Database Systems II B: DBMS-Implementation

Chapter 14: Query Optimization

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2021/22

http://www.informatik.uni-halle.de/~brass/dbi21/

Objectives

After completing this chapter, you should be able to:

- use equivalences in relational algebra to transform a given algebra expression into a more efficient form.
- develop Oracle QEPs as Oracle's older rule-based optimizer would do it.
- explain the concept of selectivity of conditions, estimate the selectivity of given conditions.
- explain how costs of QEPs are estimated.
- expain how cost-based optimization works.

Contents

Query Optimization (1)

Goal:

- Given a query in SQL or its naive translation into relational algebra (or some similar formalism).
- Determine an efficient query evaluation plan.
- Especially, the optimizer should make good use of objects which only exist on the physical level (e.g. indexes), and which cannot be directly mentioned in the SQL query.

Declarative Languages make Powerful Optimizers ...

- Necessary: The naive execution would be too inefficient.
- Possible: They do not prescribe a specific execution model.

Query Optimization (2)

Main Tasks:

- Generation of alternative query evaluation plans.

  This can often be done using algebraic identities. The system must prove that the alternative plans are really equivalent to the given query (or its naive translation).

- Cost estimation for each generated plan and selection of the plan with the least estimated cost.

- Heuristics must be applied to limit the search space.

  The time needed for optimization should not exceed the time saved for the execution of the query. Of course, if the query is executed many times, it might be acceptable to invest more time into a thorough optimization.

Naive Translation into RA/QEPs (1)

- Given an SQL query without aggregations, subqueries, etc.:

```
SELECT  $A_1$, ..., $A_n$
FROM    $R_1$ $X_1$, ..., $R_m$ $X_m$
WHERE   $\varphi$
```

- The naive/direct translation into relational algebra is:

$$\pi_{A_1,\ldots,A_n}\Big(\sigma_{\varphi}\big(\rho_{\mathtt{X}_1}(R_1) \times \cdots \times \rho_{\mathtt{X}_m}(R_m)\big)\Big).$$

- The operation $\rho_{\mathtt{X}}(R)$ renames every attribute A to $\mathtt{X}.A$.
- This needs a relational algebra with duplicates, and the π operator here does no duplicate elimination.

  Also the selection σ must treat null values correctly.

Naive Translation into RA/QEPs (2)

Translation of Subqueries:

- Also SQL queries with subqueries can be translated into relational algebra, but the mapping is more complicated.
- System R (RDBMS research prototype, 1976) did simply evaluate correlated subqueries once for every assignment of tuples to the tuple variables of the outer query.

  So that references to the outer tuplevariables in the inner query could be replaced by constants.
- Uncorrelated subqueries were evaluated only once.
- Oracle is able to replace some applications of subqueries by joins or anti-joins.

Contents

Algebraic Identities

- You know e.g. the following identities for numbers:

  $x + y = y + x$ (commutativity law)

  $x * (y + z) = x * y \; + \; x * z$ (distribution law)

- Similar laws hold for relational algebra, e.g.

  $$\sigma_{\varphi_1}\big(\sigma_{\varphi_2}(R)\big) = \sigma_{\varphi_2}\big(\sigma_{\varphi_1}(R)\big)$$

  are equivalent.

- Two relational algebra expressions E_1 and E_2 are equivalent if for all database states $\mathcal{I}$: $\quad \mathcal{I}(E_1) = \mathcal{I}(E_2)$.

  I.e. the two queries return the same answer relation, independent of the DB state.

- Equivalence of RA queries is undecidable.

Join Order (1)

- $\bowtie$, $\times$, $\cup$, $\cap$ are commutative, e.g.

$$E_1 \times E_2 = E_2 \times E_1$$

- If we treat the sequence of attributes as important, $\times$ and $\bowtie$ are not quite commutative: we must reorder the attributes.

  E.g. if E_1 has attributes A, B, and E_2 has attribute C, the identity really is $E_1 \times E_2 = \pi_{A,B,C}(E_2 \times E_1)$. Probably, in many systems the SQL parser computes the final sequence of attributes and generates such a projection to be done at the very end. Then for all the other operations, we don't have to care about the attribute sequence.

- $\bowtie$, $\times$, $\cup$, $\cap$ are also associative (parentheses don't matter):

$$E_1 \times (E_2 \times E_3) = (E_1 \times E_2) \times E_3$$

Join Order (2)

- Consider the following query which looks for departments which have a hierarchy inside them:

```
SELECT D.DNAME
FROM   DEPT D, EMP X, EMP Y
WHERE  D.DEPTNO = X.DEPTNO
AND    D.DEPTNO = Y.DEPTNO
AND    X.MGR = Y.EMPNO
```

- An important task in query evaluation is to determine a good join order. In the example, one possibility is:

$$\pi_{\mathtt{D.DNAME}}\Big(\big(\rho_{\mathtt{D}}(\mathtt{DEPT}) \underset{\mathtt{D.DEPTNO=X.DEPTNO}}{\bowtie} \rho_{\mathtt{X}}(\mathtt{EMP})\big) \underset{\dots}{\bowtie} \rho_{\mathtt{Y}}(\mathtt{EMP})\Big)$$

- Exercise: Find more possibilities.

