1. Introduction to Object-Relational Database Systems (ORDBMS)

Student ID

:

Module Code :

Student Referral Sheet

–

Faculty of Computing (Final Examination Jan

-

June 2025)

Evolution of Database Models

Relational Model (1970s)

- Simple, table-based structure with SQL.

- Limitations: Poor handling of complex data (e.g., multimedia, hierarchical relationships).

Object-Oriented Model (1980s)

- Supported complex objects, inheritance, and methods (ODMG standards).

- Limited adoption due to compatibility issues.

Object-Relational Model (Mid-1990s)

- Combines relational and OO features (SQL:99/SQL:2003 standards).

- Backward-compatible with relational databases. Limitations of Relational Databases

Set-Valued Attributes

- Example: Storing multiple phone numbers or children for a person requires normalized tables (1NF → 3NF), leading to complex joins.

- ORDB solution: Allow attributes to be sets (e.g., `PhoneN: {string}`).

Artificial Keys

- Addresses or composite keys are cumbersome.

ISA Hierarchies

- Poor support for inheritance (e.g., "Student is a Person"). Object-Relational Features in SQL

User-Defined Types (UDTs)

- Define classes with attributes and methods.

- Used for:

Row Objects-Entire table rows (e.g., `CREATE TABLE bars OF BarType`).

Column Objects-Attributes within a table (e.g., `beer BeerType`).

Object References (`REF`)

- Pointer-like mechanism to link objects (e.g., `REF BarType`).

- Dereferencing: Automatic with dot notation (e.g., `s.beer.name`).

Constraints

- Primary keys, `NOT NULL` allowed; no uniqueness on `REF` columns.

Key Oracle-Specific Syntax

Type Creation

sql

CREATE TYPE BarType AS OBJECT (name CHAR(20), addr CHAR(20));

Row Object Table

sql

CREATE TABLE bars OF BarType;

Inserting Values:

- As multi-column: `INSERT INTO bars VALUES ('Joe''s Bar', 'Maple St.');

- As object: `INSERT INTO bars VALUES(BarType('Joe''s Bar', 'Maple St.'));

Advanced Concepts

Incomplete Types

- Forward declarations for co-dependent types (e.g., `Employee` and `Department`).

Scoped REFs

- Restrict references to specific tables (e.g., `REF dept\_t REFERENCES dept\_table`).

Null Handling

- NULL allowed for objects, attributes, or collections.

2.Object-Relational Databases (ORDB) – Collections

Introduction to Collections

-Purpose-Model one-to-many relationships(e.g., an investor with multiple share purchases).

Types in Oracle

VARRAY (Variable-Size Array):Fixed maximum size (e.g., CREATE TYPE price\_arr AS VARRAY(10) OF NUMBER(12,2)).

Nested Tables: Flexible-size relations stored within a column (e.g.-CREATE TYPE BeerTableType AS TABLE OF BeerType).

VARRAYs

Definition:Ordered collections with a maximum limit.

Usage:

As column datatypes or attributes in UDTs.

Inserted using type constructors(e.g., `excharray('Sydney', 'New York')).

Example:

CREATE TABLE pricelist (pno INTEGER, prices price\_arr);

INSERT INTO pricelist VALUES(1, price\_arr(2.50, 3.75));

Querying: Use `TABLE()` to unnest:

SELECT pno, s.COLUMN\_VALUE FROM pricelist p, TABLE(p.prices) s;

Nested Tables

Definition:Tables stored within a column of another table.

Syntax:

CREATE TYPE BeerTableType AS TABLE OF BeerType;

CREATE TABLE Manfs (name CHAR(30), beers BeerTableType)

NESTED TABLE beers STORE AS beer\_table;

Querying:

Retrieve nested tables directly:

SELECT beers FROM Manfs WHERE name = 'Anheuser-Busch';

- Flatten with `TABLE()` for joins:

SELECT b.name FROM TABLE(SELECT beers FROM Manfs WHERE name = 'Anheuser-Busch') b;

DML Operations on Collections

Insert/Update/Delete:Target nested tables using `TABLE()`:

INSERT INTO TABLE(SELECT e.projects FROM employees e WHERE e.eno = 1000)

VALUES (103, 'Project Neptune');

Handling NULLs:

- Set nested table to `NULL` or initialize empty (`proj\_list()`).

