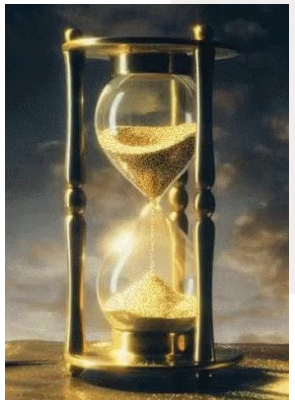
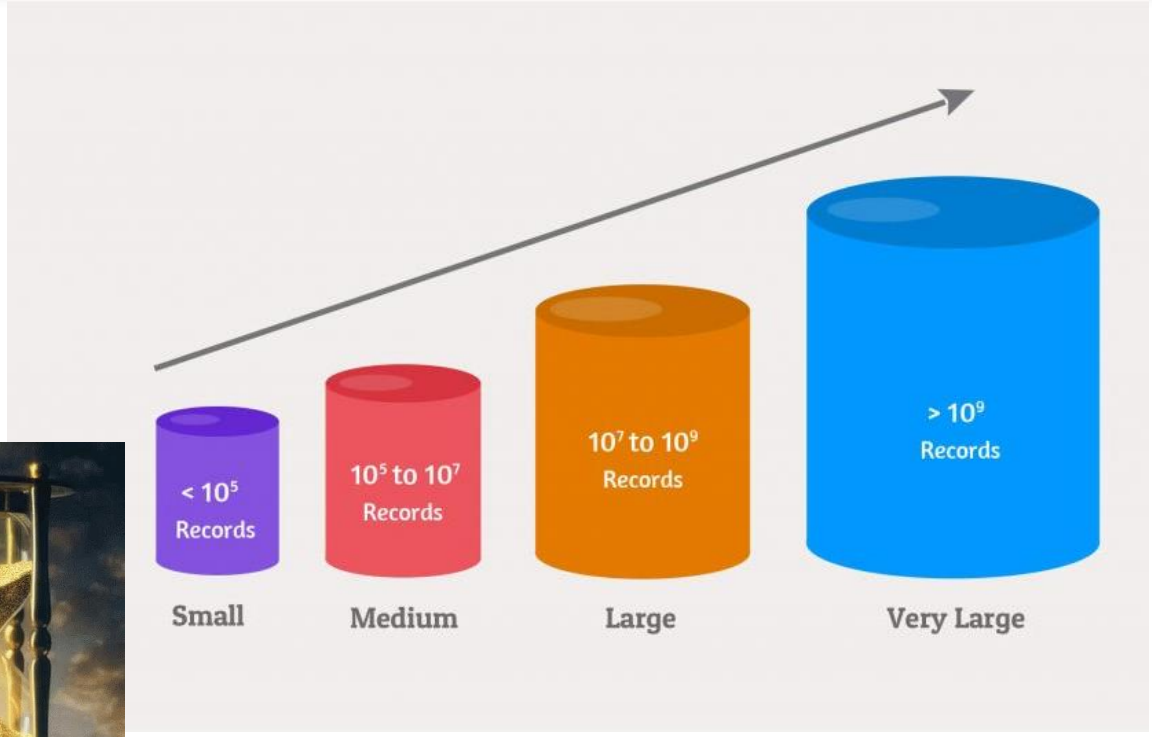
Performance Tuning

CS 341 Database Systems

SQL Queries in the real world

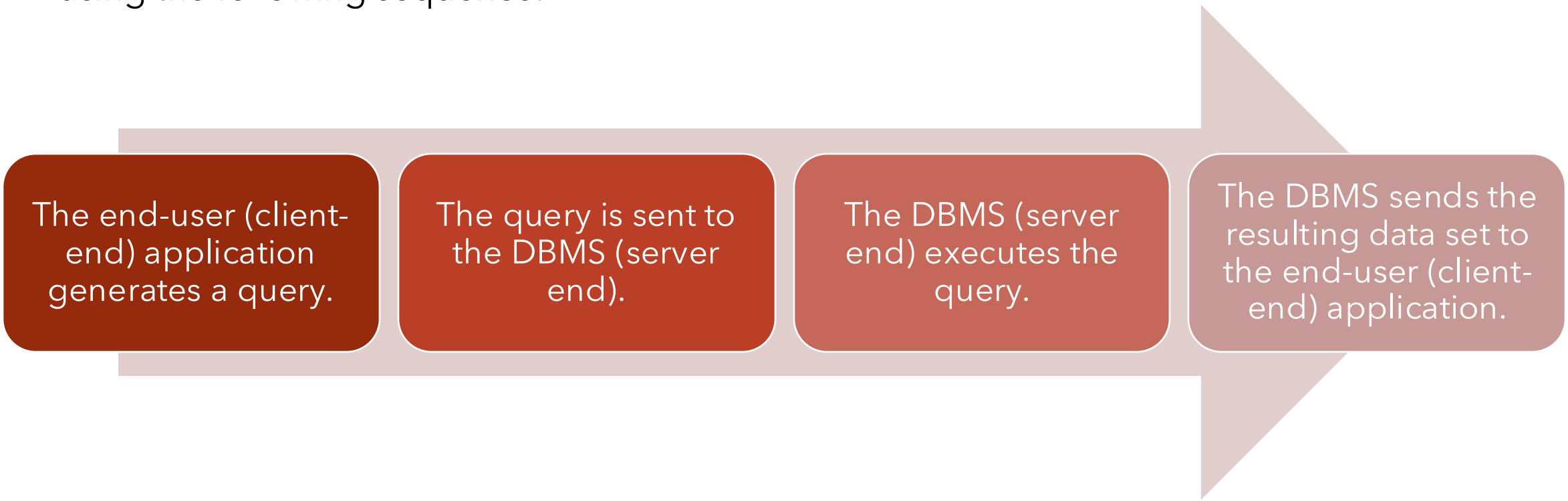
- Data in the real world is vast as opposed to small databases used for academic learning
- Lack of attention to query efficiency can yield unacceptably **slow results** in the real world when queries are executed over tens of millions of records.