Join Order (3)

- Many systems (including Oracle) construct only QEPs which start with one table, join it with a second, join the result with a third, and so on:

$$\Big(\big((R_1 \bowtie R_2) \bowtie R_3\big) \bowtie R_4\Big)$$

- These joins give normally good results.

- In rare situations, a "bushy" join would be better:

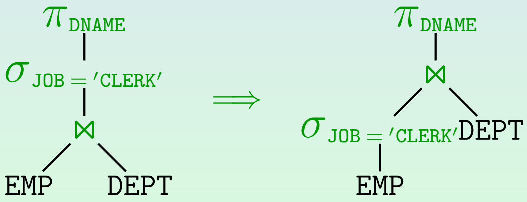
$$\big((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)\big)$$

  E.g. R_1 and R_4 are very small and R_2 and R_3 are large. The join with R_1 and R_4 might reduce the size of R_2 and R_3 (like σ).

- If one consider all possible sequences, already without bushy joins a `FROM` clause with n tables gives rise to $n!$ join orders.

Moving Selections (1)

- General Heuristic: Try to do selections as early as possible (i.e. push it down in the QEPs).
- In this way, the input relations to more expensive operations (e.g. joins) are smaller.



Moving Selections (2)

- Selections can be splitted or combined:

$$\sigma_{\varphi_1 \texttt{ AND } \varphi_2}(E) = \sigma_{\varphi_1}\big(\sigma_{\varphi_2}(E)\big).$$

- The order of selections can be exchanged:

$$\sigma_{\varphi_1}\big(\sigma_{\varphi_2}(E)\big) = \sigma_{\varphi_2}\big(\sigma_{\varphi_1}(E)\big).$$

- All kind of logical equivalence transformations can be applied to the selection condition.

  If φ_1 and φ_2 are logically equivalent:

$$\sigma_{\varphi_1}(E) = \sigma_{\varphi_2}(E).$$

- E.g. `NOT SAL` < 3000 is equivalent to `SAL` >= 3000, and the second condition may make an index applicable.

Moving Selections (3)

- A selection can be moved past $\times$ and $\bowtie$ if its selection condition φ contains only attributes of one relation (e.g. E_1):

  If φ contains only attributes of E_1:
  $$\sigma_{\varphi}(E_1 \times E_2) = \Big(\sigma_{\varphi}(E_1)\Big) \times E_2.$$

- A selection condition can also be moved past $\cup$, $\cap$, $\setminus$:
  $$\sigma_{\varphi}(E_1 \cup E_2) = \sigma_{\varphi}(E_1) \cup \sigma_{\varphi}(E_2).$$

- The system should detect joins built from σ and $\times$ (since there are more efficient algorithms for joins):

  If A is an attribute from E_1, and B from E_2:
  $$\sigma_{A=B}(E_1 \times E_2) = E_1 \underset{A=B}{\bowtie} E_2.$$

Moving Selections (4)

- If the system detects that a condition is contradictory (e.g. `YEAR=1997 AND YEAR=1998`), the entire selection can be removed:
$$\sigma_{\texttt{FALSE}}(E) = \emptyset.$$
- Such conditions may occur when views are expanded.

  See above under partitioned tables. If the user wrote such a condition, he/she should probably be informed.

- There are also rules for simplifying relational algebra expressions containing $\emptyset$, e.g. $E \times \emptyset = \emptyset$.
- Conditions which are equivalent to `TRUE` (e.g. `YEAR=1997 OR YEAR <> 1997`) need no selection:
$$\sigma_{\texttt{TRUE}}(E) = E.$$

Moving Selections (5)

Exercise:

- Consider the following SQL query:

```
SELECT E.ENAME
FROM   DEPT D, EMP E
WHERE  D.LOC = 'BOSTON'
AND    D.DEPTNO = E.DEPTNO
AND    E.SAL >= 2800
```

- The naive translation into relational algebra is

$$\pi_{\mathtt{E.ENAME}}\Big(\sigma_{\mathtt{D.LOC} = \mathtt{'BOSTON'}\ \mathtt{AND}\ \mathtt{D.DEPTNO} = \mathtt{E.DEPTNO}\ \mathtt{AND}\ \mathtt{E.SAL} >= \mathtt{2800}} \big(\rho_{\mathtt{D}}(\mathtt{DEPT}) \times \rho_{\mathtt{E}}(\mathtt{EMP})\big)\Big)$$

- Optimize this step by step by using the above equations.