Multilevel Collections

Concept:Nested collections within collections (e.g., stars → planets → satellites).

Example:

CREATE TYPE sat\_ntt AS TABLE OF sat\_t; -- Satellites

CREATE TYPE planet\_ntt AS TABLE OF planet\_t; -- Planets (with satellites)

CREATE TABLE stars\_tab OF star\_t

NESTED TABLE planets STORE AS planets\_nttab

(NESTED TABLE satellites STORE AS satellites\_nttab);

Insertion: Hierarchical data insertion:

INSERT INTO stars VALUES('Sun', 23, planet\_ntt(planet\_t('Neptune', 10, sat\_ntt(...)));

Key Takeaways

VARRAYs:Fixed-size, ordered. Ideal for small, predictable datasets.

Nested Tables:Dynamic, stored as separate relations. Better for complex hierarchies.

DML Flexibility:Use `TABLE()` to manipulate nested data directly.

Multilevel Support:Model real-world hierarchies (e.g., cosmic systems, organizational structures).

3.Object-Relational Databases (ORDB) – Methods and Inheritance

Methods in User-Defined Types (UDTs)

Purpose:Encapsulate behavior within object types (e.g., compute derived attributes, enforce business logic).

Types:

Member Methods:Bound to object instances (use `SELF` keyword to reference the current object).

CREATE TYPE MenuType AS OBJECT (

price FLOAT,

MEMBER FUNCTION priceInYen(rate FLOAT) RETURN FLOAT

);

Static Methods:Independent of instances

Implementation:

Declaration:In `CREATE TYPE`.

Definition:In `CREATE TYPE BODY` (PL/SQL or Java).

CREATE TYPE BODY MenuType AS

MEMBER FUNCTION priceInYen(rate FLOAT) RETURN FLOAT IS

BEGIN

RETURN rate \* SELF.price;

END;

END;

Example Usage:

SELECT s.beer.name, s.priceInYen(106.0) FROM Sells s;

Object Comparison: Map vs. Order Methods

Challenge:Objects lack default comparison semantics.

Solutions:

Map Method:Converts objects to scalars (e.g., compare rectangles by area).

MAP MEMBER FUNCTION area RETURN NUMBER IS

BEGIN RETURN length \* width; END;

Order Method:Directly compares two objects (returns -1/0/1).

ORDER MEMBER FUNCTION match(c Customer\_typ) RETURN INTEGER IS

BEGIN IF id < c.id THEN RETURN -1; END IF; ... END;

Key Difference:

Map:Efficient for sorting large datasets (one scalar conversion per object).

Order:Flexible for complex comparisons (e.g., image brightness).

Inheritance in ORDBMS

Concept:Organize types hierarchically (e.g., `Student` IS-A `Person`).

Oracle Implementation:

Supertype (Parent):Declared with `NOT FINAL` to allow subtypes.

CREATE TYPE Person\_type AS OBJECT (...) NOT FINAL;

Subtype (Child):Inherits attributes/methods via `UNDER`.

CREATE TYPE Student\_type UNDER Person\_type (deptid NUMBER, ...);

Specialization:

- Add new attributes/methods.

- Override inherited methods (e.g., `calculate\_area()` for `Circle` vs. `Rectangle`).

Advanced Inheritance Features

Type Modifiers:

FINAL/NOT FINAL: Control subtype creation.

INSTANTIABLE/NOT INSTANTIABLE: Abstract types (no direct instances).

Method Modifiers:

OVERRIDING: Subtypes redefine inherited methods.