End Users ⟷ DBMS

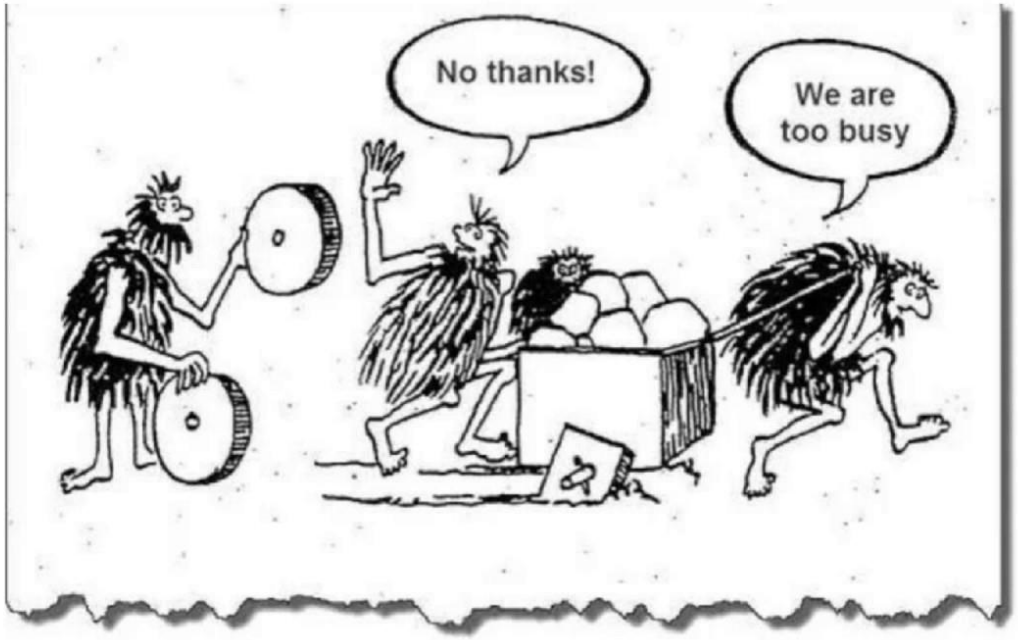
One of the main functions of a database system is to provide **timely** answers to end users. End users interact with the DBMS through queries to generate information, using the following sequence:



DB Performance Tuning

Database performance tuning refers to a set of activities and procedures designed ***to reduce the response time*** *of the database system–*

To ensure that an end-user query is processed by the DBMS in the minimum amount of time.



Good database Performance *begins with a* ***Good database Design***

No amount of fine-tuning will make a poorly designed database perform as well as a well-designed database.

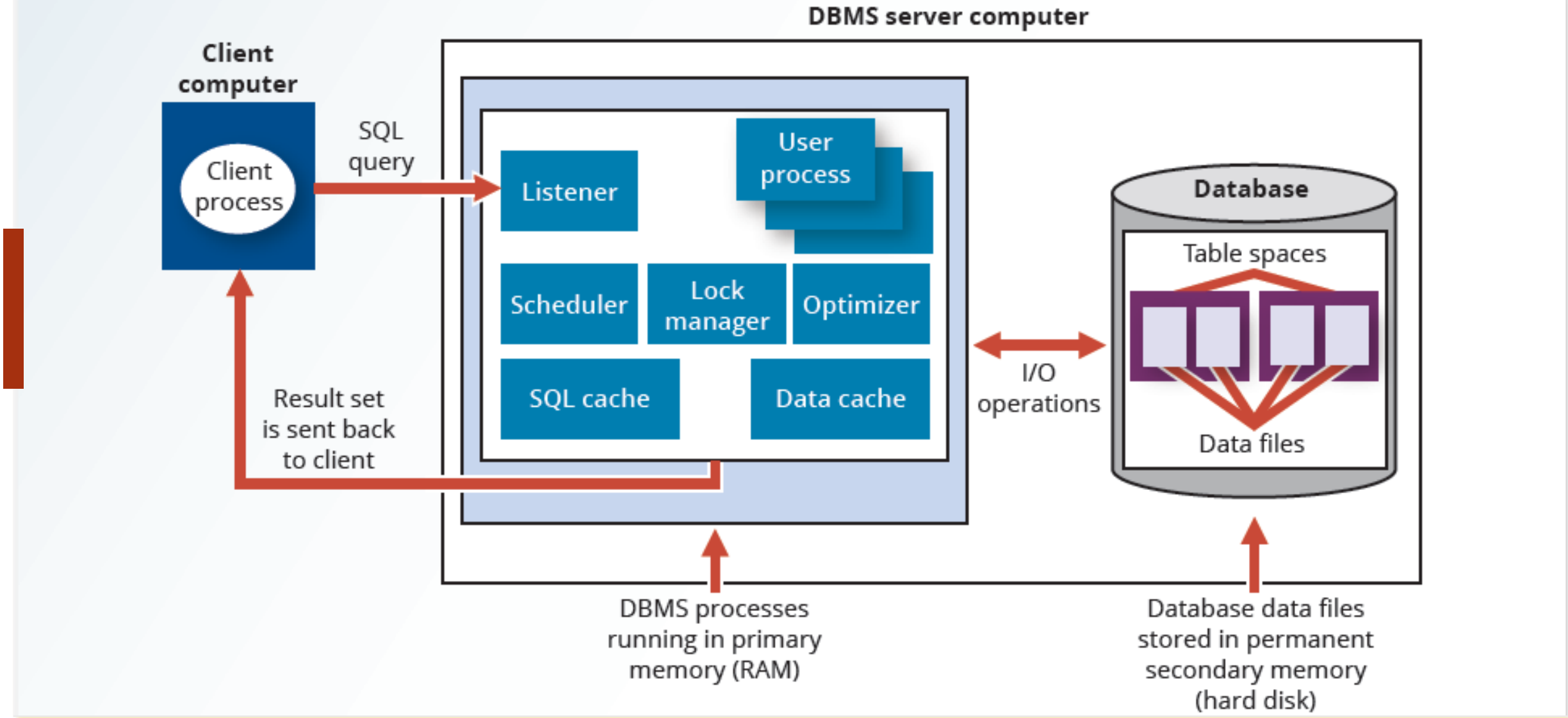
Poor Design Consequences:

