Fachhochschule Nordwestschweiz
Hochschule für Technik

DATENBANK-ARCHITEKTUR FÜR FORTGESCHRITTENE

Performanceoptimierung

Dani Schnider

Performance Tuning Tasks

Performance Planning

- Scalability
- System Architecture
- Application Design
- Data Model
- Testing
- etc.

Instance Tuning

- Memory Allocation
- I/O Balancing
- Database Configuration
- System Configuration
- etc.

SQL Tuning

- Indexing
- Partitioning
- Rewrite SQL Statements
- Gathering Statistics
- Hints
- etc.

See Oracle® Database Performance Tuning Guide, Chapter 1, Performance Tuning Overview

QUERY OPTIMIZER

Performanceoptimierung

Query Optimizer

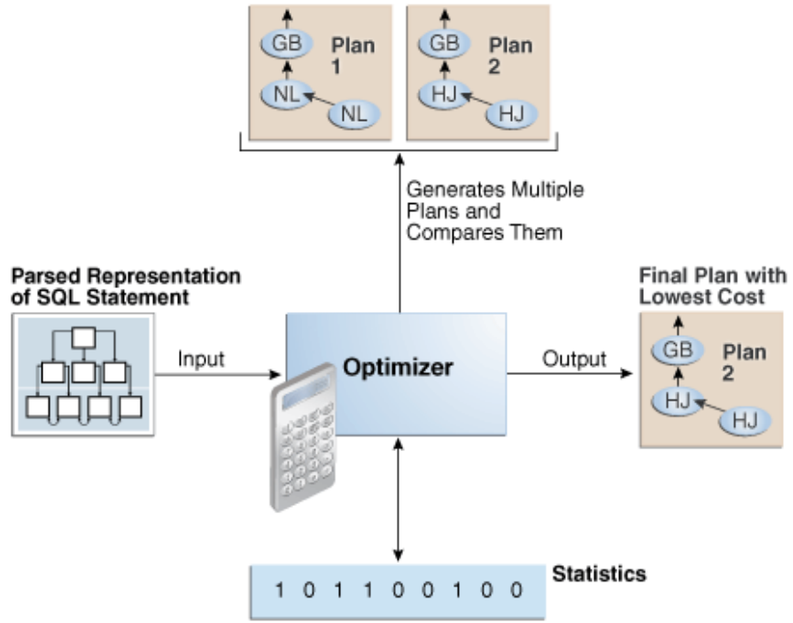
Ziel:

- Finde effizientesten Weg zur Ausführung einer SQL-Befehls

Vorgehensweise:

- Für jeden SQL-Befehl werden folgende Schritte durchgeführt:
  1. Generierung von mehreren möglichen Execution Plans
  2. Bewertung jedes Execution Plans mit Kosten*
  3. Ausführung des Execution Plans mit den geringsten Kosten
- Als Basis für die Kostenberechnung werden Optimizer-Statistiken verwendet

* Die Kosten sind ein geschätzter Wert für die I/O- und CPU-Ressourcen die zur Ausführung des Plans benötigt werden



Execution Plan

Execution Plan legt fest, wie der SQL-Befehl ausgeführt wird:

- Join-Reihenfolge der Tabellen
- Join-Methode pro Tabelle
  - Nested Loops Join
  - Merge Join
  - Hash Join
- Zugriffs-Methode pro Tabelle
  - Full Table Scan
  - Index Scan

```
-------------------------------------------------------------------
| Id  | Operation                                | Name            |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                 |
|*  1 |  HASH JOIN                               |                 |
|*  2 |   TABLE ACCESS FULL                      | PRODUCTS        |
|   3 |   NESTED LOOPS                           |                 |
|   4 |    NESTED LOOPS                          |                 |
|   5 |     NESTED LOOPS                         |                 |
|   6 |      TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS       |
|*  7 |       INDEX RANGE SCAN                   | CUST_LAST_NAME  |
|   8 |      TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS          |
|*  9 |       INDEX RANGE SCAN                   | ORD_CUST_ID     |
|* 10 |     INDEX RANGE SCAN                     | ORDI_ORDER_ID   |
|  11 |    TABLE ACCESS BY INDEX ROWID           | ORDER_ITEMS     |
-------------------------------------------------------------------
```