Moving Selections (6)

- Note that “Do selections as early as possible” is only a heuristic rule. Normally it is right, but there are exceptions.
- Suppose that `DEPT` is quite large, there are many departments in New York, few employees who earn at least $3000, and there is an index on `DEPT(DEPTNO)`, but no index on `DEPT(LOC)`.
- Then it would make things worse to move the selection `LOC = 'NEW YORK'` before the join:

$$\pi_{\texttt{DNAME}}\Big(\sigma_{\texttt{LOC='NEW YORK'}}\big(\sigma_{\texttt{SAL>=3000}}(\texttt{EMP}) \bowtie \texttt{DEPT}\big)\Big)$$

Moving Projections (1)

- Projections are normally not done explicitly. (At least with our QEP interface, attributes are only accessed as needed.)
- However, it is good to know at each step which of the attributes are actually needed.
- E.g. when an intermediate result has to be sorted, one wants to store the tuples as compactly as possible, and especially retain only the needed attributes.
- Also, index-only access plans may become available when not all attributes of the relation are needed.
- Projections with duplicate elimination on intermediate results is worth the effort only in rare circumstances.

Moving Projections (2)

- Projections can be splitted or combined:

  If the attributes in $\mathcal{A}$ are a subset of those on $\mathcal{B}$, which are in turn a subset of all attributes of E:

  $$\pi_{\mathcal{A}}\big(\pi_{\mathcal{B}}(E)\big) = \pi_{\mathcal{A}}(E).$$

- Projections can be moved past selections:

  If φ accesses only attributes in $\mathcal{A}$:

  $$\pi_{\mathcal{A}}\big(\sigma_{\varphi}(E)\big) = \sigma_{\varphi}\big(\pi_{\mathcal{A}}(E)\big).$$

- There are also rules for moving projections past other operations, e.g. joins. You only need to check which attributes are later still accessed.

Query Normalization in Oracle (1)

- After parsing the SQL query, and before QEPs are generated, Oracle replaces some constructs by equivalent constructs.
- SQL often allows equivalent formulations, and in this way the optimizer does not have to handle them all.
- Also, some of these transformations make specific optimizations applicable later.
- It is good to know about these normalizations, since then you will not wonder later which version of a query is more efficient if Oracle anyway treats them the same.
- Studying these transformations might also help you to improve your SQL knowledge.

Query Normalization in Oracle (2)

- Oracle does evaluate constant expressions, e.g. `SAL >= 24000/12` is transformed into `SAL >= 2000`.
- Oracle does not move operands from one side of a condition to the other side. E.g. `SAL*12 >= 24000` is not changed.

  And in this way an index over `EMP(SAL)` cannot be used.
- Oracle detects when the `LIKE` operator is really an equality, e.g. `ENAME LIKE 'Smith'` is mapped to `ENAME = 'Smith'`.

  There is a subtle problem with the blank-padded semantics here. If `ENAME` is of type `CHAR(10)`, the `LIKE` condition could be replaced by `FALSE` (`LIKE` uses the non-padded semantics), whereas `ENAME = 'Smith'` could return results.

Query Normalization in Oracle (3)

- `IN` with a list of values is transformed into `OR`. E.g. `ENAME IN ('Smith', 'Jones')` is transformed into `ENAME = 'Smith' OR ENAME = 'Jones'`.
- The `BETWEEN` operator is also removed. E.g.

  ```
  SAL BETWEEN 1000 AND 2000
  ```

  is replaced by

  ```
  SAL >= 1000 AND SAL <= 2000.
  ```

- `NOT` is moved down to the atomic conditions. E.g.

  ```
  NOT (SAL < 1000 OR COMM IS NULL)
  ```

  is transformed into

  ```
  SAL >= 1000 AND COMM IS NOT NULL)
  ```

Query Normalization in Oracle (4)

- ANY, SOME, and ALL are removed.
- E.g. `SAL >= ALL (1000, LOW_SAL)` is replaced by

```
SAL >= 1000 AND SAL >= LOW_SAL
```

- ANY with a subquery is normalized to EXISTS. E.g.:

```
X.SAL >=ANY (SELECT Y.SAL FROM EMP Y
                   WHERE Y.JOB = 'PROGRAMMER')
```

  is transformed to

```
EXISTS(SELECT Y.SAL FROM EMP Y
       WHERE Y.JOB = 'PROGRAMMER'
       AND X.SAL >= Y.SAL)
```

- `X.SAL >=ALL(...)` is mapped to `NOT(X.SAL <ANY(...))`, and then the above transformation yields NOT EXISTS.

Query Normalization in Oracle (5)

- The cost-based optimizer can compute certain implied conditions. E.g. given the following query:

```
SELECT E.ENAME, DNAME
FROM   DEPT D, EMP E
WHERE  D.DEPTNO = E.DEPTNO AND E.DEPTNO = 20
```

- The optimizer concludes that then also `D.DEPTNO = 20`, which might be useful for applying an index in a QEP with `DEPT` in the outer loop of a nested loop join.

- Oracle only generates conditions of the form

$$\langle\text{Column}\rangle\ \langle\text{Comparison}\rangle\ \langle\text{Constant}\rangle$$

E.g. `X.DEPTNO=Y.DEPTNO` is not derived in the query on slide 14-11.

