CSCI 5408

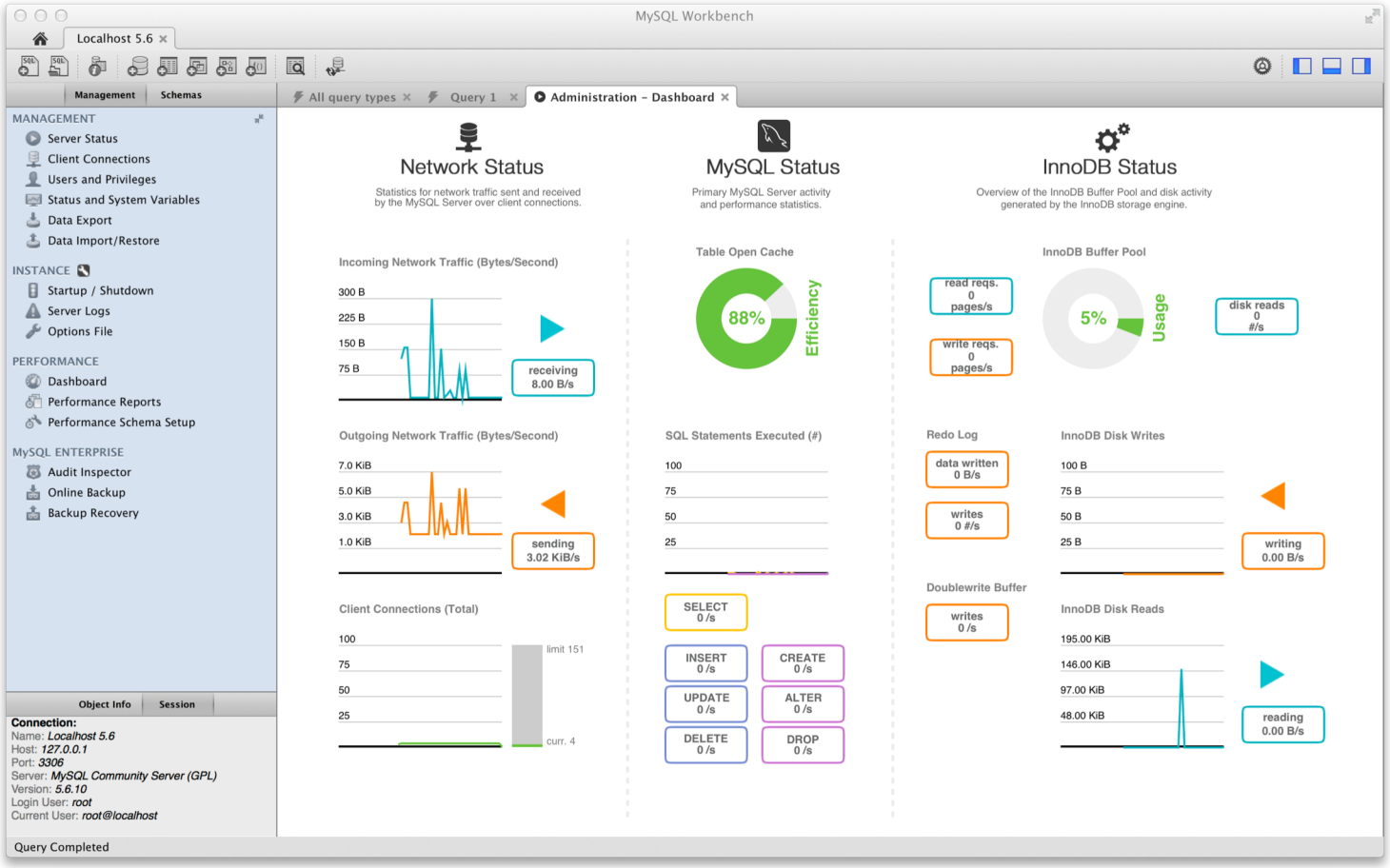
Dr. Saurabh Dey

saurabh.dey@dal.ca

Outline

- Introduction to Database performance
- Query processing
- Database performance tuning

Module 5



https://www.mysql.com/common/images/products/mysql_wb_performance_dashboard_osx.png

Database Performance

Goal of database performance is to execute queries as fast as possible

One of the main functions is to provide timely answers:

End users interact with the DBMS through the use of queries to generate information, using the following sequence:

1. End-user (client-end) application generates a query
2. Query is sent to the DBMS (server end)
3. DBMS (server end) executes the query
4. DBMS sends the resulting data set to the end-user (client-end) application

