, subtypes inherit all the functions defined over all of its supertypes. The FDM also supports the principle of substitutability (see Section 25.3.6), so that an instance of a subtype is also an instance of its supertypes. For example, we could declare the entity type Supervisor to be a subtype of the entity type Staff as follows:

Staff() > ENTITY

Supervisor() > ENTITY

IS-A-STAFF(Supervisor) > Staff

The FDM allows **derived functions** to be defined from the composition of multiple functions. Thus, we can define the following derived functions (note the overloading of function names):

fName(Staff) > fName(Name(Staff)) fName(Supervisor) › fName(IS-A-STAFF(Supervisor))

The first derived function returns the set of first names of staff by evaluating the com- posite function on the right-hand side of the definition. Following on from this, in the second case the right-hand side of the definition is evaluated as the composite function fName(Name(IS-A-STAFF(Supervisor))). This composition is called a **path expression** and may be more recognizable written in *dot notation*:

Supervisor.IS-A-STAFF.Name.fname

Figure 26.2(a) provides a declaration of part of the *DreamHome* case study as an FDM schema and Figure 26.2(b) provides a corresponding graphical representation.

Functional query languages

Path expressions are also used within a functional query language. We will not discuss query languages in any depth but refer the interested reader to the papers cited at the end of this section. Instead, we provide a simple example to illustrate the language. For example, to retrieve the surnames of clients who have viewed a property managed by staff member SG14, we could write:

RETRIEVE lName(Name(ViewedBy(Manages(Staff))))

WHERE staffNo(Staff) = 'SG14'

Working from the inside of the path expression outwards, the function Manages(Staff) returns a set of PropertyForRent entities. Applying the function ViewedBy to this result returns a set of Client entities. Finally, applying the functions Name and lName returns the surnames of these clients. Once again, the equivalent dot notation may be more recognizable:

RETRIEVE Staff.Manages.ViewedBy.Name.lName

WHERE Staff.staffNo = 'SG14'

Note, the corresponding SQL statement would require three joins and is less intuitive than the FDM statement:

SELECT c.lName

FROM Staff s, PropertyForRent p, Viewing v, Client c

WHERE s.staffNo = p.staffNo **AND** p.propertyNo = v.propertyNo **AND**

v.clientNo = c.clientNo **AND** s.staffNo = 'SG14'

Advantages

Some of the advantages of the FDM include:

- *Support for some object-oriented concepts* The FDM is capable of supporting object identity, inheritance through entity class hierarchies, function name overloading, and navigational access.

- *Support for referential integrity* The FDM is an entity-based data model and im- plicitly supports referential integrity.

- *Irreducibility* The FDM is composed of a small number of simple concepts that rep- resent semantically irreducible units of information. This allows a database schema to be depicted graphically with relative ease thereby simplifying conceptual design.

- *Easy extensibility* Entity classes and functions can be added/deleted without requiring modification to existing schema objects.

- *Suitability for schema integration* The conceptual simplicity of the FDM means that it can be used to represent a number of different data models including relational, network, hierarchical, and object-oriented. This makes the

FDM a suitable model for the integration of heterogeneous schemas within multidatabase systems (MDBSs) dis- cussed in Section 22.1.3.

n *Declarative query language* The query language is declarative with well-understood semantics (based on lambda calculus). This makes the language easy to transform and optimize.

There have been many proposals for functional data models and languages. The two earliest were FQL (Buneman and Frankel, 1979) and, perhaps the best known, DAPLEX (Shipman, 1981). The attraction of the functional style of these languages has produced many systems such as GDM (Batory *et al.*, 1988), the Extended FDM (Kulkarni and Atkinson, 1986, 1987), FDL (Poulovassilis and King, 1990), PFL (Poulovassilis and

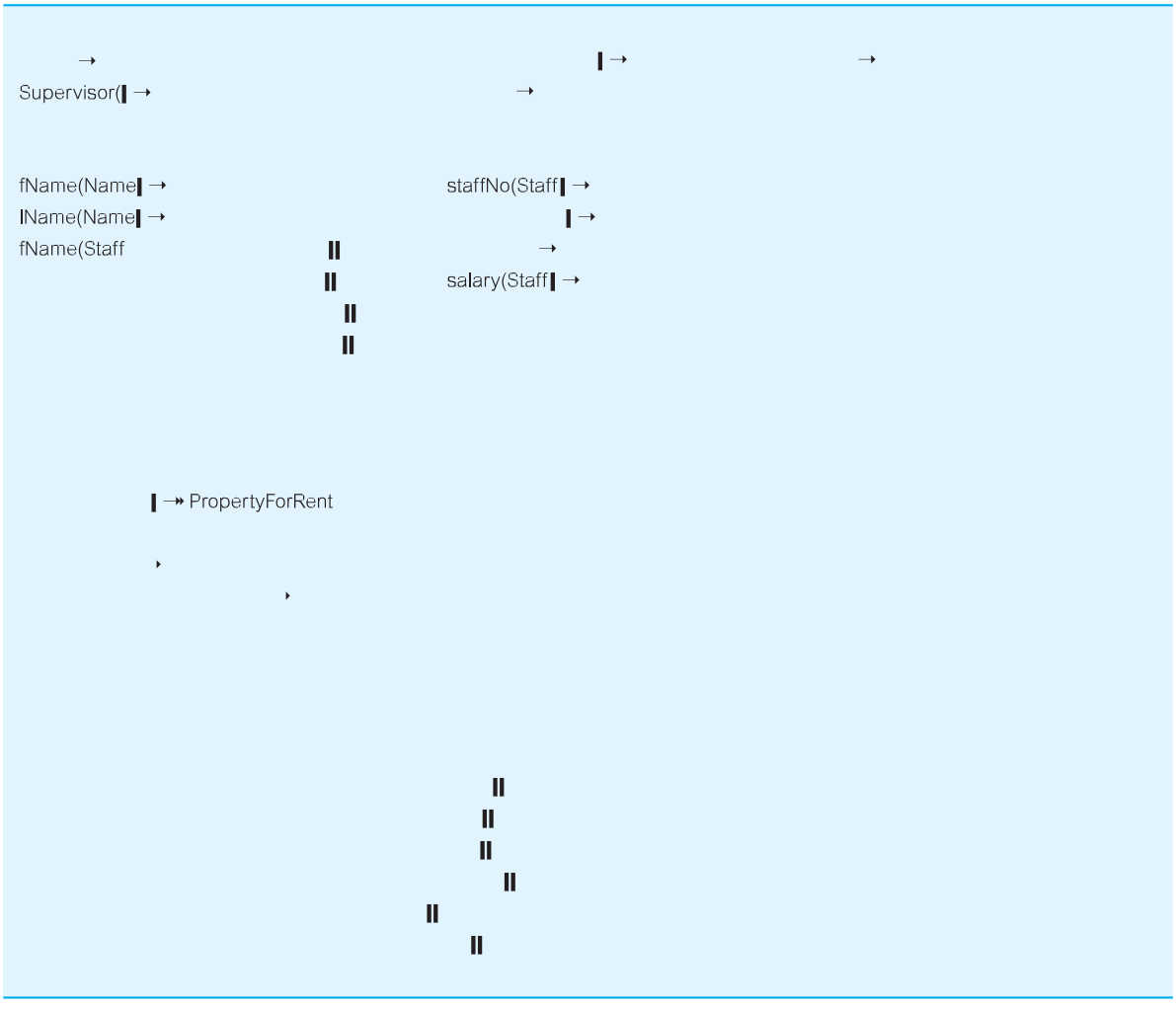


Figure 26.2 (a) Declaration of part of *DreamHome* as an FDM schema; (b) corresponding diagrammatic representation.

Small, 1991), and P/ FDM (Gray *et al.*, 1992). The functional data languages have also been used with non-functional data models, such as PDM (Manola and Dayal, 1986), IPL (Annevelink, 1991), and LIFOO (Boucelma and Le Maitre, 1991). In the next section we examine another area of research that played a role in the development of the OODBMS.

Persistent Programming Languages

Before we start to examine OODBMSs in detail, we introduce another interesting but sep- arate area of development known as *persistent programming languages*.

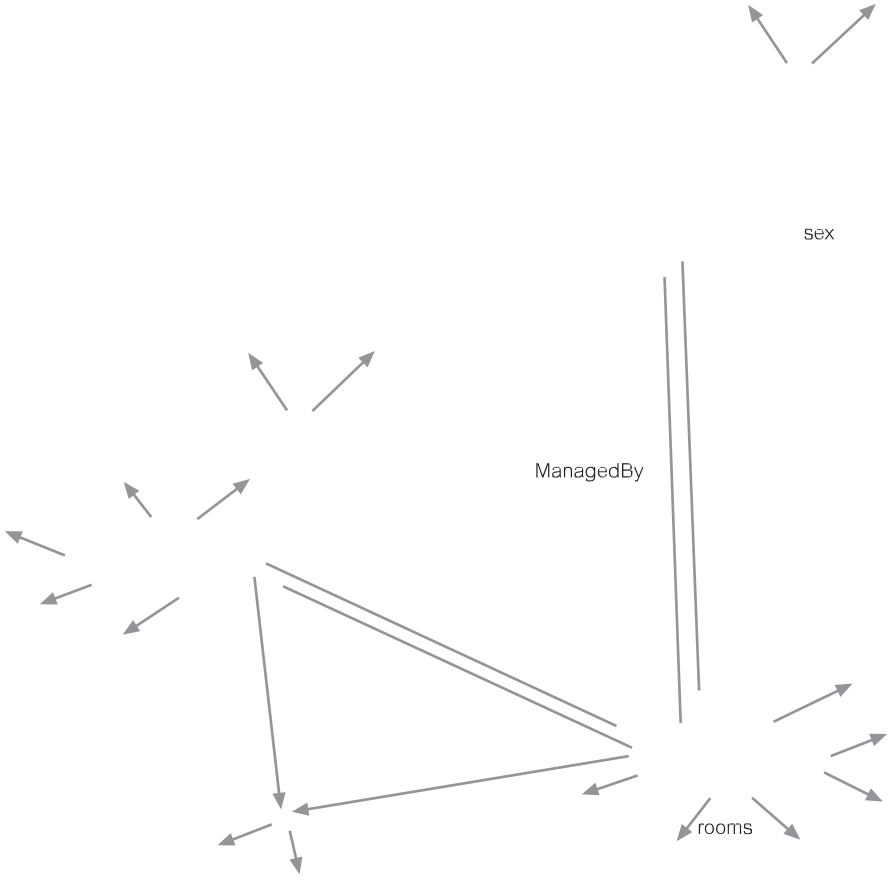


Figure 26.2
(*cont'd*)

Persistent programming language	A language that provides its users with the ability to (transparently) preserve data across successive executions of a program, and even allows such data to be used by many different programs.

Data in a persistent programming language is independent of any program, able to exist beyond the execution and lifetime of the code that created it. Such languages were originally intended to provide neither full database functionality nor access to data from multiple languages (Cattell, 1994).

Database programming language	A language that integrates some ideas from the database programming model with traditional programming language features.

In contrast, a database programming language is distinguished from a persistent programming language by its incorporation of features beyond persistence, such as transaction management, concurrency control, and recovery (Bancilhon and Buneman, 1990). The ISO SQL standard specifies that SQL can be embedded in the programming languages 'C', Fortran, Pascal, COBOL, Ada, MUMPS, and PL/1 (see Appendix E). Communication is through a set of variables in the host language, and a special preprocessor modifies the source code to replace the SQL statements with calls to DBMS routines. The source code can then be compiled and linked in the normal way. Alternatively, an API can be provided, removing the need for any precompilation. Although the embedded approach is rather clumsy, it was useful and necessary, as the SQL2 standard was not computationally complete.[†] The problems with using two different language paradigms have been collectively called the **impedance mismatch** between the application programming language and the database query language (see Section 25.2). It has been claimed that as much as 30% of programming effort and code space is devoted to converting data from database or file formats into and out of program-internal formats (Atkinson *et al.*, 1983). The integration of persistence into the programming language frees the programmer from this responsibility.

Researchers working on the development of persistent programming languages have been motivated primarily by the following aims (Morrison *et al.*, 1994):

- improving programming productivity by using simpler semantics;
- removing *ad hoc* arrangements for data translation and long-term data storage;
- providing protection mechanisms over the whole environment.

Persistent programming languages attempt to eliminate the impedance mismatch by extending the programming language with database capabilities. In a persistent programming language, the language's type system provides the data model, which usually contains rich structuring mechanisms. In some languages, for example PS-algol and Napier88, procedures are 'first class' objects and are treated like any other data objects in the language. For example, procedures are assignable, may be the result of expressions, other procedures or blocks, and may be elements of constructor types. Among other things, procedures can be used to implement abstract data types. The act of importing an abstract data type from the persistent store and dynamically binding it into a program is equivalent to module-linking in more traditional languages.

The second important aim of a persistent programming language is to maintain the same data representation in the application memory space as in the persistent store on secondary storage. This overcomes the difficulty and overhead of mapping between the two rep- resentations, as we see in Section 26.2.

The addition of (transparent) persistence into a programming language is an important enhancement to an interactive development environment, and the integration of the two

paradigms provides increased functionality and semantics. The research into persistent programming languages has had a significant influence on the development of OODBMSs, and many of the issues that we discuss in Sections 26.2, 26.3, and 26.4 apply to both persistent programming languages and OODBMSs. The more encompassing term

[†] The 1999 release of the SQL standard, SQL:1999, added constructs to the language to make it computationally complete.

Persistent Application System (PAS) is sometimes used now instead of persistent programming language (Atkinson and Morrison, 1995).

The *Object-Oriented Database System Manifesto*

The 1989 *Object-Oriented Database System Manifesto* proposed thirteen mandatory features for an OODBMS, based on two criteria: it should be an object-oriented system and it should be a DBMS (Atkinson *et al.*, 1989). The rules are summarized in Table 26.1. The first eight rules apply to the object-oriented characteristic.

(1) Complex objects must be supported

It must be possible to build complex objects by applying constructors to basic objects. The minimal set of constructors are SET, TUPLE, and LIST (or ARRAY). The first two are important because they have gained widespread acceptance as object constructors in the relational model. The final one is important because it allows order to be modeled. Furthermore, the manifesto requires that object constructors must be orthogonal: any constructor should apply to any object. For example, we should be able to use not only SET(TUPLE()) and LIST(TUPLE()) but also TUPLE(SET()) and TUPLE(LIST()).

(2) Object identity must be supported

All objects must have a unique identity that is independent of its attribute values.

(3) Encapsulation must be supported

In an OODBMS, proper encapsulation is achieved by ensuring that programmers have access only to the interface specification of methods, and the data and implementation of these methods are hidden in the objects. However, there may be cases where the enforcement

Table 26.1 Mandatory features in the *Object-Oriented Database System Manifesto*.

Object-oriented characteristics	DBMS characteristics
Complex objects must be supported Object identity must be supported Encapsulation must be supported Types or classes must be supported	Data persistence must be provided
Types or classes must be able to inherit from their ancestors	The DBMS must be capable of handling very large databases
Dynamic binding must be supported	The DBMS must support concurrent users
The DML must be computationally complete The set of data types must be extensible	The DBMS must be capable of recovery from hardware and software failures
	The DBMS must provide a simple way of querying data

of encapsulation is not required: for example, with *ad hoc* queries. (In Section 25.3.1 we noted that encapsulation is seen as one of the great strengths of the object-oriented approach. In which case, why should there be situations where encapsulation can be overriden? The typical argument given is that it is not an ordinary user who is examining the contents of objects but the DBMS. Second, the DBMS could invoke the 'get' method associated with every attribute of every class, but direct examination is more efficient. We leave these arguments for the reader to reflect on.)

(4) Types or classes must be supported

We mentioned the distinction between types and classes in Section 25.3.5. The manifesto requires support for only one of these concepts. The database schema in an object-oriented system comprises a set of classes or a set of types. However, it is not a requirement that the system automatically maintain the *extent* of a type, that is, the set of objects of a given type in the database, or if an extent is maintained, that the system should make it accessible to the user.

(5) Types or classes must be able to inherit from their ancestors

A subtype or subclass should inherit attributes and methods from its supertype or superclass, respectively.

(6) Dynamic binding must be supported

Methods should apply to objects of different types (overloading). The implementation of a method will depend on the type of the object it is applied to (overriding). To provide this functionality, the system cannot bind method names until runtime (dynamic binding).

(7) The DML must be computationally complete

In other words, the Data Manipulation Language (DML) of the OODBMS should be a general-purpose programming language. This was obviously not the case with the SQL2 standard (see Section 5.1), although with the release of the SQL:1999 standard the language is computationally complete (see Section 28.4).

(8) The set of data types must be extensible

The user must be able to build new types from the set of predefined system types. Further-

more, there must be no distinction in usage between system-defined and user-defined types.

The final five mandatory rules of the manifesto apply to the DBMS characteristic of the system.

(9) Data persistence must be provided

As in a conventional DBMS, data must remain (persist) after the application that created it has terminated. The user should not have to explicitly move or copy data to make it persistent.

(10) The DBMS must be capable of managing very large databases

In a conventional DBMS, there are mechanisms to manage secondary storage efficiently, such as indexes and buffers. An OODBMS should have similar mechanisms that are invisible to the user, thus providing a clear independence between the logical and phys- ical levels of the system.

(11) The DBMS must support concurrent users

An OODBMS should provide concurrency control mechanisms similar to those in con- ventional systems.

(12) The DBMS must be capable of recovery from hardware and software failures

An OODBMS should provide recovery mechanisms similar to those in conventional systems.

(13) The DBMS must provide a simple way of querying data

An OODBMS must provide an *ad hoc* query facility that is high-level (that is, reasonably declarative), efficient (suitable for query optimization), and application-independent. It is not necessary for the system to provide a query language; it could instead provide a graphical browser.

The manifesto proposes the following optional features: multiple inheritance, type check- ing and type inferencing, distribution across a network, design transactions, and versions. Interestingly, there is no direct mention of support for security, integrity, or views; even a fully declarative query language is not mandated.