Contents

Rule-Based Optimizer in Oracle (1)

- Consider the query

```
SELECT E.ENAME, D.DNAME, G.GRADE
FROM   EMP E, DEPT D, SALGRADE G
WHERE  E.DEPTNO = D.DEPTNO
AND    E.SAL BETWEEN G.LOSAL AND G.HISAL
AND    D.LOC = 'DALLAS' AND E.JOB = 'CLERK'
```

- The rule-based optimizer will generate 3 QEPs:
  - One accessing EMP first,
  - one starting the evaluation with DEPT,
  - and one accessing SALGRADE before the other two.

If the FROM list contains *m* tuple variables, *m* QEPs are produced.

Rule-Based Optimizer in Oracle (2)

- So first it tries to start the evaluation with `EMP`.
- It notices that the only condition which can be evaluated at this point is `E.JOB = 'CLERK'`.
- The rule-based optimizer has a priority list of available access paths (see below).
- E.g. using an index is higher on the list than a full table scan. The optimizer chooses the first available access path on the list.
- So if an index on `EMP(JOB)` exists, it will be used, if it does not exist, the only possibility is to use a full table scan.

Rule-Based Optimizer in Oracle (3)

- Then Oracle chooses a next relation, to be joined with the current intermediate result.
- So the optimizer must decide whether to join $\sigma_{\texttt{JOB} = \texttt{'CLERK'}}(\texttt{EMP})$ with DEPT or SALGRADE first.
- In order to do this, it evaluates the access paths for each of the possibilities and chooses the relation for which the join is considered cheapest (this is a greedy algorithm).
- If an index on DEPT(DEPTNO) exists, this access path would be 4th on the priority list, whereas using a condition like E.SAL >= G.LOSAL even with an index on SALGRADE(LOSAL) would be ranked 11th.

Rule-Based Optimizer in Oracle (4)

- If an index on `DEPT(LOC)` exists, this would also be an alternative, but single column indexes on non-key attributes are only ranked 9th on the list.
- So the rule-based optimizer would do the join with `DEPT` first, and use an index join (`NESTED LOOPS` in Oracle).
- Immediately after the join it will also evaluate the condition `D.LOC = 'DALLAS'`.
- Then finally it has to do the join with `SALGRADE`:
    - If an index exists on `SALGRADE(LOSAL)` or `SALGRADE(HISAL)`, this is used.
    - If no such index exists, a full table scan is done.

Access Path Priority List (1)

1. Access by `ROWID`: The query contains either a constant ROWID (`ROWID = '...'`) or a join `ROWID = R.A`, where `R` is already accessed, so `R.A` is known.
2. Single row by cluster join: E.g. `EMP` and `DEPT` are contained in the same cluster (by `DEPTNO`), and `EMP` is already accessed. Then the block containing the corresponding row of `DEPT` is already in the buffer.
3. Single row by hash cluster key: E.g. `DEPT` is stored in a hash cluster by `DEPTNO`, and the value for `DEPTNO` is known (either a condition like `DEPTNO=20` or `D.DEPTNO=E.DEPTNO` and `E` is already accessed). This needs ideally one block access.

Access Path Priority List (2)

4. Index over unique or primary key: E.g. accessing DEPT via an index over DEPT(DEPTNO) either because of a selection DEPTNO=20 or in form of an index join with a table which was already accessed.

   These first four possibilities will not extend the number of tuples in the intermediate result, since they all use a key for accessing the next table.

5. Cluster join: E.g. DEPT and EMP are stored together in a cluster over DEPTNO. If DEPT was already accessed before, the tuples in EMP are probably loaded with them.

6. Hash cluster (non-key): E.g. EMP is stored in a hash cluster on DEPTNO. The query contains a condition like DEPTNO=20.

Access Path Priority List (3)

7. Indexed cluster (non-key): As 6., but with an index cluster.
8. Composite index (non-key): E.g. using an index on `EMP(DEPTNO,JOB)` for `DEPTNO=20 AND JOB='CLERK'`.
9. Single-column index (non-key): E.g. using an index on `EMP(DEPTNO)` for evaluating `DEPTNO=20`. Also intersecting `ROWID`s from different indexes falls into this category.
10. Bounded-range search on indexed columns: E.g. using an index on `EMP(SAL)` for `SAL >= 2000 AND SAL <= 3000` (interval bounded on both sides). Also using an index for `ENAME LIKE 'F%'` and using an index on only a prefix of the index columns are bounded-range searches.

Access Path Priority List (4)

11. Unbounded-range search on indexed columns: E.g. using an index on `EMP(SAL)` for evaluating `SAL >= 2000`.

12. Merge-Join.

    Index joins are higher ranked. A merge join is done when no index is available and the join condition is an equality.

13. `MAX` or `MIN` of indexed column: Using an index for computing the maximal value of a column.

    Oracle seems nevertheless to do a full scan of the index.

14. `ORDER BY` on indexed columns: Using an index to get the tuples in sorted order.