Database Performance-Tuning

Set of activities and procedures that reduce response time of database system

Fine-tuning the performance of a system requires a holistic approach

- All factors must operate at optimum level with minimal bottlenecks

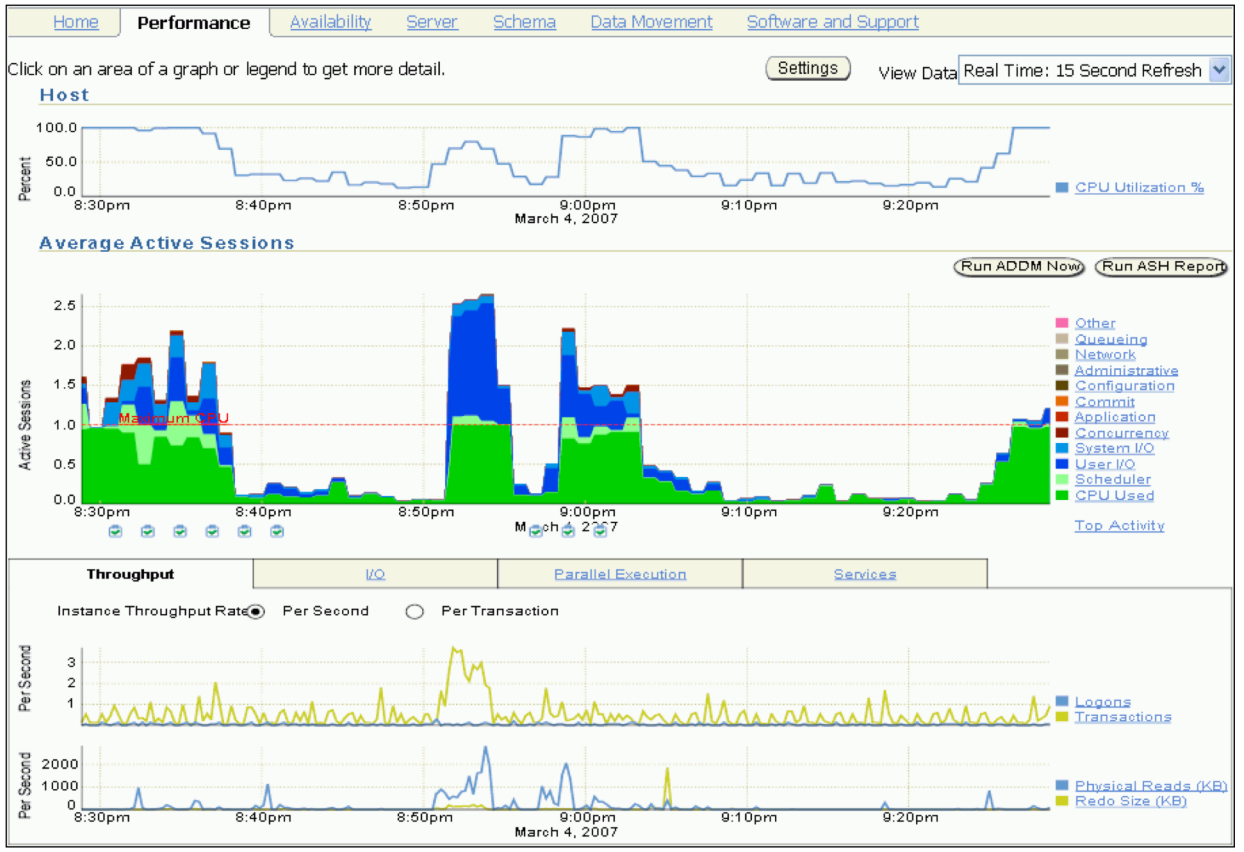


Image Source: https://docs.oracle.com/cd/B28359_01/server.111/b28275/tdppt_realtime.htm#TDPPT034

GENERAL GUIDELINES FOR BETTER SYSTEM PERFORMANCE

	SYSTEM RESOURCES	CLIENT	SERVER
Hardware	CPU	The fastest possible Dual-core CPU or higher "Virtualized Client desktop technologies could also be used."	The fastest possible Multiple processors (quad-core technology or higher) Cluster of networked computers "Virtualized server technology could be used"
	RAM	The maximum possible to avoid OS memory to disk swapping	The maximum possible to avoid OS memory to disk swapping
	Storage	Fast SATA/EIDE hard disk with sufficient free hard disk space Solid state drives (SSDs) for faster speed	Multiple high-speed, high-capacity disks Fast disk interface (SAS / SCSI / Firewire / Fibre Channel RAID configuration optimized for throughput Solid state drives (SSDs) for faster speed Separate disks for OS, DBMS, and data spaces
	Network	High-speed connection	High-speed connection
Software	Operating system (OS)	64-bit OS for larger address spaces Fine-tuned for best client application performance	64-bit OS for larger address spaces Fine-tuned for best server application performance
	Network	Fine-tuned for best throughput	Fine-tuned for best throughput
	Application	Optimize SQL in client application	Optimize DBMS server for best performance

Performance Tuning – Client and Server

Client side: SQL performance tuning

Generate SQL query that returns correct answer in least amount of time

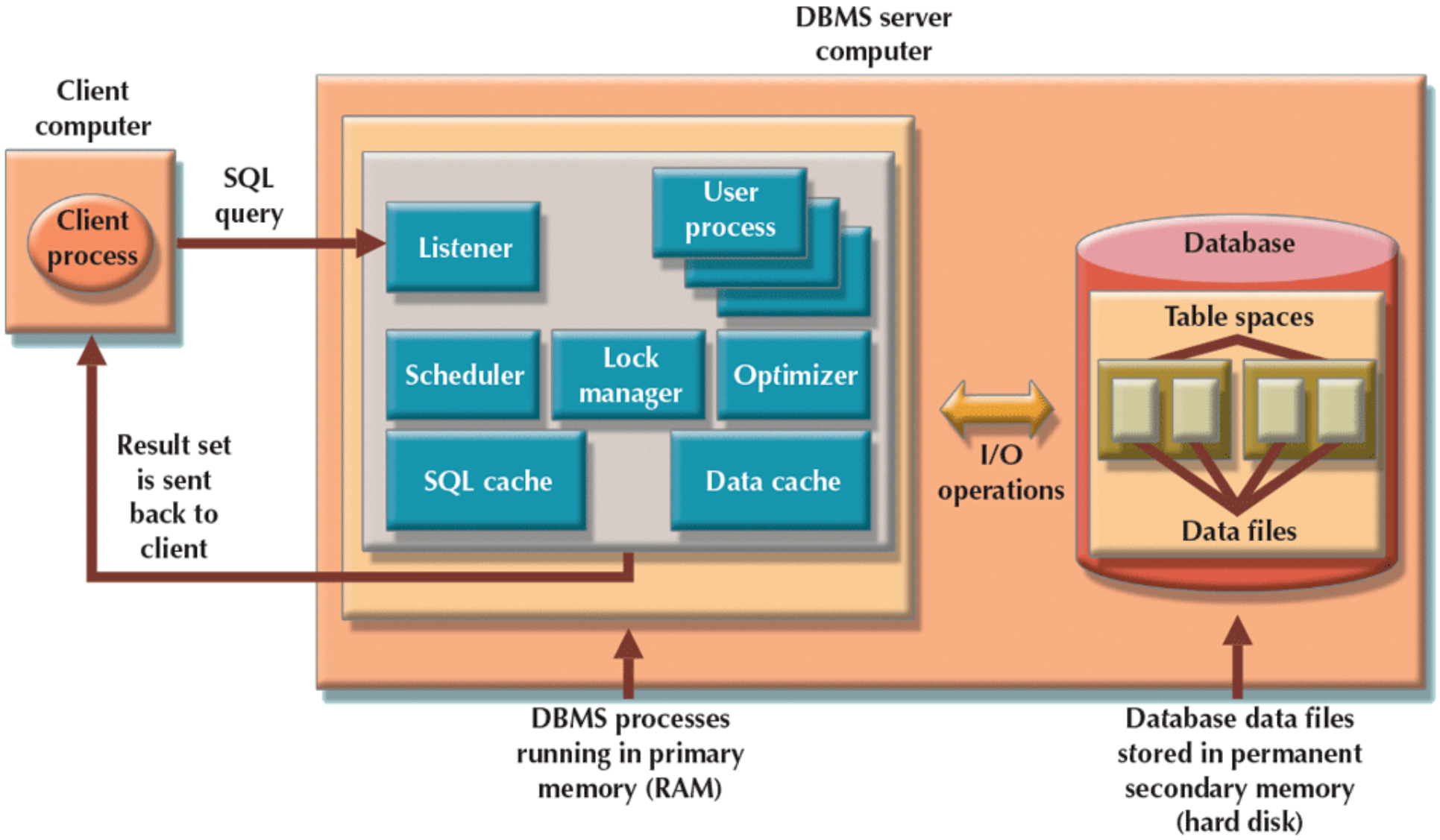
- Using minimum amount of resources at server