Alternative Strategies for Developing an OODBMS

There are several approaches to developing an OODBMS, which can be summarized as follows (Khoshafian and Abnous, 1990):

- *Extend an existing object-oriented programming language with database capabilities* This approach adds traditional database capabilities to an existing

object-oriented programming language such as Smalltalk, C++, or Java (see Figure 26.1). This is the approach taken by the product GemStone, which extends these three languages.

- *Provide extensible object-oriented DBMS libraries* This approach also adds tradi- tional database capabilities to an existing object-oriented programming language. However, rather than extending the language, class libraries are provided that support persistence, aggregation, data types, transactions, concurrency, security, and so on. This is the approach taken by the products Ontos, Versant, and ObjectStore. We discuss ObjectStore in Section 27.3.

- *Embed object-oriented database language constructs in a conventional host language* In Appendix E we describe how SQL can be embedded in a conventional host pro- gramming language. This strategy uses the same idea of embedding an object-oriented database language in a host programming language. This is the approach taken by O_2, which provided embedded extensions for the programming language 'C'.

- *Extend an existing database language with object-oriented capabilities* Owing to the widespread acceptance of SQL, vendors are extending it to provide object-oriented constructs. This approach is being pursued by both RDBMS and OODBMS vendors. The 1999 release of the SQL standard, SQL:1999, supports object-oriented features. (We review these features in Section 28.4.) In addition, the Object Database Standard by the Object Data Management Group (ODMG) specifies a standard for Object SQL, which we discuss in Section 27.2.4. The products Ontos and Versant provide a version of Object SQL and many OODBMS vendors will comply with the ODMG standard.

- *Develop a novel database data model /data language* This is a radical approach that starts from the beginning and develops an entirely new database language and DBMS with object-oriented capabilities. This is the approach taken by SIM (Semantic Information Manager), which is based on the semantic data model and has a novel DML / DDL (Jagannathan *et al*., 1988).

S Perspectives

DBMSs are primarily concerned with the creation and maintenance of large, long-lived collections of data. As we have already seen from earlier chapters, modern DBMSs are characterized by their support of the following features:

- *A data model* A particular way of describing data, relationships between data, and constraints on the data.

- *Data persistence* The ability for data to outlive the execution of a program and possibly the lifetime of the program itself.

- *Data sharing* The ability for multiple applications (or instances of the same one) to access common data, possibly at the same time.

- *Reliability* The assurance that the data in the database is protected from hardware and software failures.

- *Scalability* The ability to operate on large amounts of data in simple ways.

- *Security and integrity* The protection of the data against unauthorized access, and the assurance that the data conforms to specified correctness and consistency rules.

- *Distribution* The ability to physically distribute a logically interrelated collection of shared data over a computer network, preferably making the distribution transparent to the user.

In contrast, traditional programming languages provide constructs for procedural con- trol and for data and functional abstraction, but lack built-in support for many of the above database features. While each is useful in its respective domain, there exists an increasing number of applications that require functionality from both DBMSs and programming

languages. Such applications are characterized by their need to store and retrieve large amounts of shared, structured data, as discussed in Section 25.1. Since 1980 there has been considerable effort expended in developing systems that integrate the concepts from these two domains. However, the two domains have slightly different perspectives that have to be considered and the differences addressed.

Perhaps two of the most important concerns from the programmers' perspective are per- formance and ease of use, both achieved by having a more seamless integration between the programming language and the DBMS than that provided with traditional DBMSs. With a traditional DBMS, we find that:

- It is the programmer's responsibility to decide when to read and update objects (records).

- The programmer has to write code to translate between the application's object model and the data model of the DBMS (for example, relations), which might be quite differ- ent. With an object-oriented programming language, where an object may be composed of many subobjects represented by pointers, the translation may be particularly com- plex. As noted above, it has been claimed that as much as 30% of programming effort and code space is devoted to this type of mapping. If this mapping process can be eliminated or at least reduced, the programmer would be freed from this responsibility, the resulting code would be easier to understand and maintain, and performance may increase as a result.

- It is the programmer's responsibility to perform additional type-checking when an object is read back from the database. For example, the programmer may create an object in the strongly typed object-oriented language Java and store it in a traditional DBMS. However, another application written in a different language may modify the object, with no guarantee that the object will conform to its original type.

These difficulties stem from the fact that conventional DBMSs have a two-level storage model: the application storage model in main or virtual memory, and the database storage model on disk, as illustrated in Figure 26.3. In contrast, an OODBMS tries to give the illu- sion of a single-level storage model, with a similar representation in both memory and in the database stored on disk, as illustrated in Figure 26.4.

Although the single-level memory model looks intuitively simple, the OODBMS has to cleverly manage the representations of objects in memory and on disk to achieve this illusion. As we discussed in Section 25.3 objects, and relationships between objects, are identified by object identifiers (OIDs). There are two types of OID:

- logical OIDs that are independent of the physical location of the object on disk;

- physical OIDs that encode the location.

In the former case, a level of indirection is required to look up the physical address of the object on disk. In both cases, however, an OID is different in size from a standard in-memory pointer that need only be large enough to address all virtual memory. Thus, to achieve the required performance, an OODBMS must be able to convert OIDs to and from in-memory pointers. This conversion technique has become known as **pointer swizzling** or **object faulting**, and the approaches used to implement it have become varied, ranging from software-based residency checks to page faulting schemes used by the underlying hardware (Moss and Eliot, 1990), as we now discuss.

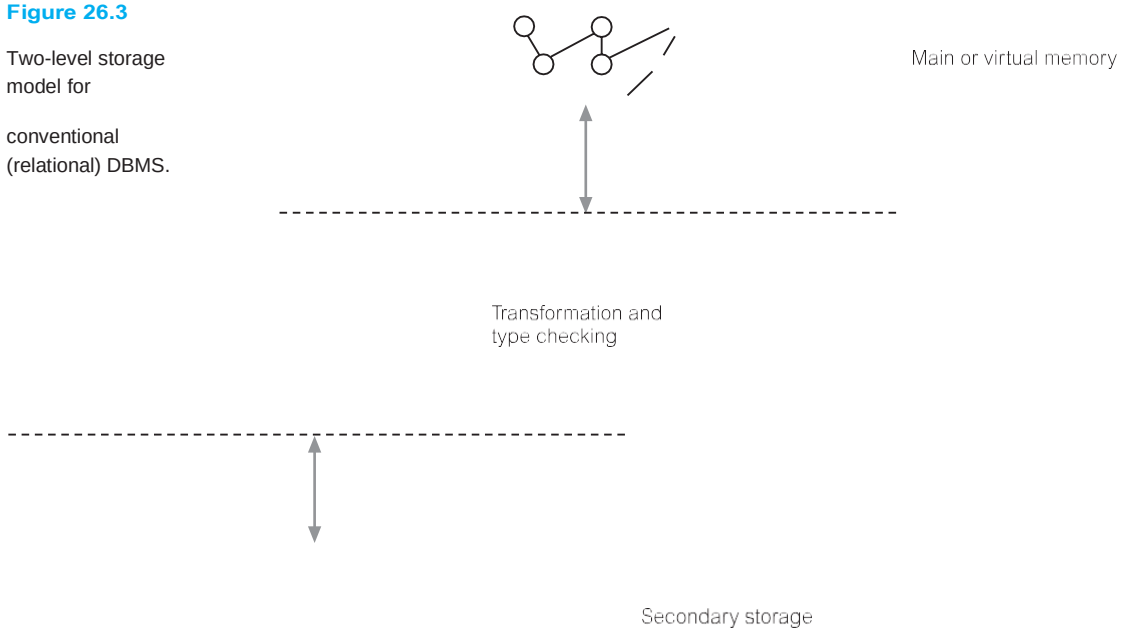


Figure 26.3

Two-level storage model for

conventional (relational) DBMS.

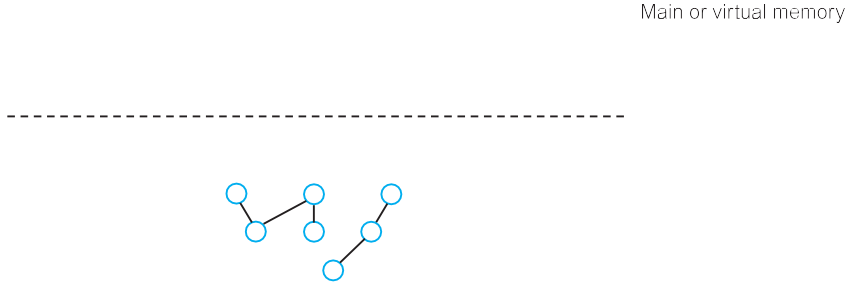


Figure 26.4 Single-level storage model for OODBMS.

26.2.1 Pointer Swizzling Techniques

Pointer swizzling	The action of converting object identifiers to main memory pointers, and back again

The aim of pointer swizzling is to optimize access to objects. As we have just mentioned, references between objects are normally represented using OIDs. If we read an object from secondary storage into the page cache, we should be able to locate any referenced objects on secondary storage using their OIDs. However, once the referenced objects have been read into the cache, we want to record that these objects are now held in main memory to prevent them being retrieved from secondary storage again. One approach is to hold a lookup table that maps OIDs to main memory pointers. We can implement the table lookup reasonably efficiently using hashing, but this is still slow compared to a pointer

dereference, particularly if the object is already in memory. However, pointer swizzling attempts to provide a more efficient strategy by storing the main memory pointers in place of the referenced OIDs and vice versa when the object has to be written back to disk.

In this section we describe some of the issues surrounding pointer swizzling, including the various techniques that can be employed.

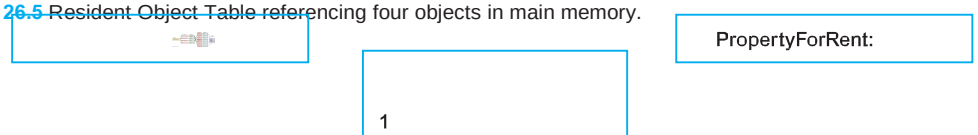
No swizzling

The easiest implementation of faulting objects into and out of memory is not to do any swizzling at all. In this case, objects are faulted into memory by the underlying object manager and a handle is passed back to the application containing the object's OID (White, 1994). The OID is used every time the object is accessed. This requires that the system maintain some type of lookup table so that the object's virtual memory pointer can be located and then used to access the object. As the lookup is required on each object access, this approach could be inefficient if the same object is accessed repeatedly. On the other hand, if an application tends only to access an object once, then this could be an acceptable approach.

Figure 26.5 shows the contents of the lookup table, sometimes called the **Resident Object Table** (**ROT**), after four objects have been read from secondary storage. If we now wish to access the Staff object with object identity OID5 from the Branch object OID1, a lookup of the ROT would indicate that the object was not in main memory and we would need to read the object from secondary storage and enter its memory address in the ROT table. On the other hand, if we try to access the Staff object with object identity OID4 from the Branch object, a lookup of the ROT would indicate that the object was already in main memory and provide its memory address.

Moss proposed an analytical model for evaluating the conditions under which swizzling is appropriate (1990). The results found suggest that if objects have a significant chance of being swapped out of main memory, or references are not followed at least several times

Figure 26.5 Resident Object Table referencing four objects in main memory.



on average, then an application would be better using efficient tables to map OIDs to object memory addresses (as in Objectivity/DB) rather than swizzling.

Object referencing

To be able to swizzle a persistent object's OID to a virtual memory pointer, a mechanism is required to distinguish between resident and non-resident objects. Most techniques are variations of either **edge marking** or **node marking** (Hoskings and Moss, 1993).

Considering virtual memory as a directed graph consisting of objects as nodes and references as directed edges, edge marking marks every object pointer with a tag bit. If the bit is set, then the reference is to a virtual memory pointer; otherwise, it is still pointing to an OID and needs to be swizzled when the object it refers to is faulted into the applica- tion's memory space. Node marking requires that all object references are immediately converted to virtual memory pointers when the object is faulted into memory. The first approach is a software-based technique but the second approach can be implemented using software- or hardware-based techniques.

In our previous example, the system replaces the value OID4 in the Branch object OID1 by its main memory address when Staff object OID4 is read into memory. This memory address provides a pointer that leads to the memory location of the Staff object identified by OID4. Thus, the traversal from Branch object OID1 to Staff object OID4 does not incur the cost of looking up an entry in the ROT, but consists now of a pointer dereference operation.

Hardware-based schemes

Hardware-based swizzling uses virtual memory access protection violations to detect accesses to non-resident objects (Lamb *et al.*, 1991). These schemes use the standard virtual memory hardware to trigger the transfer of persistent data from disk to main memory. Once a page has been faulted in, objects are accessed on that page via normal virtual memory pointers and no further object residency checking is required. The hardware approach has been used in several commercial and research systems including ObjectStore and Texas (Singhal *et al.*, 1992).

The main advantage of the hardware-based approach is that accessing memory-resident persistent objects is just as efficient as accessing transient objects because the hardware approach avoids the overhead of residency checks incurred by software approaches. A disadvantage of the hardware-based approach is that it makes the provision of many useful kinds of database functionality much more difficult, such as fine-grained locking, referential integrity, recovery, and flexible buffer management policies. In addition, the hardware approach limits the amount of data that can be accessed during a transaction to the size of

virtual memory. This limitation could be overcome by using some form of garbage collection to reclaim memory space, although this would add overhead and com- plexity to the system.

Classification of pointer swizzling

Pointer swizzling techniques can be classified according to the following three dimensions:

(1) Copy versus in-place swizzling.
(2) Eager versus lazy swizzling.
(3) Direct versus indirect swizzling.

Copy versus in-place swizzling

When faulting objects in, the data can either be copied into the application's local object cache or it can be accessed in place within the object manager's page cache (White, 1994). As discussed in Section 20.3.4, the unit of transfer from secondary storage to the cache is the page, typically consisting of many objects. Copy swizzling may be more efficient as, in the worst case, only modified objects have to be swizzled back to their OIDs, whereas an in-place technique may have to unswizzle an entire page of objects if one object on the page is modified. On the other hand, with the copy approach, every object must be explicitly copied into the local object cache, although this does allow the page of the cache to be reused.

Eager versus lazy swizzling

Moss and Eliot (1990) define eager swizzling as the swizzling of all OIDs for persistent objects on all data pages used by the application before any object can be accessed. This is rather extreme, whereas Kemper and Kossman (1993) provide a more relaxed definition, restricting the swizzling to all persistent OIDs within the object the application wishes to access. Lazy swizzling swizzles pointers only as they are accessed or discovered. Lazy swizzling involves less overhead when an object is faulted into memory, but it does mean that two different types of pointer must be handled for every object access: a swizzled pointer and an unswizzled pointer.

Direct versus indirect swizzling

This is an issue only when it is possible for a swizzled pointer to refer to an object that is no longer in virtual memory. With direct swizzling, the virtual memory pointer of the referenced object is placed directly in the swizzled pointer; with indirect swizzling, the virtual memory pointer is placed in an intermediate object, which acts as a placeholder for the actual object. Thus, with the indirect scheme objects can be uncached without requir- ing the swizzled pointers that reference the object to be unswizzled also.

These techniques can be combined to give eight possibilities (for example, in-place/ eager/direct, in-place/lazy/direct, or copy/ lazy/indirect).

Accessing an Object 26.2.2

How an object is accessed on secondary storage is another important aspect that can have a significant impact on OODBMS performance. Again, if we look at the approach taken

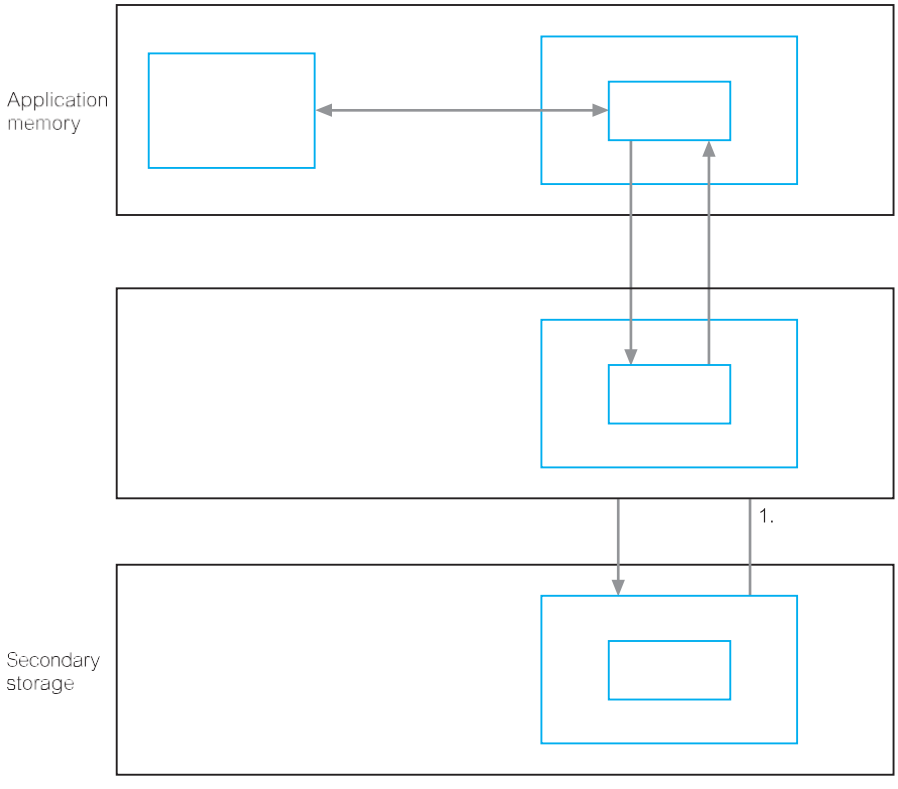


Figure 26.6
Steps in accessing a record using a conventional DBMS.

in a conventional relational DBMS with a two-level storage model, we find that the steps illustrated in Figure 26.6 are typical:

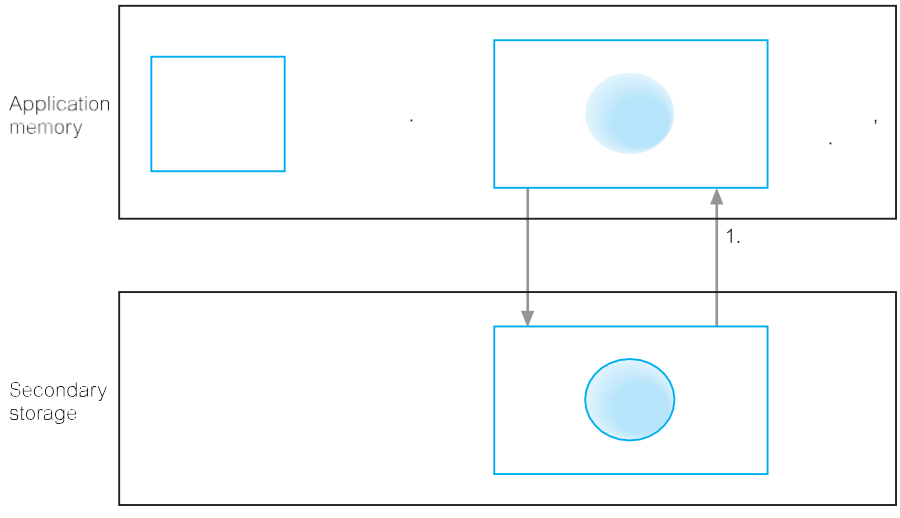
g indexes or table scans, as appropriate (see Section 21.4). The DBMS then reads that page from secondary storage and copies it into its cache.

