Part 12: Query Evaluation

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Ed., Chap. 18: "Query Processing and Optimization"
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., Chap. 12: "Query Processing"
- Ramakrishnan/Gehrke: Database Management Systems, 2nd Ed., Mc-Graw Hill, 2000, Chap. 11: "External Sorting", Chap. 12: "Evaluation of Relational Operators".
- Kemper/Eickler: Datenbanksysteme (in German), Chap. 8, Oldenbourg, 1997.
- Härder/Rahm: Datenbanksysteme — Konzepte und Techniken der Implementierung (in German), Springer, 1999.
- Garcia-Molina/Ullman/Widom: Database System Implementation. Prentice Hall, 1999, ISBN 0130402648, 672 pages.
- Oracle 8i Concepts, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76965-01. Chapter 21: "The Optimizer".
- Oracle 8i Designing and Tuning for Performance, Release 2 (8.1.6), Oracle Corporation, 1999, Part No. A76992-01.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Brass: Skript zur Vorlesung Informationssysteme II (in German), Univ. Hildesheim, 1997. `http://www-db.informatik.uni-hannover.de/~sb/isII/`

Objectives

After completing this chapter, you should be able to:

- write a short paragraph explaining what a query evaluation plan (QEP) is.
- explain pipelined evaluation and why sorting needs temporary (disk) space.
- explain different algorithms for implementing joins (especially nested loop join and merge join).
- read and explain Oracle QEPs.

  If a query performs poorly, you need to be able to understand why (and how the problem can be solved).
- develop different query evaluation plans for a given query and assess their merits.

Overview

1. Query Evaluation Plans, Pipelined Evaluation
2. Sorting
3. Algorithms for Joins
4. Operators in Oracle's Execution Plans
5. Appendix: Details, Program Code, Tricks

Introduction (1)

- A query evaluation plan (Oracle calls it an "execution plan") is a program for an abstract machine (interpreter) inside the DBMS. It is produced by the query optimizer.

  Another name is "access plan" (the DBMS has to decide how to access the rows, e.g. whether to use an index).
- In most systems, query evaluation plans (QEPs) are similar to relational algebra expressions (very system dependent).
- E.g. consider the following SQL query:

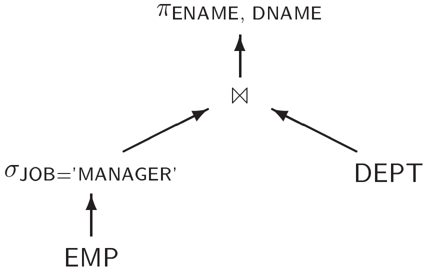
```
SELECT ENAME, DNAME
FROM   EMP, DEPT
WHERE  EMP.DEPTNO = DEPT.DEPTNO
AND    JOB = 'MANAGER'
```

Introduction (2)

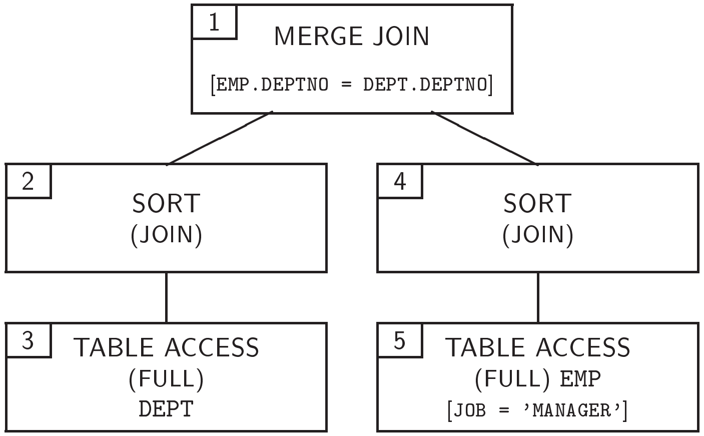
- In relational algebra, the example query would be written as:

$$\pi_{\text{ENAME, DNAME}}\big(\sigma_{\text{JOB='MANAGER'}}(\text{EMP}) \bowtie \text{DEPT}\big)$$

- Complex relational algebra expressions are best displayed as "operator trees" (showing the flow of data):

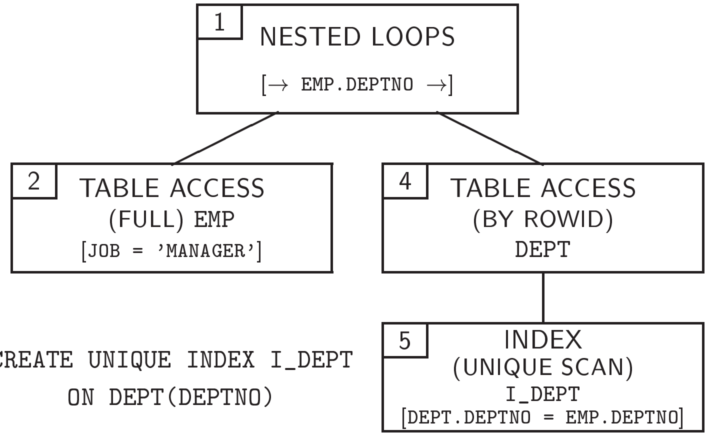


Examples of Oracle QEPs (1)



(Oracle does not show the small annotations in [...].)

Examples of Oracle QEPs (2)




```
CREATE UNIQUE INDEX I_DEPT
ON DEPT(DEPTNO)
```


QEPs vs. Relational Algebra

- Some typical differences to relational algebra are:
  - There are different implementations of the same relational algebra operator, e.g. "MERGE JOIN" is a special way to evaluate a join.
  - An implementation has to work with lists of tuples instead of relations (sets of tuples). E.g. sorting and duplicate elimination are done explicitly.
  - Indexes and ROWIDs appear explicitly in the QEP.
  - Some operations are combined, e.g. the full table scan operator can also do a selection, and the projection does not appear explictly.

How to View Oracle QEPs (1)

- First, one must create a table "PLAN_TABLE" with prescribed columns in which Oracle will store information about the QEP.