Server side: DBMS performance tuning

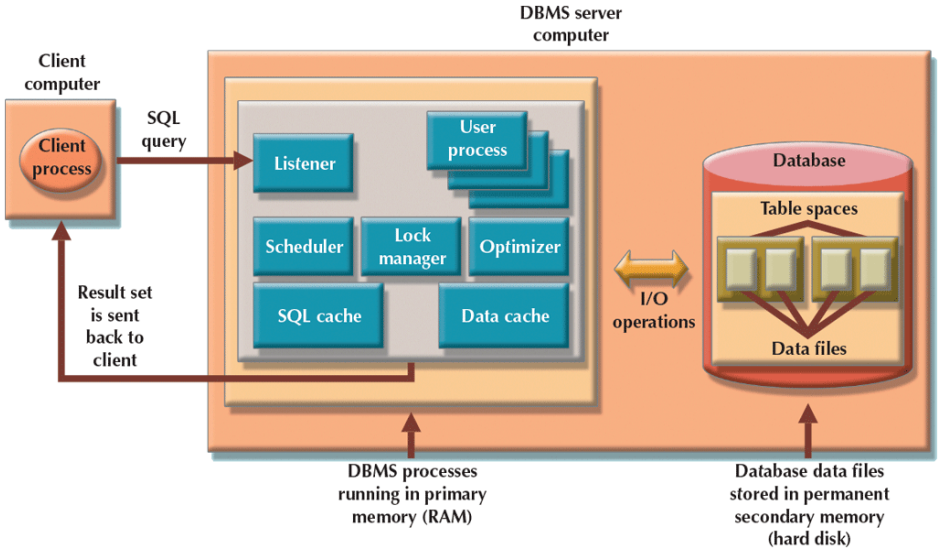
DBMS environment is configured to respond to clients' requests as fast as possible

- While making optimum use of existing resources

DBMS Architecture



DBMS Architecture



All data in a database are stored in data files

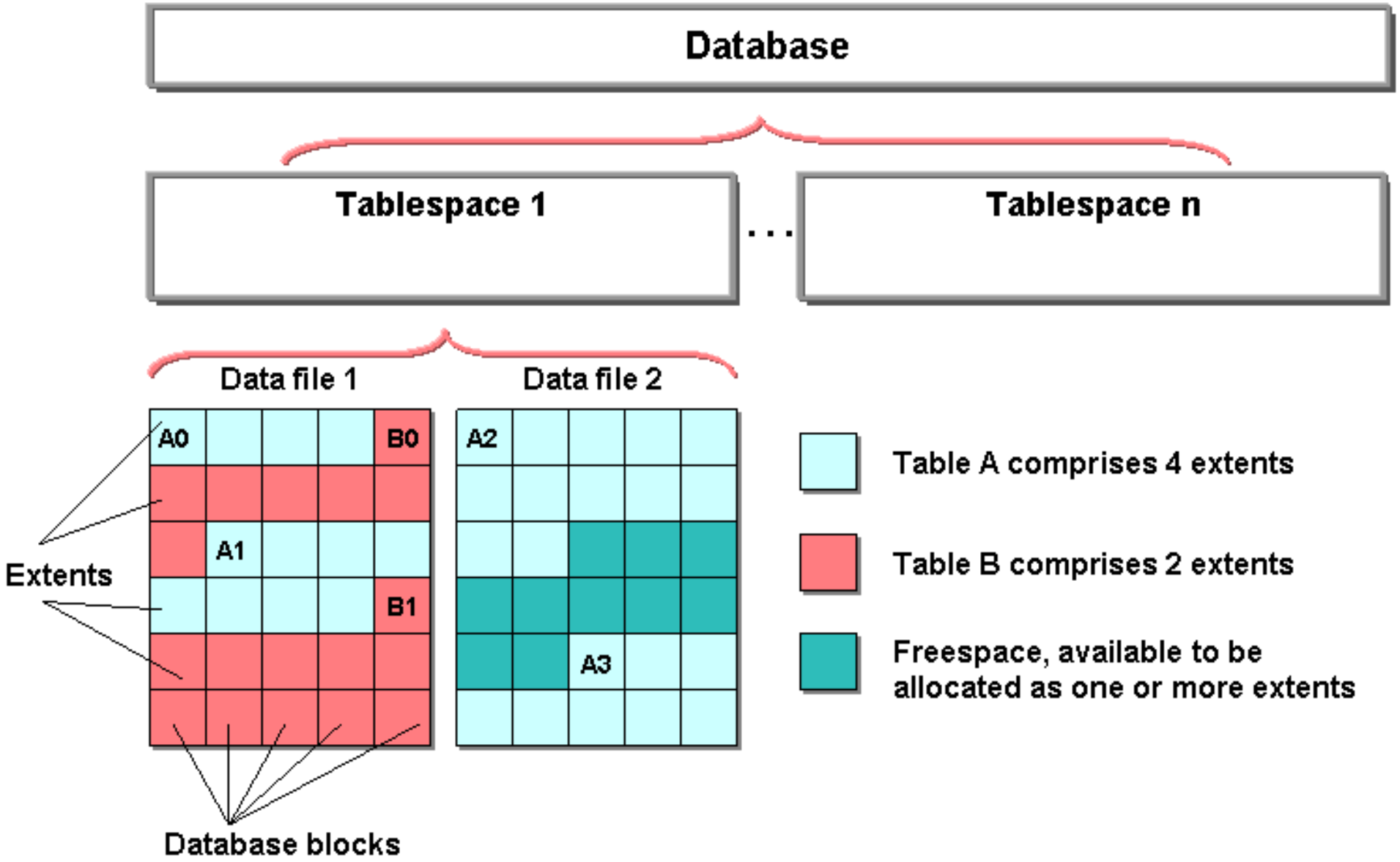
Data files automatically expand in predefined increments known as extents

Data files are grouped in file groups or table spaces

Logical grouping of several data files that store data with similar characteristics

Data cache or buffer cache: shared, reserved memory area

Stores most recently executed SQL statements or PL/SQL procedures



https://help.sap.com/doc/saphelp_qim100/1.0/en-US/45/0f8678f1394574e10000000a114a6b/content.htm?no_cache=true

Database Query Optimization Modes

Algorithms proposed for query optimization are based on two principles:

- Optimum order to achieve the fastest execution time
- Sites to be accessed to minimize communication costs

Evaluated based on:

- Operation mode
  - Automatic and Manual
- Timing of its optimization
  - Static and Dynamic

Database Query Optimization Modes

Classification of operation modes

- **Automatic query optimization:** DBMS finds the most cost-effective access path without user intervention
- **Manual query optimization:** requires that optimization be selected and scheduled by the end-user or programmer

Database Query Optimization Modes

Classification based on timing of optimization

- **Static query optimization:** best optimization strategy is selected when the query is compiled by the DBMS
  - Takes place at compilation time

- **Dynamic query optimization:** access strategy is dynamically determined by the DBMS at run time, using the most up-to-date information about the database
  - Takes place at execution time

Database Query Optimization Modes

Classification based on type of information used to optimize the query

- **Statistically based query optimization algorithm:** statistics are used by the DBMS to determine the best access strategy

- Statistical information is generated by DBMS through:
  - Dynamic statistical generation mode
  - Manual statistical generation mode

- **Rule-based query optimization algorithm:** based on a set of user-defined rules to determine the best query access strategy

Database Statistics

Another DBMS process that plays an important role in query optimization is gathering database statistics.

Measurements about database objects; provide a snapshot of database characteristics

- Number of processors used
- Processor speed
- Temporary space available

Database Statistics

SAMPLE DATABASE STATISTICS MEASUREMENTS	
DATABASE OBJECT	**SAMPLE MEASUREMENTS**
Tables	Number of rows, number of disk blocks used, row length, number of columns in each row, number of distinct values in each column, maximum value in each column, minimum value in each column, and columns that have indexes
Indexes	Number and name of columns in the index key, number of key values in the index, number of distinct key values in the index key, histogram of key values in an index, and number of disk pages used by the index
Environment Resources	Logical and physical disk block size, location and size of data files, and number of extends per data file

Query Processing

Parsing

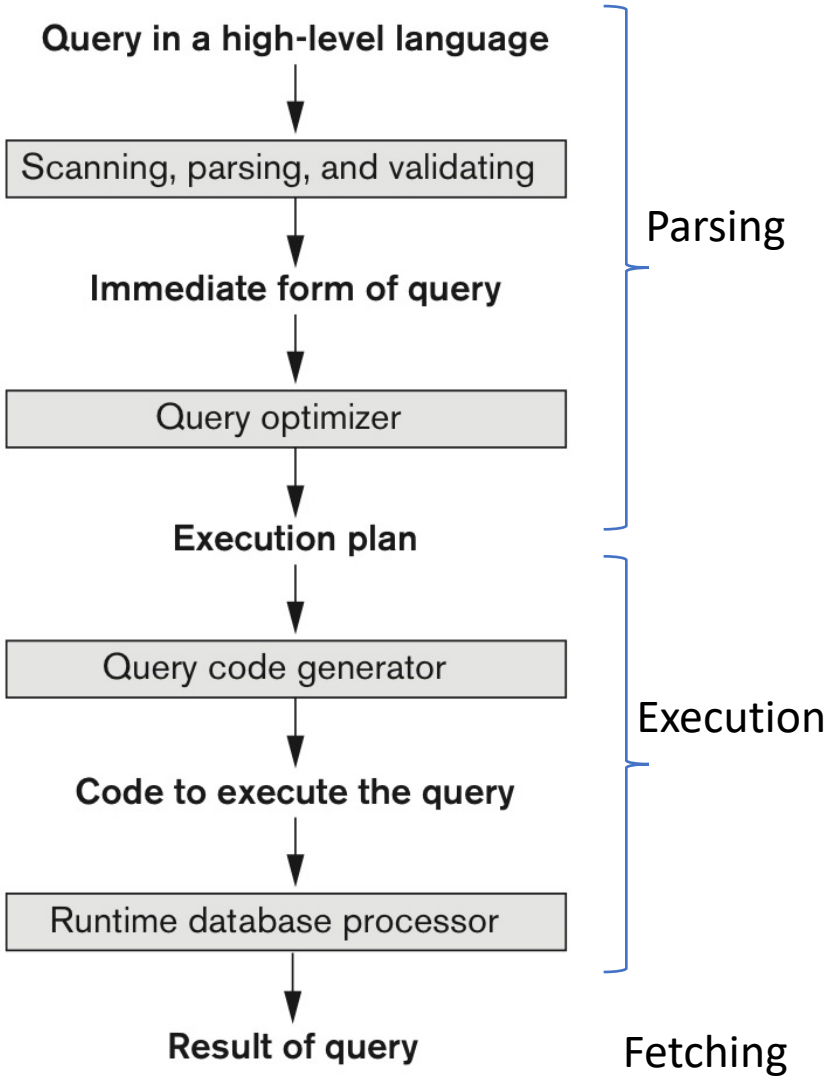
DBMS parses the SQL query and chooses the most efficient access/execution plan

Execution

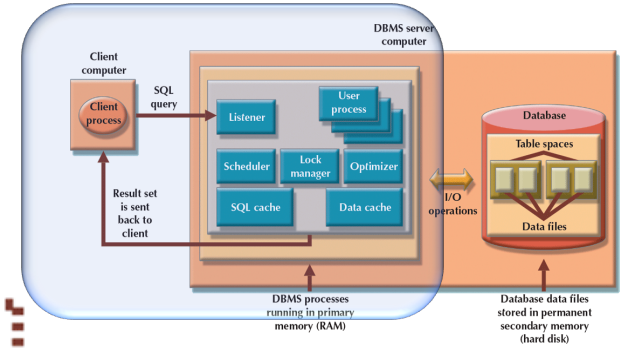
DBMS executes the SQL query using the chosen execution plan

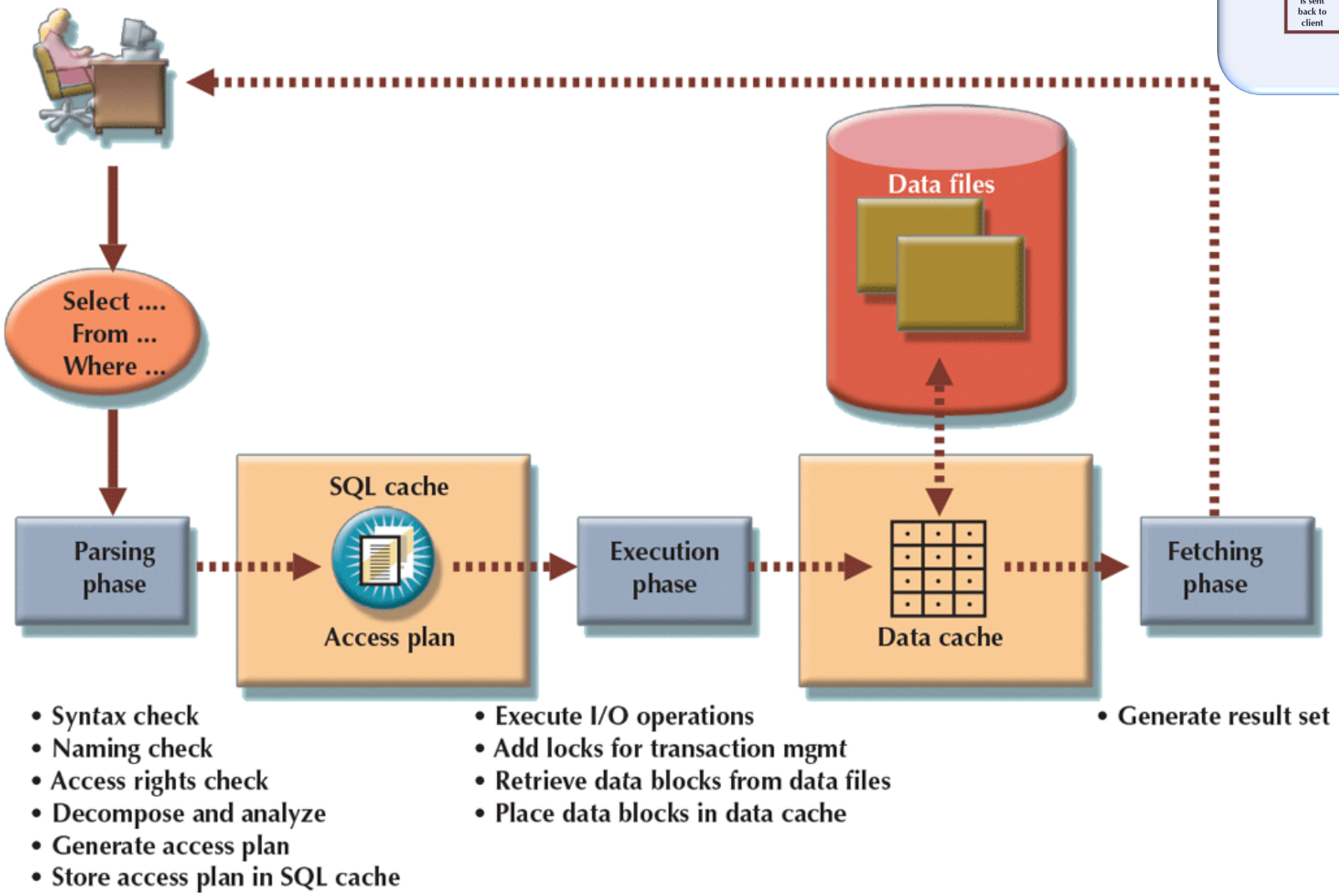
Fetching

DBMS fetches the data and sends the result set back to the client



Query Processing





Query Processing – Parsing Phase

Query is broken down into smaller units

- Original SQL query transformed into slightly different version of original SQL code which is fully equivalent and more efficient

Query optimizer: analyzes SQL query

- Finds most efficient way to access data

Query Processing – Parsing Phase (Contd.)

Access plans: result of parsing a SQL statement; contains a series of steps the DBMS will use to execute the query and return the result set in the most efficient way

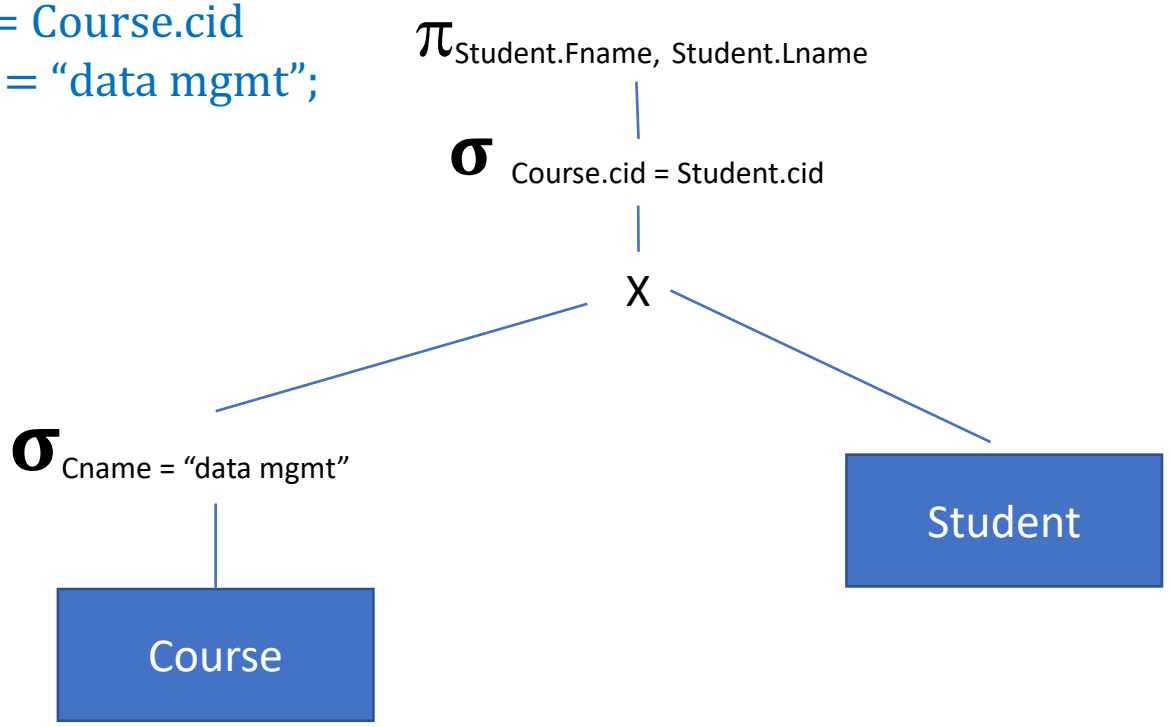
- **Access plan exists for query in SQL cache:** DBMS reuses it
- **No access plan:** optimizer evaluates various plans and chooses one to be placed in SQL cache for use

Display Last name, first name of student taking course “data mgmt.”

Student table = R(B00, Lname, Fname, cid)
Course table – R(cid, Cname, term)

```
SELECT Student.Lname, Student.Fname
FROM Student, Course
Where Student.cid = Course.cid
AND Course.Cname = “data mgmt”;
```

Query Tree



Query Processing – Parsing Phase (Contd.)

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)

Query Processing – SQL Execution Phase

All I/O operations indicated in the access plan are executed

- Locks are acquired
- Data are retrieved and placed in data cache
- Transaction management commands are processed

Query Processing – SQL Fetching Phase

Rows of resulting query result set are returned to client

- DBMS may use temporary table space to store temporary data
- Database server coordinates the movement of the result set rows from the server cache to the client cache

Query Processing Bottlenecks

Delay introduced in the processing of an I/O operation that slows the system

Caused by the:

CPU – The CPU processing power of the DBMS should match the system's expected work load. A CPU bottleneck will affect DBMS and all processes

RAM – If RAM is less, sharing with other processes, data cache, SQL cache can create a bottleneck

Hard disk – Less storage space and slower data transfer can create a bottleneck

Network – If bandwidth is limited and many users access network

Application code – Poor database design and application code can create a bottleneck

Indexes and Query Optimization

Indexes

- Help speed up data access
- Facilitate searching, sorting, using aggregate functions, and join operations
- Ordered set of values that contain the index key and pointers
- More efficient than a full table scan; index data is preordered and the amount of data is usually much smaller

No Indexes

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

CUSTOMER TABLE
(14,786 rows)

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

Indexes

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

STATE_NDX INDEX

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

CUSTOMER TABLE
(14,786 rows)

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

Indexes and Query Optimization

Data sparsity: number of different values a column could have

- High or low

Data structures used to implement indexes

- Hash indexes
- B-tree indexes
- Bitmap indexes

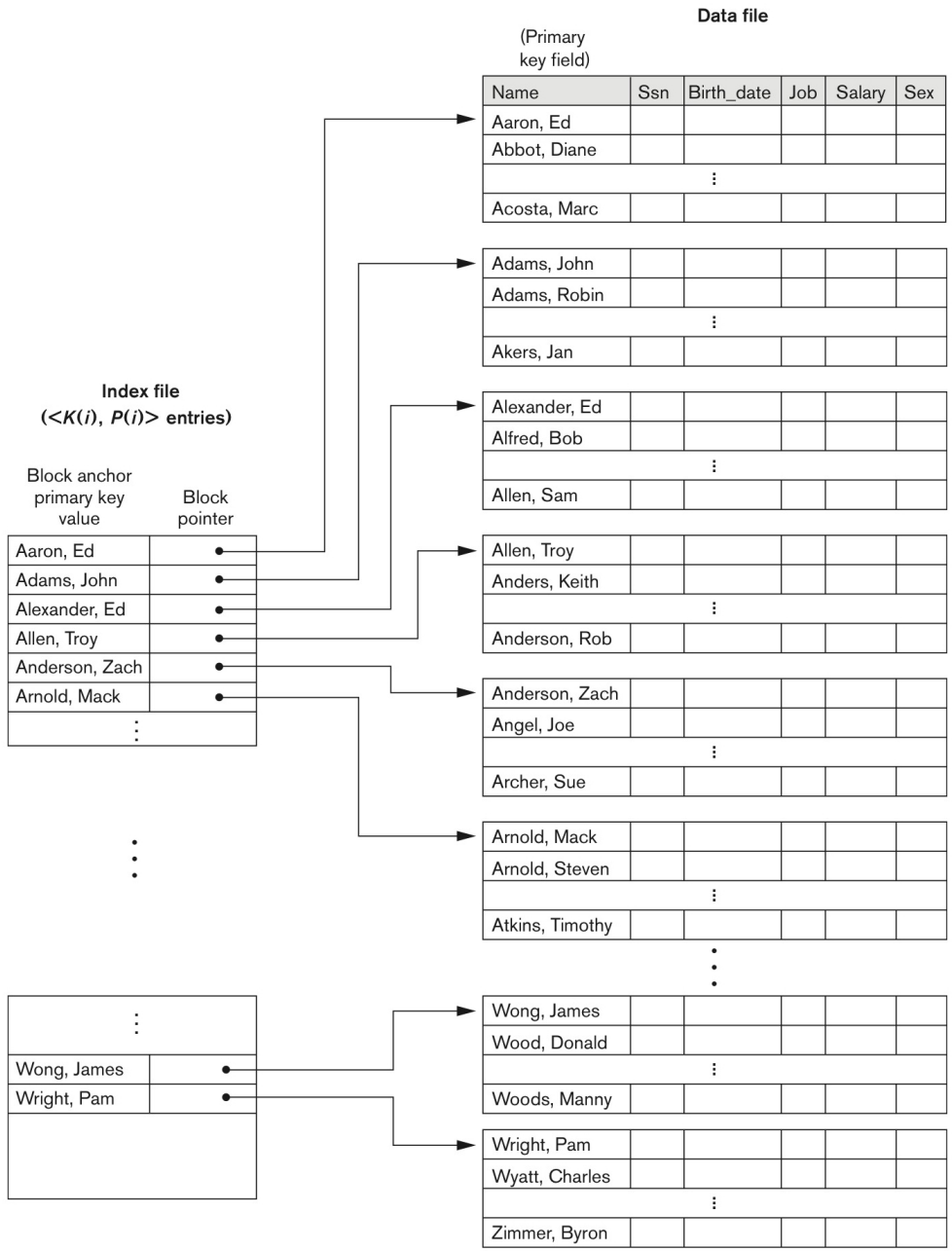
DBMS determines best type of index to use

Hash Index

A hash index is based on an ordered list of hash values.

This value points to an entry in a hash table, which in turn points to the actual location of the data row.

LNAME="Scott" and FNAME="Shannon"



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.

This is the default and most common type of index used in databases.

Used mainly in tables in which column values repeat a relatively small number of times.

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.

Bitmap indexes tend to use less space than B-tree indexes because they use bits instead of bytes to store their data.

B-Tree index is used in columns with high data sparsity—that is, columns with many different values relative to the total number of rows.