FINAL Methods:Prevent further overriding.

Overloading:Same method name, different parameters.

-- In supertype:

MEMBER FUNCTION fun(x NUMBER)...

-- In subtype:

MEMBER FUNCTION fun(x DATE)...

Practical Examples

Querying Subtypes:

`IS OF` filters by type:

SELECT VALUE(p) FROM person\_tab p WHERE VALUE(p) IS OF (Student\_type);

`TREAT()` accesses subtype attributes:

SELECT TREAT(VALUE(p) AS PartTimeStudent\_type).numhours FROM person\_tab p;

Hierarchical Data Insertion:

INSERT INTO person\_tab VALUES(Student\_type(4, 'Alice', '123 Main St', 40, 'CS'));

Key Takeaways

Methods:Enable object behavior encapsulation (e.g., currency conversion).

Comparison: Use `MAP` for efficiency, `ORDER` for flexibility.

Inheritance: Promotes code reuse and logical hierarchy (e.g., `Person` → `Student` → `PartTimeStudent`).

Polymorphism:Overriding/overloading supports diverse implementations.

4.Object-Relational Databases (ORDB) – Triggers and ER-to-ORDB Mapping

Triggers in ORDBMS

Definition:Procedures automatically executed ("fired") on specific events (DML operations, system events).

Types:

DML Triggers: `INSERT`, `UPDATE`, `DELETE` on tables/views.

System Triggers: Database/schema-level events.

Example: Trigger to log office changes for staff:

CREATE TRIGGER trig1

BEFORE UPDATE OF office ON staff\_tab

FOR EACH ROW

WHEN (new.office.campus = 'Bentley')

BEGIN

INSERT INTO movement VALUES (:old.sid, :old.office, :new.office);

END;

ER-to-ORDB Mapping

Extended ER Model Features:

Multi-valued attributes (e.g., `VARRAY`/nested tables).

Structured attributes (e.g., `address` as a UDT).

Inheritance (ISA hierarchies).

Mapping Rules:

Entity Types → UDTs:

CREATE TYPE Emp\_t (eno CHAR(11), name CHAR(20), ...);

CREATE TABLE employees OF Emp\_t (eno PRIMARY KEY);

ISA Hierarchies: Use `UNDER` for subtypes:

CREATE TYPE hourly\_emps\_t UNDER emp\_t (hourly\_wages NUMBER);

Relationships:

1:1/N:1: Embed `REF` in one type (e.g., `workdept REF dept\_t`).

M:N: Create a separate relationship type with `REF`s:

CREATE TYPE WorksIn\_t (emp REF emp\_t, dept REF dept\_t);

Weak Entities:

Option 1: Treat as regular entities.

Option 2: Nested collections (e.g., `dependents policy\_t`).

N-ary Relationships: Separate UDT with `REF`s to all participants.

Practical Example: Cyber Shop Schema

Entities:

CREATE TYPE StockItem\_typ (StockNo NUMBER) NOT FINAL;

CREATE TYPE Book\_typ UNDER StockItem\_typ (...);

Weak Entity (Line Items): Nested table in `PurchaseOrder\_typ`:

CREATE TYPE LineItemList\_ntabtyp AS TABLE OF LineItem\_typ;

CREATE TYPE PurchaseOrder\_typ (..., LineItemList LineItemList\_ntabtyp);

Constraints:

CREATE TABLE Orders OF PurchaseOrder\_typ (

PONo PRIMARY KEY,

Cust\_ref REFERENCES Customers

) NESTED TABLE LineItemList STORE AS LineItems\_ntab;

Key Takeaways

Triggers: Automate data integrity checks or logging (e.g., tracking changes).

ER-to-ORDB Mapping:

- UDTs for entities, nested tables for multi-valued attributes.

- `REF` for relationships, `UNDER` for inheritance.

Design Flexibility: Supports complex hierarchies (e.g., weak entities, n-ary relationships).

