Part 6: 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:

- explain 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.
- develop different query evaluation plans for a given query and assess their merits.

Overview

Introduction (1)

- A query evaluation plan (or "execution plan") QEP is a program for an abstract machine (interpreter) inside the DBMS.

  Another name is "access plan" (the DBMS has to decide how to access the rows, e.g. whether to use an index).
- QEPs are internal representations of the query produced by the query optimizer.

  By executing the QEP, the query result is computed. Whereas SQL is declarative, QEPs describe a concrete way for evaluating teh query.
- In most systems, QEPs are similar to relational algebra expressions (very system dependent).

Introduction (2)

- In this chapter, we use a standard example database from Oracle about employees and departments:

  EMP(<u>EMPNO</u>, ENAME, JOB, SAL, MGR→EMP, DEPTNO→DEPT)
  DEPT(<u>DEPTNO</u>, DNAME, LOC)

- Consider the following SQL query:

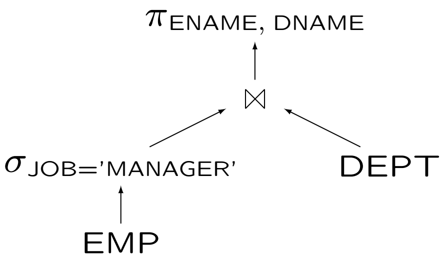
```
SELECT ENAME, DNAME
FROM   EMP, DEPT
WHERE  EMP.DEPTNO = DEPT.DEPTNO
AND    JOB = 'MANAGER'
```

- In relational algebra, this is:

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

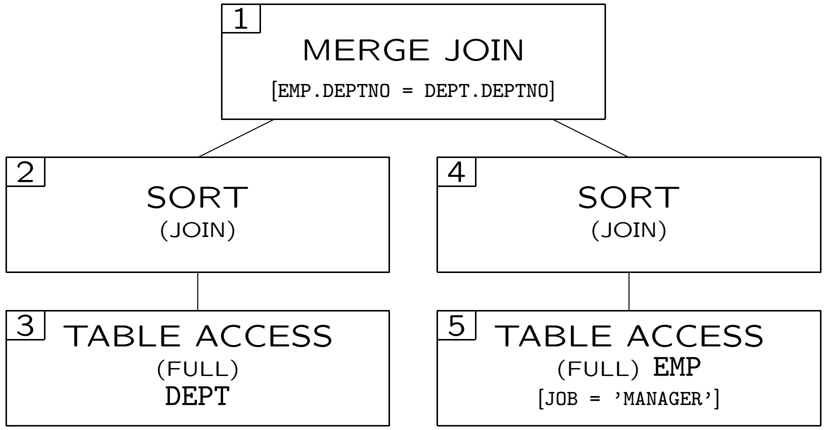
Introduction (3)

- Complex relational algebra expressions are best displayed as "operator trees":



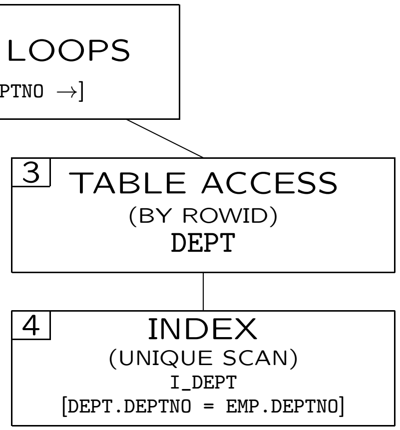
This shows the flow of data. One can view relations/tuples as being pushed from the base relations in the leaf nodes through the relational algebra operators towards the root, where the final result is computed.

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.
  - ◇ Some operations are combined.

    E.g. the full table scan operator can also do a selection, and the projection does not appear explictly.

Viewing Oracle QEPs (1)

- First, create a table "PLAN_TABLE" in which Oracle will store information about the QEP.

  The table must exist under the account of each user who wants to view QEPs. It has prescribed columns, see slide 6-6-12 for details.
- 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 structure). If one logs out from SQL*Plus, the AUTOTRACE is forgotten, but the PLAN_TABLE still exists.