  The table must exist under the account of each user who wants to view QEPs. See slide 12-67 for details of the table.

- The simplest way to do this is to execute the script

  `$ORACLE_HOME/rdbms/admin/utlxplan.sql`

- Then enter the following command in SQL*Plus:

  `SET AUTOTRACE ON EXPLAIN`

- Then Oracle will show information about the QEPs for all following queries (not all details, only the general structure).

  If one logs out from SQL*Plus, the `AUTOTRACE` setting is forgotten, but the `PLAN_TABLE` still exists.

How to View Oracle QEPs (2)

- The output you get from `AUTOTRACE` is not in graphical form as shown above, but in textual form (with indentation):

```
Execution Plan
----------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   MERGE JOIN
   2    1     SORT (JOIN)
   3    2       TABLE ACCESS (FULL) OF 'DEPT'
   4    1     SORT (JOIN)
   5    4       TABLE ACCESS (FULL) OF 'EMP'
```

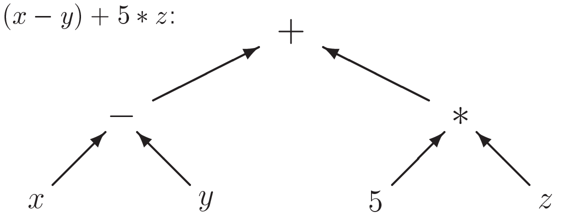
- The first number identifies the tree node (shown above in the upper left corner), the second number is the parent node.

Processing of Evaluation Plans (1)

- So a query evaluation plan is a tree with an operation attached to every node.
- Every node computes a relation which is passed as input to its parent node (up in the tree).
- The relation computed by the root node (the tree top) is returned to the user as the answer to the given SQL query.
- The leaf nodes access tables or indexes to compute their relation.
- Operations in other nodes process or combine relations which they get from their child nodes.

Processing of Evaluation Plans (2)

- Also arithmetic expressions can be displayed as a tree, e.g. $(x - y) + 5 * z$:



- Arithmetic expressions are usually compiled by using registers as temporary storage:

```
R1 := x + y;
R2 := 5 * z;
R1 := R1 + R2; // R1 contains now result
```

Processing of Evaluation Plans (3)

- It would be possible to compute every operation at once completely, store the result in a temporary relation, and let the parent operation read this temporary relation:

  R1 := $\sigma_{\texttt{JOB='MANAGER'}}(\texttt{EMP})$;
  R2 := R1 $\bowtie$ DEPT;
  R3 := $\pi_{\texttt{ENAME, DNAME}}(\texttt{R2})$;
  print R3;

- But then a lot of memory is needed for the intermediate results, and sometimes intermediate results are so large that they have to be written to disk and then read again.
- E.g. when the join has computed a tuple, one can immediately compute the projection result for that tuple.

Processing of Evaluation Plans (4)

- One can eliminate nearly all temporary storage since most operations work "tuple by tuple" (sorting is an exception).
- Most operations compute tuples only on demand (when the parent node needs them), and only one tuple at a time.

  E.g. the join node requests a tuple from the selection node. In order to satisfy the request, it requests a tuple from the relation `EMP` and checks the condition `JOB='MANAGER'`. If the condition is satisfied, it returns the tuple and is done. If not, it requests another tuple from the relation `EMP`.

- Thus, tuples flow immediately from the child to the parent, even before the child has computed the complete result.
- This is called "Pipelined Execution" (or lazy evaluation).

Overview

1. Query Evaluation Plans, Pipelined Evaluation
2. Sorting
3. Algorithms for Joins
4. Operators in Oracle's Execution Plans
5. Appendix: Details, Program Code, Tricks

Sorting and Temporary Storage (1)

- Not all operations can compute their results "on demand".
- E.g. a sort operation needs to see all input tuples before it can return the first result tuple. Otherwise it is possible that a tuple which is earlier in the sort order is still to come.
- Thus, a sort operation needs temporary space for storing all input tuples.
- Of course, the sort operation has the same external interface as all other QEP nodes (`open`, `next`, `close`).
- However, during the `open`, it will already read and sort all its input tuples. Then later requests for the next result tuple will be answered from the intermediate storage.

Sorting and Temporary Storage (2)

- Sometimes it would also be good to materialize other intermediate results which have to be read more than once (e.g. in a nested loop join).
- Some systems have a special operator for doing this (“Bucket”). However, it seems that in Oracle intermediate space is needed only for sorting.
- Oracle has an initialization parameter SORT_AREA_SIZE which is the maximal size of temporary storage a single sort operation can request in memory.
- If the space needed for sorting is larger, Oracle will use temporary segments on disk.

Sorting and Temporary Storage (3)

- How to find out the current setting of this parameter:

```
SELECT VALUE
FROM   V$PARAMETER
WHERE  NAME = 'sort_area_size';
```

  On our UNIX systems, the default is 65536 Bytes.
- The parameter can be changed with

```
ALTER SESSION SET SORT_AREA_SIZE = 131072;
```

- The memory is taken from the Program Global Area (PGA), i.e. inside the dedicated server process, not from the SGA. However, in the multithreaded server configuration, it is taken from the SGA.

Sorting and Temporary Storage (4)

- After the sort is done, the sorted rows must be temporarily stored until they are fetched from the parent node.
- The parameter SORT_AREA_RETAINED_SIZE controls how much memory can be used for this purpose.

  By default it is the same as SORT_AREA_SIZE. But if memory is scarce, it should be used for running sorts rather than afterwards when the rows only wait to be fetched.
- There are more initialization parameters controlling the sorting:

```
SELECT NAME, VALUE, DESCRIPTION
FROM   V$PARAMETER
WHERE  NAME LIKE '%sort%';
```


Sorting and Temporary Storage (5)

- Temporary segments can be allocated in any tablespace, but it is better to use a special “temporary tablespace”.
- The storage parameters INITIAL, NEXT etc. for the temporary segments are inherited from the tablespace in which they are allocated.