- **Sluggish Performance:** Even with optimization, complex queries and joins can significantly slow down the system.
- **Data Anomalies:** Leads to errors and inconsistencies during operations.
- **Hard to Scale:** Modifying a poorly designed structure often results in more complications.

Performance Tuning Activities:
Client and Server

SQL Performance Tuning	DBMS Performance Tuning
Client side	Server side
SQL query returns the correct answer in the least amount of time, using the minimum amount of resources at the server end.	DBMS environment must be properly configured to respond to clients' requests in the fastest way possible, while making optimum use of existing resources.

Figure 11.1 Basic DBMS Architecture



Database Query Optimization Modes

- Most of the algorithms proposed for query optimization are based on two principles:
  - The *selection of the optimum execution order* to achieve the *fastest execution time*
  - The selection of sites to be accessed to *minimize communication costs*
- Within those two principles, a query optimization algorithm can be evaluated based on its **operation mode** or the **timing** of its optimization.

Operation Modes

- **Automatic query optimization**
  - means that the DBMS finds the most cost-effective access path without user intervention.
- **Manual query optimization**
  - requires that the optimization be selected and scheduled by the end user or programmer.
- Automatic query optimization is clearly more desirable from the end user's point of view, but the cost of such convenience is the increased overhead that it imposes on the DBMS.

Timing Classification

Static query optimization

- Takes place at *compilation time*.
- Best optimization strategy is selected when the query is compiled by the DBMS.

Dynamic query optimization

- Takes place at *execution time*.
- Strategy dynamically determined by the DBMS at run time, using the most up-to-date information about the database.

Optimization Algorithm

A statistically based query optimization algorithm:

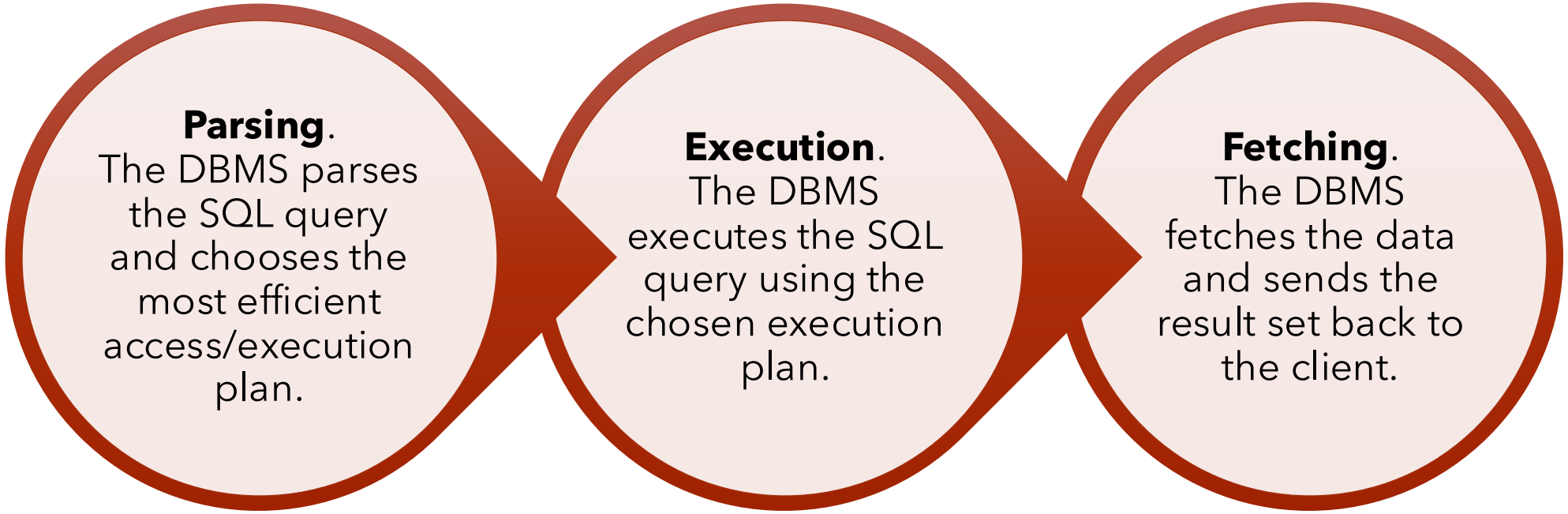
- Uses statistical information about the database to determine best access strategy.
- Database statistics: size, number of records, average access time, number of requests serviced, and number of users with access rights.

A rule-based query optimization algorithm

- Is based on a set of user-defined rules to determine the best query access strategy.
- The rules are entered by the end user or database administrator, and they are typically general in nature.