- The DBMS subsequently transfers the required parts of the record from the cache into the application's memory space. Conversions may be necessary to convert the SQL data types into the application's data types.
- The application can then update the record's fields in its own memory space.
- The application transfers the modified fields back to the DBMS cache using SQL, again requiring conversions between data types.
- Finally, at an appropriate point the DBMS writes the updated page of the cache back to secondary storage.

In contrast, with a single-level storage model, an OODBMS uses the following steps to retrieve an object from secondary storage, as illustrated in Figure 26.7:

- The OODBMS determines the page on secondary storage that contains the required object using its OID or an index, as appropriate. The OODBMS then reads that

n The DBMS determines the page on secondary storage that contains the required record using



page from secondary storage and copies it into the application's page cache within its memory space.

n The OODBMS may then carry out a number of conversions, such as:

– swizzling references (pointers) between objects;
– adding some information to the object's data structure to make it conform to that required by the programming language;
– modifying the data representations for data that has come from a different hardware platform or programming language.

n The application can then directly access the object and update it, as required.

n When the application wishes to make the changes persistent, or when the OODBMS needs to swap the page out of the page cache, the OODBMS may need to carry out sim- ilar conversions as listed above, before copying the page back to secondary storage.

Persistence

A DBMS must provide support for the storage of **persistent** objects, that is, objects that survive after the user session or application program that created them has terminated. This is in contrast to **transient** objects that last only for the invocation of the program. Persistent objects are retained until they are no longer required, at which point they are deleted. Other than the embedded language approach discussed in Section 26.1.3, the schemes we present next may be used to provide persistence in programming languages. For a complete survey of persistence schemes, the interested reader is referred to Atkinson and Buneman (1989).

Although intuitively we might consider

persistence to be limited to the state of objects, persistence can also be applied to (object) code and to the program execution state. Including code in the persistent store potentially provides a more complete and elegant

Figure 26.7 Steps in accessing an object using an OODBMS.

solution. However, without a fully integrated development environment, making code persist leads to duplication, as the code will exist in the file system. Having program state and thread state persist is also attractive but, unlike code for which there is a standard definition of its format, program execution state is not easily generalized. In this section we limit our discussion to object persistence.

26.3.1 Persistence Schemes

In this section we briefly examine three schemes for implementing persistence within an OODBMS, namely checkpointing, serialization, and explicit paging.

Checkpointing

Some systems implement persistence by copying all or part of a program's address space to secondary storage. In cases where the complete address space is saved, the program can restart from the checkpoint. In other cases, only the contents of the program's heap are saved. Checkpointing has two main drawbacks: typically, a checkpoint can be used only by the program that created it; second, a checkpoint may contain a large amount of data that is of no use in subsequent executions.

Serialization

Some systems implement persistence by copying the closure of a data structure to disk. In this scheme, a write operation on a data value typically involves the traversal of the graph of objects reachable from the value, and the writing of a flattened version of the structure to disk. Reading back this flattened data structure produces a new copy of the original data structure. This process is sometimes called **serialization**, **pickling**, or in a distributed computing context, **marshaling**.

Serialization has two inherent problems. First, it does not preserve object identity, so that if two data structures that share a common substructure are separately serialized, then on retrieval the substructure will no longer be shared in the new copies. Further, serialization is not incremental, and so saving small changes to a large data structure is not efficient.

Explicit paging

Some persistence schemes involve the application programmer explicitly ‘paging’ objects between the application heap and the persistent store. As discussed above, this usually requires the conversion of object pointers from a disk-based scheme to a memory-based scheme. With the explicit paging mechanism, there are two common methods for creating/updating persistent objects: reachability-based and allocation-based.

Reachability-based persistence means that an object will persist if it is reachable from a persistent root object. This method has some advantages including the notion that the programmer does not need to decide at object creation time whether the object should be persistent. At any time after creation, an object can become persistent by adding it to the

reachability tree. Such a model maps well on to a language such as Smalltalk or Java that contains some form of garbage collection mechanism, which automatically deletes objects when they are no longer accessible from any other object.

Allocation-based persistence means that an object is made persistent only if it is ex- plicitly declared as such within the application program. This can be achieved in several ways, for example:

- *By class* A class is statically declared to be persistent and all instances of the class are made persistent when they are created. Alternatively, a class may be a subclass of a system-supplied persistent class. This is the approach taken by the products Ontos and Objectivity/DB.
- *By explicit call* An object may be specified as persistent when it is created or, in some cases, dynamically at runtime. This is the approach taken by the product ObjectStore. Alternatively, the object may be dynamically added to a persistent collection.

In the absence of pervasive garbage collection, an object will exist in the persistent store until it is explicitly deleted by the application. This potentially leads to storage leaks and dangling pointer problems.

With either of these approaches to persistence, the programmer needs to handle two different types of object pointer, which reduces the reliability and maintainability of the software. These problems can be avoided if the persistence mechanism is fully integrated with the application programming language, and it is this approach that we discuss next.

Orthogonal Persistence

An alternative mechanism for providing persistence in a programming language is known as **orthogonal persistence** (Atkinson *et al.*, 1983; Cockshott, 1983), which is based on the following three fundamental principles.

Persistence independence

The persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of a program is expressed independently of the per- sistence of data it manipulates. For example, it should be possible to call a function with its parameters sometimes objects with long-term persistence and at other times transient. Thus, the programmer does not need to (indeed cannot) program to control the movement of data between long- and short-term storage.

Data type

orthogonality

26.3.2

All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where an object is not allowed to be long-lived or is not allowed to be transient. In some persistent languages, persistence is a quality attributable to only a subset of the language data types. This approach is exemplified by Pascal/R, Amber, Avalon/C++, and E. The orthogonal approach has been adopted by a number of systems, including PS-algol, Napier88, Galileo, and GemStone (Connolly, 1997).

Transitive persistence

The choice of how to identify and provide persistent objects at the language level is independent of the choice of data types in the language. The technique that is now widely used for identification is reachability-based, as discussed in the previous section. This principle was originally referred to as persistence identification but the more suggestive ODMG term 'transitive persistence' is used here.

Advantages and disadvantages of orthogonal persistence

The uniform treatment of objects in a system based on the principle of orthogonal persistence is more convenient for both the programmer and the system:

- there is no need to define long-term data in a separate schema language;
- no special application code is required to access or update persistent data;
- there is no limit to the complexity of the data structures that can be made persistent.

Consequently, orthogonal persistence provides the following advantages:

- improved programmer productivity from simpler semantics;
- improved maintenance – persistence mechanisms are centralized, leaving programmers to concentrate on the provision of business functionality;
- consistent protection mechanisms over the whole environment;
- support for incremental evolution;
- automatic referential integrity.

However, there is some runtime expense in a system where every pointer reference might be addressing a persistent object, as the system is required to test whether the object must be loaded from secondary storage. Further, although orthogonal persistence pro- motes transparency, a system with support for sharing among concurrent processes cannot be fully transparent.

Although the principles of orthogonal persistence are desirable, many OODBMSs do not implement them completely. There are some areas that require careful consideration and we briefly discuss two here, namely queries and transactions.

What objects do queries apply to?

From a traditional DBMS perspective, declarative queries range over persistent objects, that is, objects that are stored in the database. However, with orthogonal persistence we should treat persistent and transient objects in the same way. Thus, queries should range over both persistent and transient objects. But what is the scope for transient objects? Should the scope be restricted to the transient objects in the current user's run unit or should it also include the run units of other concurrent users? In either case, for efficiency we may wish to maintain indexes on transient as well as persistent objects. This may require some form of query processing within the client process in addition to the tra- ditional query processing within the server.

What objects are part of transaction semantics?

From a traditional DBMS perspective, the ACID (Atomicity, Consistency, Isolation, and Durability) properties of a transaction apply to persistent objects (see Section 20.1.1). For example, whenever a transaction aborts, any updates that have been applied to persistent objects have to be undone. However, with orthogonal persistence we should treat per- sistent and transient objects in the same way. Thus, should the semantics of transactions apply also to transient objects? In our example, when we undo the updates to persistent objects should we also undo the changes to transient objects that have been made within the scope of the transaction? If this were the case, the OODBMS would have to log both the changes that are made to persistent objects and the changes that are made to transient objects. If a transient object were destroyed within a transaction, how would the OODBMS recreate this object within the user's run unit? There are a considerable number of issues that need to be addressed if transaction semantics range over both types of object. Unsurprisingly, few OODBMSs guarantee transaction consistency of transient objects.

Issues in OODBMSs

In Section 25.2 we mentioned three areas that are problematic for relational DBMSs, namely:

- long-duration transactions;
- versions;
- schema evolution.

In this section we discuss how these issues are addressed in OODBMSs. We also examine possible architectures for OODBMSs and briefly consider benchmarking.

Transactions

As discussed in Section 20.1, a transaction is a *logical unit of work*, which should always transform the database from one consistent state to another. The types of transaction found in business applications are typically of short duration. In contrast, transactions involving

26.4.1

complex objects, such as those found in engineering and design applications, can continue for several hours, or even several days. Clearly, to support **long-duration transactions** we need to use different protocols from those used for traditional database applications in which transactions are typically of a very short duration.

In an OODBMS, the unit of concurrency control and recovery is logically an object, although for performance reasons a more coarse granularity may be used. Locking-based protocols are the most common type of concurrency control mechanism used by OODBMSs to prevent conflict from occurring. However, it would be totally unacceptable for a user who initiated a long-duration transaction to find that the transaction has been aborted owing to a lock conflict and the work has been lost. Two of the solutions that have been proposed are:

- *Multiversion concurrency control protocols*, which we discussed in Section 20.2.6.
- *Advanced transaction models* such as nested transactions, sagas, and multilevel transactions, which we discussed in Section 20.4.

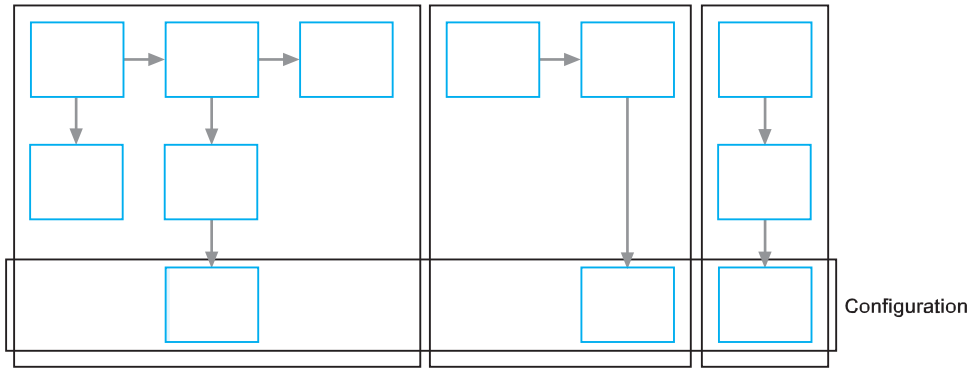


Figure 26.8 Versions and configurations.

26.4.2 Versions

There are many applications that need access to the previous state of an object. For example, the development of a particular design is often an experimental and incremental process, the scope of which changes with time. It is therefore necessary in databases that store designs to keep track of the evolution of design objects and the changes made to a design by various transactions (see for example, Atwood, 1985; Katz *et al.*, 1986; Banerjee *et al.*, 1987a).

The process of maintaining the evolution of objects is known as **version management**. An **object version** represents an identifiable state of an object; a

version history repres- ents the evolution of an object. Versioning should allow changes to the properties of objects to be managed in such a way that object references always point to the correct version of an object. Figure 26.8 illustrates version management for three objects: O_A, O_B, and O_C. For example, we can determine that object O_A consists of versions V_1, V_2, V_3; V_{1A} is derived from V_1, and V_{2A} and V_{2B} are derived from V_2. This figure also shows an example of a **configuration** of objects, consisting of V_{2B} of O_A, V_{2A} of O_B, and V_{1B} of O_C.

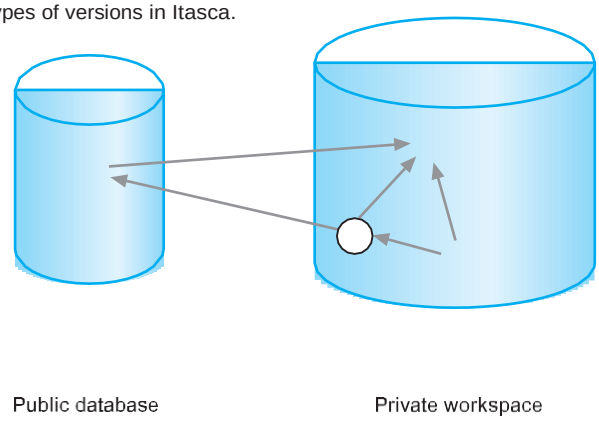
The commercial products Ontos, Versant, ObjectStore, Objectivity/DB, and Itasca provide some form of version management. Itasca identifies three types of version (Kim and Lochovsky, 1989):

- *Transient versions* A transient version is considered unstable and can be updated and deleted. It can be created from new by *checking out* a released version from a *public database* or by deriving it from a working or transient version in a *private database*. In the latter case, the base transient version is promoted to a working version. Transient versions are stored in the creator's private workspace.

- *Working versions* A working version is considered stable and cannot be updated, but it can be deleted by its creator. It is stored in the creator's private workspace.

- *Released versions* A released version is considered stable and cannot be updated or deleted. It is stored in a public database by *checking in* a working version from a private database.

Figure 26.9 Types of versions in Itasca.



These processes are illustrated in Figure 26.9. Owing to the performance and storage over- head in supporting versions, Itasca requires that the application indicate whether a class is **versionable**. When an instance of a versionable class is created, in addition to creating the first version of that instance a **generic object** for that instance is also created, which consists of version management information.

Schema Evolution

Design is an incremental process and evolves with time. To support this process, applica- tions require considerable flexibility in dynamically defining and modifying the database schema. For example, it should be possible to modify class definitions, the inheritance structure, and the specifications of attributes and methods without requiring system shutdown. Schema modification is closely related to the concept of version management discussed above. The issues that arise in schema evolution are complex and not all of them have been investigated in sufficient depth. Typical changes to the schema include (Banerjee *et al.*, 1987b):

(1) Changes to the class definition:

(a) modifying attributes;
(b) modifying methods.

(2) Changes to the inheritance hierarchy:

(a) making a class S the superclass of a class C;
(b) removing a class S from the list of superclasses of C;
(c) modifying the order of the superclasses of C.

(3) Changes to the set of classes, such as creating and deleting classes and modifying class names.