  INITIAL should be a multiple of the SORT_AREA_SIZE plus one block for the segment header.
- The tablespace used for temporary segments can be defined separately for each user in the CREATE USER statement.

  It can be changed with ALTER USER.
- Information about temporary segments can be found in V$SORT_SEGMENT and V$SORT_USAGE.

Performance Statistics for Sorts (1)

- How many sorts in my session were done in memory / on disk? How many rows were sorted?

```
SELECT X.VALUE, Y.NAME
FROM   V$SESSTAT X, V$STATNAME Y, V$SESSION Z
WHERE  X.STATISTIC# = Y.STATISTIC#
AND    Y.NAME LIKE '%sort%'
AND    X.SID = Z.SID AND Z.USERNAME = USER
```

- There is also a table V$SYSSTAT which contains accumulated counts since the DBMS was last started.
- These statistics are also contained in the report produced by utlbstat.sql/utlestat.sql (see above).

Performance Statistics for Sorts (2)

- After SET AUTOTRACE ON, SQL*Plus prints not only the QEP for every query, but also performance statistics.
  - SET AUTOTRACE ON STATISTICS prints only the statistics.
- Before one can access the statistics, the DBA (user SYS) must execute the script plustrce.sql.
  - It is located in $ORACLE_HOME/sqlplus/admin.
- It declares a role PLUSTRACE which has access to some dynamic performance views.
  - sys.v_$sesstat, sys.v_$statname, sys.v_$session.
- This role must be granted to all users who should be able to use the trace.

Sort Algorithm (1)

- Sorting is needed quite often, and it is a relatively expensive operation.
- Thus, many thoughts were put into developing an efficient sort algorithm, and new improvements are still proposed in the literature.
- Sorting with external memory is usually based on the merge sort algorithm, which you should know from your data structures course.
- Mergesort is based on the notion of "runs", which are already sorted sequences of elements. E.g. when you want to sort n elements, you start with n runs of length 1.

Sort Algorithm (2)

- Then you always merge two such sorted sequences ("runs") of length l to one sorted sequence of length $2 * l$.
- This merging can be done in linear time: You look at the first element of both runs, take the smaller one and put it into the output. Repeat this until both runs are empty.
- Since the size of the runs doubles every time, you need a logarithmic number of iterations until you have only one run which contains all elements. $\rightarrow$ Complexity $O\big(n * \log(n)\big)$.
- You can implement it with four files: Two for the input runs and two for the output runs.
  - Output runs are written to the two files in alternating fashion so that they contain the same number of runs.

Example (Basic Mergesort)

- Input (16 runs of length 1):

12	5	9	20	16	18	3	7	17	10	2	25	13	15	6	8

- After first step (8 runs of length 2):

5 12		9 20		16 18		3 7		10 17		2 25		13 15		6 8	

- Second and third step:

5 9 12 20	3 7 16 18	2 10 17 25	6 8 13 15

3 5 7 9 12 16 18 20	2 6 8 10 13 15 17 25

- After fourth step (1 runs of length 16: final result):

2	3	5	6	7	8	9	10	12	13	15	16	17	18	20	25

Example (Merging of Runs)

- One first compares the first elements of both runs:

5 9 12 20   3 7 16 18

- 3 is smaller, so it is written to the output and the current position in the second file is moved forward:

5 9 12 20   3 7 16 18 $\rightarrow$ 3

- Now 5 is smaller, and written to the output:

5 9 12 20   3 7 16 18 $\rightarrow$ 3 5

- And so on (exercise). When the end of file is reached on one side, the rest of the other side is written to the output.

Overview

1. Query Evaluation Plans, Pipelined Evaluation
2. Sorting
3. Algorithms for Joins
4. Operators in Oracle's Execution Plans
5. Appendix: Details, Program Code, Tricks

Nested Loop Join (1)

- The nested loop join looks at all combinations of tuples from both relations, evaluates the join condition, and returns those combinations for which the condition is true.
- So the $R \underset{A_i = B_j}{\bowtie} S$ is evaluated similarly to $\sigma_{A_i = B_j}(R \times S)$ but without materializing the intermediate result of $\times$.

  Our pipelined evaluation anyway wouldn't materialize the result, but we nevertheless save many function calls.
- If both relations have approximately n tuples each, these will be n^2 combinations. $\rightarrow$ Complexity $O(n^2)$.
- The nested loop join works for arbitrary join conditions, not only equality conditions like $A = B$.

Nested Loop Join (2)

- Without the pipelined evaluation, the algorithm for $\mathsf{R} \underset{\mathsf{A}_i=\mathsf{B}_j}{\bowtie} \mathsf{S}$ looks as follows (thus the name “nested loop”):

foreach tuple $t = (A_1{:}\, d_1, \ldots, A_n{:}\, d_n)$ **in** R **do**
    foreach tuple $u = (B_1{:}\, e_1, \ldots, B_m{:}\, e_m)$ **in** S **do**
        if $d_i = e_j$ **then**
            output $t \circ u = (A_1{:}\, d_1, \ldots, A_n{:}\, d_n,$
                $B_1{:}\, e_1, \ldots, B_m{:}\, e_m)$;
        fi;
    od;
od;

Merge Join (1)

- The merge join works very similar to merge sort.
- Both input relations must be sorted on the join attribute.
- Then the algorithm does a parallel pass on both relations. It advances always the scan with the smaller value in the join attribute. In this way it finds all matches (equal values).
- In this way, the time needed for the join is linear in the size of the two relations. If we have to sort them, the total complexity is $O\big(n * \log(n)\big)$.

    The runtime of the nested loop join is always quadratic in the sizes of the input relations. The number of block accesses is only quadratic if neither one fits into memory.

- The merge join works only for equality conditions.