5.Introduction to XML

Definition:

- XML (extensible Markup Language) is a semantic markup language used for data representation and exchange.

- Unlike HTML, XML tags describe data content rather than formatting.

Well-Formed XML:

- Starts with `<?xml version="1.0"?>`.

- Contains a root tag with nested tags (case-sensitive).

Example:

<catalog>

<product dept="WMN">

<number>557</number>

<name>Fleece Pullover</name>

</product>

</catalog>

XML Schema

Purpose: Defines structure and constraints for XML documents.

Types:

- DTD (Document Type Definition): Older, simpler.

- XML Schema (XSD): Supports data types and namespaces.

Example (XSD):

<xs:complexType name="PersonType">

<xs:sequence>

<xs:element name="Name" type="xs:string"/>

</xs:sequence>

</xs:complexType>

XML Data Storage

Native XML Databases:

- Store XML natively (e.g., Tamino, Xindice).

- Best for unstructured/semi-structured data.

XML-Enabled Databases:

- Use relational databases (e.g., Oracle, SQL Server) with XML extensions.

- Best for structured data with static schemas.

Storage Options:

-Untyped XML: No schema validation.

- Typed XML: Validated against a schema (optimized for queries).

-Example (SQL Server):

CREATE TABLE AdminDocs (id INT PRIMARY KEY, xDoc XML);

Querying XML

XPath:

- Path expressions to navigate XML (e.g., /catalog/product/name).

- Supports predicates (e.g.-//product[number > 500]).

XQuery:

- Full query language (like SQL for XML).

-FLWOR Expressions:

xquery

for $prod in doc("catalog.xml")//product

where $prod/@dept = "ACC"

return $prod/name

Advanced Features

Conditionals:

xquery

if ($prod/@dept = 'ACC') then <accessory/> else <other/>

Functions:

- Built-in (e.g., `substring()`, `sum()`).

- User-defined (modular).

SQL Server XML Methods

query(): Extract XML fragments.

SELECT xDoc.query('/catalog/product') FROM AdminDocs;

value(): Extract scalar values.

SELECT xDoc.value('(/product/name)[1]', 'VARCHAR(50)');

exist(): Check for node existence.

modify(): Update XML (insert/delete/replace).

Key Takeaways

- XML is ideal for hierarchical, semi-structured data.

Storage: Choose native (flexible) vs. enabled (powerful) based on schema stability.

Querying: XPath for simple navigation, XQuery/FLWOR for complex queries.

Integration: Databases like SQL Server provide robust XML support.

6.File Organization and Indexes

Files of Records

- Databases operate on records stored in files.

- A file is a collection of pages, each containing records.

- Files must support:

- Insert, delete, modify records.

- Read a specific record (using record id).

- Scan all records (possibly with conditions).

File Organization

Three main types:

a) Heap File Organization

- Structure:Records stored in no particular order.

-Operations:

- Search (Equality/Range): Requires scanning the entire file.

- Insert: Adds records at the end of the file.

- Delete: Searches for the record and removes it.

- Best for: Scenarios where most operations involve full file scans.

b) Sequential (Sorted) File Organization

-Structure: Records stored in sorted order (based on a key).

- Operations:

- Search (Equality/Range): Efficient due to sorted order (binary search possible).

- Insert: Requires finding the correct position and shifting records.

- Delete: Searches for the record, removes it, and shifts records.

- Best for: Range queries or retrieving records in order.

c) Hashed File Organization

- Structure: Uses a hash function to map records to buckets (primary pages + overflow pages).

- Operations:

- Search (Equality): Very efficient (direct access via hash function).

- Search (Range): Requires scanning the entire file.

- Insert: Uses hash function to find the bucket and inserts the record.

- Delete: Searches for the record using the hash function and removes it.

- Best for: Equality searches (not suitable for range queries).

Indexes

- Purpose: Speed up selections on search key fields.

- Search Key: A subset of fields (not necessarily a primary key).