The changes proposed to a schema must not leave the schema in an inconsistent state. Itasca and GemStone define rules for schema consistency, called **schema invariants**, which must be complied with as the schema is modified. By way of

26.4.3

an example, we con- sider the schema shown in Figure 26.10. In this figure, inherited attributes and methods are represented by a rectangle. For example, in the Staff class the attributes name and DOB

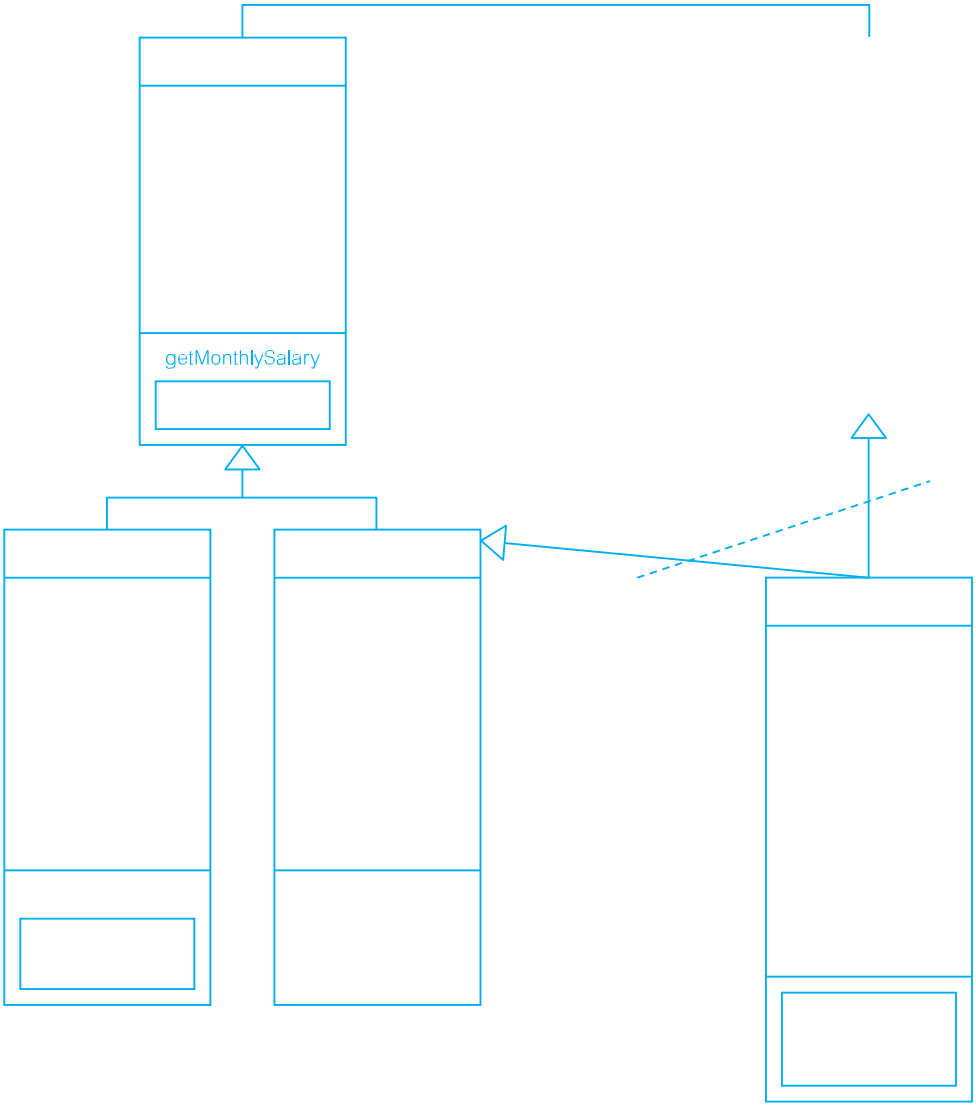


Figure 26.10 Example schema with both single and multiple inheritance.

and the method getAge have been inherited from Person. The rules can be divided into four groups with the following responsibilities:

(1) The resolution of conflicts caused by multiple inheritance and the redefinition of attributes and methods in a subclass.

Rule of precedence of subclasses over superclasses

If an attribute/method of one class is defined with the same name as an attribute/ method of a superclass, the definition specified in the subclass takes precedence over the definition of the superclass.

Rule of precedence between superclasses of a different origin

If several superclasses have attributes/methods with the same name but with a different origin, the attribute/method of the first superclass is inherited by the subclass. For example, consider the subclass SalesStaffClient in Figure 26.10, which inherits from SalesStaff and Client. Both these superclasses have an attribute telNo, which is not inherited from a common superclass (which in this case is Person). In this instance, the definition of the telNo attribute in SalesStaffClient is inherited from the first superclass, namely SalesStaff.

Rule of precedence between superclasses of the same origin

If several superclasses have attributes/methods with the same name and the same origin, the attribute/method is inherited only once. If the domain of the attribute has been redefined in any superclass, the attribute with the most specialized domain is inherited by the subclass. If domains cannot be compared, the attribute is inherited from the first superclass. For example, SalesStaffClient inherits name and DOB from both SalesStaff and Client; however, as these attributes are themselves inherited ultimately from Person, they are inherited only once by SalesStaffClient.

(2) The propagation of modifications to subclasses.

Rule for propagation of modifications

Modifications to an attribute/method in a class are always inherited by subclasses, except by those subclasses in which the attribute/method has been redefined. For example, if we deleted the method getAge from Person, this change would be reflected in all subclasses in the entire schema. Note that we could not delete the method getAge directly from a subclass as it is defined in the superclass Person. As another example, if we deleted the method getMonthSalary from Staff, this change would also ripple to Manager, but it would not affect SalesStaff as the method has been redefined in this subclass. If we deleted the attribute telNo from SalesStaff, this version of the attribute telNo would also be deleted from SalesStaffClient but SalesStaffClient would then inherit telNo from Client (see rule 1.2 above).

Rule for propagation of modifications in the event of conflicts

The introduction of a new attribute/method or the modification of the name of an attribute/method is propagated only to those subclasses for which there would be no resulting name conflict.

Rule for modification of domains

The domain of an attribute can only be modified using generalization. The domain of an inherited attribute cannot be made more general than the domain of the original attribute in the superclass.

(3) The aggregation and deletion of inheritance relationships between classes and the creation and removal of classes.

Rule for inserting superclasses

If a class C is added to the list of superclasses of a class C_s, C becomes the last of the superclasses of C_s. Any resulting inheritance conflict is resolved by rules 1.1, 1.2, and 1.3.

Rule for removing superclasses

If a class C has a single superclass C_s, and C_s is deleted from the list of superclasses of C, then C becomes a direct subclass of each direct superclass of C_s. The order- ing of the new superclasses of C is the same as that of the superclasses of C_s. For example, if we were to delete the superclass Staff, the subclasses Manager and SalesStaff would then become direct subclasses of Person.

Rule for inserting a class into a schema

If C has no specified superclass, C becomes the subclass of OBJECT (the root of the entire schema).

Rule for removing a class from a schema

To delete a class C from a schema, rule 3.2 is applied successively to remove C

from the list of superclasses of all its subclasses. OBJECT cannot be deleted.

(4) Handling of composite objects.

The fourth group relates to those data models that support the concept of composite objects. This group has one rule, which is based on different types of composite object. We omit the detail of this rule and refer the interested reader to the papers by Banerjee *et al.* (1987b) and Kim *et al.* (1989).

Architecture

In this section we discuss two architectural issues: how best to apply the client–server architecture to the OODBMS environment, and the storage of methods.

Client–server

Many commercial OODBMSs are based on the client–server architecture to provide data to users, applications, and tools in a distributed environment (see Section 2.6). However, not all systems use the same client–server model. We can distinguish three basic architec- tures for a client–server DBMS that vary in the functionality assigned to each component (Loomis, 1992), as depicted in Figure 26.11:

- *Object server* This approach attempts to distribute the processing between the two components. Typically, the server process is responsible for managing storage, locks, commits to secondary storage, logging and recovery, enforcing security and integrity, query optimization, and executing stored procedures. The client is responsible for transaction management and interfacing to the programming language. This is the best architecture for cooperative, object-to-object processing in an open, distributed environment.

- *Page server* In this approach, most of the database processing is performed by the client. The server is responsible for secondary storage and providing pages at the client's request.

- *Database server* In this approach, most of the database processing is performed by the server. The client simply passes requests to the server, receives results, and passes them on to the application. This is the approach taken by many RDBMSs.

In each case, the server resides on the same machine as the physical database; the client may reside on the same or different machine. If the client needs access to databases distributed across multiple machines, then the client communicates with a server on each machine. There may also be a number of clients communicating with one server, for example, one client for each user or application.

Storing and executing methods

There are two approaches to handling methods: store the methods in external files, as shown in Figure 26.12(a), and store the methods in the database, as shown in Figure 26.12(b). The first approach is similar to function libraries or Application Pro- gramming Interfaces (APIs) found in traditional DBMSs, in which an application program interacts with a DBMS by linking in functions supplied by the DBMS vendor. With the second approach, methods are stored in the database and are dynamically bound to the application at runtime. The second approach offers several benefits:

- *It eliminates redundant code* Instead of placing a copy of a method that accesses a data element in every program that deals with that data, the method is stored only once in the database.

- *It simplifies modifications* Changing a method requires changing it in one place only. All the programs automatically use the updated method. Depending on the nature of the change, rebuilding, testing, and redistribution of programs may be eliminated.

Fi

Methods are more secure Storing the methods in the database gives them all the benefits of security provided automatically by the OODBMS.

- *Methods can be shared concurrently* Again, concurrent access is provided automatically by the OODBMS. This also prevents multiple users making different changes to a method simultaneously.
- *Improved integrity* Storing the methods in the database means that integrity constraints can be enforced consistently by the OODBMS across all applications.

The products GemStone and Itasca allow methods to be stored and activated from within the database.

Benchmarking

Over the years, various database benchmarks have been developed as a tool for comparing the performance of DBMSs and are frequently referred to in academic, technical, and commercial literature. Before we examine two object-oriented benchmarks, we first pro- vide some background to the discussion. Complete descriptions of these benchmarks are outwith the scope of this book but for full details of the benchmarks the interested reader is referred to Gray (1993).

Wisconsin benchmark

Perhaps the earliest DBMS benchmark was the Wisconsin benchmark, which was developed to allow comparison of particular DBMS features (Bitton *et al.*, 1983). It consists of a set of tests as a single user covering:

- updates and deletes involving both key and non-key attributes;
- projections involving different degrees of duplication in the attributes and selections with different selectivities on indexed, non-index, and clustered attributes;
- joins with different selectivities;
- aggregate functions.

The original Wisconsin benchmark was based on three relations: one relation called Onektup with 1000 tuples, and two others called Tenktup1/ Tenktup2 with 10,000 tuples each. This benchmark has been generally useful although it does not cater for highly skewed attribute distributions and the join queries used are relatively simplistic.

Owing to the importance of accurate benchmarking information, a consortium of manu- facturers formed the **Transaction Processing Council** (TPC) in 1988 to formulate a series of transaction-based test suites to measure database/ TP environments. Each consists of a printed specification and is accompanied by ANSI 'C' source code, which populates a database with data according to a preset standardized structure.

TPC-A and TPC-B benchmarks

TPC-A and TPC-B are based on a simple banking transaction. TPC-A measures online transaction processing (OLTP) performance covering the time taken by the database server, network, and any other components of the system but excluding user interaction. TPC-B measures only the performance of the database server. A transaction simulates the transfer of money to or from an account with the following actions:

- update the account record (Account relation has 100,000 tuples);
- update the teller record (Teller relation has 10 tuples);
- update the branch record (Branch relation has 1 tuple);
- update a history record (History relation has 2,592,000 tuples);
- return the account balance.

The cardinalities quoted above are for a minimal configuration but the database can be scaled in multiples of this configuration. As these actions

are performed on single tuples, important aspects of the system are not measured (for example, query planning and join execution).

TPC-C benchmark

TPC-A and TPC-B are obsolescent and are being replaced by TPC-C, which is based on an order entry application. The underlying database schema and the range of queries are more complex than TPC-A, thereby providing a much more comprehensive test of a DBMS's performance. There are five transactions defined covering a new order, a pay- ment, an order status inquiry, a delivery, and a stock level inquiry.

Other benchmarks

The Transaction Processing Council has defined a number of other benchmarks, such as:

- TPC-H, for *ad hoc*, decision support environments where users do not know which queries will be executed;
- TPC-R, for business reporting within decision support environments where users run a standard set of queries against a database system;
- TPC-W, a transactional Web benchmark for e-Commerce, where the workload is performed in a controlled Internet commerce environment that simulates the activities of a business-oriented transactional Web server.

The Transaction Processing Council publishes the results of the benchmarks on its Web site (www.tpc.org).

OO1 benchmark

The Object Operations Version 1 (OO1) benchmark is intended as a generic measure of OODBMS performance (Cattell and Skeen, 1992). It was designed to reproduce operations that are common in the advanced engineering applications discussed in Section 25.1, such as finding all parts connected to a random part, all parts connected to one of those parts, and so on, to a depth of seven levels. The benchmark involves:

- random retrieval of 1000 parts based on the primary key (the part number);
- random insertion of 100 new parts and 300 randomly selected connections to these new parts, committed as one transaction;
- random parts explosion up to seven levels deep, retrieving up to 3280 parts.

In 1989 and 1990, the OO1 benchmark was run on the OODBMSs GemStone, Ontos, ObjectStore, Objectivity/ DB, and Versant, and the RDBMSs INGRES and Sybase. The results showed an average 30-fold performance improvement for the OODBMSs over the RDBMSs. The main criticism of this benchmark is that objects are connected in such a way as to prevent clustering (the closure of any object is the entire database). Thus, systems that have good navigational access at the expense of any other operations perform well against this benchmark.

OO7 benchmark

In 1993, the University of Wisconsin released the OO7 benchmark, based on a more comprehensive set of tests and a more complex database. OO7 was designed for detailed comparisons of OODBMS products (Carey *et al.*, 1993). It simulates a CAD/CAM envir- onment and tests system performance in the area of object-to-object

navigation over cached data, disk-resident data, and both sparse and dense traversals. It also tests indexed and non-indexed updates of objects, repeated updates, and the creation and deletion of objects.

The OO7 database schema is based on a complex parts hierarchy, where each part has associated documentation, and modules (objects at the top level of the hierarchy) have a manual. The tests are split into two groups. The first group is designed to test:

- traversal speed (simple test of navigational performance similar to that measured in OO1);

- traversal with updates (similar to the first test, but with updates covering every atomic part visited, a part in every composite part, every part in a composite part four times);
- operations on the documentation.

The second group contains declarative queries covering exact match, range searches, path lookup, scan, a simulation of the make utility, and join. To facilitate its use, a number of sample implementations are available via anonymous ftp from ftp.cs.wisc.edu.

Advantages and Disadvantages of OODBMSs

OODBMSs can provide appropriate solutions for many types of advanced database applications. However, there are also disadvantages. In this section we examine these advantages and disadvantages.

Advantages

The advantages of OODBMSs are listed in Table 26.2.

Table 26.2 Advantages of OODBMSs.

Enriched modeling capabilities

The object-oriented data model allows the 'real world' to be modeled more closely. The object, which encapsulates both state and behavior, is a more natural and realistic repres- entation of real-world objects. An object can store all the relationships it has with other objects, including many-to-many relationships, and objects can be formed into complex objects that the traditional data models cannot cope with easily.

Extensibility

OODBMSs allow new data types to be built from existing types. The ability to factor out common properties of several classes and form them into a superclass that can be shared

Enriched modeling

eling capabilities Extensibility

26.5.1

Removal of impedance mismatch More expressive query language Support for schema evolution

Support for long-duration transactions Applicability to advanced database applications Improved performance

with subclasses can greatly reduce redundancy within systems and, as we stated at the start of this chapter, is regarded as one of the main advantages of object orientation. Overriding is an important feature of inheritance as it allows special cases to be handled easily, with minimal impact on the rest of the system. Further, the reusability of classes promotes faster development and easier maintenance of the database and its applications.

It is worthwhile pointing out that if domains were properly implemented, RDBMSs would be able to provide the same functionality as OODBMSs are claimed to have. A domain can be perceived as a data type of arbitrary complexity with scalar values that are encapsulated, and that can be operated on only by predefined functions. Therefore, an attribute defined on a domain in the relational model can contain anything, for example, drawings, documents, images, arrays, and so on (Date, 2000). In this respect, domains and object classes are arguably the same thing. We return to this point in Section 28.2.2.

Removal of impedance mismatch

A single language interface between the Data Manipulation Language (DML) and the programming language overcomes the impedance mismatch. This eliminates many of the inefficiencies that occur in mapping a declarative language such as SQL to an imperative language such as 'C'. We also find that most OODBMSs provide a DML that is com- putationally complete compared with SQL, the standard language for RDBMSs.

More expressive query language

Navigational access from one object to the next is the most common form of data access in an OODBMS. This is in contrast to the associative access of SQL (that is, declarative statements with selection based on one or more predicates). Navigational access is more suitable for handling parts explosion, recursive queries, and so on. However, it is argued that most OODBMSs are tied to a particular programming language that, although con- venient for programmers, is not generally usable by end-users who require a declarative language. In recognition of this, the ODMG standard specifies a declarative query lan- guage based on an object-oriented form of SQL (see Section 27.2.4).

Support for schema evolution

The tight coupling between data and applications in an OODBMS makes schema evolu- tion more feasible. Generalization and inheritance allow the schema to be better structured, to be more intuitive, and to capture more of the semantics of the application.

Support for long-duration transactions

Current relational DBMSs enforce serializability on concurrent transactions to maintain database consistency (see Section 20.2.2). Some OODBMSs use a different protocol to handle the types of long-duration transaction that are common in many advanced database applications. This is an arguable advantage: as we have already mentioned in Section 25.2, there is no structural reason why such transactions cannot be provided by an RDBMS.

Applicability to advanced database applications

As we discussed in Section 25.1, there are many areas where traditional DBMSs have not been particularly successful, such as, computer-aided design (CAD), computer-aided soft- ware engineering (CASE), office information systems (OISs), and multimedia systems. The enriched modeling capabilities of OODBMSs have made them suitable for these applications.

Improved performance

As we mentioned in Section 26.4.5, there have been a number of benchmarks that have suggested OODBMSs provide significant performance improvements over relational DBMSs. For example, in 1989 and 1990, the OO1 benchmark was run on the OODBMSs GemStone, Ontos, ObjectStore, Objectivity/DB, and Versant, and the RDBMSs INGRES and Sybase. The results showed an average 30-fold performance improvement for the OODBMS over the RDBMS, although it has been argued that this difference in per- formance can be attributed to architecture-based differences, as opposed to model-based differences. However, dynamic binding and garbage collection in OODBMSs may com- promise this performance improvement.

It has also been argued that these benchmarks target engineering applications, which are more suited to object-oriented systems. In contrast, it has been suggested that RDBMSs outperform OODBMSs with traditional database applications, such as online transaction processing (OLTP).

Disadvantages

The disadvantages of OODBMSs are listed in Table 26.3.

Table 26

Lack of universal data model

As we discussed in Section 26.1, there is no universally agreed data model for an OODBMS, and most models lack a theoretical foundation. This disadvantage is seen as a

.3 Disadvantages of OODBMSs. Lack of universal data model 26.5.2

Lack of experience Lack of standards
Competition

Query optimization compromises encapsulation Locking at object level may impact performance Complexity

Lack of support for views Lack of support for security

significant drawback and is comparable to pre-relational systems. However, the ODMG proposed an object model that has become the *de facto* standard for OODBMSs. We dis- cuss the ODMG object model in Section 27.2.

Lack of experience

In comparison to RDBMSs, the use of OODBMSs is still relatively limited. This means that we do not yet have the level of experience that we have with traditional systems. OODBMSs are still very much geared towards the programmer, rather than the naïve end-user. Furthermore, the learning curve for the design and management of OODBMSs may be steep, resulting in resistance to the acceptance of the technology. While the OODBMS is limited to a small niche market, this problem will continue to exist.