15. Full table scan.

Exercises

- How will Oracle (with the rule-based optimizer) evaluate this query?

```
SELECT E.ENAME
FROM   EMP E
WHERE  DEPTNO = 20
AND    SAL >= 2000
AND    ENAME LIKE 'F%'
```

  How does the answer depend on the existence of indexes over `EMP(DEPNO)`, `EMP(SAL)`, and `EMP(ENAME)`?

- Compute the other two QEPs for the query on slide 14-27 (starting with `DEPT` and starting with `SALGRADE`).

Rule-Based QEP Selection

- Among the generated QEPs, the rule-based optimizer chooses the one with the smallest number of nested-loop joins (Proper nested loop joins, where the inner table is accessed with a full table scan.).
- If this does not bring a decision, the optimizer chooses the QEP with the smallest number of merge joins.
- If there is still a tie, it chooses the plan with the more efficient access path to the first table.
- Finally it uses the sequence in the `FROM` list.

  It chooses the plan of which the relation accessed first appears later in the `FROM` list.

Contents

Cost-Based Optimization (1)

- The cost-based optimizer generates a larger number of alternative query evaluation plans.

  The Oracle documentation does not say much about this, but e.g. a full table scan is considered even if there is an index. The rule-based optimizer checks the access paths on this list in the given order and stops once it has found an available path.

- It then estimates the cost of each generated plan and picks the cheapest.

  It is possible to use cost estimates already on partial plans while they are generated in order to restrict the search space of possible alternatives.

Cost-Based Optimization (2)

- In order to estimate the costs, all tables should be analyzed.

  If there are no statistics in the data dictionary, the optimizer will guess them based on the number of blocks allocated to the table. However, this will often result in sub-optimal plans.

- Important input data are:
  - Size of tables (number of rows, number of blocks)
  - Key constraints, foreign key constraints
  - Distribution of values for each attribute used in conditions (number of different values, minimal/maximal value).
- It might also be useful to compute histograms (see below).

Cost-Based Optimization (3)

- With the cost-based approach, the chosen QEP may depend on the constant values in the query. E.g. the optimizer may treat these queries differently:

```
SELECT * FROM EMP WHERE SAL >= 1000
SELECT * FROM EMP WHERE SAL >= 5000
SELECT * FROM EMP WHERE SAL >= :X
```

- Some systems defer certian decisions to runtime (“runtime optimization).

  E.g. they first assume that a certain intermediate result will be small, but as soon as more than a few tuples are computed, they switch to a QEP for larger results.

Selectivity (1)

- Suppose we have a query

```
SELECT *
FROM   Customers
WHERE  SEX = 'M' AND AGE >= 80
```

- If there are indexes over both attributes, it is better to use the one over AGE, since it will select less rows.
- An important part of cost-based query optimization is the estimation of the "selectivity" of conditions, that is the percentage of rows which will satisfy the condition:

$$\frac{\text{Number of rows in } R \text{ which satisfy the condition } \varphi}{\text{Total number of rows in } R.}$$

Selectivity (2)

- So the selectivity of a condition ranges from 0 to 1, and the smaller it is, the better.

  You can understand the selectivity also as the probability that a row will fulfill the condition.

- The optimizer can only use estimates for the selectivity.

  The actual selectivity is only known after the query is executed (and varies if the QEP is executed multiple times).

- Bad estimates can lead to sub-optimal query evaluation plans, but usually a high accuracy is not needed.

- The selectivity is used to estimate the number of rows in intermediate results.

Selectivity (3)

Selectivity of $A = c$:

- If A is a primary key, only one row can satisfy this condition.

  So the selectivity esitimate is $1/\text{number of rows}$.

- If the number of different values of A is n, the selectivity of $A = c$ is estimated to be $1/n$.

  This assumes a uniform distribution. When the table is analyzed, Oracle stores the number of different data values in each column in the table `COLS`, column `NUM_DISTINCT`.

- If nothing was known about the column values, System R guessed a selectivity of 0.1. Probably Oracle uses a similar value if the table is not analyzed.

- Exercise: What is the selectivity of `SEX = 'M'`?

Selectivity (4)

Selectivity of $A > c$:

- If A is of numerical type, and the minimal and the maximal value of A are known, the selectivity of $A > c$ can be estimated as
$$\frac{\max(A) - c}{\max(A) - \min(A)}$$
This assumes that there are very many different possible values.

- Otherwise System R has estimated it as 0.3.

- Oracle uses the above formula also for string-valued columns, and uses the internal string encoding (e.g. ASCII values).

  Maximal and minimal value in hexadecimal are stored in the columns `LOW_VALUE` and `HIGH_VALUE` of the table `COLS`.