3 Phases of Query Processing

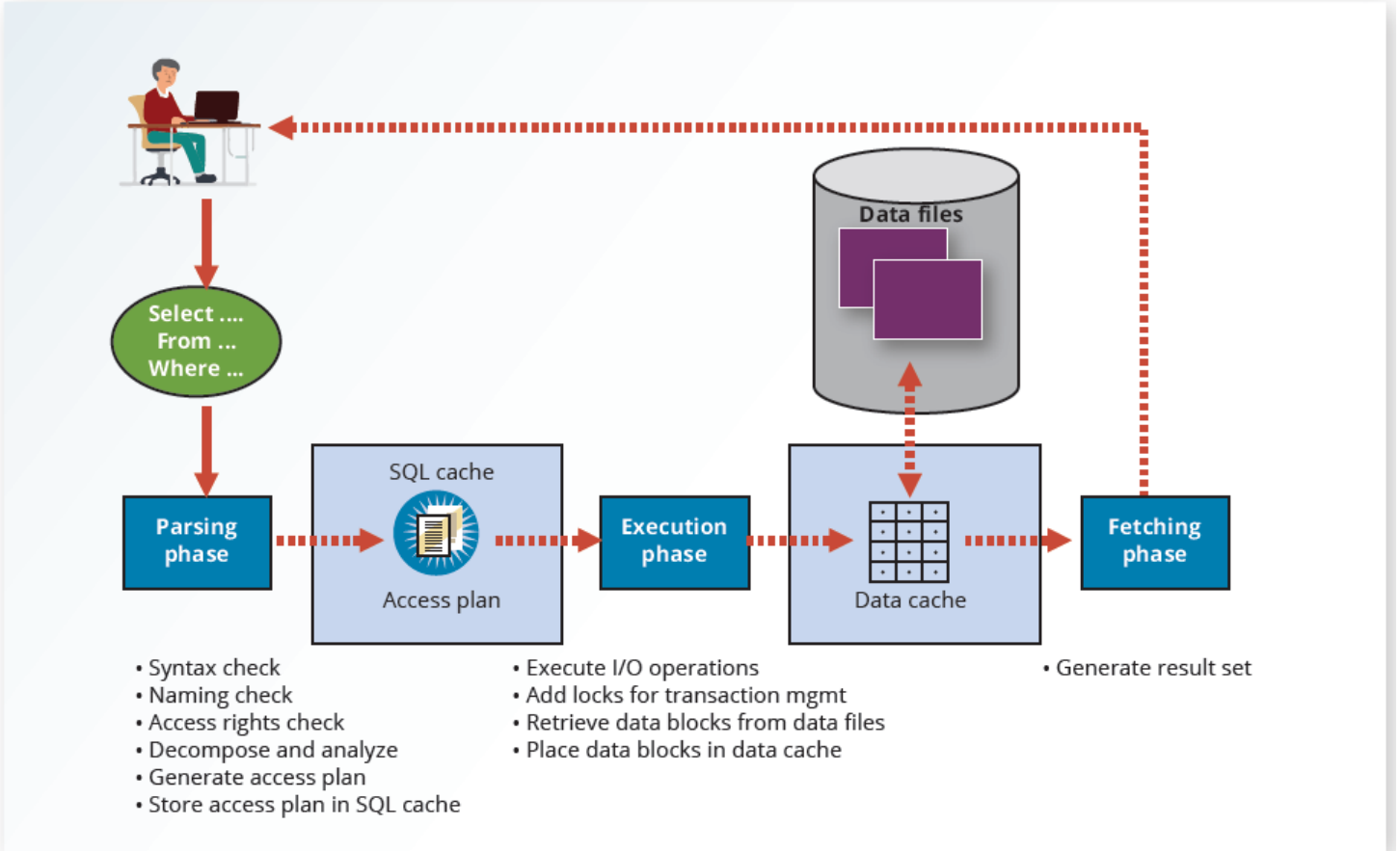
Server response to client's SQL statement



Query Processing

- Optimization Process: break query into smaller units transforming into ***fully equivalent*** and ***more efficient*** SQL code.
- The SQL parsing activities are performed by the **query optimizer**, which analyzes the SQL query and finds the most efficient way to access the data. This process is the most time-consuming phase in query processing.

Figure 11.2 Query Processing



Optimizer Choices

A rule-based optimizer

- uses preset rules and points to determine the best approach to execute a query.
- The rules assign a "fixed cost" to each SQL operation; the costs are then added to yield the cost of the execution plan.
- For example, a full table scan has a set cost of 10, while a table access by row ID has a set cost of 3.

A cost-based optimizer

- uses sophisticated algorithms based on statistics about the objects being accessed to determine the best approach to execute a query.
- In this case, the optimizer process adds up the processing cost, the I/O costs, and the resource costs (RAM and temporary space) to determine the total cost of a given execution plan.