- Characteristics:

- Improve query performance.

- Consume additional storage space.

- May slow down inserts, updates, and deletes (due to maintenance overhead).

Alternatives for Data Entries in Indexes

1. Alternative 1: Store the actual data record with key value k.

2. Alternative 2: Store <k, rid> (record identifier of the data record).

3. Alternative 3: Store <k, list of rids> (for duplicate keys).

Types of Indexes

a) Clustered vs. Unclustered

- Clustered Index:

- Data records are physically ordered by the index key.

- Only one clustered index per table (since data can’t be sorted in multiple ways).

- Faster for range queries (records are contiguous).

- Unclustered Index:

- Data records are not physically ordered by the index key.

- Slower for range queries (records may be scattered).

b) Dense vs. Sparse

- Dense Index:

- At least one index entry per search key value.

- Alternative 1 always leads to a dense index.

- Sparse Index:

- Only some key values have index entries.

- Every sparse index is clustered.

- Smaller in size but less flexible for some optimizations.

c) Primary vs. Secondary

- Primary Index: Search key includes the primary key.

- Unique Index: Search key is a candidate key (no duplicates).

d) Composite Search Keys

- Indexes on multiple fields (e.g., `<age, salary>`).

- Supports equality queries (all fields specified) and range queries (some fields unspecified).

- Entries are sorted lexicographically.

B+ Trees

- Most widely used index structure (supports both equality and range searches).

- Structure:

- Balanced tree with leaf nodes (containing data entries) and internal nodes(containing index entries).

-Fanout (F): Number of children per node.

- Minimum occupancy: 50% (except root).

- Operations:

- Search: Starts at root, traverses to leaf.

- Insert: If leaf is full, it splits (copy-up middle key).

- Delete: If leaf underflows, redistribute or merge with siblings.

- Advantages:

- Efficient for range queries (leaf nodes are linked).

- Log F N cost for insert/delete operations.

- Example:

- Height 3 B+ tree with fanout 133 can hold ~2.3 million records.

- Height 4 can hold ~312 million records.

Hashing

- Best for equality searches (not range queries).

- Static Hashing:

- Fixed number of buckets.

- Uses overflow chains to handle collisions.

- Problems: Long overflow chains degrade performance.

- Dynamic Hashing (Extendible Hashing):

- Uses a directory to point to buckets.

- Doubles directory size when a bucket overflows.

- Global depth: Number of bits used to determine bucket location.

- Local depth: Number of bits for a specific bucket.

- Advantages:

- No overflow chains.

- Directory fits in memory for fast access.

7.Query Processing

Introduction to Query Processing

Query processing refers to the sequence of steps a Database Management System (DBMS) performs to execute a query and retrieve the requested data. The goal is to transform a high-level query (e.g., SQL) into an efficient execution plan.

Key Steps in Query Processing:

1. Parsing and Translation

- Verifies the query's syntax and semantics.

- Converts the query into an internal form (e.g., relational algebra expression, query tree, or query graph).

- Example:

SELECT s.sname FROM S, SP WHERE S.sno = SP.sno AND SP.pno = 'P2'

Translates to:

*π*sname​(*σ*SP.pno=’P2’∧S.sno=SP.sno​(*S*×*SP*))

2. Optimization

- Generates an efficient execution strategy.

- Uses heuristic rules and cost-based techniques to minimize resource usage (e.g., disk I/O).

3. Evaluation

- The evaluation engine executes the optimized plan to produce the query result.

Query Optimization

Optimization is critical because the same query can be executed in many ways, with vastly different costs.

Why Optimize?

- Example: Computing a Cartesian product followed by a selection is inefficient.

- Cost without optimization: ~3,000,000 disk I/Os (for 100-page S and 10,000-page SP).

- Optimized cost: ~10,100 disk I/Os (by pushing selections early and avoiding Cartesian products).