Selectivity (5)

- The selectivity of φ_1 `AND` φ_2 can be estimated as $s_1 * s_2$, if s_1 and s_2 are the selectivity estimates for φ_1 and φ_2.
- This assumes that the conditions are independent.
- E.g. `SAL >= 1000 AND SAL <= 2000` must be treated specially. Oracle uses the formula $s_1 + s_2 - 1$ in this case.
- The selectivity of φ_1 `OR` φ_2 can be estimated as $s_1 + s_2 - s_1 * s_2$ (again assuming independence).
- Selectivity can also be estimated by sampling: If 5 rows out of 100 randomly chosen rows satisfy the condition, the selectivity is estimated as 0.05.
- However, this needs DB accesses (expensive method).

Histograms (1)

- Suppose that 90% of your customers are from Pittsburgh, but you also have customers from 99 other cities.

  The above formulas would estimate the selectivity of `CITY = 'PITTSBURGH'` as 0.01 (1%).

- Suppose that the director of a company makes a yearly salary of \$530 000, but the salaries of the other employees are mostly in the range \$30 000 to \$70 000.

  The above formulas would estimate the selectivity of `SAL < 100 000` as 0.2 (20%).

- Oracle allows to create histograms for such non-uniformly distributed data.

Histograms (2)

- You can request that a histogram is computed on a specific column with a version of the ANALYZE command:

```
ANALYZE TABLE EMP
COMPUTE STATISTICS FOR COLUMNS SAL SIZE 10
```

- Then Oracle will sort the table EMP on the attribute SAL, split it into 10 groups (intervals) containing the same number of rows, and write the maximal SAL value in each interval into the table

```
USER_HISTOGRAMS(TABLE_NAME, COLUMN_NAME,
        ENDPOINT_NUMBER, ENDPOINT_VALUE)
```

  In addition, MIN(SAL) is stored as endpoint number 0.

  String-valued column values are shown as numbers.

Histograms (3)

EMP	
···	SAL
···	800
···	950
···	1100
···	1250
···	1250
···	1300
···	1500
···	1600
···	2450
···	2850
···	2975
···	3000
···	3000
···	4000
···	50000

USER_HISTOGRAMS			
TAB···	COL···	ENDPOINT_NUMBER	···VALUE
EMP	SAL	0	800
EMP	SAL	1	1100
EMP	SAL	2	1300
EMP	SAL	3	2450
EMP	SAL	4	3000
EMP	SAL	5	50000

Histograms (4)

- Histograms are not automatically computed when you analyze the table.

  However, there will be entries for the minimum and maximum column value in `USER_HISTOGRAMS`.

- In the example with nearly all customers coming from Pittsburgh, the value “Pittsburgh” will be the endpoint of most intervals. In this way, Oracle notices that the selectivity of `CITY='Pittsburgh'` is not good.

  It would also be possible to store the number of different values in each of the intervals (not done in Oracle).

- Queries in Embedded SQL which use a program variable in place of a constant value do not profit from histograms.

Cost Estimation (1)

- The main cost factor is the number of blocks read.

  Some systems might also consider CPU costs, but it has a much smaller weight. The CPU is usually not the bottleneck in DBMSs, unless they have large amounts of memory. Often, the CPU will be idle waiting for the disk.

- In order to compute the cost of a QEP node, an estimate for the number of result rows of its child nodes is needed.

- If we do nested loop joins, it might be good to have a method for estimating the cost of executing a QEP n times.

  This may be cheaper than n times the cost of a single execution: buffer cache, storing intermediate results.

- Oracle's cost formulas are not published.

Cost Estimation (2)

Full Table Scans:

- A full table scan needs to read all blocks below the "high water mark" (column `BLOCKS` in `TABS`).
- But reading these blocks should be given a big discount, since they are stored consecutively on the disk.

  If the number of extents is small (ideally 1). The number of extents is stored in `USER_SEGMENTS.EXTENTS`.

- Also, a full table scan becomes cheaper if the parameter

  `DB_FILE_MULTIBLOCK_READ_COUNT`

  (number of blocks to read in a single OS call) is larger.

  For Oracle on our Solaris systems, it is set to 8.

Cost Estimation (3)

Index Scans:

- For a unique index scan, the number of blocks is the height of the B-tree.

  `IND` contains this information (`BLEVEL+1`).

- The root node will often be in the buffer cache.

- For other index scans, the number of leaf blocks accessed can be estimated based on the selectivity of the condition and the total number of leaf blocks.

  `IND.LEAF_BLOCKS` is the total number of leaf blocks. E.g. suppose it is 50. It the condition has a selectivity of 0.1 (10%), we will assume that 5 leaf blocks will be accessed. In addition `IND.BLEVEL` branch blocks will be accessed.

Cost Estimation (4)

Table Access by ROWID:

- In principle, every `ROWID` can request a different data block.
- So "Table Access by ROWID" can cost as many block accesses as it has input ROWIDs.

  The number of input ROWIDs can be estimated from the selectivity of the condition and `IND.NUM_ROWS`.

- However, if the table is small compared with the buffer cache, no block will be read twice.

  So for small tables, their number of blocks (`TABS.BLOCKS`) is an upper limit to the cost of "Table Access by ROWID".