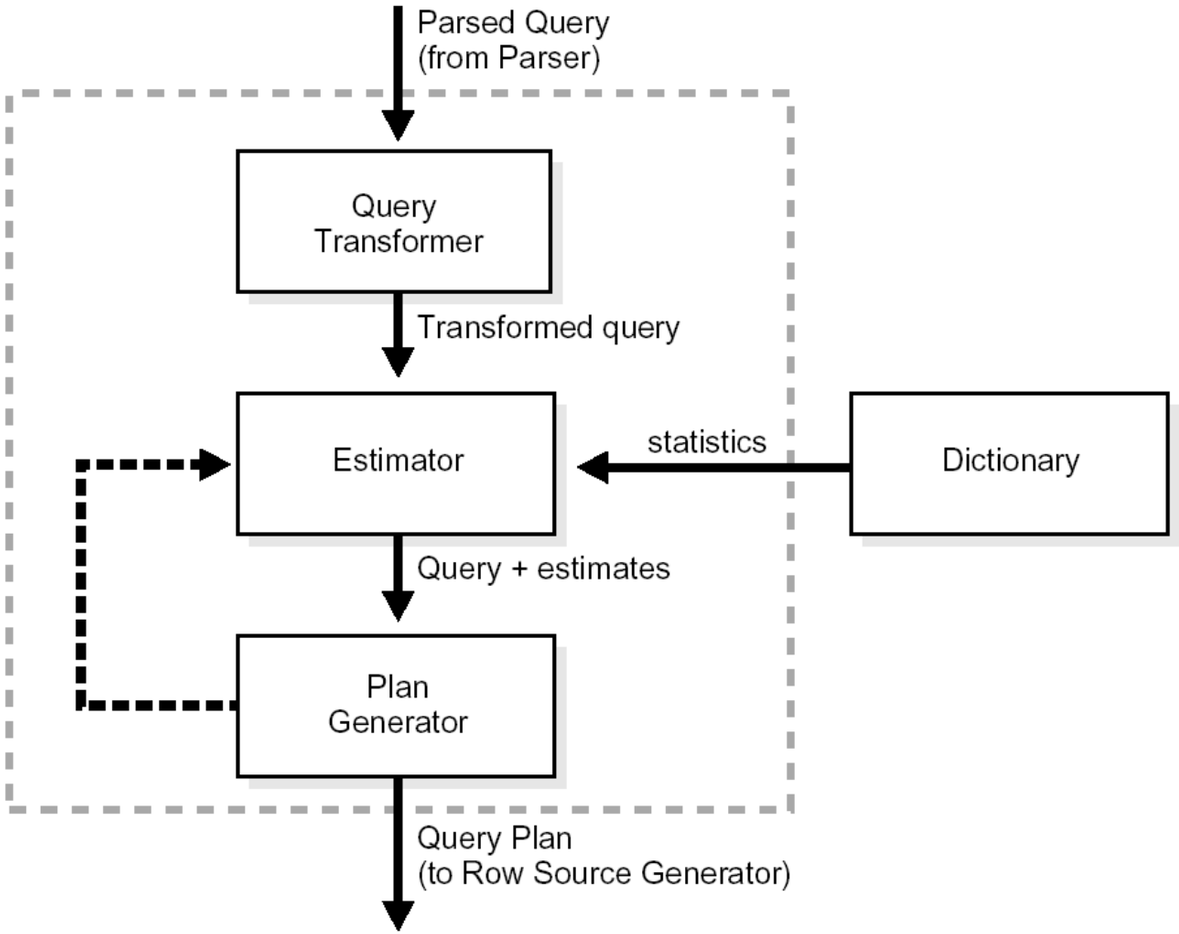
Kardinalität und Selektivität

$$Cardinality = Selectivity \cdot TotalNumberOfRows$$

Cardinality is derived from

- Table statistics (total number of rows)
- Column statistics (number of distinct values, histograms)
- WHERE condition

Architektur des Oracle Optimizers



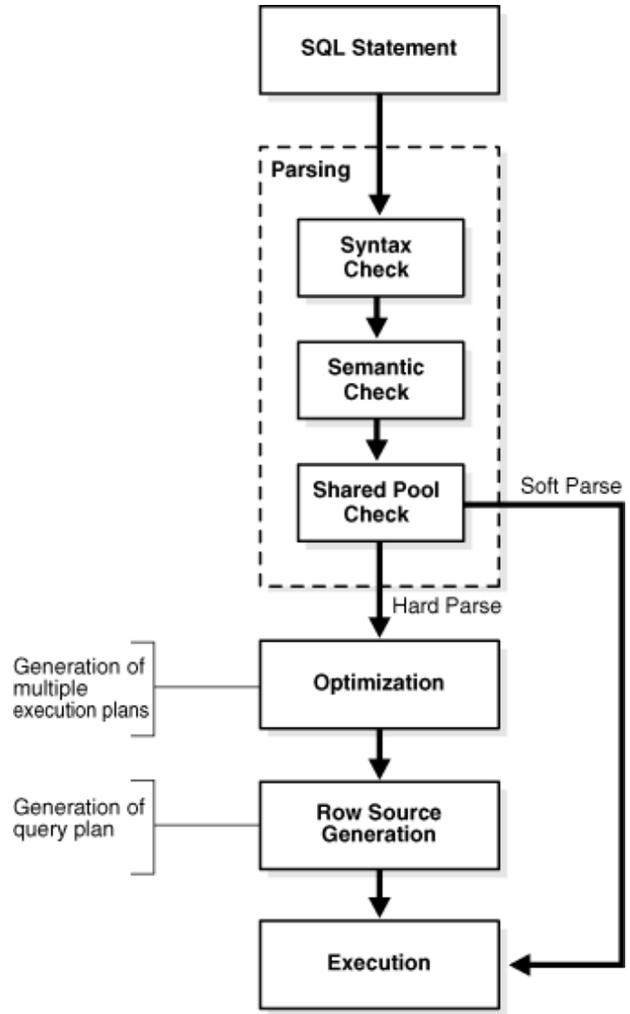
Parsing

Optimierung wird während Parse-Phase durchgeführt

- Parser überprüft SQL-Statement
- Optimizer berechnet günstigsten Execution Plan
- SQL und Plan werden in Shared SQL Area (Shared Pool) gespeichert