Merge Join (2)

open(R); open(S);
read $t = (A_1{:}\, d_1, \ldots, A_n{:}\, d_n)$ from R;
read $u = (B_1{:}\, e_1, \ldots, B_m{:}\, e_m)$ from S;
while not end of file(R) **and not** end of file(S) **do**
    if $d_i < e_j$ **then**
        read $t = (A_1{:}\, d_1, \ldots, A_n{:}\, d_n)$ from R;
    else if $d_i > e_j$ **then**
        read $u = (B_1{:}\, e_1, \ldots, B_m{:}\, e_m)$ from S;
    else /* $d_i = e_j$, assumes only one row on each side */
        output $t \circ u = (A_1{:}\, d_1, \ldots, A_n{:}\, d_n,$
            $B_1{:}\, e_1, \ldots, B_m{:}\, e_m)$;

Example (Merge Join)

Selection on EMP		
EMPNO	ENAME	DEPTNO
7782	CLARK	2
7839	KING	2
7934	MILLER	2
7369	SMITH	3
7876	JONES	3
7788	SCOTT	3
7566	ADAMS	6
7499	ALLEN	7
7654	MARTIN	7

DEPT	
DEPTNO	DNAME
1	ACCOUNTING
2	RESEARCH
3	SALES
4	OPERATIONS
7	SHIPPING

ADAMS violates the foreign key, but makes the example more interesting.

Index Join

- Suppose we have to compute $R \underset{A=B}{\bowtie} S$ and that there is an index on S(B).
- Then we can loop over all tuples in R and locate the corresponding tuples from S via the index.
- Since every access to S via the index potentially needs one or more block accesses, this is only useful if R contains only relatively few tuples (less tuples than S has blocks).
- Otherwise the merge sort is better.

  The index is also useful if S is small and will be completely buffered, but then there probably should be no index.

Hash Join

- The idea of the hash join is to partition both relations into small pieces by applying a hash function to the join attribute.
- Possible matches can only occur between tuples with the same hash value. Only such tuple combinations must be tried, not all tuple combinations.
- The partitioning is done such that the smaller parts fit into main memory. If needed, the partitioning step is iterated.
- Then a hash table is built in memory for each such partition and an index join is done with the corresponding partition of the other table. The result is the union of the joins of the pairs of partitions with the same hash value.

Example (Hash Join)

E.g. hash(row)=1 if DEPTNO odd, otherwise hash(row)=2:

Selection on EMP

EMPNO	ENAME	DEPTNO
7369	SMITH	3
7499	ALLEN	7
7654	MARTIN	7
7788	SCOTT	3
7782	CLARK	2
7839	KING	2
7566	ADAMS	6
7934	MILLER	2

DEPT

DEPTNO	DNAME
1	ACCOUNTING
3	SALES
7	SHIPPING
2	RESEARCH
4	OPERATIONS

Overview

1. Query Evaluation Plans, Pipelined Evaluation
2. Sorting
3. Algorithms for Joins
4. Operators in Oracle's Execution Plans
5. Appendix: Details, Program Code, Tricks

Optimizer Modes

- Oracle has four possible optimizer modes (see next chapter):
  - RULE: Older optimizer which does not depend on statistics about table sizes (rule-based optimizer).
  - FIRST_ROWS: Newer cost-based optimizer with the goal to produce the first row fast (best response time).
  - ALL_ROWS: Newer cost-based optimizer with the goal to produce the last row fast (best throughput).
  - CHOOSE: Use the cost-based approach at least one table used in the statement was analyzed.
- ALTER SESSION SET OPTIMIZER_MODE = ...;
- The QEPs shown below were generated with the rule-based optimizer.

Oracle QEPs for Simple Queries (1)

- Simple Query = Join, Selection, and Projection.
- Above we have looked at query evaluation plans for:

```
SELECT ENAME, DNAME
FROM   EMP, DEPT
WHERE  EMP.DEPTNO = DEPT.DEPTNO
```

- It should now be understandable what the merge join means and why it requires that both inputs are sorted.
- The other access plan (page 4-6) is really an index join but uses Oracle's NESTED LOOPS operator.
- It seems that Oracle tries to evaluate conditions as far down in the tree as possible (which is good).

Oracle QEPs for Simple Queries (2)

- So Oracle's NESTED LOOPS operator does not itself evaluate the join condition.
- E.g. when the current employee in the outer loop works in department 20, the condition DEPTNO=20 is passed to the right child when it is opened.
- Also the TABLE ACCESS operator in Oracle can contain a selection which is not explicitly shown. E.g. the QEP for the following query consists only of the TABLE ACCESS node:

```
SELECT EMPNO
FROM   EMP
WHERE  ENAME LIKE 'F%'
```


Oracle QEPs for Simple Queries (3)

- Suppose we change the join query by adding a condition on EMP:

```
SELECT ENAME, DNAME
FROM   EMP, DEPT
WHERE  EMP.DEPTNO = DEPT.DEPTNO
AND    EMP.SAL > 5000
```

- Here Oracle would show the same access plan as without the condition EMP.SAL > 5000.
- But of course, this condition is evaluated in the TABLE ACCESS operator before the sorting for the merge join.
- In general, one tries to evaluate selections as early as possible to make the intermediate results smaller.

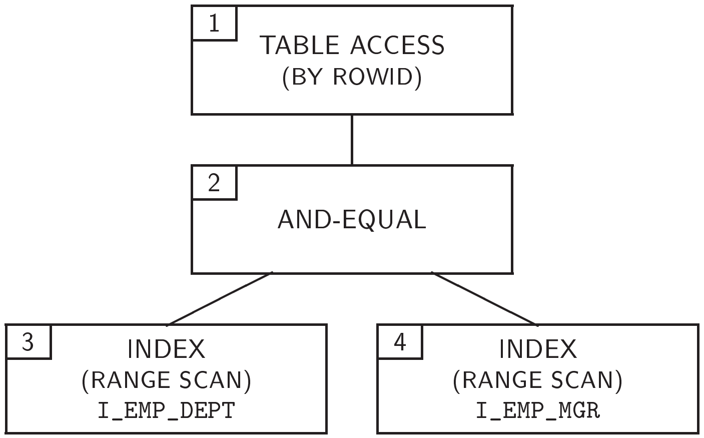
Oracle QEPs for Simple Queries (4)

- Consider the following query:

```
SELECT *
FROM   EMP
WHERE  DEPTNO = 20 AND MGR = 7566
```

- If there are indexes on both attributes, Oracle produces an evaluation plan which intersects the ROWIDs returned from both indexes (see next page).
- These access plans are computed by the rule-based optimizer.
- If you analyze the tables EMP and DEPT, the cost-based optimizer becomes applicable. It creates a full table scan in this case since the table EMP fits into one block.