Heuristic Optimization

- Applies rules to simplify and improve query plans:

- Cascade of selections:

*σc*1∧*c*2​(*E*)=*σc*1​(*σc*2​(*E*))

- Commutativity of selections:

*σc*1​(*σc*2​(*E*))=*σc*2​(*σc*1​(*E*))

- Push selections down to reduce intermediate result sizes.

- Avoid Cartesian products by using joins directly.

Equivalence Rules

- Transform relational algebra expressions into equivalent but more efficient forms.

- Example: Combining selections with joins:

*σθ*​(*E*1​×*E*2​)=*E*1​⋈*θ*​*E*2

Cost Estimation and Execution Plans

The optimizer estimates the cost of different plans to choose the best one.

Cost Metrics

-Disk I/O: Primary cost metric (number of pages read/written).

- Statistics Used:

- Number of tuples (`NTuples`) and pages (`NPages`) for each relation.

- Index details (e.g., height, distinct keys).

Algorithms for Operations

(a) Selection Operations

- No index, unsorted data: Full scan (cost = `NPages`).

- With index:

- Clustered index: ~1 I/O per qualifying page.

- Unclustered index: Up to 1 I/O per qualifying tuple.

(b) Join Operations

1. Nested Loops Join

- Simple: For each tuple in R, scan all of S (cost: M+(pR×M×N)*M*+(*pR*​×*M*×*N*)).

- Page-oriented: For each page of R, scan all pages of S (cost: M+(M×N)*M*+(*M*×*N*)).

- Block-oriented: Uses buffer space to reduce I/O (cost: M+(⌈M/B⌉×N)*M*+(⌈*M*/*B*⌉×*N*), where *B* is block size).

2. Index Nested Loops Join

- Uses an index on the join column of the inner relation.

- Cost: *M*+(*M*×cost of probing index).

3. Sort-Merge Join

- Sorts R and S on the join column, then merges them.

- Cost: *O*(*M*log*M*+*N*log*N*)+(*M*+*N*).

Cost-Based Optimization

- Compares alternative plans by estimating their costs.

- Considers:

- Size of intermediate results.

- Available indexes.

- Operator implementations (e.g., join algorithms).

Reduction Factors (RF)

- Estimate how predicates reduce result size:

- For col = value: RF=1/NKeys(I)*RF*=1/NKeys(I) (if index exists).

- For col1 = col2: RF=1/max⁡(NKeys(I1),NKeys(I2))*RF*=1/max(NKeys(I1),NKeys(I2)).

- For col > value: RF=(High(I)−value)/(High(I)−Low(I))*RF*=(High(I)−value)/(High(I)−Low(I)).

8.Query Optimization

Introduction to Query Optimization

Query optimization is a crucial step in database systems that aims to find the most efficient way to execute a given query. The process involves several stages:

1. Cast to relational algebra expression: Convert the SQL query into a formal relational algebra expression.

2. Convert to an efficient relational algebra expression: Apply equivalence rules and heuristics to optimize the expression.

3. Choose low-level procedures: Select efficient algorithms for operations like selections, projections, and joins.

Join Algorithms

Nested Loop Join

A fundamental join algorithm that compares each tuple from one relation with every tuple from another relation.

Types:

1. Simple Nested Loop Join:

- For each tuple in R, scan all tuples in S.

- Cost: M+M×P×N*M*+*M*×*P*×*N* IOs (where M*M* and N*N* are pages in R and S, respectively, and P*P* is tuples per page in R).

- Minimum buffer: 3 pages.

2. Page-Oriented Nested Loop Join:

- For each page in R, scan all pages in S.

-Cost: M+M×N*M*+*M*×*N* IOs.

- Minimum buffer: 3 pages.

3. Block-Oriented Nested Loop Join:

- For each block of R (using buffer), scan all pages of S.

- Cost: M+⌈M/B−2⌉×N*M*+⌈*B*−2*M*​⌉×*N* IOs (where *B* is buffer size).