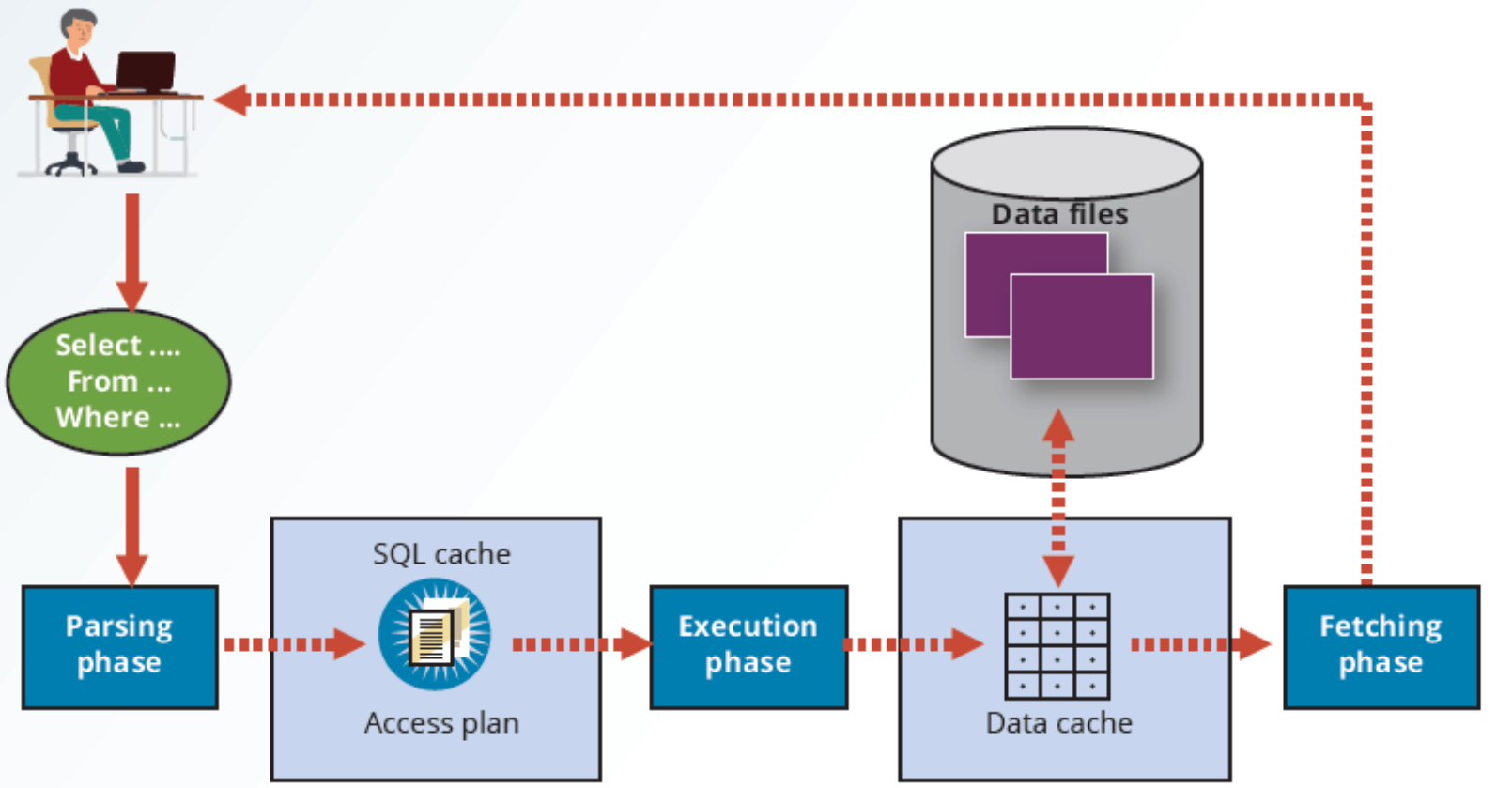
Access Plan

- Access plans are DBMS-specific and translate the client's SQL query into the series of complex I/O operations required to read the data from the physical data files and generate the result set.

Table 11.3 Sample DBMS Access Plan I/O Operations

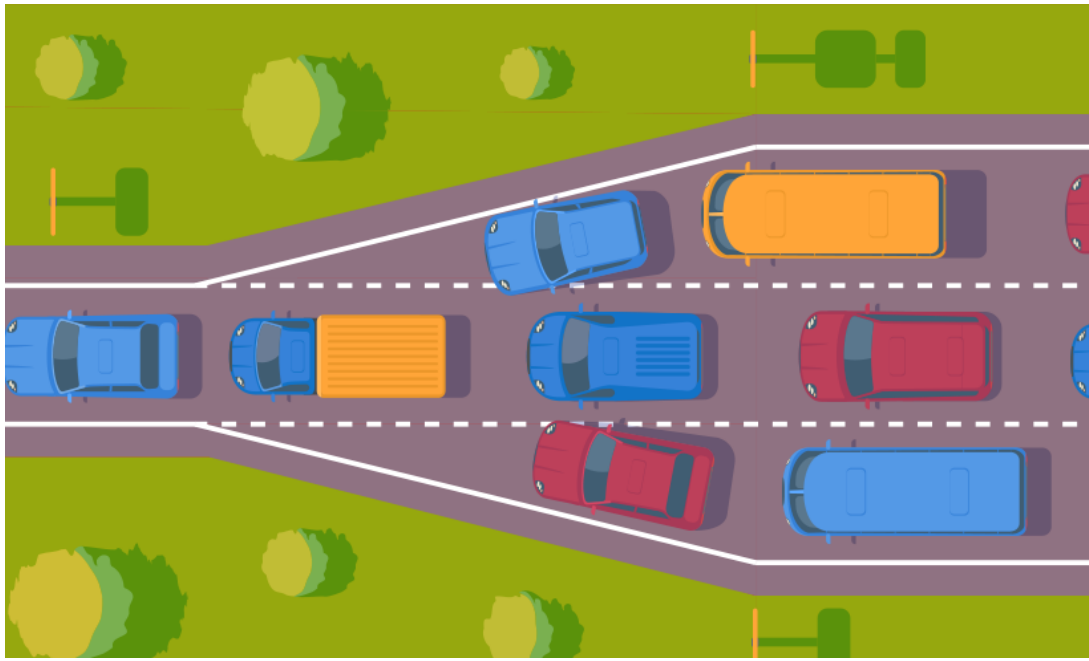
Operation	Description
Table scan (full)	Reads the entire table sequentially, from the first row to the last, one row at a time (slowest)
Table access (row ID)	Reads a table row directly, using the row ID value (fastest)
Index scan (range)	Reads the index first to obtain the row IDs and then accesses the table rows directly (faster than a full table scan)
Index access (unique)	Used when a table has a unique index in a column
Nested loop	Reads and compares a set of values to another set of values, using a nested loop style (slow)
Merge	Merges two data sets (slow)
Sort	Sorts a data set (slow)

Figure 11.2 Query Processing



Bottlenecks

- **Bottlenecks** are the result of multiple database transactions competing for the use of database resources (CPU, RAM, hard disk, indexes, locks, buffers, etc.).
- Most typical bottlenecks is caused by transactions competing for the
  - **same data rows.**
  - **shared memory resources**, particularly shared buffers and locks.