Oracle QEPs for Simple Queries (5)



Oracle QEPs for Simple Queries (6)

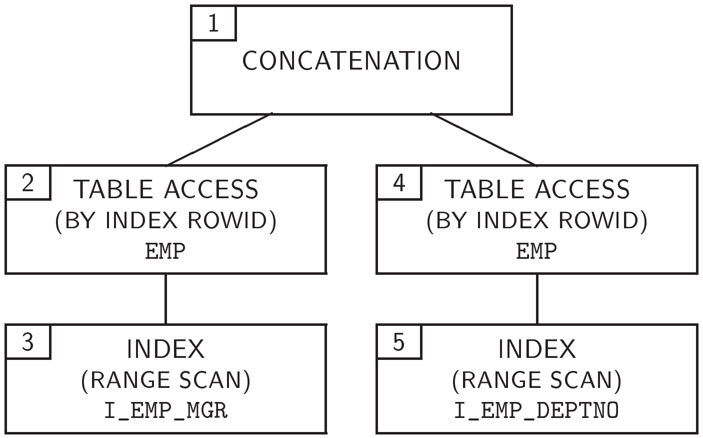
- Consider the following query:

```
SELECT * FROM EMP
WHERE  DEPTNO = 20 OR MGR = 7566
```

- If there are indexes on both attributes, the rule-based optimizer produces an evaluation plan which uses both indexes and concatenates the results (see next page).
- Tuples which satisfy both conditions are printed only once (as it should be) although the cheap concatenation does no duplicate elimination. Probably it is evaluated this way:

```
SELECT * FROM EMP WHERE MGR = 7566
UNION ALL
SELECT * FROM EMP WHERE DEPTNO = 20
                    AND NOT (MGR = 7566)
```


Oracle QEPs for Simple Queries (7)

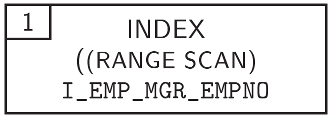


Oracle QEPs for Simple Queries (8)

- Oracle can produce index-only QEPs:

```
SELECT EMPNO
FROM   EMP
WHERE  MGR = 7566
```

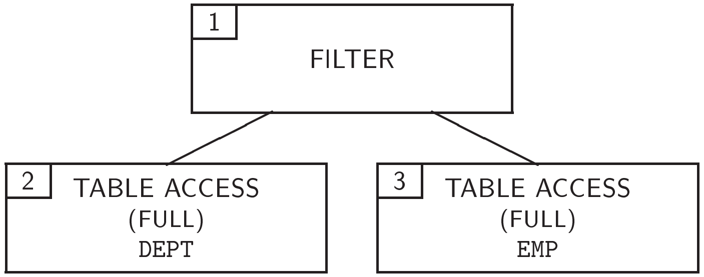
- If there is an index on `EMP(MGR, EMPNO)`, Oracle will produce this plan:



- However, if there is an index on `EMP(MGR)`, Oracle will prefer this, even though afterwards it has to access the table.

Oracle QEPs for NOT EXISTS (1)

```
SELECT *
FROM   DEPT D
WHERE  NOT EXISTS(SELECT * FROM EMP E
                  WHERE E.DEPTNO = D.DEPTNO)
```



Oracle QEPs for NOT EXISTS (2)

- `FILTER` is an operation that accepts a set of rows, eliminates some of them, and returns the rest.
  See Oracle8 Tuning, page 23-8.
- So `FILTER` is something like a selection, but Oracle never shows proper selections explicitly, they are done as a by-product in the other nodes.
- `FILTER` is only needed to evaluate complex conditions, especially with subqueries (e.g. `NOT EXISTS`).
- `FILTER` may have any number of child nodes (at least two). The left child of `FILTER` computes the rows which are to be filtered. All other childs correspond to subqueries.

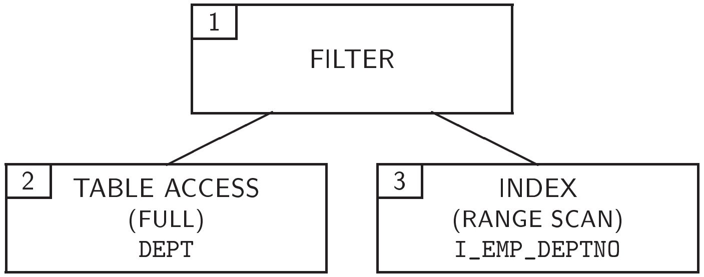
Oracle QEPs for NOT EXISTS (3)

- Consider again the query for departments without employees:

```
SELECT *
FROM   DEPT D
WHERE  NOT EXISTS(SELECT * FROM EMP E
                  WHERE E.DEPTNO = D.DEPTNO)
```

- If there is an index `I_EMP_DEPTNO` on `EMP(DEPTNO)`, Oracle produces an access plan which does not access the relation `EMP`, but only the index (see next page).