- Minimum buffer: 3 pages.

Optimization: Choosing the smaller relation as the outer relation reduces cost.

Index Nested Loop Join

Utilizes an index on the join column of the inner relation to improve performance.

Cost Formula:

Cost=*M*+*M*×*P*×(Cost of getting matching S tuples)

Factors Affecting Cost:

- Type of index (B+ tree or hash).

- Clustering of the index.

- Number of matching tuples for each outer tuple.

Examples:

- Clustered Hash Index: Cost of probing S index is ~1.2 IOs.

- Unclustered B+ Tree Index: Cost depends on height of tree and number of matching tuples.

Sort-Merge Join

Joins relations by first sorting them on the join column and then merging.

Steps:

1. Sort R and S on the join column.

2. Merge the sorted relations to find matching tuples.

Cost:

Cost=Sort R+Sort S+Merge R and S

Best Case: Each R page matches one S page during merge.

Cost=4×*M*+4×*N*+(*M*+*N*)

Worst Case: Every tuple in R matches all tuples in S.

Cost=4×*M*+4×*N*+*M*×*N*

Buffer Requirement: Minimum buffer size is R+1*R*​+1

External Merge Sort

Used for sorting large relations that do not fit in memory.

Two-Pass Algorithm:

1. Pass 0:

- Read relation in chunks of B pages.

- Sort each chunk in memory.

- Write sorted chunks to disk.

- Cost: 2×R2×*R* IOs

2. Pass 1:

- Merge all sorted chunks.

- Write the final sorted relation to disk.

- Cost: 2×R2×*R* IOs.

Total Cost:  4×R4×*R* IOs.

**Buffer Requirement**: R+1*R*​+1 pages.

Aggregate Operations

Without Grouping

- Requires scanning the relation.

- If an index includes all attributes in SELECT or WHERE clauses, an index-only scan can be used.

With Grouping

1. Sort-Based Approach:

- Sort the relation on group-by attributes.

- Compute aggregates for each group during the final pass of sorting.

- Cost: Sort cost plus scan cost.

2. Hash-Based Approach:

- Hash tuples on group-by attributes.

- Compute aggregates for each bucket.

3. Index-Only Scan:

- If the index includes all attributes in SELECT, WHERE, and GROUP BY clauses, use the index to retrieve tuples in group-by order.

Query Plan Generation

Steps

1. Single-Relation Plans:

- Evaluate all access paths (file scan, index scan) for each relation.

- Choose the path with the least cost.

- Combine operations (e.g., selection, projection) into a pipeline.

2. Multiple-Relation Plans:

- Pass 1: Find the best plan for each relation individually.

- Pass 2: Find the best way to join the results of the 1-relation plans.

- Retain the cheapest overall plan and plans with interesting orders (for ORDER BY, GROUP BY).

Cost Estimation

- Result Cardinality: Product of relation sizes multiplied by reduction factors (RF) of selection conditions.

- Join Cost: Sum of individual operation costs (selection, join, projection).

Example

- Relations: S (with B+ tree on status, hash on sno) and SP (with B+ tree on pno).

- Query: Join S and SP on sno, with selections on pno='P5'and status > 50.

- Plan Options:

- Index scan on SP with B+ tree, then nested loop join with S.

- Index scan on SP with B+ tree, then index nested loop join using hash index on S<sno>.

9.Transactions and Concurrency Control

Introduction to Transactions

A transaction is a sequence of operations (reads and writes) performed by a user program that the DBMS treats as a single logical unit of work.

Example: Transferring money from Account A to Account B involves:

- Reading balances (A and B).

- Updating balances (A = A - amount, B = B + amount).

Key Properties of Transactions (ACID)

| Property | Description |

|----------|-------------|

| Atomicity | A transaction is executed entirely or not at all. If it fails midway, all changes are rolled back. |

| Consistency | A transaction brings the database from one valid state to another, preserving integrity constraints. |

| Isolation | Concurrent transactions do not interfere with each other; execution appears serial. |

| Durability | Once committed, a transaction’s changes persist even after system failures. |

Transaction Execution and Schedules

Actions in a Transaction

- Read (R): Retrieve data from the database.