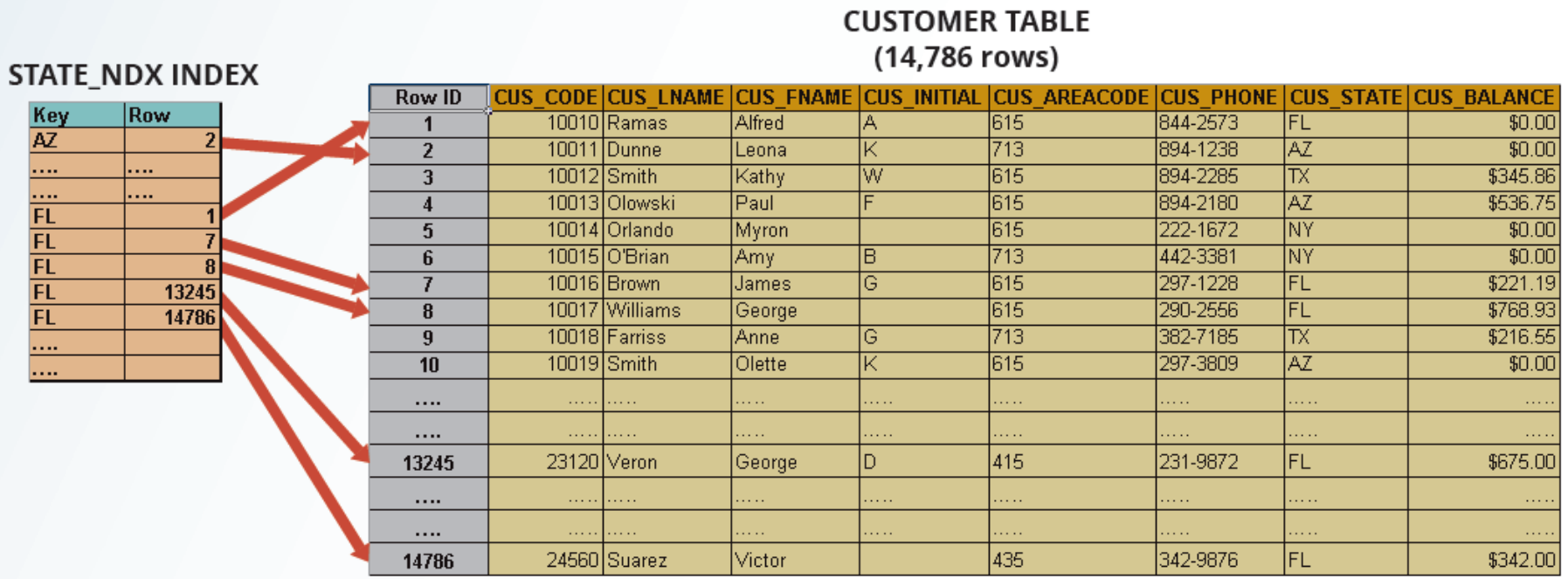
Indexes and Query Optimization

- Indexes → Speed up data access by facilitating search, sort, and use of aggregate functions or even join operations.

An index is an ordered set of values that contains the index key and pointers.

  - The pointers are the row IDs for the actual table rows.
  - An index scan is more efficient than a full table scan because the index data is preordered, and the amount of data is usually much smaller.

Figure 11.3 Index Representation for the Customer Table



Key	Row
AZ	2
....	....
....	....
FL	1
FL	7
FL	8
FL	13245
FL	14786
....	
....	

Row ID	CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_STATE	CUS_BALANCE
1	10010	Ramas	Alfred	A	615	844-2573	FL	$0.00
2	10011	Dunne	Leona	K	713	894-1238	AZ	$0.00
3	10012	Smith	Kathy	W	615	894-2285	TX	$345.86
4	10013	Olowski	Paul	F	615	894-2180	AZ	$536.75
5	10014	Orlando	Myron		615	222-1672	NY	$0.00
6	10015	O'Brian	Amy	B	713	442-3381	NY	$0.00
7	10016	Brown	James	G	615	297-1228	FL	$221.19
8	10017	Williams	George		615	290-2556	FL	$768.93
9	10018	Farriss	Anne	G	713	382-7185	TX	$216.55
10	10019	Smith	Olette	K	615	297-3809	AZ	$0.00
....	.....	.....	.....	.....	.....	.....	.....	.....
....	.....	.....	.....	.....	.....	.....	.....	.....
13245	23120	Veron	George	D	415	231-9872	FL	$675.00
....	.....	.....	.....	.....	.....	.....	.....	.....
....	.....	.....	.....	.....	.....	.....	.....	.....
14786	24560	Suarez	Victor		435	342-9876	FL	$342.00

Suppose you submit the following query:

```
SELECT      CUS_NAME, CUS_STATE
FROM        CUSTOMER
WHERE       CUS_STATE = 'FL';
```

Put Indexes on All Columns?

- Check for **Data sparsity**
  - number of different values a column can have.
- Gender has 2 values, M and F → Low Sparsity
- DOB has multiple values → High Sparsity
- Key tip: Avoid low sparsity columns for indexing