Viewing Oracle QEPs (2)

- The output you get from AUTOTRACE is not in graphical form as shown above, but in textual form:

```
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.

Details: 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 (can be retrieved with SQL). 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 ...
```

Details: 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: 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: Time when EXPLAIN PLAN was issued.

Details: 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 child nodes from left to right.
  - ◇ REMARKS: Normally null (can be set with UPDATE).
  - ◇ OPTIMIZER: Current mode of the optimizer.

Details: Plan Table (4)

- Columns of the PLAN_TABLE, continued:
  - ◇ The following columns describe operations.
  - ◇ OPERATION: E.g. "TABLE ACCESS", "MERGE JOIN".
  - ◇ OPTIONS: E.g. "FULL" for operation "TABLE ACCESS".
  - ◇ OBJECT_OWNER, OBJECT_NAME: Identifies the table or index used in the operation.

    Null for operations which get input only from their children.
  - ◇ OBJECT_INSTANCE: Position of table in the FROM-list.

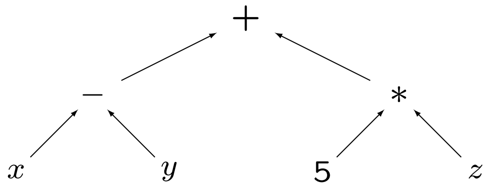
    E.g. useful if there are two tuple variables over one table.
  - ◇ OBJECT_TYPE: "UNIQUE" / "NON-UNIQUE" for indexes.

Processing of QEPs (1)

- A QEP is a tree with operations attached to nodes.
- 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 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 QEPs (2)

- Such a tree representation is known for arithmetic expressions. For example: $(x - y) + 5 * z$



- 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 QEPs (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 relation.
- This corresponds to the compilation of arithmetic expressions with registers as temporary storage for intermediate results:

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

Processing of QEPs (4)

- However, in this way a lot of memory is needed for the intermediate results.
- Sometimes intermediate results are so large that they have to be written to disk and then read again.
- But one can eliminate nearly all temporary storage since most operations work "tuple by tuple".

  Sorting is an exception (see next section).
- In the example, when the join has computed some tuple, one can immediately compute the projection result for that tuple (instead of first storing it).

Processing of QEPs (5)

- 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/Lazy Execution".

Pipelined Eval.: Interface (1)

Interface of QEP Nodes (Example):

- The interface is very similar to an SQL cursor.

  One opens the relation that is the result of this operation, fetches every tuple in a loop, and closes it. (Other names: "scan", "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 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 Eval.: 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 `i`-th attribute of 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 Eval.: Interface (3)

Less common operations:

- `save/restore`: Remember the current position in the stream of result tuples / switch back to it.

  Needed for merge join if duplicate values on both sides ($\times$ for subset).

- `back`: Switch back to previous result tuple.

  This operation is inverse to `next`. Needed for zig-zag nested loop join.

- `num_attrs`: 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 $\sigma_{Attr=Val}(Input)$.

  In a real system we must be able to pass any condition on tuples (with $\neg$, $\vee$, $\wedge$ and $<$, $>$, `like`, `is null`, ...).

- `QEP_Node_Selection(Input, AttrNo, Val)`:
  The constructor stores the three parameters in instance variables (attributes) of this object.

- `open()`:
```
Input->open(); // Simply pass to child node
```

- `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());
```


Overview

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.

Temporary Storage (2)

- 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 (i.e. the real work is done here).
  - ◇ Then later requests for the next result tuple will be answered from the intermediate storage.

Temporary Storage (3)

- 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"). But Oracle seems to use intermediate space only for sorting.

- In Oracle, the maximal size of temporary storage that a single sort operation can request in memory is set by the initialization parameter SORT_AREA_SIZE.
- If the space needed for sorting is larger, Oracle will use temporary segments on disk.

Temporary Storage (4)

- The current value of this parameter is shown with:

```
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.