- Write (W): Modify data in the database.

- Commit (C): Finalize changes permanently.

- Abort (A): Undo all changes made by the transaction.

Notation:

- *RT*​(*O*): Transaction T reads object O.

- *WT*​(*O*): Transaction T writes object O.

Schedules

A schedule is an ordered sequence of actions (reads, writes, commits, aborts) from multiple transactions.

- Serial Schedule: No interleaving of transactions (one completes before the next starts).

- Serializable Schedule: A schedule whose effect is equivalent to some serial execution.

Example Schedule:

| T1 | T2 |

|----------|----------|

| R(A) | |

| W(A) | R(A) |

| R(B) | W(A) |

| W(B) | Commit |

| Commit | |

Concurrency Control

Why Allow Concurrent Transactions?

- Performance: Overlap I/O and CPU processing (e.g., while T1 waits for disk, T2 uses CPU).

- Fairness: Short transactions shouldn’t wait for long ones.

Problems with Interleaved Execution (Conflicts)

| Conflict Type | Description | Example |

|--------------|-------------|---------|

| WR (Dirty Read) | T2 reads data written by uncommitted T1. | T1 writes A, T2 reads A, T1 aborts → T2 reads invalid data. |

| RW (Unrepeatable Read) | T1 reads data twice, but T2 modifies it in between. | T1 reads A, T2 writes A, T1 reads A again (different value). |

| WW (Overwrite) | T1 and T2 write the same object; one overwrites the other. | T1 sets A=100, T2 sets A=200 → Final value depends on order. |

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)

- Phase 1 (Growing): Acquire locks (no releases).

- Phase 2 (Shrinking): Release locks (no acquisitions).

- Lock Types:

- Shared (S): For reads (multiple transactions can hold S-locks).

- Exclusive (X): For writes (only one transaction can hold X-lock).

Rules:

1. Request S-lock before reading, X-lock before writing.

2. Hold all locks until commit/abort.

Advantages:

- Ensures serializable schedules.

- Prevents dirty reads, unrepeatable reads, and overwrites.

Transaction Failures and Recovery

Cascading Aborts

- Problem: Abort of T1 forces abort of T2 if T2 read data written by T1.

- Solution: Strict 2PL prevents this by holding locks until commit.

Unrecoverable Schedules

- Problem: T2 commits before T1 (which wrote data read by T2) aborts.

- Solution: Strict 2PL ensures T1 commits before T2.

The Log

- Records all writes (old/new values), commits, and aborts.

- Used for:

- Undo: Roll back aborted transactions.

- Redo: Reapply committed transactions after a crash.

Advanced Topics

Deadlocks

- Definition: Cycle of transactions waiting for each other’s locks.

- Solutions:

- Prevention: Wait-Die or Wound-Wait policies.

- Detection: Waits-for graph + victim selection.

Phantom Problem

- Issue: New records appear in a range query due to concurrent inserts.

- Solution: Predicate locking or index locking.

Multiple Granularity Locking

- Locks can be applied at different levels (database, table, page, row).

- Intention Locks:

- IS (Intent Shared): Indicates S-locks may be requested at a lower level.

- IX (Intent Exclusive): Indicates X-locks may be requested.

- SIX (Shared+IX): Combines S and IX.

SQL Transaction Control

Isolation Levels

| Level | Dirty Reads | Unrepeatable Reads | Phantoms |

|--------------------|-------------|---------------------|----------|

| READ UNCOMMITTED | Yes | Yes | Yes |

| READ COMMITTED | No | Yes | Yes |

| REPEATABLE READ | No | No | Yes |

| SERIALIZABLE | No | No | No |

Example:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

-- SQL operations

COMMIT;