- `AVG_DATA_BLOCKS_PER_KEY` and `CLUSTERING_FACTOR` (in `IND`) might indicate that the ROWIDs are not randomly distributed.

Cost Estimation (5)

Nested Loop Join (Unoptimized Version):

- Suppose the left child needs b block accesses and returns n rows. The nested loop join will then cost b block accesses plus the cost of executing the right child n times.
- Suppose L has 40 rows, stored in 1 block, and R has 100 rows stored in 50 blocks. Without caching, the nested loop join $L \bowtie R$ will need $1 + 40 * 50 = 2001$ block accesses.
- The nested loop join $R \bowtie L$ will need $50 + 100 * 1 = 150$ block accesses.
- So for the unoptimized nested loop join, it is not always better to use the smaller relation in the outer loop.

Cost Estimation (6)

- It is also not always better to use the larger relation in the outer loop.
- E.g. suppose that L has 1000 rows stored in 100 blocks and R has 10 rows stored in 5 blocks.
- Then $L \bowtie R$ will need $100 + 1000 * 5 = 5100$ block accesses, and $R \bowtie L$ will need $5 + 10 * 100 = 1005$ block accesses.
- So the nested loop join behaves asymmetric, and the cost of both variants must be estimated and compared.
- This also clarifies why Oracle tries to avoid a full table scan as the right child of a nested loop join with higher priority than on the left side: As right child, it is done repeatedly.

Cost Estimation (7)

- For the unoptimized version, if one relation is small enough to fit into the buffer, it is better to use it as inner relation.

  Since the outer relation will anyway be read only once.

- However, Oracle uses only a small number of buffer frames for full table scans (buffer frames are immediately reused).

  Older versions of Oracle had a parameter `_SMALL_TABLE_THRESHOLD` for the number of buffers for a full table scan that will be normally cached using LRU (Default 5). Since Oracle 7.1, the `CREATE TABLE` command has an option `CACHE` which requests to do normal caching for full table scans for this table. You should set this for small tables which are right arguments in nested loop joins.

Cost Estimation (8)

Nested Loop Join, Optimized Version:

- For the unoptimized version, the question whether one table fits into memory is all or nothing: If it does not fit entirely, very little is gained.
- The optimized version can make good use of any amount of buffer space available.
- E.g. if the smaller table is double the size of the available memory, the larger table has to be read twice.
- In the optimized version, the buffer space is used for the outer table in order to process as many tuples as possible from the outer table for each pass through the inner table.

Cost Estimation (9)

Result Size Estimation for Joins:

- In order to estimate the cost of the parent nodes, we need to estimate the number of tuples produced by the child nodes.

  E.g. if we join three tables, we must know the size of the intermediate result after joining two tables.

- For the join $R \underset{A=B}{\bowtie} S$, if $R.A$ is a foreign key referencing $S.B$, the result will have exactly as many tuples as R has.

- For the cartesian product $R \times S$, the result will have $|R| * |S|$ tuples (where $|R|$ denotes the number of tuples in R).

- Other join estimates are beyond the scope of this course.

Cost Estimation (10)

Index Joins:

- Suppose we evaluate $R \underset{A=B}{\bowtie} S$ with an index on $S.B$.
- If R returns n rows, we have to do n index lookups and table accesses by ROWID on S.

  This is very similar to a nested loops join, which is why Oracle has only one `NESTED LOOPS` operator for both joins.
- If n is small and S is large, this join is cheapest, since in this way we can avoid reading all tuples of S.

  All other joins need to look at each tuple at least once.
- If S is small, all of its blocks will end up in the cache.

  But then a nested loops join is also very efficient.

Cost Estimation (11)

Merge Join:

- The main cost of the merge join is the sorting phase.

  The merging can be done on the fly when the last sort step produces the tuples.

- If the sorting can be done in main memory, the number of block accesses will be the sum of the costs of the children.

  Due to pipelined evaluation, no block accesses are added.

- If the input is b blocks, and we have m buffer frames available ($m < b$), mergesort needs $2 * b * ceil\big(\log_{m-1}(n)\big)$ block read/writes where $n = ceil(b/m)$.

  n is the number of initial runs.

  $ceil$ rounds to the next heigher integer.

Comparison of Join Methods

- Suppose we have to evaluate $R \underset{A=B}{\bowtie} S$.
- If R is small (few tuples) and S is large, and there is an index on $S.B$, choose an index join.

  And vice versa: R large, S small, index on $R.A$.
- Otherwise, if one relation is small (fits into memory), the other is large, but no index is available for the large relation, choose the nested loops join.

  Since this avoids sorting the large relation.
- If both relations are small, it doesn't matter.
- If both relations are large, and no index is available, the merge join or the hash join are best.

Optimizer Hints (1)

- If you discover that the optimizer produces a bad QEP, you can try to give it hints for a better QEP.

  Only the cost-based optimizer understands hints.