Temporary Storage (5)

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

  By default, this parameter 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%'
```


Temporary Storage (6)

- Temporary segments can be allocated in any tablespace, but it is better to use a special "temporary tablespace".

  The storage parameters 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.

  See CREATE USER statement. It can be changed with ALTER USER.

- Information about temporary segments is available in V$SORT_SEGMENT and V$SORT_USAGE.

Performance Statistics (1)

- How many sorts in the current session were done in memory? How many on disk? And 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 (2)

- After `SET AUTOTRACE ON`, SQL*Plus prints not only the QEP for every query, but also performance statistics (including information about sorts).

  `SET AUTOTRACE ON STATISTICS` prints only the statistics.

- The role `PLUSTRACE` gives access to some dynamic performance views. It must be granted to all users who should be able to use this feature.

  It contains access to `sys.v_$sesstat`, `sys.v_$statname`, `sys.v_$session`. To declare this role, the DBA (user `SYS`) must execute the script `plustrce.sql`. It is located in `$ORACLE_HOME/sqlplus/admin`.

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.

Sort Algorithm (2)

- 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.
- Then you always merge two such sorted sequences ("runs") of length l to one sorted sequence of length $2 * l$.

Sort Algorithm (3)

- The 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. → Complexity $O(n * \log(n))$.

  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:

  $\boxed{5}$ 9 12 20 $\boxed{3}$ 7 16 18

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

  $\boxed{5}$ 9 12 20   3 $\boxed{7}$ 16 18 → 3

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

  5 $\boxed{9}$ 12 20   3 $\boxed{7}$ 16 18 → 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.

Sort Algorithm Optimizations

- There are many optimizations to Mergesort, 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.
  - ◇ If one has k buffer frames available during the merge phase, one merges $k-1$ runs instead of only 2 runs.

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.
- $R \underset{A_i=B_j}{\bowtie} S$ 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.

Nested Loop Join (2)

- Without the pipelined evaluation, the algorithm for $R \underset{A_i=B_j}{\bowtie} S$ looks as follows:

```
foreach tuple t = (d_1, ..., d_n) in R do
    foreach tuple u = (e_1, ..., e_m) in S do
        if d_i = e_j then
            output t ∘ u = (d_1, ..., d_n,
                            e_1, ..., e_m);
        fi;
    od;
od;
```

- Thus the name "nested loop".

Nested Loop Join (3)

- If both relations have approximately n tuples each, n^2 tuple combinations are checked.
- Thus, the nested loop join needs quadratic time, i.e. its complexity is $O(n^2)$.
- The merge join (see below) is asymptotically faster: It has complexity $O(n * \log(n))$.
- However, the nested loop join works for arbitrary join conditions, not only equality conditions.

  The merge join and other specialized join methods work only with equality conditions like $A = B$.

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.

    That value cannot have a join partner on the other side, since all following values there will be even bigger than the current one.
  - ◇ In this way it finds all matches (equal values).

Merge Join (2)

$R \underset{A_i = B_j}{\bowtie} S$:

```
open(R); open(S);
read t = (d_1,...,d_n) from R;
read u = (e_1,...,e_m) from S;
while not eof(R) and not eof(S) do
    if d_i < e_j then
        read t = (d_1,...,d_n) from R;
    else if d_i > e_j then
        read u = (e_1,...,e_m) from S;
    else /* d_i = e_j */
        output t ∘ u = (d_1,...,d_n, e_1,...,e_m);
        read u = (e_1,...,e_m) from S;
```

This program code assumes that A_i is a key in R. Therefore, after a match is found, the other side S is advanced for a possible further match.

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.

Merge Join (4)

- The time needed for the join itself is linear in the size of the two relations.

  We assume again that the join attribute on one side is a key of that relation, so there are no duplicate values on that side. If duplicate values were allowed on both sides, the extreme case (a single value repeated n times) would always lead to quadratic complexity: This would simply be a kartesian product.
- If we have to sort them, the total complexity is $O(n * \log(n))$.

  In comparison, the runtime (CPU time) 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. However, the merge join works only for equality conditions.

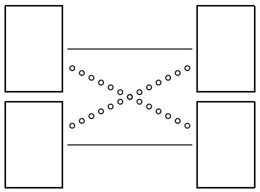
Index Join

- 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 (1)

- 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.



Hash Join (2)

- 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.

Hash Join (3)

Example: $\text{hash(row)} = \begin{cases} 1 & \text{if DEPTNO odd} \\ 2 & \text{otherwise} \end{cases}$

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