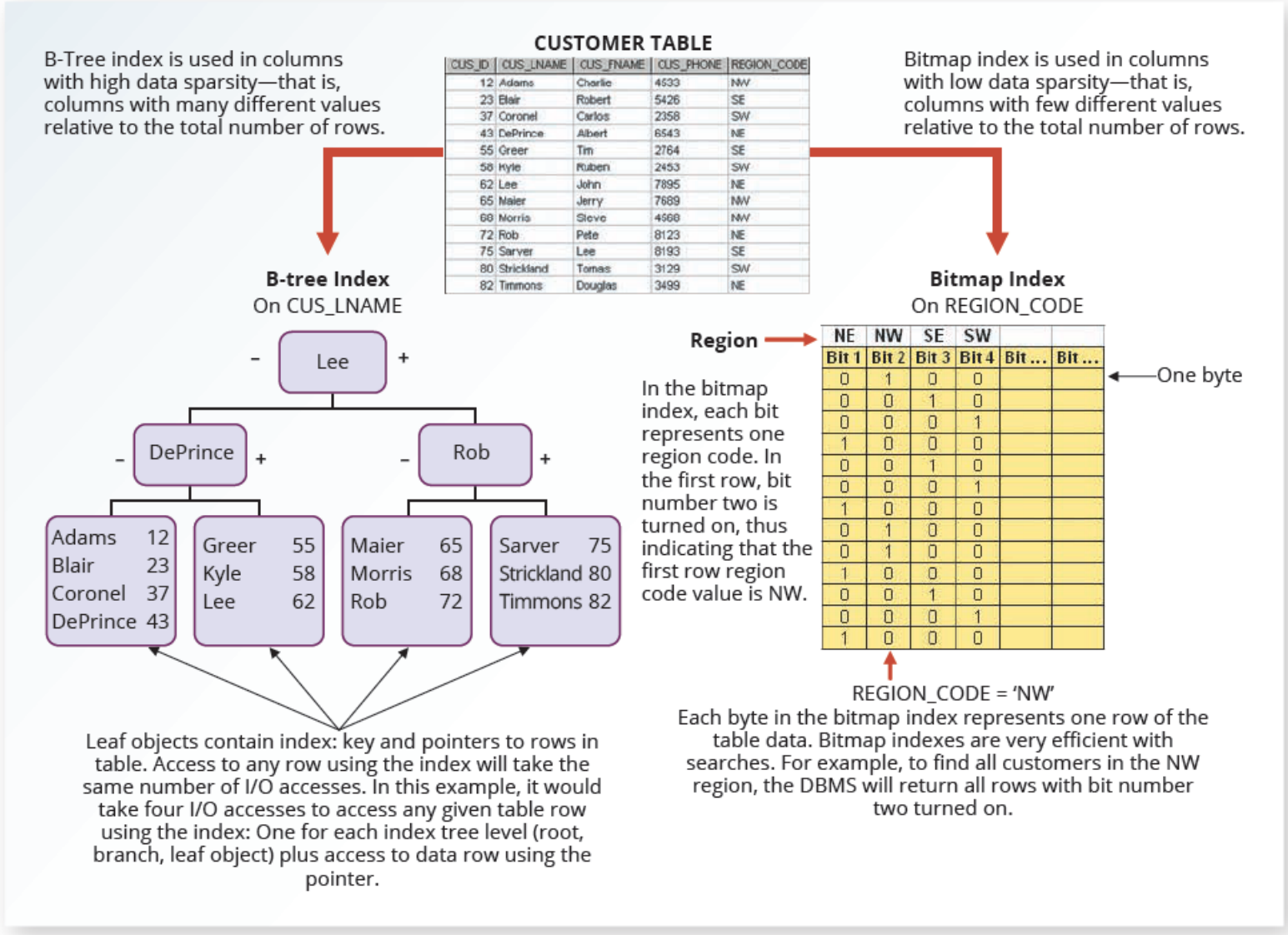
Types of Indexes used by DBMS

- **Hash index.**
  - A hash index is based on an ordered list of hash values.
  - A hash algorithm is used to create a hash value from a key column.
  - This value points to an entry in a hash table, which in turn points to the actual location of the data row.
  - This type of index is good for simple and fast lookup operations based on equality conditions

Types of Indexes used by DBMS

- **B-tree index.**
  - The B-tree index is an ordered data structure organized as an upside-down tree. The index tree is stored separately from the data. The lower-level leaves of the B-tree index contain the pointers to the actual data rows.
  - Default and most common type of index used in databases.
- **Bitmap index.**
  - A bitmap index uses a bit array (0s and 1s) to represent the existence of a value or condition. These indexes are used mostly in data warehouse applications in tables with a large number of rows in which a small number of column values repeat many times.

Figure 11.4 B-Tree and Bitmap Index Representation



List all products provided by a vendor based in Florida

```
SELECT    P_CODE, P_DESCRIPT, P_PRICE, V_NAME, V_STATE
FROM      PRODUCT, VENDOR
WHERE     PRODUCT.V_CODE = VENDOR.V_CODE AND
          VENDOR.V_STATE = 'FL';
```

Database Statistics:

- PRODUCT table has 7000 rows
- VENDOR table has 300 rows
- 10 Vendors are located in ‘Florida’
- 1000 products come from vendors in ‘Florida’

2 Different Access Plans

Which plan should the optimizer choose?

Table 11.4 Comparing Access Plans and I/O Costs

Plan	Step	Operation	I/O Operations	I/O Cost	Resulting Set Rows	Total I/O Cost
A	A1	Cartesian product (PRODUCT, VENDOR)	7,000 + 300	7,300	2,100,000	7,300
	A2	Select rows in A1 with matching vendor codes	2,100,000	2,100,000	7,000	2,107,300
	A3	Select rows in A2 with V_STATE = 'FL'	7,000	7,000	1,000	**2,114,300**
B	B1	Select rows in VENDOR with V_STATE = 'FL'	300	300	10	300
	B2	Cartesian Product (PRODUCT, B1)	7,000 + 10	7,010	70,000	7,310
	B3	Select rows in B2 with matching vendor codes	70,000	70,000	1,000	**77,310**

Operations and I/O Cost columns in estimate only the number of I/O disk reads the DBMS must perform. For simplicity's sake, it is assumed that there are no indexes and that each row read has an I/O cost of 1.

Indexes to the Rescue

Some queries answered entirely by using only the index

- PRODUCT table and the index P_QOH_NDX in the P_QOH attribute.
- **SELECT MIN(P_QOH) FROM PRODUCT**
- *This query could be resolved by reading only the first entry in the P_QOH_NDX index, without the need to access any of the data blocks for the PRODUCT table*.
- (Remember that the index defaults to ascending order.)