- Hints have the form of special comments:

```
SELECT /*+ ORDERED */ ENAME, DNAME
FROM   EMP E, DEPT D
WHERE  E.JOB = 'CLERK' AND D.LOC = 'BOSTON'
AND    E.DEPTNO = D.DEPTNP
```

- `/*+ ORDERED */` means to access the relations in the order given in the `FROM` clause.

- You can specify several hints in one comment, e.g.

```
/*+ ORDERED INDEX(E I_EMP_JOB) */
```

Optimizer Hints (2)

- Oracle does not print an error message if such a hint contains an error or can not be respected.
- In these cases, the hint is treated as a comment (ignored).

  One reason for this is that it is legal to intersperse hints with normal comments (e.g. explaining the hints).
- Hints must be put immediately after the first keyword (`SELECT`, `UPDATE`, `DELETE`). No space between `/*` and `+`.
- Hints apply only to their statement block.

  E.g. if you have two `SELECT` queries, combined by `UNION`, you must put separate hints into both parts. Also subqueries might need their own hint.
- Hints must use the tuple variable names, not table names.

Optimizer Hints (3)

- Hints for optimizer selection:

  `ALL_ROWS`, `FIRST_ROWS`, `CHOOSE`, `RULE`.

- Hints for access methods:

  `FULL`(table), `ROWID`(table), `CLUSTER`(table), `HASH`(table), `HASH_AJ`, `HASH_SJ`, `INDEX`(table index ...), `INDEX_ASC(...)`, `INDEX_COMBINE(...)`, `INDEX_DESC(...)`, `INDEX_FFS(...)`, `MERGE_AJ`, `MERGE_SJ`, `USE_CONCAT_SJ`, `AND_EQUAL`(table index index ...),

- Hints for joins:

  `ORDERED`, `STAR`, `USE_NL(table ...)`, `USE_MERGE(table ...)`, `USE_HASH(table ...)`,

- More hints for parallel execution, views, caching, ...

Contents

Conclusion (1)

- SQL is a declarative language, so when you write queries, you should not worry too much about query evaluation.
- However, this depends on how good the optimizer of your system is and which performance you have to reach.
- If your database is large and you have to support many concurrent users, performance becomes an issue.
- There should be quantifiable performance requirements.
- If you discover performance problems only when the system goes into production, you have a big problem.
- It is much cheaper to think about performance already during design and development.

Conclusion (2)

- Some optimizations in SQL formulations can be done without making the query more complicated:
  - E.g. write `SAL > 24000/12` instead of `SAL*12 > 24000`.

    In Oracle, any datatype function applied to a column makes an index unusable for that column.
  - E.g. avoid unnecessary complications (in any case better).

    E.g. some students used `UNION` when a simple `OR` would have sufficed, some used `DISTINCT` without need, or `GROUP BY` a key, unnecessary subqueries under `FROM`, etc.
  - E.g. express “for all” with comparing counts instead of a doubly nested `NOT EXISTS` subquery.

    Nested subqueries might be difficult for some optimizers.

Conclusion (3)

- If performance is an issue, the application programmers should think about which indexes can be used for their queries, and check whether the indexes exist.
- Sometimes redundant data must be added to the schema.
- E.g. `UPPER(NAME) = :X` requires to store `NAME` in uppercase in the database, or indexes are not usable.
- Sometimes (very seldom) redundant data must be stored for performance reasons ("denormalization").
- However, index selection and denormalization should be done before the actual programming begins.
- The later the change, the more expensive it is.

Conclusion (4)

- However, with regard to QEP selection (only that!), I would suggest a reactive, rather than a proactive approach:
  - Try to write good SQL, create the indexes/clusters which seem useful (based on your knowledge from this course).
  - Check the QEP only for a few especially critical queries.
  - Test your application programs under actual system loads.
  - If there is a performance problem, find out which queries use the most system resources and tune these queries.
  - I.e. keep performance in mind over the entire life-cycle, do what is sensible (the earlier, the better), but invest your time where it is most useful to satisfy the goals.

Conclusion (5)

- Reasons for reactive approach:
    - QEPs might change if your table sizes etc. change.
    - With a new version of Oracle, the optimizer might have changed, and different QEPs can be produced.
    - Also, adding an index later in order to improve a certain query might confuse the optimizer, so that it produces worse QEPs for other queries.
    - Unless you put optimizer hints in every query or use the rule-based optimizer (which is unlikely to change), QEPs are not sufficiently stable to invest too much work in them.
    - After all, you program in SQL and not in QEPs.

Conclusion (6)

Good performance depends also on many other issues, e.g.:

- The right setting of DBMS and OS parameters.
- If you have many concurrent users, the reason for delays might be locks by other users.
- Your interface to the DBMS server:
  - It might be more efficient to do more work on the server by means of stored procedures (less network traffic).
  - In Embedded SQL, fetching whole arrays of results is more effient than fetching only a tuple at a time.
  - Using static Embedded SQL rather than dynamic SQL.
- Good database and application program design.

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. 13: "Introduction to Query Optimization", Chap. 14: "A Typical Relational Query Optimizer".
- 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.