Lack of standards

There is a general lack of standards for OODBMSs. We have already mentioned that there is no universally agreed data model. Similarly, there is no standard object-oriented query language. Again, the ODMG specified an Object Query Language (OQL) that has become a *de facto* standard, at least in the short term (see Section 27.2.4). This lack of standards may be the single most damaging factor for the adoption of OODBMSs.

Competition

Perhaps one of the most significant issues that face OODBMS vendors is the competition posed by the RDBMS and the emerging ORDBMS products. These products have an established user base with significant experience available, SQL is an approved standard and ODBC is a *de facto* standard, the relational data model has a solid theoretical foundation, and relational products have many supporting tools to help both end-users and developers.

Query optimization compromises encapsulation

Query optimization requires an understanding of the underlying implementation to access the database efficiently. However, this compromises the concept of encapsulation. The OODBMS Manifesto, discussed in Section 26.1.4, suggests that

this may be acceptable although, as we discussed, this seems questionable.

Locking at object level may impact performance

Many OODBMSs use locking as the basis for a concurrency control protocol. However, if locking is applied at the object level, locking of an inheritance hierarchy may be problematic, as well as impacting performance. We examined how to lock hierarchies in Section 20.2.8.

Complexity

The increased functionality provided by an OODBMS, such as the illusion of a single-level storage model, pointer swizzling, long-duration transactions, version management, and

schema evolution, is inherently more complex than that of traditional DBMSs. In general, complexity leads to products that are more expensive to buy and more difficult to use.

Lack of support for views

Currently, most OODBMSs do not provide a view mechanism, which, as we have seen previously, provides many advantages such as data independence, security, reduced com- plexity, and customization (see Section 6.4).

Lack of support for security

Currently, OODBMSs do not provide adequate security mechanisms. Most mechanisms are based on a coarse granularity, and the user cannot grant access rights on individual objects or classes. If OODBMSs are to expand fully into the business field, this deficiency must be rectified.

Review Questions

26.1 Compare and contrast the different definitions of object-oriented data models.

26.3 Describe the main modeling component of the functional data model.

26.4 What is a persistent programming language and how does it differ from an OODBMS? Discuss the difference between the two-level storage model used by conventional DBMSs and the single-level storage model used by OODBMSs.

26.5

26.6 How does this single-level storage model

26.7 What is pointer swizzling? Describe the different approaches to pointer swizzling.

26.8

26.9 Describe the types of transaction protocol that can be useful in design applications.

Discuss why schema control may be a useful facility for some applications.

Describe the different architectures for an OODBMS

Exercises

You have been asked by the Managing Director of *DreamHome* to investigate and prepare a report on the applicability of an OODBMS for the organization. The report should compare the technology of the RDBMS with that of the OODBMS, and should address the advantages and disadvantages of implementing an OODBMS within the organization, and any perceived problem areas. Finally, the report should contain a fully justified set of conclusions on the applicability of the OODBMS for *DreamHome*.

For the relational Hotel schema in the Exercises at the end of Chapter 3, suggest a number of methods that may be applicable to the system. Produce an object-oriented schema for the system.

Produce an object-oriented database design for the *DreamHome* case study documented in Appendix A. State any assumptions necessary to support your design.

Produce an object-oriented database design for the *University Accommodation Office* case study presented in Appendix B.1. State any assumptions necessary to support your design.

Produce an object-oriented database design for the *EasyDrive School of Motoring* case study presented in Appendix B.2. State any assumptions necessary to support your design.

Produce an object-oriented database design for the *Wellmeadows Hospital* case study presented in Appendix B.3. State any assumptions necessary to support your design.

Repeat Exercises 26.14 to 26.18 but produce a schema using the functional data model. Diagrammatically illustrate each schema.

Using the rules for schema consistency given in Section 26.4.3 and the sample schema given in Figure 26.10, consider each of the following modifications and state what the effect of the change should be to the schema:

(a) adding an attribute to a class;
(b) deleting an attribute from a class;
(c) renaming an attribute;
(d) making a class *S* a superclass of a class *C*;
(e) removing a class *S* from the list of superclasses of a class *C*; (f) creating a new class *C*;
(g) deleting a class;
(h) modifying class names.

UNIT-V

OMG(OBJECT MANAGEMENT GROUP):

The Object Management Group (OMG) is the world's largest software consortium with an international membership of vendors, developers, and end users. Established in 1989, its mission is to help computer users solve enterprise integration problems by supplying open, vendor neutral portability, interoperability and reusability specifications based on Model Driven Architecture (MDA). MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform, and provides a set of guidelines for structuring specifications expressed as models. OMG has established numerous widely used

standards such as OMG IDL, CORBA, Realtime CORBA, GIOP/IIOP, UML, MOF, XMI and CWM to name a few significant ones.

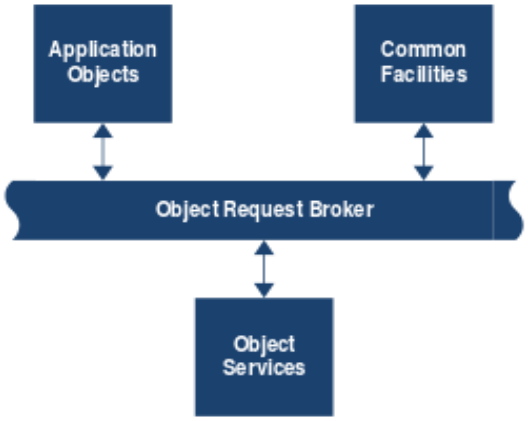


Fig. 1. The object management architecture.

Object Request Broker, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.

Object Services, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating,deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in CORBAservices: Common Object Services Specification.

Common Facilities, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.Information about Common Facilities will be contained in CORBAfacilities.

Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the

Reference Model.

OBJECT REQUEST BROKER ARCHITECTURE:

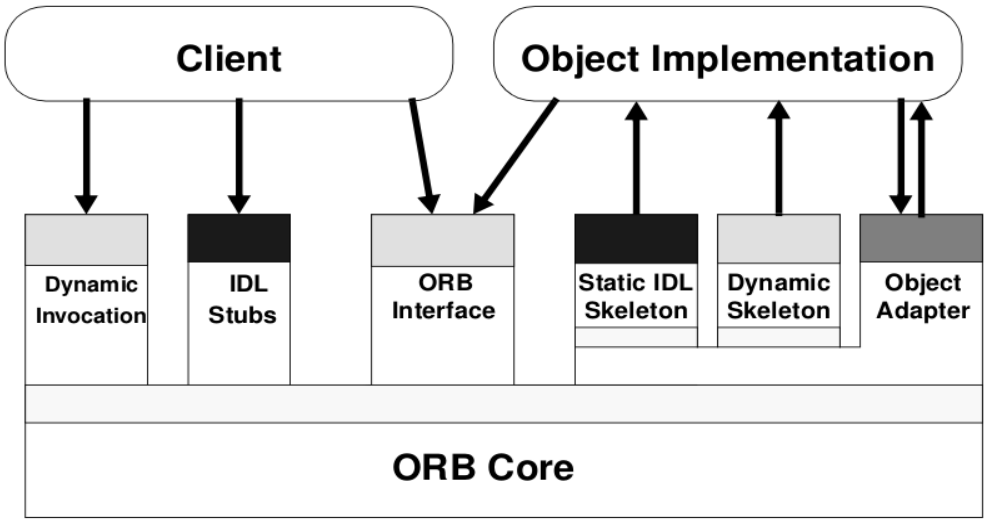


Fig:The Structure of Object Request Interfaces

Dynamic Invocation Interface:

An interface is also available that allows the dynamic construction of object invocations, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call orsequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed.

Object Adapters:

An object adapter is the primary way that an object implementation accesses services provided by the ORB.Services provided by the ORB through an Object Adapter often include: generationand interpretation of object references, method invocation, security of interactions,object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.

The ORB Interface :

It is the interface that goes directly to the ORB which is the same for all ORBs and does not depend on the object's interface or object adapter. Because most of the functionality of the ORB is provided through the object adapter, stubs, skeleton, or dynamic invocation, there are only a few operations that are common across all objects. These operations are useful to both clients and implementations of objects.

Client Stubs:

For the mapping of a non–object–oriented language, there will be a programming interface to the stubs for each interface type.The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.Object-oriented programming languages, such as C++ and Smalltalk, do not require stub interfaces.

Implementation Skeleton:

For a particular language mapping, and possibly depending on the object adapter, there

will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface, in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

Model-driven architecture (MDA):

It is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as **models**.

The Model Driven Architecture (MDA) is an open, vendor-neutral approach to interoperability using OMG's modeling specifications: Unified Modeling Language (UML),Meta-Object Facility (MOF), and Common Warehouse Meta-model (CWM).

->The Unified Modeling Language™ (UML™) helps you specify, visualize, and document models of software systems. Using any one of the large number of UML-based tools on the market, you can analyze your future application's requirements and design a solution that meets them.

->The Meta-Object Facility (MOF) is a set of standard interfaces that can be used to define and manipulate a set of interoperable meta-models and their corresponding models.MOF Model is referred to as a meta-metamodel because it is being used to define metamodels such as the UML.

->The Common Warehouse metamodel (CWM) specifies interfaces that can be used to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments. CWM is based on three standards:

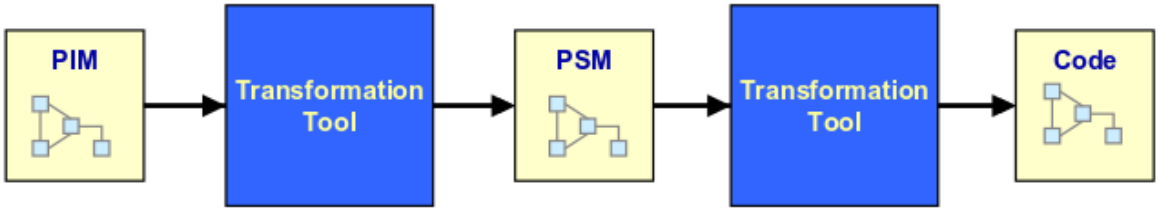
- UML - Unified Modeling Language, an OMG modeling standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard
- XMI - XML Metadata Interchange, an OMG metadata interchange standard

To enable MDA we need

->Modeling languages

->Transformation definitions

->Tools



Important model types:

PIM : Platform Independent Model

PSM : Platform Specific Model

Code

ODMG(Object Data Management Group):

The Object Data Management Group (ODMG) was in 1991 with object databasevendors that was organized by Rick Cattell of Sun Microsystems. The primary goal of the ODMG was to put forward a set of specifications that allowed a developer to write portable applications for object database and object-relational mapping products. In order to do that, the data schema, programming language bindings, and data manipulation and query languagesneeded to be portable.

Major components of the ODMG 3.0 specification:

*Object Model:*This was based on the Object Management Group's Object Model. The OMG core model was designed to be a common denominator for object request brokers, object database systems, object programming languages, etc. The ODMG designed a profile by adding components to the OMG core object model.

- *Object Specification Languages:*The ODMG Object Definition Language (ODL) was used to define the object types that conform to the ODMG Object Model. The ODMG Object Interchange Format (OIF) was used to dump and load the current state to or from a file or set of files.
- *Object Query Language (OQL):*The ODMG OQL was a declarative (nonprocedural) language for query and updating. It used SQL as a basis, where possible, though OQL supports more powerful object-oriented capabilities.
- *C++ Language Binding.* This defined a C++ binding of the ODMG ODL and a

C++ Object Manipulation Language (OML). The C++ ODL was expressed as a library that provides classes and functions to implement the concepts defined in the ODMG Object Model. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library. The C++ binding also provided a mechanism to invoke OQL.

Object Model:

the Object Model supported by ODMG-compliant object data management systems (ODMSs). The Object Model is important because it specifies the kinds of semantics that can be defined explicitly to an ODMS.

The Object Model specifies the constructs that are supported by an ODMS:

- The basic modeling primitives are the object and the literal.
- Each object has a unique identifier. A literal has no identifier.
- Objects and literals can be categorized by their types.
- All elements of a given type have a common range of states (i.e., the same set of properties) and common behavior (i.e., the same set of defined operations). An object is sometimes referred to as an instanceof its type.
- The state of an object is defined by the values it carries for a set of properties
- These properties can be attributes of the object itself or relationships
- between the object and one or more other objects. Typically, the values of an
- object's properties can change over time.
- The behavior of an object is defined by the set of operations that can be executed on or by the object. Operations may have a list of input and output parameters, each with a specified type.
- Each operation may also return a typed result.
- An ODMS stores objects, enabling them to be shared by multiple users and applications.
- An ODMS is based on a schema that is defined in ODL and contains instances of the types defined by its schema.

Atomic Objects:

An atomic object type is user-defined**.** There are no built-in atomic object types included in the ODMG Object Model.

Collection Objects :

In the ODMG Object Model, instances of collection objects are composed of distinct elements, each of which can be an instance of an atomic type, another collection, or a literal type.

The collections supported by the ODMG Object Model include

•Set<t>

•Bag<t>

•List<t>

•Array<t>

•Dictionary<t,v>

Extents:

The extent of a type is the set of all instances of the type within a particular ODMS

Keys :

The individual instances of a type can be uniquely identified by the values they carry for some property or set of properties. These identifying properties are called keys.

Objects:

This section considers each of the following aspects of objects:

•Creation, which refers to the manner in which objects are created by the

programmer.

•Identifiers, which are used by an ODMS to distinguish one object from another and to find objects.

•Names, which are designated by programmers or end users as convenient

ways to refer to particular objects.

•Lifetimes, which determine how the memory and storage allocated to objects

are managed.

•Structure, which can be either atomic or not, in which case the object is com-

posed of other objects.

All of the object definitions, defined in this chapter, are to be grouped into an enclosing module that defines a name scope for the types of the model

a)Object Creation :

Objects are created by invoking creation operations on factory interfacesprovided on factory objects supplied to the programmer by the language binding implementation.

The new operation, defined below, causes the creation of a new instance of an object of the Objecttype.

```
interface ObjectFactory {
Object new();
};
```

All objects have the following ODL interface, which is implicitly inherited by the defi-nitions of all user-defined objects:

```
interface Object {

enum Lock_Type{read, write, upgrade};

void lock(in Lock_Type mode) raises(LockNotGranted);

boolean try_lock(in Lock_Type mode);

boolean same_as(in Object anObject);

Object copy();

void delete();

};
```

b)Object Identifiers:

All objects have identifiers, an object can always be distinguished from all other objects within its storage domain

- All identifiers of objects in an ODMS are unique, relative to each other.
- An object retains the same object identifier for its entire lifetime.
- Object identifiers are generated by the ODMS, not by applications.
- Literals do not have their own identifiers and cannot stand alone as objects; they are embedded in objects and cannot be individually referenced.

c) Object Names:

An object may be given one or more names that are meaningful to the programmer or end user. The ODMS provides a function that it uses to map from an object name to an object.

d) Object Lifetimes:

The lifetime of an object determines how the memory and storage allocated to the object are managed. The lifetime of an object is specified at the time the object is created.

Two lifetimes are supportedin the Object Model:

- •transient
- •persistent

An object whose lifetime is transient is allocated memory that is managed by the programming language runtime system.

An object whose lifetime is persistent is allocated memory and storage managed by the ODMS runtime system.

Collections are created by invoking the operations on the factory interfaces defined for each particular collection.

Collections all have the following operations:

```
interface Collection : Object {
exception          InvalidCollectionType{};
exception          ElementNotFound{Object element; };
unsigned long      cardinality();
boolean is_empty();
boolean is_ordered();
boolean allows_duplicates();
boolean contains_element(in Object element);
void insert_element(in Object element);
void remove_element(in Object element)raises(ElementNotFound);
Iterator create_iterator(in boolean stable);
BidirectionalIterator create_bidirectional_iterator(in boolean stable)
raises(InvalidCollectionType);
Object                 select_element(in string OQL_predicate);
Iterator           select(in string OQL_predicate);
boolean              query(in string OQL_predicate, inout  Collection result);
boolean          exists_element(in string OQL_predicate);
};
```

An Iterator which is a mechanism for accessing the elements of a Collection object, can be created to traverse a collection.