Oracle QEPs for NOT EXISTS (4)

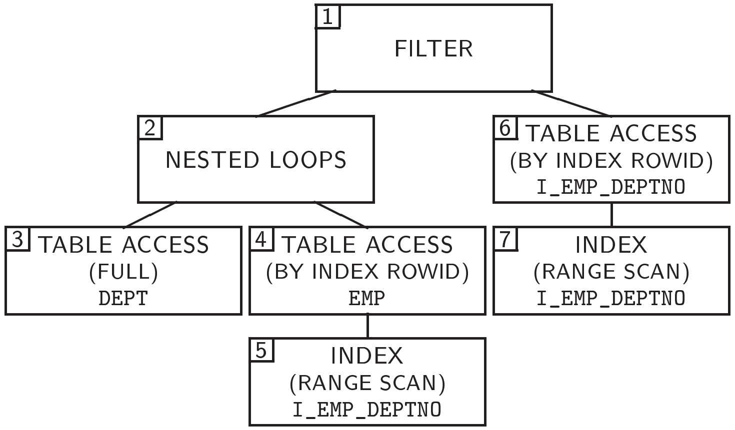


Oracle QEPs for NOT EXISTS (5)

- The set of rows which have to be filtered can also be computed by a join.
- E.g. consider the following query which computes employees in the research department which do not have a superviser in that department (the QEP is shown on the next page):

```
SELECT E.ENAME
FROM   EMP E, DEPT D
WHERE  E.DEPTNO=D.DEPTNO AND D.DNAME='RESEARCH'
AND    NOT EXISTS (SELECT * FROM EMP X
                   WHERE  X.EMPNO = E.MGR
                   AND    X.DEPTNO = D.DEPTNO)
```


Oracle QEPs for NOT EXISTS (6)



Oracle QEPs for NOT EXISTS (7)

- Oracle also has Anti-Join Operations which can be used for `NOT EXISTS` and `NOT IN` subqueries.
- An antijoin works like a join, but returns those tuples from the first relation which have no join-partner in the second relation.
- Oracle has anti-join versions of the merge join and the hash join. It seems that `FILTER` is something like an anti-join version of a nested loop join.
- You can request to use a specific kind of anti-join with the initialization parameter `ALWAYS_ANTI_JOIN`.

  But this parameter is not modifiable while the system runs.

Oracle QEPs for Aggregations (1)

```
SELECT   DEPTNO, COUNT(*)
FROM     EMP
WHERE    SAL >= 2000
GROUP BY DEPTNO
```

1 SORT
(GROUP BY)

2 TABLE ACCESS
(FULL)
EMP

Oracle QEPs for Aggregations (2)

- A query without aggregation will use the operator `SORT (AGGREGATE)`:

```
SELECT COUNT(*)
FROM   EMP
WHERE  SAL >= 2000
```

- This operator does not sort, but computes the single tuple which is the result of the aggregation.
- A query with `DISTINCT` uses the operator `SORT (UNIQUE)`:

```
SELECT DISTINCT DEPTNO
FROM   EMP
```


Oracle QEPs for Sorting (1)

- Oracle can use an index for producing a sorted sequence:

```
SELECT   *
FROM     EMP
ORDER BY EMPNO
```

1 TABLE ACCESS
(BY INDEX ROWID)
EMP

2 INDEX
(FULL SCAN)
I_EMP_EMPNO

Oracle QEPs for Sorting (2)

- But this is only possible if the sort attribute is `NOT NULL`. Otherwise, not all tuples are represented in the index.
- E.g. `ORDER BY MGR` needs a full table scan, even if there is an index on `EMP(MGR)`:

1 SORT
(ORDER BY)

2 TABLE ACCESS
(FULL)
EMP

Operations in Oracle QEPs (1)

Access Methods (Tables):

- Access methods are used for getting data from the disk (as opposed to operations which combine data, e.g. joins).
- `TABLE ACCESS (FULL)`: Retrieval of all rows from a table by reading all blocks below the high water mark.
- `TABLE ACCESS (BY ROWID)`: Retrieval of rows from a table given their ROWIDs.
- `TABLE ACCESS (CLUSTER)`: Retrieval of rows from a table stored in an index cluster, given the value of the cluster key.
- `TABLE ACCESS (HASH)`: Retrieval of rows from a table stored in a hash cluster, given the value of the cluster key.

Operations in Oracle QEPs (2)

Access Methods (Indexes):

- `INDEX (UNIQUE SCAN)`: Retrieval of a single ROWID from an index for a key, given a value for the key attribute.
- `INDEX (RANGE SCAN)`: Retrieval of attribute values and ROWIDs from an index, given an interval for the values of the indexed attributes (or a single attribute value).
- `INDEX (RANGE SCAN DESCENDING)`: As before, but attribute values are produced in descending order.

  I have not seen any Oracle QEP using this operation.
- `INDEX (FULL SCAN)`: Retrieval of all values for the indexed column in sorted order (plus the corresponding ROWIDs).

Operations in Oracle QEPs (3)

- `INDEX (FULL SCAN DESCENDING)`: Full index scan, but in inverse order (used for `ORDER BY ... DESC`).
- `INDEX (FAST FULL SCAN)`: Full index scan in the order in which the blocks are stored on the disk (reading multiple blocks in one OS call as in a full table scan).

  This operation does not return the attribute values in sorted order, but it can be useful for index-only QEPs. Branch blocks which are read are simply ignored (there are much more leaf blocks than branch blocks, so this is not a problem). In order to use this operation, the initialization parameter `FAST_FULL_SCAN_ENABLED` must be set to `TRUE` (before the system is started).

Operations in Oracle QEPs (4)

Join Methods:

- `NESTED LOOPS`: Nested Loops or Index Join.
- `NESTED LOOPS (OUTER)`: The same method, but used for an outer join (protecting tuples from one or both tables).
- `MERGE JOIN`: Merge Join (needs sorted inputs).
- `MERGE JOIN (OUTER)`: Outer join based on merge method.
- `MERGE JOIN (SEMI)`: The special case of a semi-join (only checking the existence of a join partner).
- `MERGE JOIN (ANTI)`: The merge method applied for finding tuples without a join partner (for `NOT EXISTS`, `NOT IN`).

Operations in Oracle QEPs (5)

Join Methods, Continued:

- `HASH`: A hash join (partitioning both tables).
- `HASH (SEMI)`: The special case of a semi-join.
- `HASH (ANTI)`: The hash method applied for an anti-join.

Sort Operations:

- `SORT (JOIN)`: Sorting before a merge join.
- `SORT (UNIQUE)`: Sorting and duplicate elimination.
- `SORT (GROUP BY)`: Sorting, grouping, aggregation.
- `SORT (AGGREGATE)`: Aggregation to a single row (no sort).
- `SORT (ORDER BY)`: Sorting because of an `ORDER BY` clause.