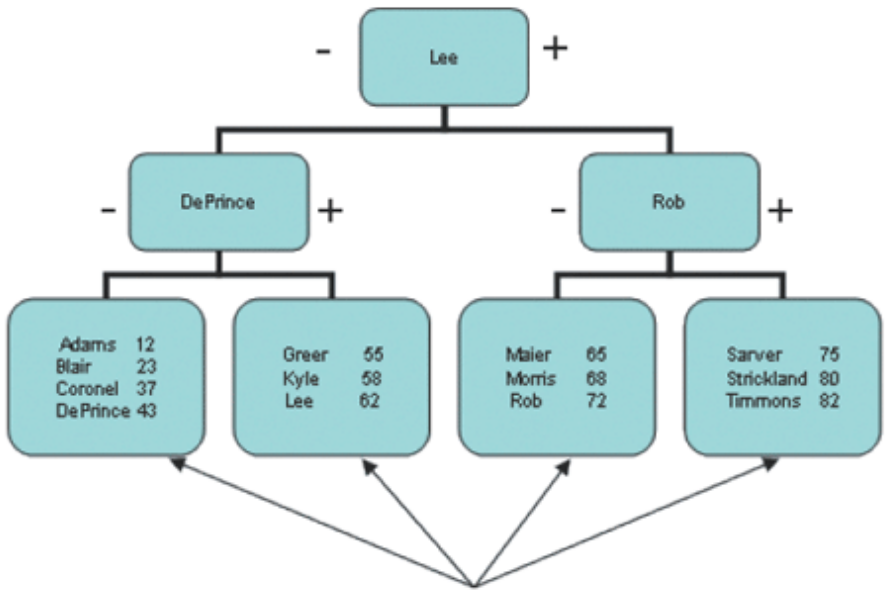
Bitmap index is used in columns with low data sparsity—that is, columns with few different values relative to the total number of rows.

CUSTOMER TABLE

CUS_ID	CUS_LNAME	CUS_FNAME	CUS_PHONE	REGION_CODE
12	Adams	Charlie	4533	NW
23	Blair	Robert	5426	SE
37	Coronel	Carlos	2358	SW
43	DePrince	Albert	6543	NE
55	Greer	Tim	2764	SE
58	Kyle	Ruben	2453	SW
62	Lee	John	7895	NE
65	Maier	Jerry	7689	NW
68	Morris	Steve	4568	NW
72	Rob	Pete	8123	NE
75	Sarver	Lee	8193	SE
80	Strickland	Tomas	3129	SW
82	Timmons	Douglas	3499	NE

B-tree Index

On CUS_LNAME



Leaf objects contain index: key and pointers to rows in table. Access to any row using the index will take the same number of I/O accesses. In this example, it would take four I/O accesses to access any given table row using the index: One for each index tree level (root, branch, leaf object) plus access to data row using the pointer.

Bitmap Index

On REGION_CODE

Region →

NE	NW	SE	SW		
Bit 1	Bit 2	Bit 3	Bit 4	Bit ...	Bit ...
0	1	0	0		
0	0	1	0		
0	0	0	1		
1	0	0	0		
0	0	1	0		
0	0	0	1		
1	0	0	0		
0	1	0	0		
0	1	0	0		
1	0	0	0		
0	0	1	0		
0	0	0	1		
1	0	0	0		

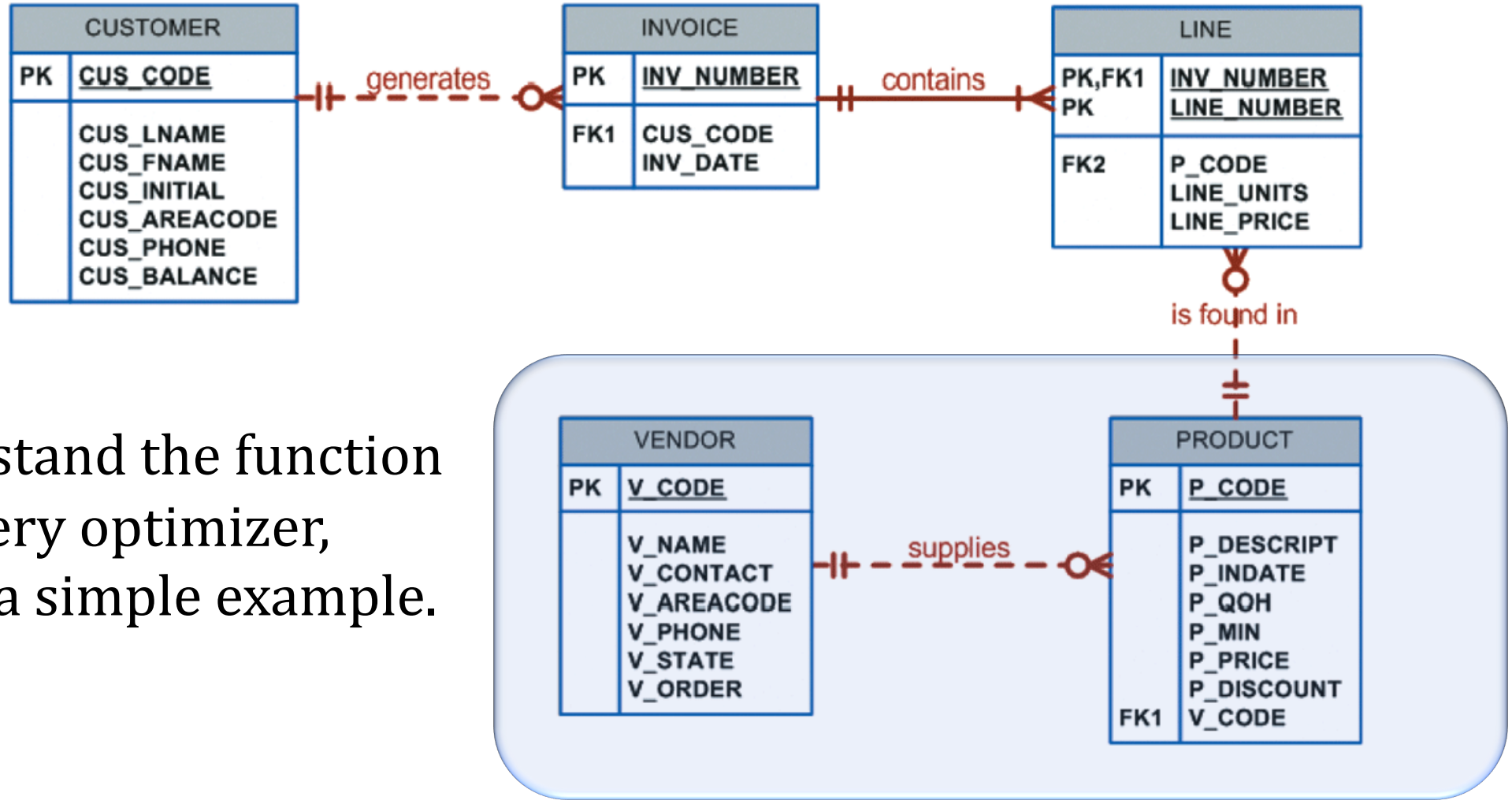
← One byte

In the bitmap index, each bit represents one region code. In the first row, bit number two is turned on, thus indicating that the first row region code value is NW.

REGION_CODE = 'NW'

Each byte in the bitmap index represents one row of the table data. Bitmap indexes are very efficient with searches. For example, to find all customers in the NW region, the DBMS will return all rows with bit number two turned on.

Example



To understand the function of the query optimizer, consider a simple example.

Example (Contd.)

Assume that you want to list all products provided by a vendor based in Florida. To acquire that information, you could write the following query:

```
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';
```

- The PRODUCT table has 7,000 rows.
- The VENDOR table has 300 rows.
- Ten vendors are located in Florida.
- One thousand products come from vendors in Florida.

Database Statistics

```
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';
```

- The PRODUCT table has 7,000 rows.
- The VENDOR table has 300 rows.
- 10 vendors are located in Florida.
- 1000 products come from vendors in Florida.

Assume no index and each row read has I/O cost "1"

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**

Example (Contd.)

Using Hints to Affect Optimizer Choices

Optimizer might not choose the best execution plan

- Makes decisions based on existing statistics; might be old
- Might choose less-efficient decisions

Optimizer hints: special instructions for the optimizer

- Embedded in the SQL command text

Using Hints to Affect Optimizer Choices (Contd.)

OPTIMIZER HINTS

HINT	USAGE
ALL_ROWS	Instructs the optimizer to minimize the overall execution time—that is, to minimize the time needed to return all rows in the query result set. This hint is generally used for batch mode processes. For example: SELECT /*+ ALL_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;
FIRST_ROWS	Instructs the optimizer to minimize the time needed to process the first set of rows—that is, to minimize the time needed to return only the first set of rows in the query result set. This hint is generally used for interactive mode processes. For example: SELECT /*+ FIRST_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;
INDEX(name)	Forces the optimizer to use the P_QOH_NDX index to process this query. For example: SELECT /*+ INDEX(P_QOH_NDX) */ * FROM PRODUCT WHERE P_QOH < 10

SQL Performance Tuning

Evaluated from client perspective

- Most current relational DBMSs perform automatic query optimization at the server end
- Most SQL performance optimization techniques are DBMS-specific and thus rarely portable

Majority of performance problems are related to poorly written SQL code

- A carefully written query almost always outperforms a poorly written one

Index Selectivity (1-2)

Measure of the likelihood that an index will be used in query processing

Indexes are used when a subset of rows from a large table is to be selected based on a given condition

- Cannot always be used to improve performance

Function-based index: based on a specific SQL function or expression

Index Selectivity (Contd.) (2-2)

As general rule, indexes are likely to be used:

- When an indexed column appears by itself in the search criteria of a WHERE or HAVING clause
- When an indexed column appears by itself in a GROUP BY or ORDER BY clause
- When a MAX or MIN function is applied to an indexed column
- When the data sparsity on the indexed column is high

Conditional Expression (1-4)

Expressed within WHERE or HAVING clauses of a SQL statement

Restricts the output of a query to only rows matching conditional criteria

Conditional Expression (Contd.) (2-4)

Guidelines to write efficient conditional expressions in SQL code

- Use simple columns or literals as operands

  WHERE P_Value = 10 * TAX is slower than WHERE P_Value = 10

- Numeric field comparisons are faster than character, date, and NULL comparisons

  WHERE P_Name = 'item1' is slower than WHERE P_Value = 10

- Equality comparisons are faster than inequality comparisons

  (>, >=, <, <=) are slower than "="

Conditional Expression (Contd.) (3-4)

Guidelines to write efficient conditional expressions in SQL code

- Transform conditional expressions to use literals

P_QOH < P_MIN AND P_MIN = P_REORDER AND P_QOH = 10 [X]

P_QOH = 10 AND P_MIN = P_REORDER AND P_MIN > 10

- Write equality conditions first when using multiple conditional expressions

  First use =, then use <>

Conditional Expression (Contd.) (4-4)

Guidelines to write efficient conditional expressions in SQL code

- When using multiple AND conditions, write the condition most likely to be false first

  P_PRICE > 10 AND V_STATE = 'FL'

  (if few vendors are located in Florida) V_STATE = 'FL' AND P_PRICE > 10

- When using multiple OR conditions, put the condition most likely to be true first

  V_STATE = 'FL' OR V_STATE = 'TX'

- Avoid the use of NOT logical operator

  NOT (P_PRICE > 10.00) can be written as P_PRICE <= 10.00

Query Formulation

Step to formulate a query

Pinpoint what columns and computations are required

- Granularity? Aggregation? Simple expression?

Identify source tables

- Use least number of tables in JOIN

Decide how to join tables

- Inner or Outer

Establish action criteria are needed

- Single value, Nested comparison

Determine the order in which to display the output

- Use ORDER BY clause. This is resource-intensive

DBMS Performance Tuning (1-2)

Managing DBMS processes in primary memory and the structures in physical storage

DBMS performance tuning at server end focuses on setting parameters used for:

Data cache: The data cache size must be set large enough to permit as many data requests as possible to be serviced from the cache.

SQL cache: The SQL cache stores the most recently executed SQL statements (after the SQL statements have been parsed by the optimizer).

Sort cache: The sort cache is used as a temporary storage area for ORDER BY or GROUP BY operations, as well as for index-creation functions.

Optimizer mode: Most DBMSs operate in one of two optimization modes: cost-based or rule-based.

DBMS Performance Tuning (Contd.) (2-2)

In-memory database: store large portions of the database in primary storage

- These systems are becoming popular
  - Increasing performance demands of modern database applications
  - Diminishing costs
  - Technology advances of components

DBMS Performance Tuning (Contd.) (2-2)

Recommendations for physical storage of databases

- Utilize I/O accelerators
- Use RAID (Redundant Array of Independent Disks) to provide a balance between performance improvement and fault tolerance
- Minimize disk contention
- Put high-usage tables in their own table spaces
- Assign separate data files in separate storage volumes for indexes, system, and high-usage tables
- Take advantage of the various table storage organizations in the database
- Partition tables based on usage
- Apply denormalized tables where appropriate
- Store computed and aggregate attributes in tables

Questions to Consider

- Is performance tuning important for all types of DBMS servers?
- Can more number of denormalized tables cause performance issues?
- How to reduce I/O cost in DBMS query processing phase?