The following operations are defined in the Iterator interface:

```
interface Iterator {
exception NoMoreElements{};
```

```
exception InvalidCollectionType{};
boolean is_stable();
boolean at_end();
void reset();
Object get_element() raises(NoMoreElements);
void next_position() raises(NoMoreElements);
void replace_element (in Object element) raises(InvalidCollectionType);
};
interface BidirectionalIterator : Iterator {
boolean at_beginning();
void previous_position() raises(NoMoreElements);
};
```

Set Objects:

A Set object is an unordered collection of elements, with no duplicates allowed. The following operations are defined in the Set interface:

```
interface SetFactory : ObjectFactory {
Set new_of_size(in long size);   };

class Set : Collection {
attribute set<t> value;
Set create_union(in Set other_set);
Set create_intersection(in Set other_set);
Set create_difference(in Set other_set);
boolean is_subset_of(in Set other_set);
boolean is_proper_subset_of(in Set other_set);
boolean is_superset_of(in Set other_set);
```

boolean is_proper_superset_of(in Set other_set);

};

Bag Objects:

A Bag object is an unordered collection of elements that may contain duplicates. The following interfaces are defined in the Baginterface:

interface BagFactory : ObjectFactory {

Bag new_of_size(in long size);

};

class Bag : Collection {

attribute bag<t>value;

unsigned long occurrences_of(in Object element);

Bag create_union(in Bag other_bag);

Bag create_intersection(in Bag other_bag);

Bag create_difference(in Bag other_bag);

};

List Objects:

A List object is an ordered collection of elements.

interface ListFactory : ObjectFactory {

List new_of_size(in long size);

};

class List : Collection {

exception InvalidIndex{unsigned long index; };

attribute list<t>value;

void remove_element_at(inunsigned long index)raises(InvalidIndex);

Object retrieve_element_at(in unsigned long index) raises(InvalidIndex);

void replace_element_at(in Object element, in unsigned long index) raises(InvalidIndex);

```
void insert_element_after(in Object element, in unsigned long index)
raises(InvalidIndex);
void insert_element_before(in Object element, in unsigned long index)
raises(InvalidIndex);
void insert_element_first (in Object element);
void insert_element_last (in Object element);
void remove_first_element() raises(ElementNotFound);
void remove_last_element()  raises(ElementNotFound);
Object retrieve_first_element() raises(ElementNotFound);
Object retrieve_last_element() raises(ElementNotFound);
List concat(in List other_list);
void append(in List other_list);
};
```

Dictionary Objects:

A Dictionary object is an unordered sequence of key-value pairs with no duplicate keys. Each key-value pair is constructed as an instance of the following structure:

```
struct Association {Object key; Object value; };
```

The following operations are defined in the Dictionary interface:

```
interface DictionaryFactory : ObjectFactory {
Dictionary new_of_size(in long size);
};
class Dictionary: Collection {
exception DuplicateName{string key; };
exception KeyNotFound{Object key; };
attribute dictionary<t,v>value;
void bind(in Object key, in Object value) raises(DuplicateName);
```

void unbind(in Object key) raises(KeyNotFound);

Object lookup(in Object key) raises(KeyNotFound);

boolean contains_key(in Object key);

};

Structured Objects:

All structured objects support the Object ODL interface. The ODMG Object Model defines the following structured objects:

•Date

•Interval

•Time

•Timestamp

OBJECT DEFINITION LANGUAGE :

The Object Definition Language is a language for defining the specifications of object types for ODMG-complaint systems. The ODL defines the attributes and relationships of types and specifies the signature of the operations, but it does not address the implementation of signatures. The ODMG hoped that the ODL would be basis for integrating schemas from multiple sources and applications.

OBJECT QUERY LANGUAGE :

Object Query Language provides declarative access to the object database using an SQL-like syntax.

ODL can be used both for associative and navigational access.

- An Associative query returns a collection of objects.
- A Navigational query accesses individual objects and object relationships are used to navigate from one object to another.

EXPRESSIONS

QUERY DEFINITION EXPRESSION :

A Query definition expression is of the form : **DEFINE** Q **AS** e. This defines a named query(that is, view) with name Q , given a query expression e.

ELEMENTARY EXPRESSIONS :

An Expression can be an atomic literal, a named object, an iterator object from the SELECT-FROM-WHERE statement, a query definition expression(Q above).

CONSTRUCTION EXPRESSIONS :

If T is a type name with properties p1, ...pn, and e1,.....en are expressions, then T(p1:e1,.....pn :en) is an expression of type T. For Example,

Manager (staffNo: “SL21”, fname: “john”, iname: “white”, position: “Manager”, salary: 30000)

ATOMIC TYPE EXPRESSIONS :

Expressions can be formed using the standard unary and binary operations, concatenation, string offset Si, S [low : up] etc.. on expressions.

OBJECT EXPRESSIONS :

Expressions can be formed using the equality and inequality operations(‘=’ and ‘!=’), returning a Boolean value. If e is an expression of a type having an attribute or a relationship p of type T, then we can extract the attribute or traverse the relationship using the expressions e.p and e->p, which are of type T.

COLLECTION EXPRESSIONS **:**

Expressions can be formed using universal FOR ALL, EXISTS, IN, SELECT FROM WHERE, ORDER BY, MIN, MAX, COUNT, SUM, AVG, GROUP BYclauses.

For example, FOR ALL * IN managers: x.salary > 12000.

CONVERSION EXPRESSIONS :

- If e is a list expression, then listtoset(e) is an expression that converts the list into set.
- If e is a collection-valued expression, then flatten(e) is an expression that converts a collection of collections into a collection.
- If e is an expression and c is a typename, then c(e) is an expression that asserts e is an object of type c.

INDEXED COLLECTIONS EXPRESSIONS :
If e1, e2 are lists or arrays and e3, e4 are integers, then e1[e3], e1[e3:e4], first(e1), last(e1), and (e1+e2) are expressions. For example, first(element(SELECT b FROM b IN branchoffices WHERE b.branchNo = “B001”)Has);
BINARY SET EXPRESSIONS :

If e1, e2 are sets or bags, then the set operators union, except, and intersect of e1 and e2 are expressions.

QUERIES :

A query consists of a set of query definition expressions followed by an expression. The result of a query is an object with or without identity.

- Object query Language - use of extents and traversal paths:
- We can use the extent of class staff to produce the required set using the following simple expression: *staff*
- To get the set of all branch managers(with identity):

Branchoffices.ManagedBy

- Object query Language- use of DEFINE :
- We can express this query as:

DEFINE Londoners AS SELECT s FROM s IN salesStaff WHERE s.WorksAt.address.city = "London";

SELECT s.name.iname FROM s IN Londoners;

- Object query Language- use of structures :
- To get the structured set(without identity) containing the name, sex, and age of all sales *staff* who work in London.

We can express the query using struct as:

SELECT **struct**(iname: s.name.iname, sex: s.sex, age: s.getAge) FROM s IN salesStaff WHERE s. WorksAt.address.city="London";

Which returns a literal of type set<struct>.

- Object query Language- use of aggregates :

OQL aggregate functions can be applied within the select clause or to the result of the select operation. For example,

SELECT COUNT(s) FROM s IN saleStaff WHERE s.WorksAt.branchNo="B003";

COUNT(SELECT s FROM s IN salesStaff WHERE s.WorksAt.branchNo="B003";

- GROUP BY and HAVING clauses :

As with SQL, the Having clause can be filter the positions and GroupBy clause is used to group more than one table.

For example,

SELECT branchNumber, averageSalary: AVG(SELECT p.s.salary FROM p IN partition) FROM s IN salesStaff GROUP BY branchNumber: s.WorksAt.branchNo HAVING COUNT(partition) > 10;

OTHER PARTS OF ODMG STANDARD :

OBJECT INTERCHANGE FORMAT **:**

The object interchange format is a specification language used to dump and load the current state of an ODMS to and from one or more files. It was also designed according to NCITS and PDES/STEP for mechanical CAD.

The OIF file is made up of one or more object definitions, where object definition is an object identifier and a class name.

For example,

John {salesStaff} - an instance of class salesStaff is created with name John.

John (Mary) {salesStaff} - an instance of class salesStaff is created with name John physically near to persistent object Mary.

ODMG LANGUAGE BINDINGS :

OML is used to specify how database objects are retrieved and manipulated within the application program. To create a working application, the c++ ODL declarations are passed through a C++ ODL Preprocessor, which has the effect of generating a C++ header file containing the object database definition and storing the ODMS metadata in the database. The users C++ applications which contains OML is then compiled in the normal way along with the generated object database definition C++ header file. Finally the output is linked with the ODMS runtime library to produce the required executable image. In addition, to ODL & OML the programmer can use a set of constructs, called physical programs.

In the C++ class library, features that implement the interface to the ODMG object model are prefixed d_. Examples are d_Float, d_String, d_Short for base data types and d_List, d_set, and d_Bag for collection types.

Relationships are handled by including either a reference or a collection. For example, to represent 1: * *Has* relationship in the Branch class we would write :

d_Rel_Set<salesStaff, _WorksAt>Has;

const char_WorksAt[] = “WorksAt”;

and to represent the same relationship in the salesStaff class we would write :

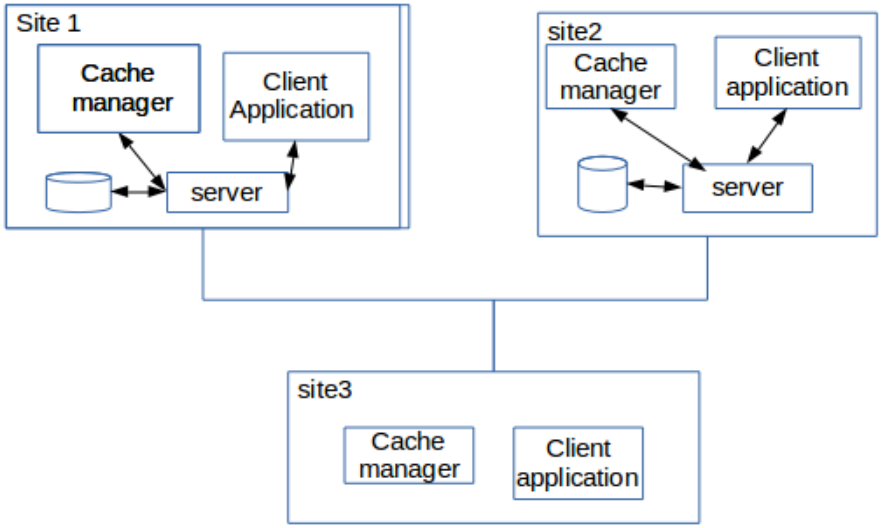
d_Rel_Set<Branch, _ Has > WorksAt;

const char_ Has[] = “Has”;

OBJECT STORE

ARCHITECTURE**:**

Object store is based on the multi –client/multi –server architecture. with each server is responsible for controlling access to an object store and for managing concurrency control,data recovery,and the transaction , among other tasks. A client can contact the object server on is hosts machine or any object store in any other server in the network. For each host machine running one application there is associated cache manager process those primary functions s to facilitate concurrent access to data by handling call back messages from server to client applications.



OBJECT STORE SERVER:

- Storage and retrieval of data.
- Handling concurrent access by multiple client applications.
- Database recovery.

CLIENT APPLICATION:

The object store clent library is linked into each clent applicaton,allowing the client application to:

- Map persistent object to virtual address.
- Allocate and de allocate storage for persistent object.
- Maintain a cache of recently used pages and lock status for those pages.
- Handle page fault on address that to persistent object.

CLIENT MANAGER:

When client application need to access an persistent object ,a page fault is generated in following situations:

- The object is not in physical memory and client cache.
- The object is client cache but not accessed yet.
- The object is client cache but has been previously accessed with different read/write operations.

OWNERSHIP,LOCKING AND THE CACHE MANAGER:

To understand the function of the cache manger, first have to understand object stores ownership and locking mechanism .a client may request read or write operation from a server .

Read ownership granted as many clients request it. provided client has no write ownership ,but there will be a only client that can be only one client with write operation at one time. Then a client read or write page during transaction he places read or write clock on that pages ,a client read or write operation permission for that page.

Once transaction completed client releases the lock.

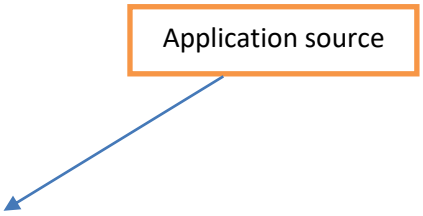
When client request permission to read a page and no other client has permission to update that page server can grant read ownership and the cache manager is at involved .however ,the cache manager is involved when:

- A client request read or write permission on page and other client has write permission on that page.
- A client request rite permission on a page and one other client has read permission on that page.

BUILDING OBJECT STORE APPLICATION:

Building C++ object store application is different from that described for the ODMG C++ language building.an object store is built from number of files:

- C++ application that contain main code.
- C ++ header files that contain the persistent classes.
- The necessary object store header files.
- A schema source fil e that defines the persistent classes for schema generator.


Application source

The object store generate can get two output file

- The applicable schema database,which contain type information about the objects the application can also persistently.
- An application schema object file ,which can linked to application program.

OBJECT STORE DATABASE:

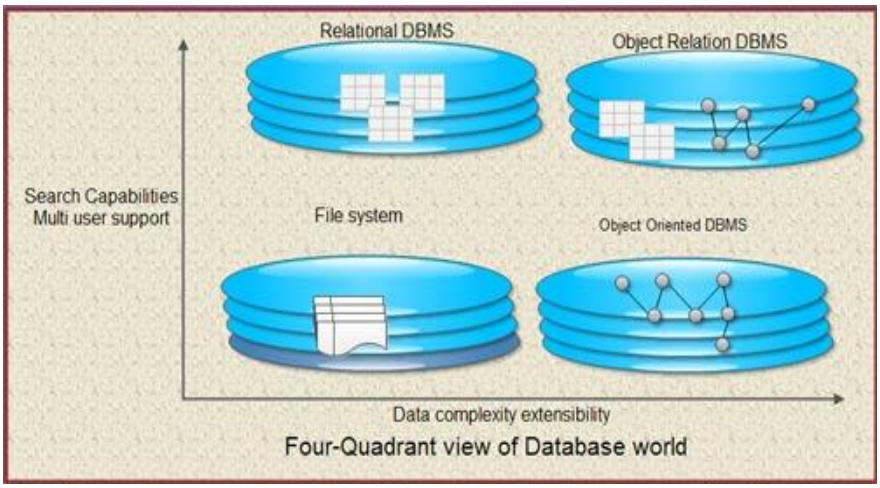
An object store darabase persistent objects and can be created using the os_database::create function.object store supports two types ofvdatabase:

- File database
- Raws(raw file system)

Object relational database(ORDBMS):

An object relational database management system (ORDBMS) is a database management system with that is similar to a relational database, except that it has an object-oriented database model. This system supports objects, classes and inheritance in database schemas and query language.The characteristic properties of ORDBMS are 1) complex data, 2) type inheritance, and 3) object behavior.

Stonebraker's view:



In a four-quadrant view of the database world, as illustrated in the figure,

The lower-left quadrant are those applications that process simple data and have

no requirements for querying the data. These types of application, for example standard text processing packages such as Word.

In the lower-right quadrant are those applications that process complex data but again have no significant requirements for querying the data. For these types of application, for example computer-aided design packages, an OODBMS may be an appropriate choice of DBMS.

In the top-left quadrant are those applications that process simple data and also have requirements for complex querying. Many traditional business applications fall into this quadrant and an RDBMS may be the most appropriate DBMS.

Finally, in the top-right quadrant are those applications that process completed data and have complex querying requirements. This represents many of the advanced database applications and for these applications an ORDBMS may be the appropriate choice of DBMS.

Advantages of ORDBMSs:

1)Reuse and Sharing: Reuse comes from the ability to extend the DBMS server to perform standard functionality centrally, rather than have it coded in each application.

2)Increased Productivity**:** ORDBMS provides increased productivity both for the developer and for the, end user

3)Use of experience in developing RDBMS: If the new functionality is designed appropriately, this approach should allow organizations to take advantage of the new extensions in an evolutionary way without losing the benefits of current database features and functions.

Disadvantages of ORDBMSs :

- Complexity and associated increased costs.
- Essential simplicity' and purity of the .relational model are lost with these types of extension.
- potentially misses the point of object orientation, highlighting the large semantic gap between these two technologies.
- Object applications are simply not as data-centric as relational-based ones.

Third-generation database system manifesto :

->A rich type system is needed (including extensions to query language).

-> Inheritance is a good idea.

-> Functions, including database procedures and methods, and encapsulation are a good idea.

-> Unique IDs should only be reassigned if human defined primary key is not available.

-> Rules are important

-> Programmatic access should be through a non-procedural high- level access language (minimal navigation).

-> Collections should be definable both through member enumeration and through queries (extensionally vs. intentionally).

-> Updatable views are essential.

->Performance indicators must not appear in data models.

-> DBMSs must be accessible through multiple higher level languages.

-> Persistent objects is a good idea.

-> SQL will prevail.

-> Queries and result set should be the lowest level of communication.

Postgres :

PostgreSQL, often simply **Postgres**, is an object-relational database (ORDBMS) – i.e. an ,RDBMS with additional (optional use) "object" features – with an emphasis on extensibility and standards compliance. As a database server, its primary functions are to store data securely and return that data in response to requests from other software applications.

Postgres extended the relational model to include :1)abstract data type

2) data of type procedure

3)rules.

Abstract data type (ADT) :

POSTGRES provides a collection of atomic and structured types. The predefined atomic

types include:int2,int4,float4,float8,bool,char, and date.

The command :

define typeint4 is(InternalLength = 4,InputProc = CharToInt4 ,OutputProc = Int4ToChar, Default = ‘‘0’’)

defines the type int4 which is predefined in the system. CharToInt4 and Int4ToChar are procedures that are coded in a conventional programming language

the command below defines the plus operator.

define operator‘‘+’’(int4, int4) returns int4 is (Proc = Plus, Precedence =5,

Associativity = ‘‘left’’)

Inheritence and relations:

create EMPLOYEE (Dept = char[25],

Status = int2, Mgr = char[25],

JobTitle = char[25], Salary = money)

inherits (PERSON)

retrieve(S.name) from S in STUDENT* where S.city = ‘‘Berkeley’.

SQL3:

SQL:1999 (also called SQL 3) was the fourth revision of the SQL database query language.. It introduced a large number of new features, many of them include

user-defined types :

eg: create type age as integer FINAL;

SQL:1999 has four new data types:

1)Large Object (LOB) type: CHARACTER LARGE OBJECT (CLOB)

BINARY LARGE OBJECT (BLOB)

2)Boolean type

3)Two new *composite* types: ARRAY (storing collections of values in a column) and ROW (storing structured values in single columns of the database)

4)Distinct types.

New Predicates : Using “SIMILAR” besides “LIKE”: gives programs UNIX-like regular expressions.

WHERE NAME SIMILAR TO '(SQL-(86|89|92|99)) I (SQL(I|2|3))‘ (which would match the various names given to the SQL standard over the years.)

DISTINCT predicate.

Enhanced Security :

Adding *role* facility.

Granting privileges to the *roles*.

Simplifying the difficult job of managing security in a Database environment.

Active Database : This facility is provided through a feature known as *triggers*.

Object Orientation The structured user-defined types.

->They may be defined to have one or more attributes.

->All aspects of their behaviors are provided through methods, functions, and procedures.

->Their attributes are encapsulated through the use of system-generated observer.

-> They may participate in type hierarchies.

UNIT VI

6.1.BIG DATA:

'Big Data' is also a **data** but with a **huge size**. 'Big Data' is a term used to describe collection of data that is huge in size and yet growing exponentially with time.In short, such a data is so large and complex that none of the traditional data management tools are able to store it or process it efficiently.

Categories Of 'Big Data'
Big data' could be found in three forms:

1. **Structured**

2. **Unstructured**
3. **Semi-structured**

Structured

Any data that can be stored, accessed and processed in the form of fixed format is termed as a 'structured' data. Over the period of time, talent in computer science have achieved greater success in developing techniques for working with such kind of data (where the format is well known in advance) and also deriving value out of it. However, now days, we are foreseeing issues when size of such data grows to a huge extent, typical sizes are being in the rage of multiple zettabyte.

Examples Of Structured Data

An 'Employee' table in a database is an example of Structured Data

Employee_ID	Employee_Name	Gender	Department	Salary_In_lacs
2365	Rajesh Kulkarni	Male	Finance	650000
3398	Pratibha Joshi	Female	Admin	650000
7465	Shushil Roy	Male	Admin	500000
7500	Shubhojit Das	Male	Finance	500000
7699	Priya Sane	Female	Finance	550000

Unstructured

Any data with unknown form or the structure is classified as unstructured data. In addition to the size being huge, un-structured data poses multiple challenges in terms of its processing for deriving value out of it. Typical example of unstructured data is, a heterogeneous data source containing a combination of simple text files, images, videos etc. Now a day organizations have wealth of data available with them but unfortunately they don't know how to derive value out of it since this data is in its raw form or unstructured format.

Examples Of Un-structured Data

Output returned by 'Google Search'

Semi-structured

Semi-structured data can contain both the forms of data. We can see semi-structured data as a strcutured in form but it is actually not defined with e.g. a table definition in relational DBMS. Example of semi-structured data is a data represented in XML file.

Examples Of Semi-structured Data

Personal data stored in a XML file-

Characteristics Of 'Big Data'

(i)Volume – The name 'Big Data' itself is related to a size which is enormous. Size of data plays very crucial role in determining value out of data. Also, whether a particular data can actually be considered as a Big Data or not, is dependent upon volume of data. Hence, **'Volume'** is one characteristic which needs to be considered while dealing with 'Big Data'.

(ii)Variety – The next aspect of 'Big Data' is its **variety**.

Variety refers to heterogeneous sources and the nature of data, both structured and unstructured. During earlier days, spreadsheets and databases were the only sources of data considered by most of the applications. Now days, data in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. is also being considered in the analysis applications. This variety of unstructured data poses certain issues for storage, mining and analysing data.

(iii)Velocity – The term **'velocity'** refers to the speed of generation of data. How fast the data is generated and processed to meet the demands, determines real potential in the data.

Big Data Velocity deals with the speed at which data flows in from sources like business processes, application logs, networks and social media sites, sensors, Mobile devices, etc. The flow of data is massive and continuous.

(iv)Variability – This refers to the inconsistency which can be shown by the data at times, thus hampering the process of being able to handle and manage the data effectively.

Benefits of Big Data Processing

Ability to process 'Big Data' brings in multiple benefits, such as-

• Businesses can utilize outside intelligence while taking decisions

Access to social data from search engines and sites like facebook, twitter are enabling organizations to fine tune their business strategies.

• Improved customer service

Traditional customer feedback systems are getting replaced by new systems designed with 'Big Data' technologies. In these new systems, Big Data and natural language processing technologies are being used to read and evaluate consumer responses.

• Early identification of risk to the product/services, if any

• Better operational efficiency

'Big Data' technologies can be used for creating staging area or landing zone for new data before identifying what data should be moved to the data warehouse. In addition, such integration of 'Big Data' technologies and data warehouse helps organization to offload infrequently accessed data.

6.2. HADOOP

data residing in a local file system of personal computer system, in Hadoop, data resides in a distributed file system which is called as a **Hadoop Distributed File system**.

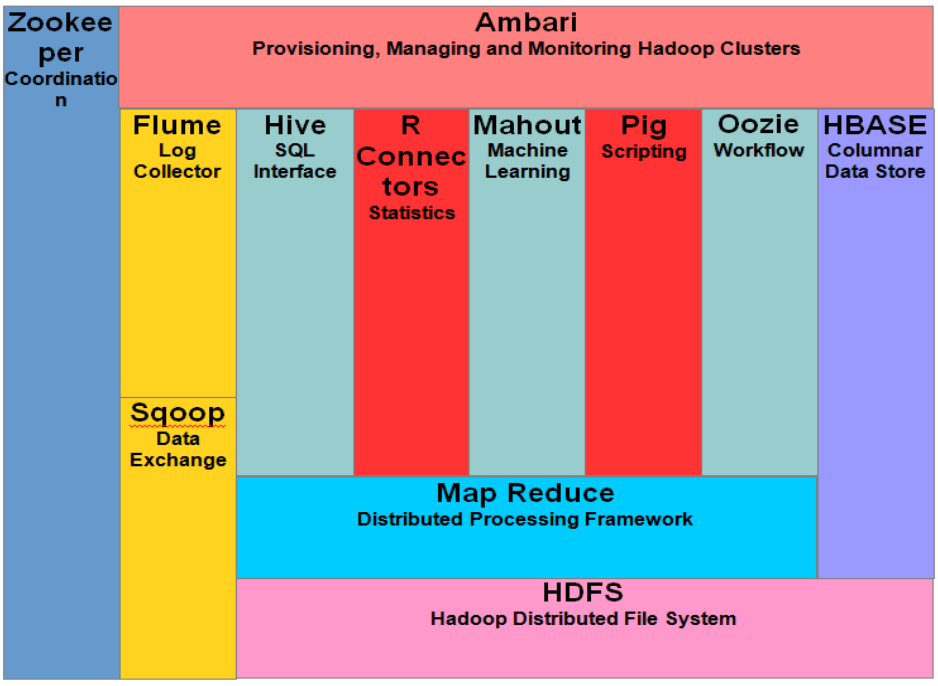
Processing model is based on **'Data Locality'** concept wherein computational logic is sent to cluster nodes(server) containing data.

This computational logic is nothing but a compiled version of a program written in a high level language such as Java. Such a program, processes data stored in Hadoop HDFS.

HADOOP is an open source software framework. Applications built using HADOOP are run on large data sets distributed across clusters of commodity computers.

Commodity computers are cheap and widely available. These are mainly useful for achieving greater computational power at low cost.

Components of Hadoop
Below diagram shows various components in Hadoop ecosystem-



Apache Hadoop consists of two sub-projects –

1. **Hadoop MapReduce** : MapReduce is a computational model and software framework for writing applications which are run on Hadoop. These MapReduce programs are capable of processing enormous data in parallel on large clusters of computation nodes.
2. **HDFS** (**Hadoop Distributed File System**): HDFS takes care of storage part of Hadoop applications. MapReduce applications consume data from HDFS. HDFS creates multiple replicas of data blocks and distributes them on compute nodes in cluster. This distribution enables reliable and extremely rapid computations.

Although Hadoop is best known for MapReduce and its distributed file system- HDFS, the term is also used for a family of related projects that fall under the umbrella of distributed computing and large-scale data processing. Other Hadoop-related projects at Apache include
are **Hive**, **HBase**, **Mahout**, **Sqoop** , **Flume** and **ZooKeeper**.

Features Of 'Hadoop'

• Suitable for Big Data Analysis

As Big Data tends to be distributed and unstructured in nature, HADOOP clusters are best suited for analysis of Big Data. Since, it is processing logic (not the actual data) that flows to the computing nodes, less network bandwidth is consumed. This concept is called as **data locality concept** which helps increase efficiency of Hadoop based applications.

• Scalability

HADOOP clusters can easily be scaled to any extent by adding additional cluster nodes, and thus allows for growth of Big Data. Also, scaling does not require modifications to application logic.

• Fault Tolerance

HADOOP ecosystem has a provision to replicate the input data on to other cluster nodes. That way, in the event of a cluster node failure, data processing can still proceed by using data stored on another cluster node.

HADOOP Distributed File Systems:

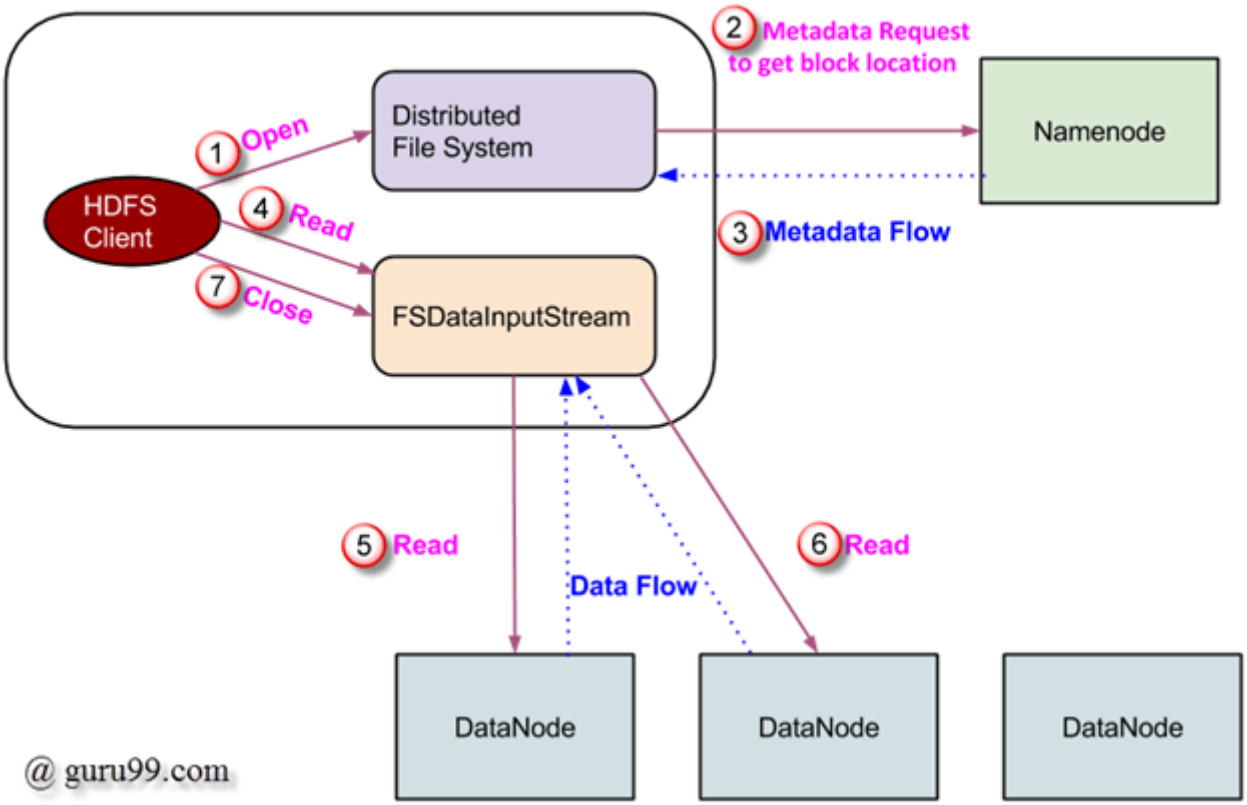
Hadoop comes with a distributed file system called **HDFS** (**HADOOP Distributed File Systems**) HADOOP based applications make use of HDFS. HDFS is designed for storing very large data files, running on clusters of commodity hardware. It is fault tolerant, scalable, and extremely simple to expand.

HDFS cluster primarily consists of a **NameNode** that manages the file system **Metadata** and a **DataNodes** that stores the **actual data**.

- **NameNode:** NameNode can be considered as a master of the system. It maintains the file system tree and the metadata for all the files and directories present in the system. Two files **'Namespace image'** and the **'edit log'** are used to store metadata information. Namenode has knowledge of all the datanodes containing data blocks for a given file, however, it does not store block locations persistently. This information is reconstructed every time from datanodes when the system starts.
- **DataNode :** DataNodes are slaves which reside on each machine in a cluster and provide the actual storage. It is responsible for serving, read and write requests for the clients.

Read/write operations in HDFS operate at a block level. Data files in HDFS are broken into block-sized chunks, which are stored as independent units. Default block-size is 64 MB.

Read Operation In HDFS
Data read request is served by HDFS, NameNode and DataNode. Let's call reader as a 'client'. Below diagram depicts file read operation in Hadoop.

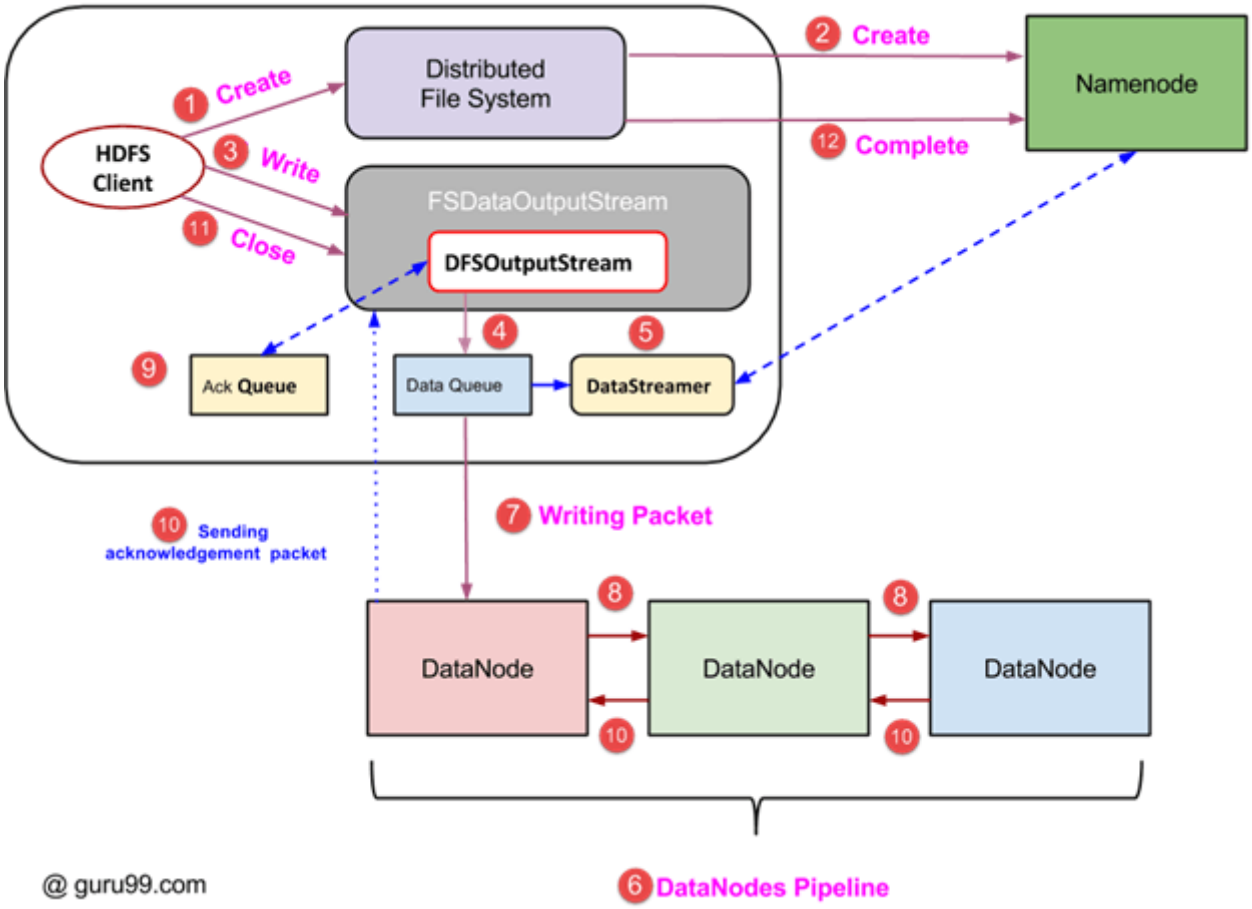


Client initiates read request by calling **'open()'** method of FileSystem object; it is an object of type **DistributedFileSystem**.

1. This object connects to namenode using RPC and gets metadata information such as the locations of the blocks of the file. Please note that these addresses are of first few block of file.
2. In response to this metadata request, addresses of the DataNodes having copy of that block, is returned back.
3. Once addresses of DataNodes are received, an object of type **FSDataInputStream** is returned to the client. **FSDataInputStream** contains **DFSInputStream** which takes care of interactions with DataNode and NameNode. In step 4 shown in above diagram, client invokes **'read()'** method which causes **DFSInputStream** to establish a connection with the first DataNode with the first block of file.
4. Data is read in the form of streams wherein client invokes **'read()'** method repeatedly. This process of **read()** operation continues till it reaches end of block.
5. Once end of block is reached, DFSInputStream closes the connection and moves on to locate the next DataNode for the next block

6. Once client has done with the reading, it calls **close()** method.

Write Operation In HDFS



In this section, we will understand how data is written into HDFS through files.

1. Client initiates write operation by calling 'create()' method of DistributedFileSystem object which creates a new file - Step no. 1 in above diagram.
2. DistributedFileSystem object connects to the NameNode using RPC call and initiates new file creation. However, this file create operation does not associate any blocks with the file. It is the responsibility of NameNode to verify that the file (which is being created) does not exist already and client has correct permissions to create new file. If file already exists or client does not have sufficient permission to create a new file, then **IOException** is thrown to client. Otherwise, operation succeeds and a new record for the file is created by the NameNode.
3. Once new record in NameNode is created, an object of type

FSDataOutputStream is returned to the client. Client uses it to write data into the HDFS. Data write method is invoked (step 3 in diagram).

4. FSDataOutputStream contains DFSOutputStream object which looks after communication with DataNodes and NameNode. While client continues writing data, **DFSOutputStream** continues creating packets with this data. These packets are en-queued into a queue which is called as **DataQueue**.
5. There is one more component called **DataStreamer** which consumes this **DataQueue**. DataStreamer also asks NameNode for allocation of new blocks thereby picking desirable DataNodes to be used for replication.
6. Now, the process of replication starts by creating a pipeline using DataNodes. In our case, we have chosen replication level of 3 and hence there are 3 DataNodes in the pipeline.
7. The DataStreamer pours packets into the first DataNode in the pipeline.
8. Every DataNode in a pipeline stores packet received by it and forwards the same to the second DataNode in pipeline.
9. Another queue, 'Ack Queue' is maintained by DFSOutputStream to store packets which are waiting for acknowledgement from DataNodes.
10. Once acknowledgement for a packet in queue is received from all DataNodes in the pipeline, it is removed from the 'Ack Queue'. In the event of any DataNode failure, packets from this queue are used to reinitiate the operation.
11. After client is done with the writing data, it calls close() method (Step 9 in the diagram) Call to close(), results into flushing remaining data packets to the pipeline followed by waiting for acknowledgement.
12. Once final acknowledgement is received, NameNode is contacted to tell it that the file write operation is complete.

MapReduce : MapReduce is a programming model suitable for processing of huge data. Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

MapReduce programs work in two phases:

1. Map phase
2. Reduce phase.

Input to each phase are **key-value** pairs. In addition, every programmer needs to specify two functions: **map function** and **reduce function**.

The whole process goes through three phase of execution namely,

How MapReduce works

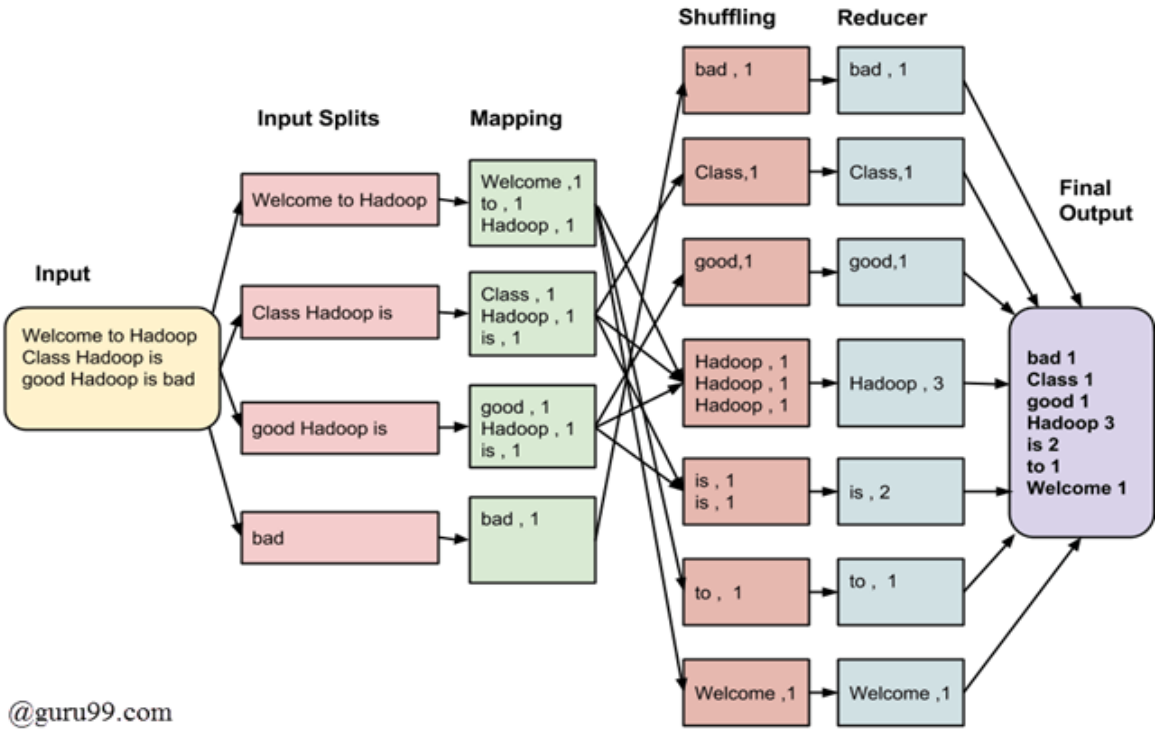
Lets understand this with an example –

Consider you have following input data for your MapReduce Program

Welcome to Hadoop Class

Hadoop is good

Hadoop is bad



The final output of the MapReduce task is

bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

The data goes through following phases

Input Splits:

Input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

Mapping

This is very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, job of mapping phase is to count number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling

This phase consumes output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, same words are clubed together along with their respective frequency.

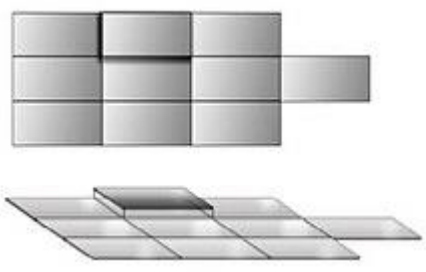
Reducing

In this phase, output values from Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

6.3. NO SQL DATABASES:

A **NoSQL** (originally referring to "non SQL" or "non relational")database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.NoSQL databases are increasingly used in big data and real-time web applications.NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.

The term *NoSQL* was used by Carlo Strozzi in 1998 to name his lightweight, Strozzi NoSQL open-source relational database that did not expose the standard Structured Query Language (SQL) interface, but was still relational.His NoSQL RDBMS is distinct from the circa-2009 general concept of NoSQL databases. Strozzi suggests that, because the current NoSQL movement "departs from the relational model altogether, it should therefore have been called more appropriately 'NoREL',referring to 'No Relational'.



First visual representation of NoSQL initiatives

Types and examples of NoSQL databases

There have been various approaches to classify NoSQL databases, each with different categories and subcategories, some of which overlap. What follows is a basic classification by data model, with examples:

- **Column**: Accumulo, Cassandra, Druid, HBase, Vertica, SAP HANA
- **Document**: Apache CouchDB, ArangoDB, Clusterpoint, Couchbase, Cosmos DB, HyperDex, IBM Domino, MarkLogic, MongoDB, OrientDB, Qizx, RethinkDB
- **Key-value**: Aerospike, ArangoDB, Couchbase, Dynamo, FairCom c-treeACE, FoundationDB, HyperDex, InfinityDB, MemcacheDB, MUMPS, Oracle NoSQL Database, OrientDB, Redis, Riak, Berkeley DB, SDBM/Flat File dbm
- **Graph**: AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso
- **Multi-model**: ArangoDB, Couchbase, FoundationDB, InfinityDB, MarkLogic, OrientDB

6.4.The CAP Theorem

According to University of California, Berkeley computer scientist Eric Brewer, the theorem first appeared in autumn 1998.It was published as the CAP principle in 1999 and presented as a conjecture by Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's conjecture, rendering it a theorem.

In 2012, Brewer clarified some of his positions, including why the often-used "two out of three" concept can be misleading or misapplied, and the different definition of consistency used in CAP relative to the one used in ACID.

A similar theorem stating the trade-off between consistency and availability in distributed systems was published by Birman and Friedman in 1996.The result of

Birman and Friedman restricted this lower bound to non-commuting operations.

In theoretical computer science, the CAP theorem, also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees

Consistency	***Availability***	***Partition tolerance***
Every read receives the most recent write or an error	Every request receives a (non-error) response – without guarantee that it contains the most recent write	The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

In other words, the CAP theorem states that in the presence of a network partition, one has to choose between consistency and availability. Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in ACIDdatabase transactions.

No distributed system is safe from network failures, thus network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. When choosing consistency over availability, the system will return an error or a time-out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

In the absence of network failure – that is, when the distributed system is running normally – both availability and consistency can be satisfied.

CAP is frequently misunderstood as if one had to choose to abandon one of the three guarantees at all times. In fact, the choice is really between consistency and availability only when a network partition or failure happens ; at all other times, no trade-off has to be made.

Database systems designed with traditional ACID guarantees in mind such as RDBMS choose consistency over availability, whereas systems designed around the BASE philosophy, common in the NoSQL movement for example, choose availability over consistency.

The PACELC theorem builds on CAP by stating that even in the absence of partitioning, another trade-off between latency and consistency occurs.

6.5. comparison with relational databases and noSQL databases types

Advantages of NOSQL over Relational

- Provides a wide range of data models to choose from
- Easily scalable
- Database administrators are not required
- Some of the NOSQL DBaaS providers like Riak and Cassandra are programmed to handle hardware failures
- Faster , more efficient and flexible
- Has evolved at a very high pace

Disadvantages of NOSQL over Relational

- Immature
- No standard query language
- Some NOSQL databases are not ACID compliant
- No standard interface
- Maintenance is difficult

6.6. Relationship between CAP,ACID, and noSQL databases

The CAP Theorem
Published by Eric Brewer in 2000, the theorem is a set of basic requirements that describe any distributed system (not just storage/database systems).

Let's imagine a distributed database system with multiple servers being used by an outside person requesting information. Here's how the CAP theorem applies:

- **Consistency** - All the servers in the system will have the same data so anyone using the system will get the same copy regardless of which server answers their request.
- **Availability** - The system will always respond to a request (even if it's not the latest data or consistent across the system or just a message saying the system isn't working).
- **Partition Tolerance** - The system continues to operate as a whole even if individual servers fail or can't be reached.

It's theoretically impossible to have all 3 requirements met, so a combination of 2 must be chosen and this is usually the deciding factor in what technology is used.

ACID

ACID is a set of properties that apply specifically to database transactions, defined as follows:

- **Atomicity** - Everything in a transaction must happen successfully or none of the changes are committed. This avoids a transaction that changes multiple pieces of data from failing halfway and only making a few changes.
- **Consistency** - The data will only be committed if it passes all the rules in place in the database (ie: data types, triggers, constraints, etc).
- **Isolation** - Transactions won't affect other transactions by changing data that another operation is counting on; and other users won't see partial results of a transaction in progress (depending on isolation mode).
- **Durability** - Once data is committed, it is durably stored and safe against errors, crashes or any other (software) malfunctions within the database.

SQL / Relational DB

ACID is commonly provided by most classic relational databases like MySQL, Microsoft SQL Server, Oracle and others. These databases are known for storing data in spreadsheet-like tables that have their columns and data types strictly defined. The tables can have relationships between each other and the data is queried with SQL (Structured Query Language), which is a standardized language for working with databases.

NoSQL

With the massive amounts of data being created by modern companies, alternative databases have been developed to deal with the scaling and performance issues of existing systems as well as be a better fit for the kind of data created. NoSQL databases are what these alternatives are called because many do not support SQL as a way to query the data.

These NoSQL alternatives have matured now and while some do provide SQL abilities, they have come to be known more for their emphasis on scalable storage of a much higher magnitude of data (ie: terabytes and petabytes) by dropping direct support for database joins, storing data differently and using several distributed servers together as one.

Overview:

CAP provides the basic requirements that a distributed storage system has to follow and ACID is a set of rules that a database can choose to follow that guarantees how it handles transactions and keeps data safe.

NoSQL databases are alternatives to classic relational databases for storing lots more data or different kinds of data and they often use a distributed set of servers working together. This system has to fit 2 of the 3 requirements of the CAP theorem (depends on the software used and the needs of the application).

When it comes to how safe the committed data is, any ACID compliant system can be considered reliable but most NoSQL databases don't implement ACID and vary in how durable they are with stored data.

By the way, SQL vs NoSQL terms aren't useful in practice and it's far better to just reference the type of database itself, here are the major types:

- Relational
- Document
- Key/Value
- Wide-Column (different from relational db with columnar storage)
- Graph
- Search (optimized for storing and searching against text)

6.7.case studies usng mango db and cassendra:

Apache’s Cassandra DB:
Born at Facebook, Cassandra is great at handling massive amounts of unstructured data. If you’re struggling with making your relational database faster and more reliable—particularly when you’re at scale—Cassandra may be a great option for you. It combines Amazon’s Dynamo storage system with Google’s Bigtable paradigm, offering the near-constant availability required to support real-time querying for web and mobile apps.

Cassandra can tackle even the most massive data sets.
If you need a NoSQL database that can handle massive amounts of data with no single point of failure, the high-performing Cassandra database is an excellent option. It’s used on data-heavy apps like Instagram, which has to handle an average of 80 million photos uploaded a day, and Spotify, which stores over 20 million songs in its database.

If you're a business owner, you may be thinking "Do I need all of that? My data won't get *that* big," but Cassandra isn't just effective at scale. It has excellent compression tools and high availability thanks to an eventual consistency model—a plus if immediate consistency is not a top priority for your data.

Cassandra boasts amazing, record-setting reliability at scale.
For being so scalable, Cassandra is nearly guaranteed to run safely and reliably. It was designed to handle big data in a distributed fashion, capable of being deployed across multiple servers right out of the box with little extra work. If you anticipate a lot of growth, Cassandra is famously easy to manage at scale.

Also, Cassandra uses peer-to-peer fault-tolerance technology—no master/slave setup, failover, or leader election. This means any node in the cluster can be delegated to perform your query in the event of a failure.

Eventual consistency yields high availability.
One thing to bear in mind with Cassandra is that it tends to compromise latency and consistency for availability. It's "**eventually consistent**," a model for NoSQL database consistency that's used with distributed setups. Rather than maintain strict consistency that could really slow things down at scale, eventual consistency enables high availability—just at the cost of every instance of your data not being synced up across all servers right away.

Wide-column flexibility.
Wide-column stores are a type of NoSQL database that are a slight variation on a key-value store. Wide-column stores use rows and columns, but the names and formats of the columns can change—something that isn't possible with a relational DB.

Minimal administrative tasks at scale.
As a rule of thumb, databases are typically either easy to set up, or easy to run. While MongoDB is very easy to use right out of the box, where Cassandra really excels is being easy to maintain once you've scaled *way* up. Much of that easy maintenance is thanks to a simple tool that lets you add, remove, or edit nodes. Resyncing and balancing data are both automatic as you scale up, as well.

Cassandra is also more minimalist than MongoDB, which comes with a lot of features packed in. Its continuous availability, high scalability, performance, and strong security give it an operational simplicity while also lowering overall cost of ownership.

In summary, Cassandra is good if your app requires…
A simple setup, easy maintenance (no matter how big your data gets), flexible parsing/wide column requirements, no multiple secondary indexes, and massive scale. However, it's not as good if your app requires transactional operations, primary or financial records, needs dynamic queries on column data, low latency, or on-the-fly aggregations.

Apps that use Cassandra: *Instagram, eBay (generating personalized recommendations to users), Comcast's messaging bus, Hulu, Rackspace, Peloton stationary bikes (storing real-time user stats), Apple, and Spotify.*

MongoDB:
The headline with MongoDB? It pretty much offers the best of both worlds: relational and NoSQL. The pros of relational databases remain, as does the easy setup that lets developers build apps faster than with a NoSQL database. MongoDB is known to be a developer-driven database, and it's especially popular among startups.

As a document-based store, it offers amazing flexibility.
Need a replacement for your existing RDBMS that's faster and more reliable? MongoDB can do most things you can do with MySQL or PostgreSQL, but it doesn't limit you with their predefined columns. Being able to scale horizontally and handle unstructured data also makes it an ideal pairing for mobile app development and content management systems. In addition, MongoDB is schema-free, so your database can evolve along with your application.

MongoDB can analyze data of any kind within the database itself. In Mongo, each document has its own data, and its own unique key, which is used to retrieve it. It's a great option for data that's document-oriented, but still somewhat structured, making it more like a replacement for a relational database management system (RDBMS), just for unstructured data.

High-performance, high availability, and automatic scaling.
MongoDB has full index support for high performance, replication, and fail-over (a fault-tolerance measure achieved through redundancy) for high, real-time availability of data. It also features auto-sharding for excellent, unlimited horizontal scalability.

Feature-packed out of the box.
Whereas Cassandra is more minimalist, MongoDB has lots of features from the start. Its powerful query system lets you leverage location-based data well, too, with built-in spacial functions that allow you to harvest this data from specific locations and put it to use without complicated extraction processes.

Try new things, and fast.
Its ability to support fast iterations means you can scale up, make modifications, and give your customers new and better apps fast, and without the cost of updating your RDBMS. MongoDB's dynamic schemas let you test out and experiment on the fly. Because your data doesn't need to be prepped ahead of time to go into an SQL table, your data team can incorporate new types of data and schemas into your database quickly, and at a lower cost.

In summary, MongoDB is good if your app requires…
An RDBMS replacement, real-time analytics, high-speed logging, caching, and high scalability. However, it's not as good if your app requires a highly transactional system (although it is built to handle transactions on single documents, so may be enough for your e-commerce needs), or traditional database requirements such as foreign key constraints.

Side by Side Comparison of MongoDB versus Cassandra

Upwork™

	Cassandra	MongoDB
Written in	Java	C++
Protocol	CQL3 (similar to SQL). Cassandra's CQL language is intended to be "exactly like SQL, except when it's not."	Custom binary (BSON)
Type	Wide-column store	Document-based store
Compatibility	The C/C++, C#, Java, Node.js, Python, and Ruby drivers support Cassandra's binary protocol.	Not limited to the MEAN stack—it's compatible with .NET applications, the Java platform, and more.
Fault tolerance	Cassandra has high availability without a single point of failure.	Master-slave model + failover, with no single point of failure.
Read/Write capabilities	Cassandra meets low latency requirements for write requests, at low throughput. Not engineered for read performance.	MongoDB meets low latency requirements for read requests, at low throughput.
Replication	Replication with Cassandra is very simple—it mostly takes care of it all for you. Just indicate how many nodes you want your data copied to and it automatically handles the rest.	MongoDB's replication facility, called replica set, provides automatic failover and data redundancy. Replication needs to be set up, but Mongo has clear guidelines to help.
Sharding	Auto-sharding, with a different effect on its replication. In Cassandra, each shard is a server, with its data replicated across other servers.	Sharding is built-in, but vs. Cassandra, MongoDB shards are replications of multiple servers.
Querying	Predictable query performance, but being a key/value store it can be a bit more tricky than MongoDB. It allows querying by key, or key range.	"Query by example" model. Mongo can execute complex, dynamic queries, but it's not necessarily the best fit for reporting-style workloads or complex transactions.
Hadoop support	Native Hadoop support, including Hive and Pig	MongoDB Connector for Hadoop