Operations in Oracle QEPs (6)

Set Operations:

- `AND-EQUAL`: Intersection of ROWIDs returned from different indexes.
- `CONCATENATION`: Union without duplicate elimination.
- `UNION-ALL`: The same (?).

  The documentation also mentions a `UNION` with duplicate elimination, but I have seen no QEP which uses it.
  For queries containing `UNION`, Oracle does a `UNION-ALL` followed by `SORT (UNIQUE)`.
- `MINUS`: Set difference (needs sorted inputs).
- `INTERSECTION`: Set intersection (needs sorted inputs).

Operations in Oracle QEPs (7)

More Access Operations:

- `BITMAP (...)`: Operations for using bitmap indexes.
- `REMOTE`: Retrieval of data from a remote database.
- `SEQUENCE`: Access to sequences.

Using `ROWNUM`:

- `COUNT`: Used for queries containing `ROWNUM` in the select list.
- `COUNT (STOPKEY)`: Used for filtering the first n input rows if the query contains `ROWNUM <=` n.
- `FIRST ROW`: Retrieval of the first row only.

  I haven't seen this in an actual QEP.

Operations in Oracle QEPs (8)

Other Operations:

- `FILTER`: Evaluate complex conditions (e.g. `NOT EXISTS`). This is similar to a `NESTED LOOPS` anti-join.
- `CONNECT BY`: For evaluating queries on tree-structured data with the `CONNECT BY` clause (not in the SQL-92 standard).
- `PARTITION (CONCATENATED)`: For partitioned tables.
- `INLIST ITERATOR`: Iterates over the operation below it.
- `PROJECTION`: Projection (appears seldom explicitly).
- `VIEW`: Used to mark subqueries which correspond to a view.
- `FOR UPDATE`: Locks the rows flowing through this operator.

QEP Structural Rules

- TABLE ACCESS (FULL) and INDEX cannot have child nodes.
- TABLE ACCESS (BY INDEX ROWID) must have exactly one child node which produces ROWIDs.
- NESTED LOOPS/MERGE JOIN must have exactly two children.
- If the query contains n tuple variables, you need $n-1$ join operators. In the 1999 exam, some students used more.
- NESTED LOOPS passes the information about the current tuple from the left child down to its right child etc.

  Only there an index on the join attribute can be used.

  Some students used the index on the join attribute in the left child, or in the right child of a merge join.

Overview

1. Query Evaluation Plans, Pipelined Evaluation
2. Sorting
3. Algorithms for Joins
4. Operators in Oracle's Execution Plans
5. Appendix: Details, Program Code, Tricks

The Oracle Plan Table (1)

EXPLAIN PLAN command:

- An alternative to "SET AUTOTRACE ON" is to use

  EXPLAIN PLAN FOR ⟨SQL QUERY⟩

  Then Oracle prints only "Explained". It does not execute the query and does not automatically show the QEP. But information about the QEP is stored in the PLAN_TABLE and can be retrieved with standard SQL queries. The rows should normally be deleted before the next EXPLAIN PLAN.
- The PLAN_TABLE can contain rows for several QEPs, then one should use e.g.

```
EXPLAIN PLAN SET STATEMENT_ID = 'MyFirstQuery'
FOR SELECT ... FROM ... WHERE ...
```


The Oracle Plan Table (2)

- The PLAN_TABLE contains one row for each node in the QEP(s) stored in it.

  More precisely, not the QEP is stored in it, but only some information about the general QEP structure. Oracle does not show all details of the QEP (e.g. selection conditions).
- Columns of the PLAN_TABLE:
  - STATEMENT_ID: This value is used to distinguish the rows belonging to execution plans for different queries.

    Normally the PLAN_TABLE contains only one plan and STATEMENT_ID is null. But see SET STATEMENT_ID above.
  - TIMESTAMP: Date and time when EXPLAIN PLAN was issued.

The Oracle Plan Table (3)

- Columns of the PLAN_TABLE, continued:
  - The following three columns describe the tree structure of the QEP: Which node gets input from which other node?
  - ID: Number which identifies this node in the tree.
  - PARENT_ID: ID of the parent node.

    The parent node gets input from this node.
  - POSITION: Order of the child nodes from left to right.
  - REMARKS: Normally null. Can be set with an UPDATE statement in order to put comments into the PLAN_TABLE.
  - OPTIMIZER: Current mode of the optimizer.

The Oracle Plan Table (4)

- Columns of the PLAN_TABLE, continued:
  - The following columns contain information about operations and their Arguments.
  - OPERATION: E.g. “TABLE ACCESS”, “MERGE JOIN”.
  - OPTIONS: E.g. “FULL” for the operation “TABLE ACCESS”.
  - OBJECT_OWNER, OBJECT_NAME: Identifies the table/index used in the operation.

    Null for operations which get input only from their children (like MERGE JOIN).
  - OBJECT_INSTANCE: Position of the table in the FROM-list.

    E.g. useful if there are two tuple variables over one table.
  - OBJECT_TYPE: “UNIQUE” or “NON-UNIQUE” for indexes.

The Oracle Plan Table (5)

- Columns of the PLAN_TABLE, continued:
  - The following attributes are only set when the cost-based optimizer is used, which means that the underlying tables must be analyzed.
  - COST: Estimated cost for this step. The number is used only for comparison, the specific value has no significance.
  - CARDINALITY: Estimated number of rows which will be returned by the operation (only for cost-based optimizer).
  - BYTES: Estimated number of bytes returned by the operation (?) (only for cost-based optimizer).

Pipelined Evaluation: Interface (1)

Interface of QEP Nodes (Example):

- The interface is very similar to an SQL cursor.

  You open the relation which is the result of this operation, fetch every tuple in a loop, and close it at the end.

  This kind of interface is also called a scan or an iterator.
- In object-oriented terms, there is an abstract class QEP_Node, with subclasses for every kind of operator.