Sometimes Low Sparsity Index can be helpful

- Assume in a table of 130,000 employees, gender is recorded
- SELECT COUNT(GENDER) FROM EMPLOYEE WHERE GENDER = 'F';
- No index → full table scan
- Index → read only index data without accessing the employee data at all

Index by example

Test it out!

Suppose you often run the following query to get employees with salary in a specific range

```
1  SELECT employee_id, first_name, last_name, salary
2  FROM employees
3  WHERE salary > 5000;
4
```

Query Result

SQL | Fetched 50 rows in 0.057 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
1	201	Michael	Hartstein	13000
2	202	Pat	Fay	6000
3	203	Susan	Mavris	6500
4	204	Hermann	Baer	10000
5	205	Shelley	Higgins	12008
6	206	William	Gietz	8300

```
10 EXPLAIN PLAN FOR
11 SELECT employee_id, first_name, last_name, salary
12 FROM employees
13 WHERE salary > 5000;
14
15 SELECT * FROM table(DBMS_XPLAN.DISPLAY());
```

Script Output | Query Result

SQL | All Rows Fetched: 13 in 0.172 seconds

```
PLAN_TABLE_OUTPUT
1  Plan hash value: 1445457117
2
3  ----------------------------------------------------------------------------
4  | Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
5  ----------------------------------------------------------------------------
6  |   0 | SELECT STATEMENT   |           |    57 |  1311 |     3   (0)| 00:00:01 |
7  |*  1 |  TABLE ACCESS FULL| EMPLOYEES |    57 |  1311 |     3   (0)| 00:00:01 |
8  ----------------------------------------------------------------------------
9
10 Predicate Information (identified by operation id):
11 ---------------------------------------------------
12
13    1 - filter("SALARY">5000)
```

```sql
CREATE INDEX idx_emp_salary
ON employees (salary);
```

```sql
35 -- After index monitor
36 EXPLAIN PLAN FOR
37 SELECT employee_id, first_name, last_name, salary
38 FROM employees
39 WHERE salary > 5000;
40
41 SELECT * FROM table(DBMS_XPLAN.DISPLAY());
```

Script Output | Query Result | Query Result 1 | Query Result 2 | Query Result 3 | Query Result 4

SQL | All Rows Fetched: 13 in 0.011 seconds

PLAN_TABLE_OUTPUT

```
Plan hash value: 1445457117

-----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    57 |  1311 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |    57 |  1311 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

NOTE:
DBMS preferred to do a full table scan as there were already too many rows being scanned hence index was not helpful

What the salaries look like

```
42
43 select salary from employees order by salary;
44
```

Script Output | Query Result | Query Result 1 | Query Result 2 | Query Result 3 | Query Result 4

SQL | Fetched 50 rows in 0.005 seconds

	SALARY
1	2100
2	2200
3	2200
4	2400
5	2400
6	2500
7	2500
8	2500
9	2500
10	2500
11	2500
12	2600
13	2600
14	2600
15	2600
16	2700
17	2700
18	2800

- Many in range specified "> 5000"
- Let's try salary "< 2500" → few people

Without Index

```
39 EXPLAIN PLAN FOR
40 SELECT employee_id, first_name, last_name, salary
41 FROM employees
42 WHERE salary < 2500;
43
44 SELECT * FROM table(DBMS_XPLAN.DISPLAY());
45
```

Script Output | Query Result | Query Result 1 | Query Result 2 | Query Result 3 | Query Result 4

SQL | All Rows Fetched: 13 in 0.069 seconds

PLAN_TABLE_OUTPUT

```
Plan hash value: 1445457117

-----------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     6 |   138 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |     6 |   138 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

Table Access by Index

Worksheet | Query Builder

```
36 -- After index monitor
37 EXPLAIN PLAN FOR
38 SELECT employee_id, first_name, last_name, salary
39 FROM employees
40 WHERE salary < 2500;
41
42 SELECT * FROM table(DBMS_XPLAN.DISPLAY());
```

Script Output | Query Result | Query Result 1 | Query Result 2 | Query Result 3 | Query Result 4

SQL | All Rows Fetched: 14 in 0.078 seconds

PLAN_TABLE_OUTPUT

```
Plan hash value: 1068959935

-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                 |     6 |   138 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES       |     6 |   138 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IDX_EMP_SALARY  |     6 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Client Side – SQL Performance Tuning

Write efficient SQL Code

- Most current-generation relational DBMSs perform *automatic query optimization* at the server end.
- The DBMS uses general optimization techniques rather than focusing on specific techniques dictated by the special circumstances of the query execution
- **There is always room for improvement as**
  Bad SQL code = Bad DBMS performance

Query Formulation

- Identify what columns and computations are required
- Identify source tables
- Determine how to join the tables
  - **Minimize joins operations**
- Determine the selection criteria needed (WHERE or HAVING)
- Determine the order to display the final output (ORDER BY)

WHERE - Conditional Expressions

- Avoid use of *functions* in conditionals →adds to query execution time
- Numeric field comparisons are faster than character, date and NULL (slowest) comparisons
- *Equality* (=) faster than Inequality (>,>=, <,<=) comparisons
- In case of multiple conditions:

1	2	3	4	5
Write equality conditions first	In case of **AND**: Write the condition most likely to be **false** first	In case of **OR**: Write the condition most likely to be **true** first	Save time on unnecessary evaluation of additional conditions	Avoid NOT logical operator or <> → slow

DBMS Performance Tuning

- Includes global tasks such as managing the DBMS processes in primary memory (allocating memory for caching purposes)
- and managing the structures in physical storage (allocating space for the data files).
- DBMS performance tuning at the **server end** focuses on setting the parameters used for: