COP 4710: Database Systems
Fall 2013

Query Processing (Chapter 12)

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4710/fall2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Database Tuning Concepts

- One of the main functions of a DBMS is to provide timely answers to end users. End users interact with the DBMS through the use of queries to generate information, using the following general sequence:
  1. The end user (client-side) application generates a query.
  2. The query is sent to the DBMS (server-side).
  3. The DBMS (server-side) executes the query.
  4. The DBMS sends the resulting data set to the end-user (client-side) application.
- End users expect their queries to return results as quickly as possible. Regardless of user perceptions, the goal of database performance is to execute queries as fast as possible.

Database Tuning Concepts

- To accomplish this goal, database performance must be closely monitored and regularly tuned.
- Database performance tuning refers to a set of activities and procedures designed to reduce the response time of the database system, i.e., to ensure that an end-user query is processed by the DBMS is in the minimum amount of time.
- The time required by a query to return a result depends on many factors, which tend to be wide-ranging and vary among environments and vendors.
- The performance of a DBMS is constrained by three main factors: CPU processing power, the amount of primary memory (RAM), and I/O (disk and network) throughput. The table on the next page summarizes these performance factors.

Database Tuning Concepts

	SYSTEM RESOURCES	CLIENT	SERVER
Hardware	CPU	The fastest possible Dual-core CPU or higher	The fastest possible Multiple processors (quad-core technology)
	RAM	The maximum possible	The maximum possible
	Hard disk	Fast SATA/EIDE hard disk with sufficient free hard disk space	Multiple high-speed, high-capacity hard disks (SCSI/SATA/Firewire/Fibre Channel) in RAID configuration
	Network	High-speed connection	High-speed connection
Software	Operating system	Fine-tuned for best client application performance	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

General Guidelines For Better System Performance

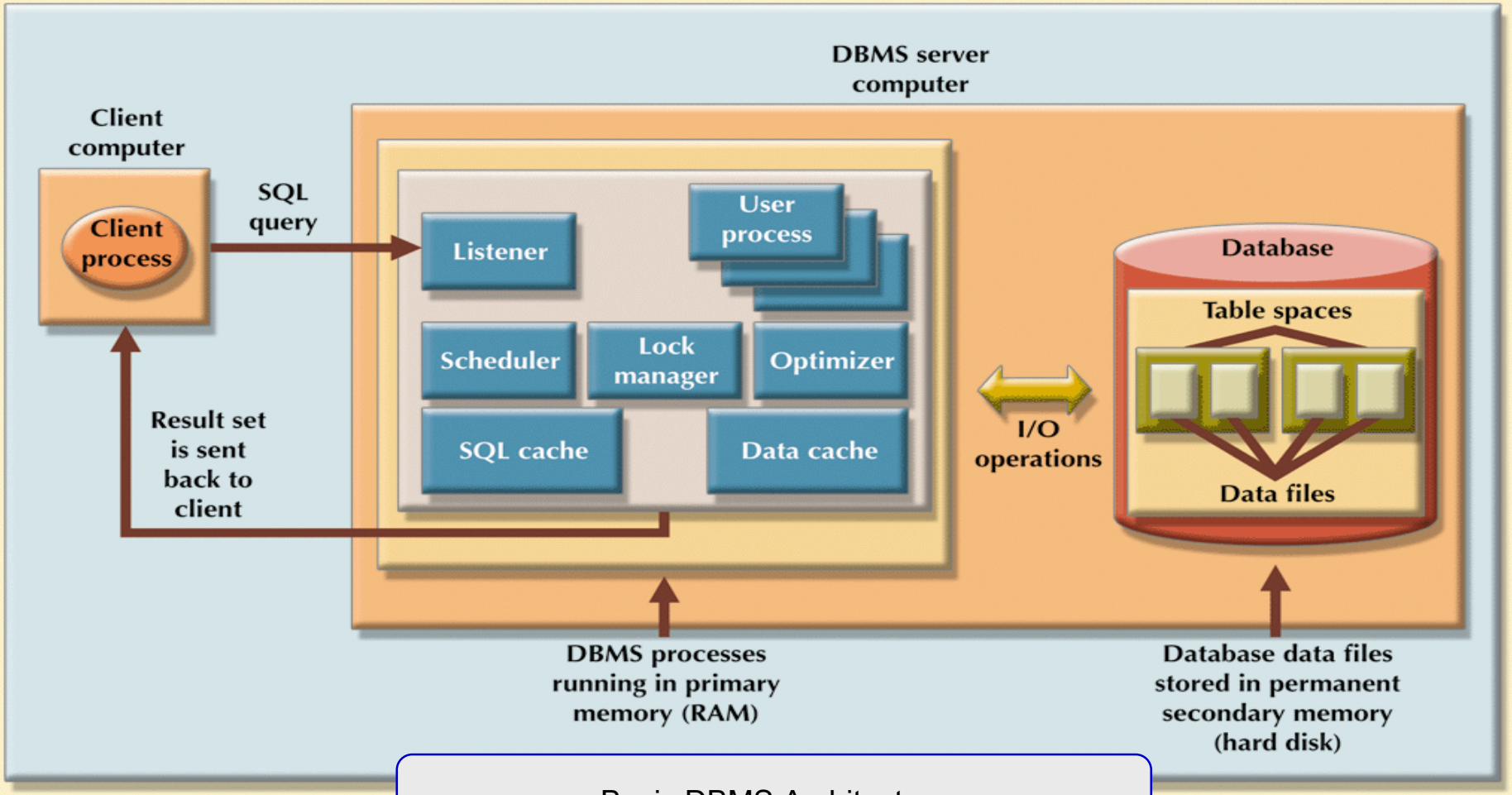
Database Tuning Concepts

- Good database performance starts with good database design. No amount of fine-tuning will make a poorly designed database perform as well as a well-designed database.
- What constitutes a good, efficient database design? We've already examined many of these aspects already in this course, but from a performance point of view, the database designer should ensure that the design makes use of the features in the DBMS that guarantee the integrity and optimal performance of the database.
- This set of notes focuses on the fundamental concepts used to optimize database performance by selecting the appropriate database server configuration, using indices, and implementing the most efficient SQL query syntax.

Database Tuning Concepts

- Client side
  - Generate SQL query that returns correct answer in least amount of time
    - Using minimum amount of resources at server
  - SQL performance tuning
- Server side
  - DBMS environment configured to respond to clients' requests as fast as possible
    - Optimum use of existing resources
  - DBMS performance tuning

Database Tuning Concepts



Basic DBMS Architecture

Database Query Optimization Modes

- Automatic query optimization
  - 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
- Static query optimization
  - Takes place at compilation time
- Dynamic query optimization
  - Takes place at execution time

Database Query Optimization Modes

- Statistically based query optimization algorithm
  - Uses statistical information about the database
  - 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

- The term database statistics refers to a number of measurements about database objects and available resources such as:
  - Tables – number of rows, number of disk blocks used, row length, number of columns, number of distinct values in each column, minimum and maximum value in each column, and columns that are indexed.
  - 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, number of disk pages used by the index, etc..
  - Environment resources - Number of processors used, processor speed, logical and physical disk block size, location and size of data files, temporary space available, etc..

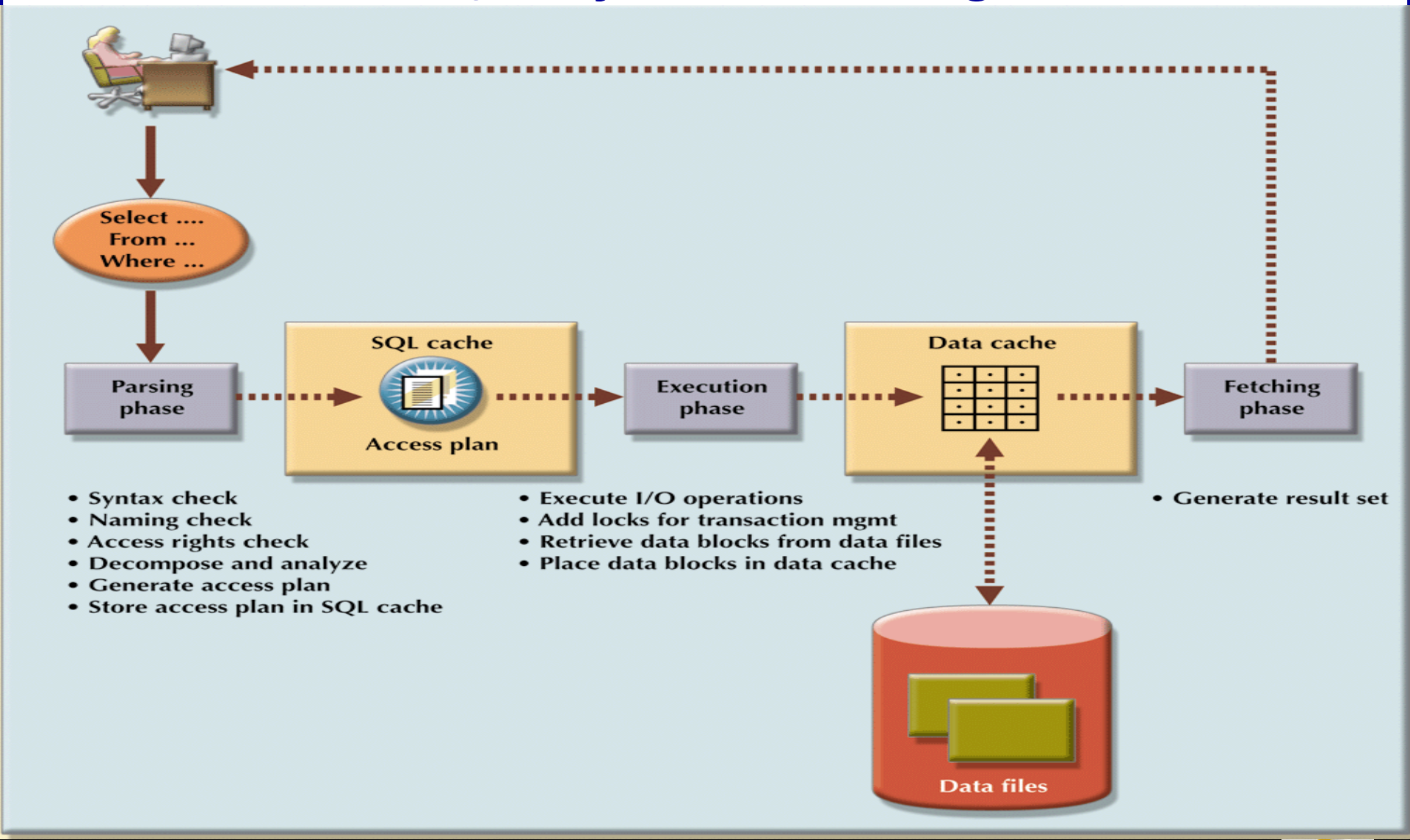
Database Statistics

- The DBMS uses database statistics to make critical decisions about improving query processing efficiency.
- They can be gathered manually by the DBA or automatically by the DBMS.
- Database statistics are stored in the system catalog in specially designated tables.
- They are periodically updated, with more frequent updates for database objects that are subject to frequent change.

Query Processing

- DBMS processes queries in three phases
  - Parsing
    - DBMS parses the query and chooses the most efficient access/execution plan
  - Execution
    - DBMS executes the query using chosen execution plan
  - Fetching
    - DBMS fetches the data and sends the result back to the client

Query Processing



SQL Parsing Phase

- Break down the query into smaller units.
- Transform the original SQL query into a slightly different version of original SQL code.
  - Fully equivalent version
    - Optimized query results are always the same as original query.
  - More efficient version
    - Optimized query will almost always execute faster than original query.

SQL Parsing Phase

- Query optimizer analyzes SQL query and finds most efficient way to access data.
  - Validated for syntax compliance.
  - Validated against data dictionary.
    - Tables and column names are correct.
    - User has proper access rights.
  - Analyzed and decomposed into components.
  - Optimized.
  - Prepared for execution.

SQL Parsing Phase

- Access plans are DBMS-specific.
  - Translate client's SQL query into a series of complex I/O operations.
  - Required to read the data from the physical data files and generate result set.
- DBMS checks if access plan already exists for query in SQL cache.
- DBMS reuses the access plan to save time.
- If not, optimizer evaluates various plans.
  - Chosen plan placed in SQL cache.

SQL Parsing Phase

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)

Sample DBMS Access Plan I/O Operations

SQL Execution Phase – Fetching/Execution

- All I/O operations indicated in access plan are executed.
  - Locks acquired.
  - Data retrieved and placed in data cache.
  - Transaction management commands processed.
- Rows of resulting query result set are returned to client.
- DBMS may use temporary table space to store temporary data.
- In this phase, the DBMS 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.
  - CPU
  - RAM
  - Hard disk
  - Network
  - Application code

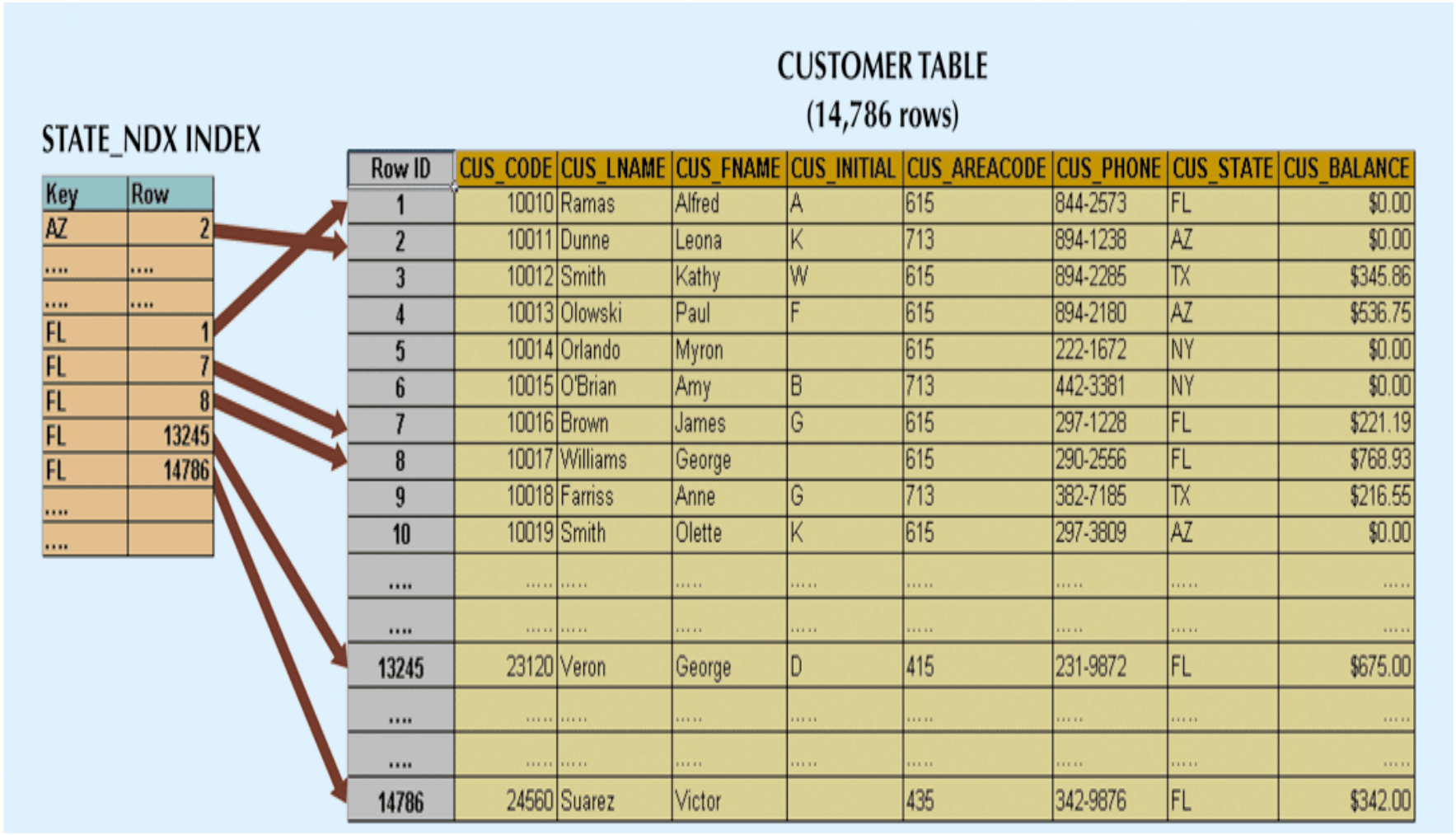
Indices and Query Optimization

- Indices
  - Crucial in speeding up data access.
  - Facilitate searching, sorting, and using aggregate functions as well as join operations.
  - Ordered set of values that contains index key and pointers.
- More efficient to use an index to access table than to scan all rows in a table sequentially.

Indices and Query Optimization

- Data sparsity: the number of different values a column could possibly have. Sparse is equivalent to few values, dense is equivalent to many values.
- Indexes implemented using:
  - Hash indexes
  - B-tree indexes
  - Bitmap indexes
- The DBMS determines the best type of index to use in a specific situation.

Indices and Query Optimization



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

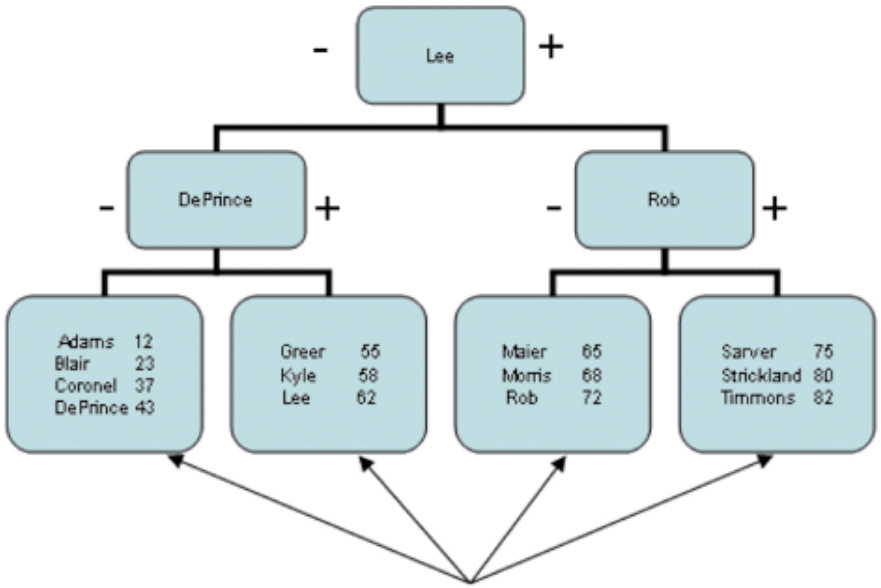
B-Tree index is used in columns with high data sparsity—that is, columns with many different values relative to the total number of rows.

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.

Optimizer Options

- Query optimization is the central activity during the parsing phase in query processing.
- During this phase, the DBMS must choose what indices to use, how to perform join operations, which table to use first, and so on.
- Each DBMS has it s own algorithms for determining the most efficient way to access the data.

Optimizer Options

- The query optimizer can work in one of two fashions:
  - 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 added to yield the cost of the execution plan.
  - 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 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.

Optimizer Options

- The optimizer's objective is to find alternative ways to execute a query - to evaluate the "cost" of each alternative and then choose the one with the lowest cost.
- Consider the following example: Assume that you want to list all products provided by a vendor based in Florida. You might write the following query:

```
SELECT  p_code, p_descript, p_price, v_name, v_date
FROM product, vendor
WHERE product.v_code = vendor.v_code
      AND vendor.v_state = "FL";
```

- Let's assume that the PRODUCT table contains 7000 rows, the VENDOR table has 300 rows, there are 10 vendors located in Florida, and 1000 products come from vendors in Florida.

Optimizer Options

- We'll assume that only the number of rows in the two tables is known to the optimizer.
- The primary factor in determining the most efficient access plan is the I/O cost.
- The table on the next page illustrates two different access plans and the respective I/O costs.
  - We'll assume that only the number of I/O disks reads is considered and to make it simpler to understand, no indices are utilized and each row read from a table represents an I/O cost of 1.
- Notice that the total I/O cost of access plan A is nearly 30 times higher than that of access plan B! Obviously, plan B is chosen.

Optimizer Options

Comparing Access Plans and 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**

Using Hints To Affect Optimizer Options

- Generally, optimizers perform quite well under most circumstances. In some cases however, the best access plan might not be selected.
- Since the optimizer makes decisions based on existing statistics, using old, outdated statistics might cause the optimizer to select a poor access plan.
- Even using current statistics, the optimizer's choice might not bee the most efficient one. Sometimes, the end user would like to change the optimizer mode for the current SQL statement. This is done with optimizer hints, which are special instructions for the optimizer that are embedded inside the SQL statement.
- Some examples of optimizer hints are shown on the next page.

Using Hints To Affect Optimizer Options

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

- SQL performance tuning is evaluated from the client perspective.
- The goal is to illustrate some common practices that are used to write efficient SQL code.
- Note:
  - Most current generation relational DBMS perform automatic query optimization at the server end.
  - Most SQL performance optimization techniques are DBMS-specific and therefore are rarely portable, even across different versions of the same DBMS.
- This does not mean that you shouldn't worry about writing good SQL code because the system will always optimize it!

SQL Performance Tuning

- Many of the performance problems that a DBMS can suffer are related to poorly written SQL code.
- Although a DBMS provides general optimizing services, a carefully written query will usually outperform a poorly written query.
- Although the DML of SQL contains many different commands, such as INSERT, DELETE, UPDATE, and SELECT, most of what we'll look at for performance tuning is related to the SELECT statement and in particular the use of indices and how to correctly write conditional expressions.

Index Selectivity

- Indices are the most important technique used in SQL performance optimization.
- Indices are used when:
  - Indexed column appears by itself in search criteria of WHERE or HAVING clause.
  - Indexed column appears by itself in GROUP BY or ORDER BY clause.
  - MAX or MIN function is applied to indexed column.
  - Data sparsity on indexed column is high.
- Indices are useful when you want to select a small subset of rows from a large table based on a given condition.

Index Selectivity

- The objective is to create indices that have high selectivity.
- Index selectivity is a measure of the likelihood that an index will be used in query processing.
- General guidelines for indexes:
  - Create indexes for each attribute in WHERE, HAVING, ORDER BY, or GROUP BY clause.
  - Do not use in small tables or tables with low sparsity.
  - Declare primary and foreign keys so optimizer can use indexes in join operations.
  - Declare indexes in join columns other than PK/FK.

Conditional Expressions

- A conditional expression is normally expressed within the WHERE or HAVING clauses of SQL statement.
- It restricts the output of a query to only rows matching conditional criteria.
- Most of the query optimization techniques that involve conditional expressions are designed to make the optimizer's work easier to perform.

Conditional Expressions

- Common practices for efficient SQL:
  - Use simple columns or literals in conditionals.
  - Numeric field comparisons are faster.
  - Equality comparisons are faster than inequality.
  - Transform conditional expressions to use literals.
  - Write equality conditions first.
  - AND: use condition most likely to be false first.
  - OR: use condition most likely to be true first.
  - Avoid NOT.

Query Formulation

- Queries are written to answer questions. To get the correct answer, you must carefully evaluate what tables, columns, and computations are required to generate the desired output.
- To formulate a query you need to:
  - Identify what columns and computations are required.
  - Identify source tables.
  - Determine how to join tables.
  - Determine what selection criteria is needed.
  - Determine in what order to display output.

DBMS Performance Tuning

- DBMS performance tuning includes global tasks such as managing DBMS processes in primary memory and structures in physical storage
- DBMS performance tuning at server end focuses on setting parameters used for the:
  - Data cache – sized to permit maximum requests.
  - SQL cache – most recently executed queries still available thus skipping the parsing phase.
  - Sort cache – temporary storage for ORDER BY and GROUP BY clauses.
  - Optimizer mode (cost-based or rule-based).

DBMS Performance Tuning

- Managing the physical storage details of the data files also plays an important role in DBMS performance tuning.
- Some of the physical storage details are dependent upon the specific hardware utilized by the system.
- Some general guidelines to follow when creating databases are shown on the next page.

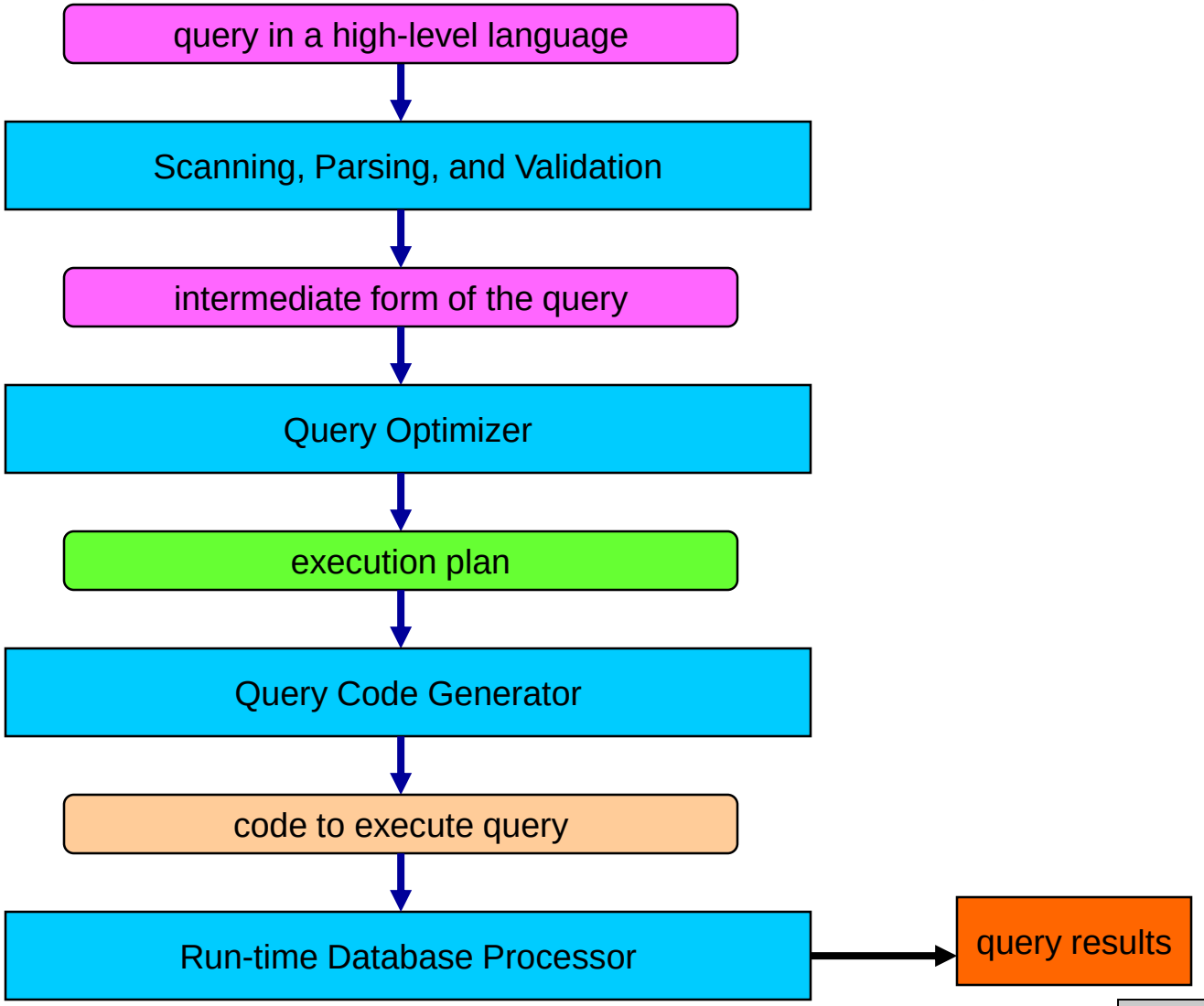
DBMS Performance Tuning

- Some general guidelines for the creation of databases:
  - Use RAID (Redundant Array of Independent Disks) to provide balance between performance 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 table storage organizations in database.
  - Partition tables based on usage.
  - Use denormalized tables where appropriate.
  - Store computed and aggregate attributes in tables.

Query Processing and Optimization

- A query expressed in a high-level language like SQL must first be scanned, parsed, and validated.
- Once the above steps are completed, an internal representation of the query is created. Typically this is either a tree or graph structure, called a query tree or query graph.
- Using the query tree or query graph the RDBMS must devise an execution strategy for retrieving the results from the internal files.
- For all but the most simple queries, several different execution strategies are possible. The process of choosing a suitable execution strategy is called query optimization.

The Steps in Query Processing



Query Optimization

- The term query optimization may be somewhat misleading. Typically, no attempt is made to achieve an optimal query execution strategy overall – merely a *reasonably efficient strategy*.
- Finding an optimal strategy is usually too time consuming except for very simple queries and for these it usually doesn't matter.
- Queries may be "hand-tuned" for optimal performance, but this is rare.
- Each RDBMS will typically maintain a number of general database access algorithms that implement basic relational operations such as select and join. Hybrid combinations of relational operations also typically exist.

Query Optimization (cont.)

- Only execution strategies that can be implemented by the DBMS access algorithms and which apply to the particular database in question can be considered by the query optimizer.
- There are two basic techniques that can be applied to query optimization:

1. Heuristic rules: these are rules that will typically reorder the operations in the query tree for a particular execution strategy.
2. Systematical estimation: the cost of various execution strategies are systematically estimated and the plan with the least "cost" is chosen. What constitutes cost can also vary. It could be a monetary cost, or it could be a cost in terms of time or other factors.

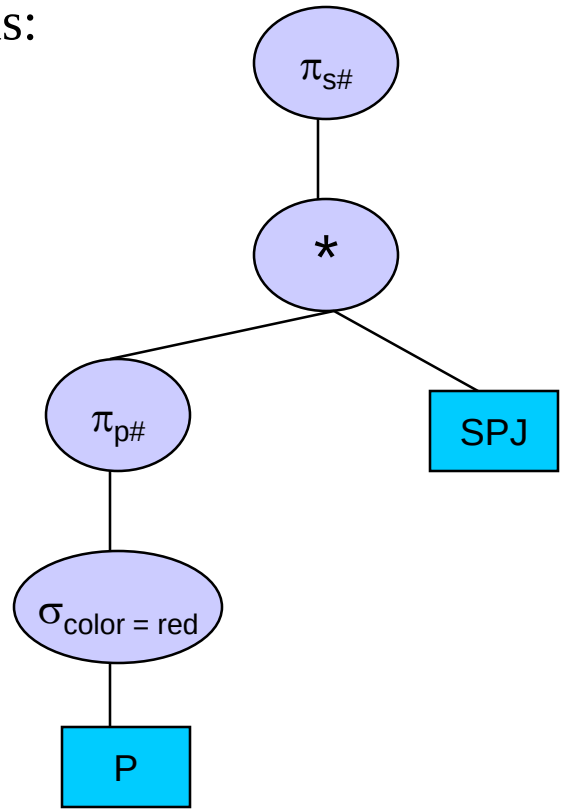
- Most query optimizers use a combination of both techniques.

Query Trees

- A query tree is a tree representation of a relational algebra expression which represents the operand relations as leaf nodes and the relational algebra operators as internal nodes.
- Execution of the query tree consists of executing and internal node operation whenever its operands are available and then replacing that internal node by the virtual relation which results from the execution of the operation.
- Execution terminates when the root node is executed and the resulting relation is produced.
- This technique is similar to what many compilers do for 3GLs like C.

Query Tree Example

- Consider the query: "list the supplier numbers for suppliers who supply a red part." (this one should be really familiar by now!!)
- In relational algebra we have: $\pi_{s\#}(spj * (\pi_{p\#}(\sigma_{color='red'}(P))))$
- The corresponding query tree is:



Query Trees

- There are usually several different ways to generate a relational algebra expression for a query. This should be quite obvious by now after doing the homework for the course.
- Since several different relational algebra expressions are possible for a given query, so too are there multiple query trees possible for the same query.
- The next page shows several different relational algebra expressions for a given query and the following couple of pages illustrate the possible query trees.

Query Expressions

- Query: list the names of those suppliers who ship both part numbers P1 and P2.

SQL Version #1:

```
Select name
From Suppliers
Where s# In (Select s#
                From Shipments
                Where p# = "P1")
  And  s# In (Select s#
                From Shipments
                 Where p# = "P2")
```

SQL Version #2:

```
Select name
From Suppliers
Where Exists (Select s#
                From Shipments
                Where p# = "P1" and
              Suppliers.s# = Shipments.s#)
  And  Exists (Select s#
                From Shipments
                 Where p# = "P2" and
              Suppliers.s# = Shipments.s#)
```

Query Expressions

- Query: list the names of those suppliers who ship both part numbers P1 and P2.

SQL Version #3:

```
Select name
From Suppliers
Where s# In (Select Shipments.s#
                From Shipments Cross Join Shipments As SPJ
                Where Shipments.s# = SPJ.s#
                        And Shipments.p# = "P1"
                        And SPJ.p# = "P2")
```


Query Expressions

- Query: list the names of those suppliers who ship both part numbers P1 and P2.

exp #1: $(\pi_{name}(s * (\pi_{s\#}(\sigma_{p\#=P1}(spj))))) \cap (\pi_{name}(s * (\pi_{s\#}(\sigma_{p\#=P2}(spj)))))$

exp #2: $\pi_{name}(s * ((\pi_{s\#}(\sigma_{p\#=P1}(spj))) \cap (\pi_{s\#}(\sigma_{p\#=P2}(spj)))))$

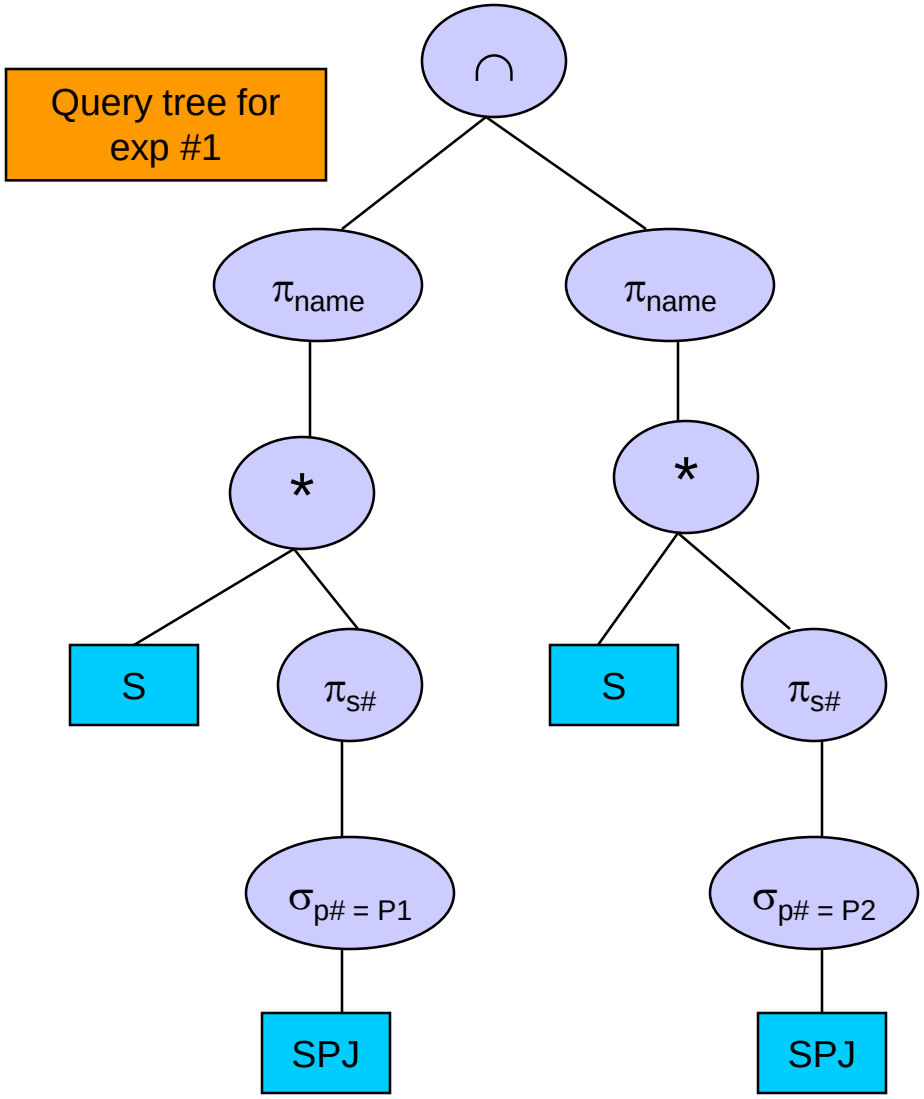
Similar to SQL Versions #1 and #2

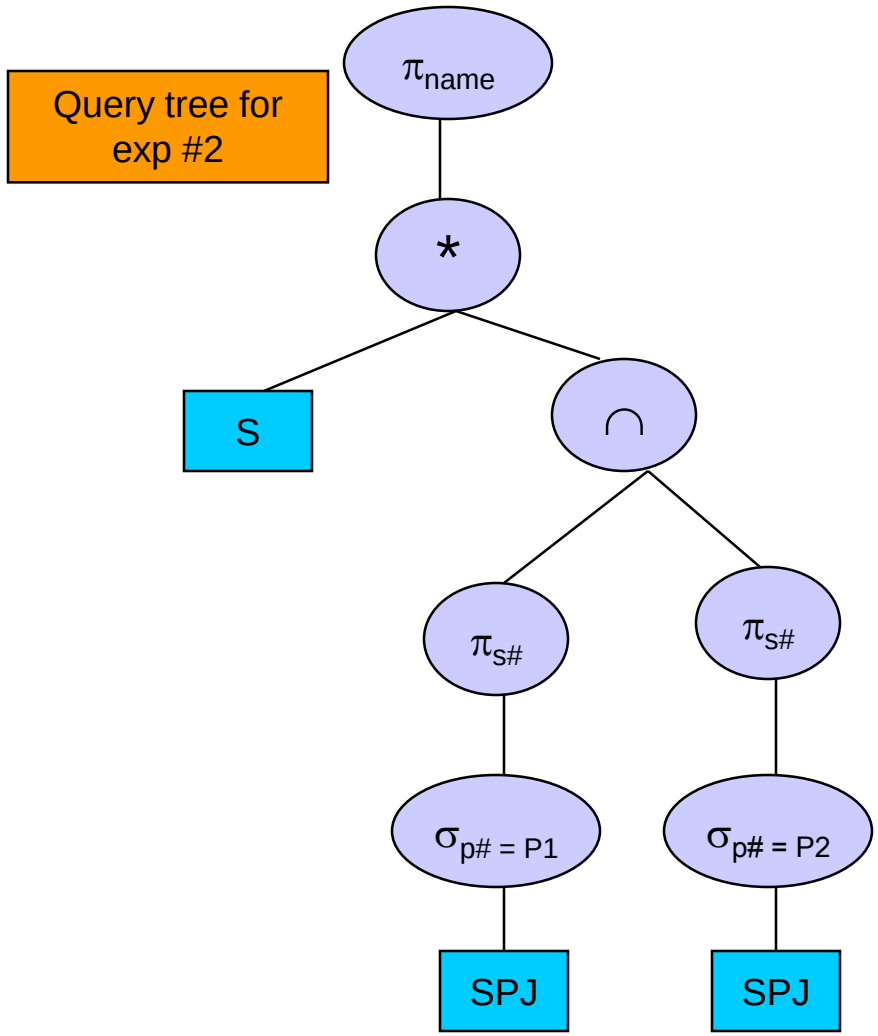
exp #3: $\pi_{name}(s * (\pi_{s\#}(\sigma_{spj.p\#=P1}(spj)(\sigma_{spj1.p\#=P2}(spj1)(spj \times spj1)))))$

Similar to SQL Version #3

exp #4: $\pi_{name}(s * (\sigma_{spj.p\#=P1}(\sigma_{spj1.p\#=P2}(\sigma_{spj.s\#=spj1.s\#}(\pi_{spj.s\#,spl1.s\#,spj.p\#,spj1.p\#}(spj \times spj1))))))$

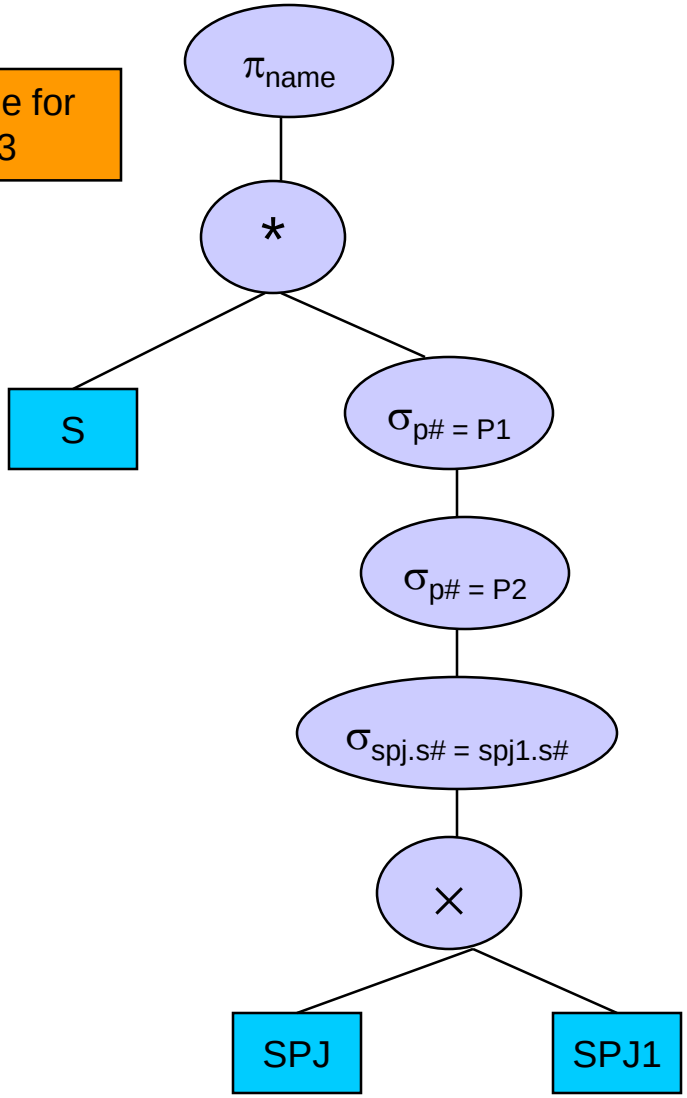
Corresponding Query Trees



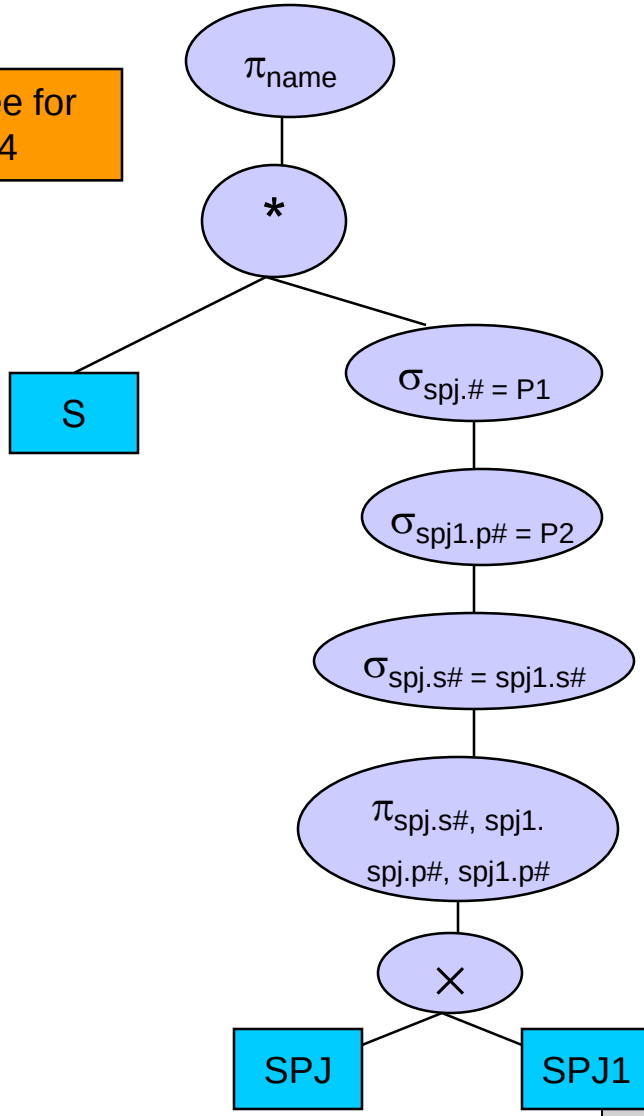


Corresponding Query Trees

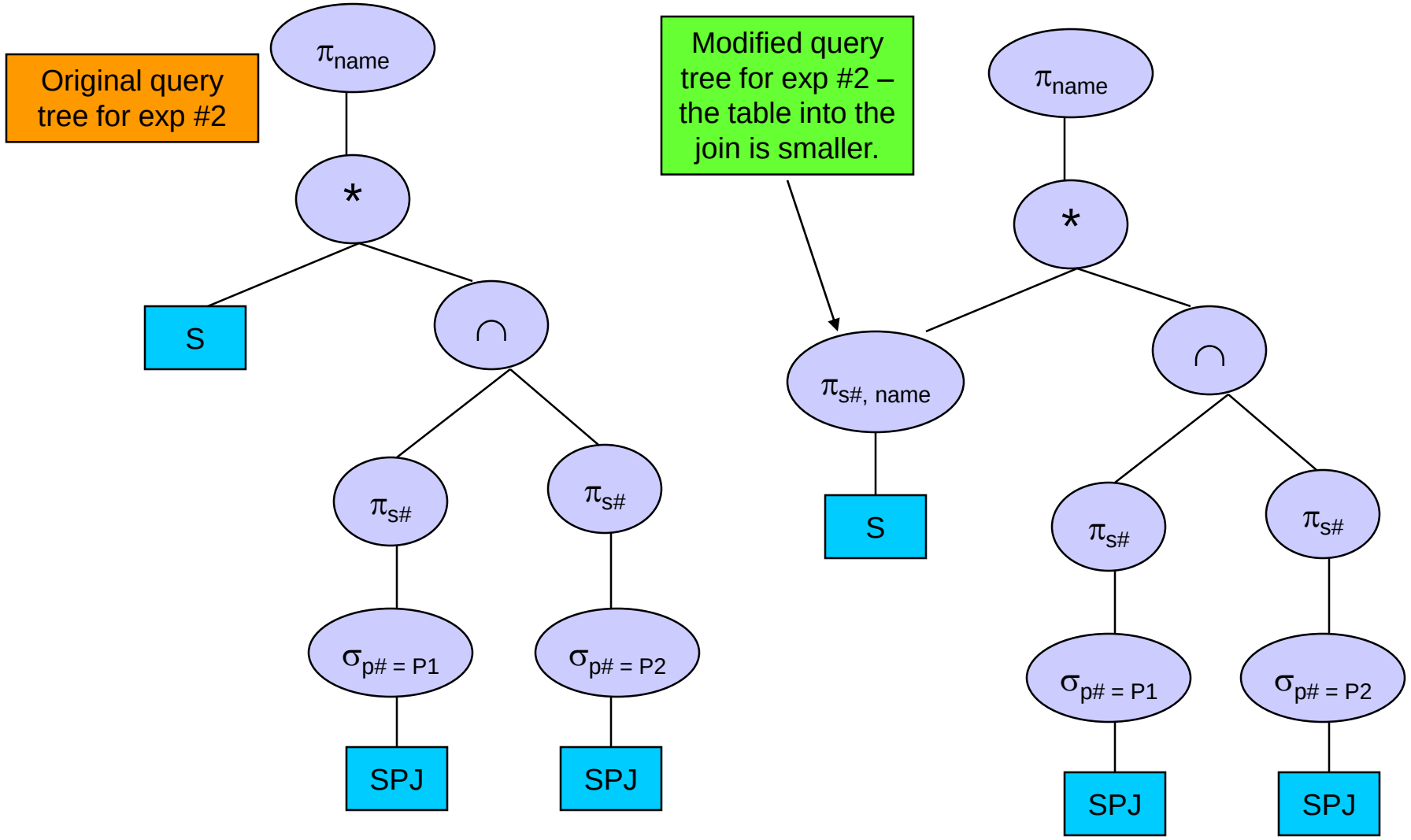
Query tree for exp #3



Query tree for exp #4



Corresponding Query Trees



Basic Query Execution Algorithms

- For each operation (relational algebra operation, plus others) as well as combinations of operations, the DBMS will maintain one or more algorithms to execute the operation.
- Certain algorithms will apply to particular storage structures and access paths and thus can only be utilized if the underlying files involved in the operation include these access paths.
- Typically, the access paths will involve indices and/or hash tables, although other hybrid access paths are also possible.
- In the next few pages will examine some of these query execution strategies for the basic relational algebra operations.

Algorithms for Selection Operations

- There are many different options for Select operations based on the availability of access paths, indices, etc.
- Search algorithms for Select operations are one of two types:
  - index scans: search is directed from an index structure.
  - file scans: records are selected directly from the file structure.
- (FS1-linear search): Heap files typically are searched with a linear search algorithm.
- (FS2-fast search): Sequential files are typically searched with a binary or jump type of search algorithm.
- (IS3-primary index or hash key to extract single record): In these cases the selection condition involves an equality comparison on a key attribute for which a primary index has been created (or a hash key can be used.)

Algorithms for Selection Operations (cont.)

- (IS4-primary index or hash key to extract multiple records): In these cases the selection condition involves a non-equality based comparison (<, <=, >, >=) on a key attribute for which a primary index has been created. The primary index is used to find the record which satisfies the equality condition and then based upon this record, all other preceding (< or <=) or subsequent (> or >=) records are retrieved from the ordered file.

- (IS5-clustering index to extract multiple records): In these cases the selection condition involves an equality comparison on a non-key attribute which has a clustering index (a secondary index). The clustering index is used to retrieve all records which satisfy the selection condition.

- (IS6 – secondary index, B^+ tree): A selection condition with an equality comparison, a secondary index can be used to retrieve a single record if the indexing field is a key or to retrieve multiple records if the indexing field is not a key. Secondary indices can also be used for any of the comparison operators, not just equality.

Algorithms for Conjunctive Selections

- Conjunctive selections are selection conditions in which several conditions are logically AND'ed together.
- For simple (non-conjunctive) selection conditions, optimization basically means that you check for the existence of an access path on the attribute involved in the condition and use it if available, otherwise a linear search is performed.
- Query optimization for selection is most useful for conjunctive conditions whenever more than one of the participating attributes has an access path.
- The optimizer should choose the access path that retrieves the fewest records in the most efficient manner.

Algorithms for Conjunctive Selections (cont.)

- The overriding concern when choosing between multiple simple conditions in a conjunctive select condition is the selectivity of each condition.
- Selectivity is defined as:

$$Selectivity = \frac{\#\ of\ \ records\ which\ satisfy\ the\ condition}{\#\ of\ \ records\ in\ the\ relation}$$

- The smaller the selectivity the fewer the tuples the condition selects.
- Thus the optimizer should schedule the conjunctive selection comparisons so that the smallest selectivity conditions are applied first followed by the higher and higher selectivity values so that the last condition applied has the highest selectivity value.

Algorithms for Conjunctive Selections (cont.)

- Usually, exact selectivity values for all conditions are not available. However, the DBMS will maintain estimates for most if not all types of conditions and these estimates will be used by the optimizer.
- For example:
  - The selectivity of an equality condition on a key attribute of a relation r(R) is:

$$\frac{1}{|r(R)|}$$

  - The selectivity of an equality condition on an attribute with n distinct values can be estimated by:

$$\frac{\left(\frac{|r(R)|}{n}\right)}{|r(R)|} = \frac{1}{n}$$

Assuming that the records are evenly distributed across the n distinct values, a total of |r(R)|/n records would satisfy an equality condition on this attribute.

Algorithms for Conjunctive Selections (cont.)

- (IS7-conjunctive selection): If an attribute is involved in any single simple condition in the conjunctive selection has an access path that permits the use of any of FS2 through IS6, use that condition to retrieve the records, then check if each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
- (IS8-conjunctive selection using a composite index): If two or more attributes are involved in an equality condition and a composite index (or hash structure) exists for the combined fields – use the composite index directly.
- (IS9-conjunctive selection by intersection of record pointers): If secondary indices are available on any or all of the attributes involved in an equality comparison (assuming that the indices use record pointer and not block pointers), then each index is used to retrieve the record pointers that satisfy the individual simple conditions. The intersection of these record pointers is the set of tuples that satisfy the conjunction.

Algorithms for Join Operations

- The join operation and its variants are the most time consuming operations in query processing.
- Most joins are either natural joins or equi-joins.
- Joins which involve two relations are called two-way joins while joins involving more that two relations are called multiway joins.
- While there are several different strategies that can be employed to process two-way joins, the number of potential strategies grows very rapidly for multiway joins.

Two-way Join Strategies

- We'll assume that the relations to be joined are named R and S, where R contains an attribute named A and S contains an attribute named B which are join compatible.
- For the time-being, we'll consider only natural or equijoin strategies involving these two attributes.
- Note that for a natural join to occur on attributes A and B, a renaming operation on one or both of the attributes must occur prior to the natural join operation.
  - Note too, that if attributes A and B are the only join compatible attributes in R and S, that the equi-join operation R $*_{A=B}$ S has the same effect as a natural join operation.

Algorithms for Two-way Join Operations

- (J1-nested loop): A brute force technique where for each record $t \in R$ (outer loop) retrieve every record $s \in S$ (inner loop) and test if the two records satisfy the join condition, namely does t.A = s.B?
- (J2-single loop w/access structure): If an index or hash key exists for one of the two join attribute, for example, $B \in S$, retrieve each record $t \in R$ one at a time and then use the access structure to retrieve directly all matching records $s \in S$ that satisfy t.A = s.B.
- (J3-sort-merge join): If the records of both R and S are physically sorted (ordered) by the values of the join attributes A and B, then the join can be processed using the most efficient strategy. Both relations are scanned in the order of the join attributes; matching the records that have the same A and B values. In this fashion, each relation is scanned only once.
- (J4-hash-join): In this technique, the records of both relations R and S are hashed using the same hashing function (on the join attributes) to the same hash file. A single pass through the smaller relation will hash its records to the hash file. A single pass through the other relation will hash its records to the same bucket as the first pass combining all similar records.

Pipelining Operations

- Query optimization can also be effected by reducing the number of intermediate relations that are produced as a result of executing a query stream.
- This reduction in the number of intermediate relations is accomplished by combining several relational operations into a single pipeline of operations. This method is also sometimes referred to as stream-based processing.
- While the combining of operations in a pipeline eliminates some of the cost of reading and writing intermediate relations, it does not eliminate all reading and writing costs associated with the operations nor does it eliminate any processing.
- As an example, consider the natural join of two relations R and S, followed by the projection of a set of attributes from the join result.

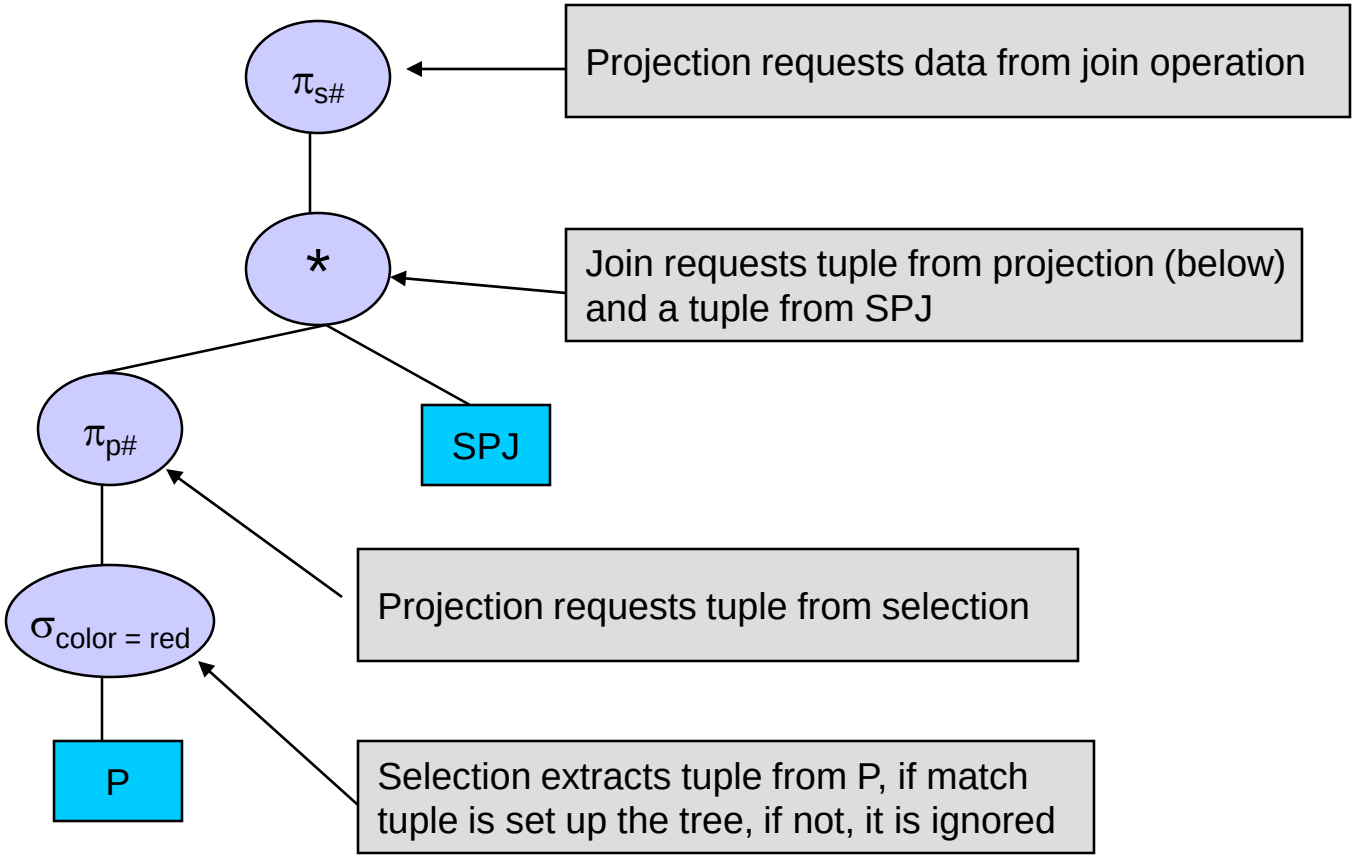
Pipelining Operations (cont.)

- In relational algebra this query looks like: $\pi_{(a, b, c)}(R * S)$
- This set of two operations could be executed as:
  - construct the join of R and S, save as intermediate table T1. [T1 = R * S]
  - project the desired set of attributed from table T1. [result = $\pi_{(a, b, c)}$(T1)]
- In the pipelined execution of this query, no intermediate relation T1 is produced. Instead, as soon as a tuple in the join of R and S is produced it is immediately passed to the projection operation to processing. The final result is created directly.
- In the pipelined version, results are being produced even before the entire join has been processed.

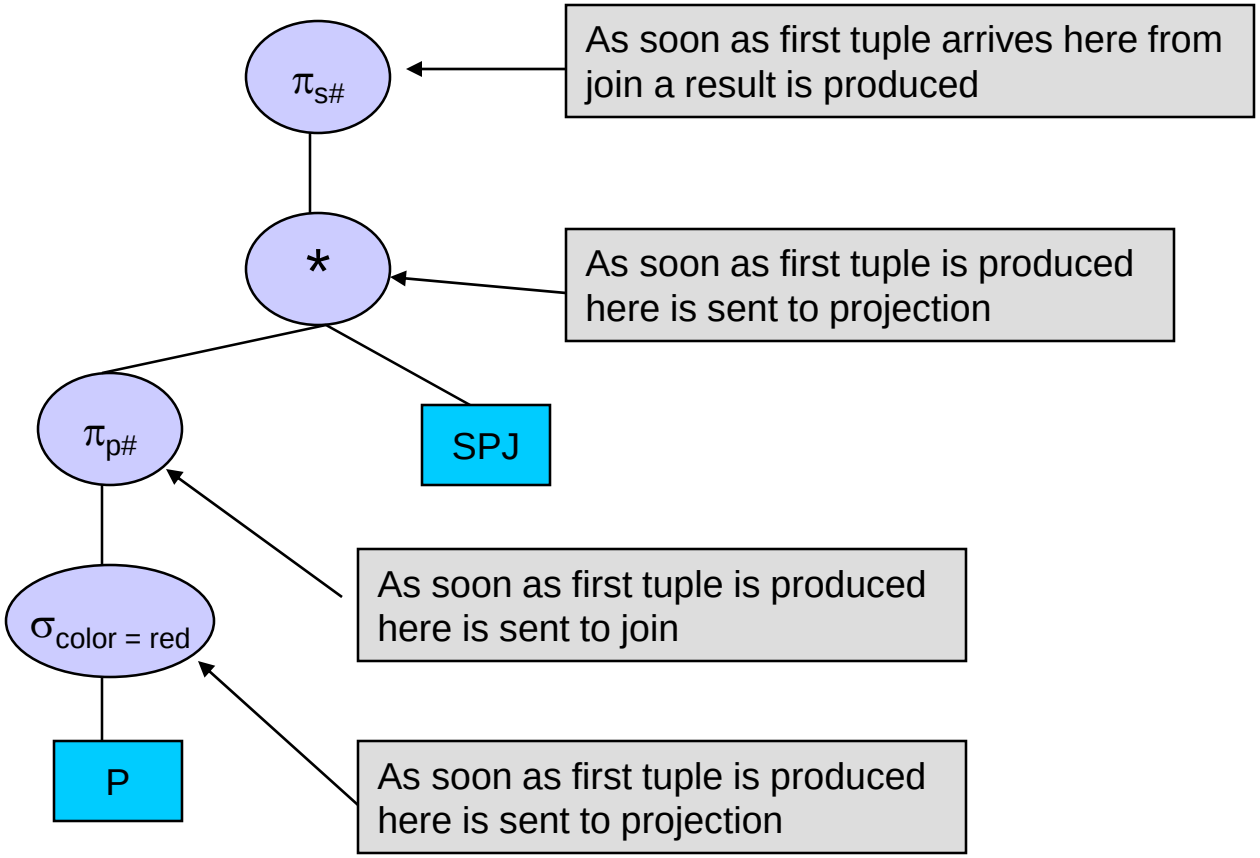
Pipelining Operations (cont.)

- There are two basic strategies that can be used to pipeline operations.
- Demand-driven pipelining: In effect, data is "pulled-up" the query tree as operations request data to operate upon.
- Producer-driven pipelining: In effect, data is "pushed-up" the query tree as lower level operations produce data which is sent to operations higher in the query tree.

Demand-Driven Pipelining Example



Producer-Driven Pipelining Example



Using Heuristics in Query Optimization

- The parser of the high-level query language generates the internal representation of the query which is optimized according to heuristic rules.
- The access routines which execute groups of operations together are based upon the access paths available for the relations involved are chosen by the query optimizer.
- One of the main heuristic rules is to apply projections and selections as early as possible. This is useful because the size of the relations involved in subsequent join operations (or other binary operations) are as small as possible.
- Basically, the query optimizer generates several different query expressions and selects the best choice.

Using Heuristics in Query Optimization (cont.)

- When an equivalent query expression is generated, you must be certain that it is in fact an equivalent expression.
- To this end, the query optimizer must follow certain transformation rules that will ensure equivalency amongst the various query expressions.
- The level of information available to the optimizer will affect the effectiveness of the equivalence generation scheme.
  - At the lowest level – only relation names are known:
    - $R \cap R \equiv R$
    - $\pi_X (R \cup S) \equiv \pi_X (R) \cup \pi_X (S)$
    - $\sigma_{A=B \text{ AND } B=C \text{ AND } A=C} (R) \equiv \sigma_{A=B \text{ AND } B=C} (R)$

Using Heuristics in Query Optimization (cont.)

- If schema information is available:
  - Given R(A, B), S(B,C) with r(R) and s(S) then,
  - $\sigma_{A=a}(R * S) \equiv \sigma_{A=a}(R) * S$
- If constraint information is known, they provide even more information and modification possibilities:
  - If you know that R(A,B,C,D) with r(R) and you also know that r satisfies $B \rightarrow C$, then $\pi_{A,B}(r) * \pi_{B,C}(r) \equiv \pi_{A,B,C}(r)$

- In general, there are many different equivalences that will hold and the optimizer can utilize as many as possible.
  - For example: $r \cup r \equiv r$, $r \cap r \equiv r$, $r - r \equiv \emptyset$

Using Heuristics in Query Optimization (cont.)

- Commutivity rules can also be applied to optimize query execution.
- For example what is the difference between R * S and S * R?
  - Suppose that R contains 3 tuples and S contains 5 tuples. Further suppose that each tuple in R is 10 bytes long and each tuple in S is 100 bytes long.
  - R * S: 1 pass through R generates 3 × 10 bytes = 30 bytes. Three passes through S (one for each tuple generated from R) generates 15 tuples × 100 bytes = 1500 bytes. Total = 1530 bytes.
  - S * R: 1 pass through S generates 5 × 100 bytes = 500 bytes. Five passes through R (one for each tuple generated from S) generates 15 tuples × 10 bytes = 150 bytes. Total = 650 bytes.
  - Clearly, S*R is a better strategy than is R*S.

Using Cost Estimation in Query Optimization

- Cost estimation is typically only used for "canned" query execution code, i.e., compiled queries that will be executed repeatedly.
- The time and effort required for this type of analysis is not justified for simple one-time query execution.
- The cost estimation technique considers the cost of executing a query from four different perspectives:

1. Access costs to secondary storage: this involves all the costs of searching, reading, and writing secondary storage.
2. Storage costs: this involves the cost of storing the intermediate files generated by the chosen execution strategy.
3. Computation costs: Sorting, merging, computation in attributes (selection and join conditions).
4. Communication costs: In a distributed environment, this includes the cost of shipping the query and/or its results to the originating site.

Semantic Query Optimization

- This technique uses the semantics of the database and the various constraints that apply to semantically modify queries into queries which are more efficient.
- For example, suppose a user issues the following query:
  - $\pi_{s\#}(\sigma_{qty > 100}(SPJ))$ {list supplier numbers for suppliers who ship at least one part in a quantity greater than 100.}
  - If a constraint exists that states: all quantities <= 75, then the optimizer could inform the system that the query did not need to be executed at all and the result is simply the empty set.