  E.g. QEP_Node_Selection or QEP_Node_Merge_Join.
- Constructor: This class method creates a new QEP node. The parameters depend on the type of operation.

  E.g. a the constructor for QEP_Node_Selection needs the child QEP node and the selection condition.

Pipeline Evaluation: Interface (2)

- `open`: Open input.
  This method may have parameters (depending on the type of operation). E.g. the search values for an index scan.
  In this way, Information can also flow down in the tree.
- `next`: Advance input to next tuple.
  Returns false if end of input.
- `attr(i)`: Value of the `i`-th attribute of the current tuple.
  This returns a pointer to the attribute value. In this way we avoid constructing an entire new tuple for the result.
- `close`: Close input. It may then be opened again.

Pipelined Evaluation: Interface (3)

Less common operations:

- `save`/`restore`: Remember the current position in the stream of result tuples / switch back to it.
  This is needed for the merge join if the same attribute value appears more than once.
- `back`: Switch back to previous result tuple (inverse to `next`).
  This operation is needed for a zig-zag nested loop join.
- `num_attrs`: Return number of attributes in the result.
- `size`/`cost`: Estimates for number of tuples in the result and the runtime needed for computing them.
  This is useful for query optimization.

Example: Selection (1)

- Suppose we want to implement a simple selection of the form $\sigma_{\mathsf{Attr}\,=\,\mathsf{Val}}(\mathsf{Input})$.
  In a real system we must be able to pass any condition on tuples (with $\neg$, $\vee$, $\wedge$ and $<$, $>$, `like`, `is null`, ...).
- Constructor `QEP_Node_Selection(Input, AttrNo, Val)`: This operation creates a new object of this class and stores the three parameters in instance variables of this object.
- `open()`:
  ```
  Input->open();
  ```
- `close()`:
  ```
  Input->close();
  ```


Example: Selection (2)

- `next()`:
  ```
  bool End_of_Input;
  End_of_Input = Input->next();
  while(!End_of_Input
          && Input->attr(AttrNo) != Val)
      End_of_Input = Input->next();
  return(End_of_Input);
  ```
- `attr(i)`:
  ```
  return(Input->attr(i));
  ```
- `num_attrs()`:
  ```
  return(Input->num_attrs());
  ```

Sort Algorithm Optimizations

- There are many optimizations to the Mergesort algorithm, e.g.:
  - One tries to produce large initial runs by sorting chunks of the given elements in the available main memory. The longer the initial runs, the less iterations are needed later.

    Once a block of such an initial run was written to disk, one can reuse the memory page for more input elements. New elements which happen to be greater than the greatest element already written to the output can still become part of the current run, other elements have to be saved for the next run.
  - If one has k buffer frames available during the merge phase, one merges $k-1$ runs instead of only 2 runs.

Nested Loop Join: Details (1)

Methods for Pipelined Evaluation:

- `QEP_Node_NestedL(L_Child, R_Child, L_Attr, R_Attr)`
  Constructor, creates object and remembers these parameters.
- `open()`:

```
L_Child->open();
R_Child->open();
L_Attrs = L_Child->num_attrs();
L_End = L_Child->next();
```


Nested Loop Join: Details (2)

- `next()`:

```
while(!L_End) {
    R_End = R_Child->next();
    while(!R_End) {
        if(L_Child->attr(L_Attr)
                == R_Child->attr(R_Attr)
            return(true);
        R_End = R_Child->next();
    }
    R_Child->close();
    R_Child->open();
    L_End = L_Child->next();
}
return(false);
```


Nested Loop Join: Details (3)

- `attr(i)`:

```
if(i < L_Attrs)
    return(L_Child->attr(i));
else
    return(R_Child->attr(i-L_Attrs));
```

- `num_attrs()`:

```
return(L_Attrs + R_Child->num_attrs());
```

Nested Loop Join: Optimizations (1)

- If the left (outer) relation has l tuples, the right (inner) relation will be read l times. This can be improved by reading as many as possible tuples from the left relation into memory, and then doing the join with all these tuples during one pass through the inner relation.

  If n tuples from the left relation fit into the available memory, the right one will be read only l/n times.

- It is better to use the smaller relation as left (outer) relation.

  It must be read once in addition to the $l * r$ or $r * l$ tuples read in the inner loop.

Nested Loop Join: Optimizations (2)

- If one of the two relations is small enough to fit into main memory, the nested loop join needs only one pass through each of the relations and is very efficient.

  Since the smaller relation is used as outer relation and the outer relation gets all available buffer space, this happens automatically in the above scheme.

- Even if only one buffer frame can be used for the inner relation, if should be read forward and backward in alternating fashion (zig-zag). This gives one block access less in every pass (starting from the second).

Nested Loop Join: Optimizations (3)

- If the left (outer) input relation is sorted, the unoptimized version would protect the sorting. The optimized version destroys it.

  If the other (unsorted) relation fits into main memory, one can use it as inner relation to protect the sorting on the outer one.

- Since there are multiple passes through the inner input relation, it might be useful to store it explictly (not using pipelined evaluation) as compact as possible.
- This discussion should demonstrate that even for a relatively simple operation like the nested loop join, there are many optimization tricks.

Merge Join: Details (1)

- The algorithm shown above does not fit into our pipelined concept, and it is also not able to treat relations which contain duplicate values in the join attribute.
- The following program code treats these problems.
  It assumes that both input relations are sorted.
- `open()`:

```
L->open();
R->open();
empty = !L->next() || !R->next();
match_found = false;
```

Merge Join: Details (2)

```
next():
    if(empty)
        return(false);
    if(match_found) {
        if(R->next() && L->attr(A) == R->attr(B))
            return(true);
        else {
            if(!L->next()) return(false);
            R->restore();
            match_found = false;
        }
    }
```


Merge Join: Details (3)

```
while(true) {
    if(L->attr(A) == R->attr(B) {
        R->save();
        match_found = true;
        return(true);
    }
    if(L->attr(A) < R->attr(B))
        if(!L->next())
            return(false);
    if(L->attr(A) > R->attr(B))
        if(!R->next())
            return(false);
}
```