Bei erneuter Ausführung des SQL-Befehls wird Soft Parse ausgeführt

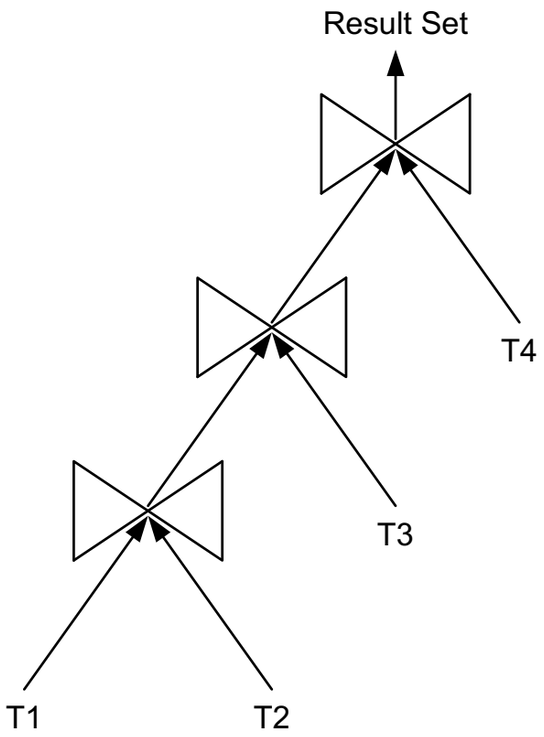
- SQL im Shared Pool vorhanden,
  => gleicher Exeuction Plan wird verwendet



JOIN-METHODEN

Performanceoptimierung

Join Trees – Deep Trees

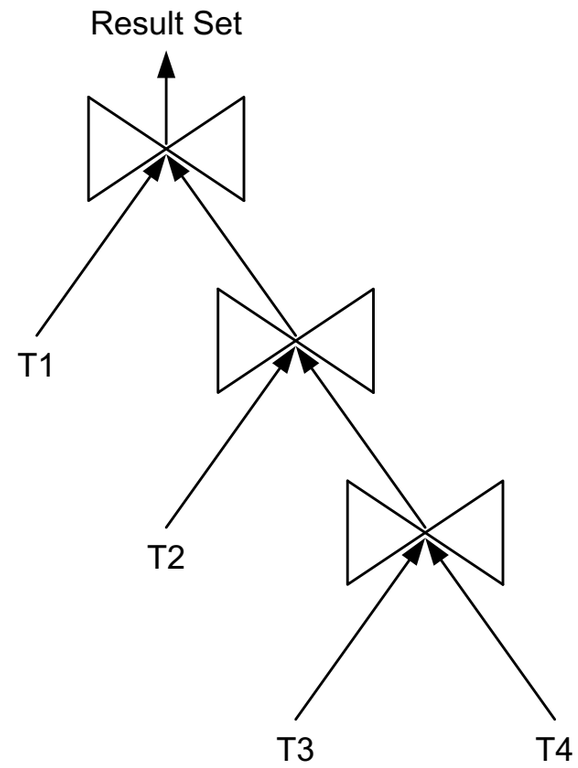


Left Deep Tree

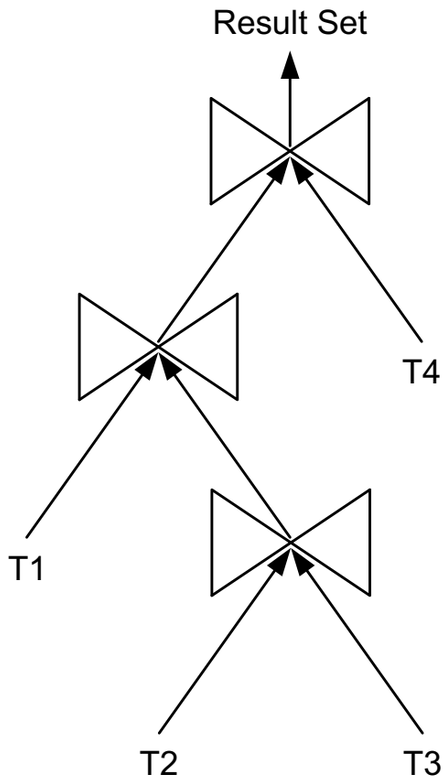
```
-----------------------------------
| Id | Operation           | Name |
-----------------------------------
|  1 | HASH JOIN           |      |
|  2 |  HASH JOIN          |      |
|  3 |   HASH JOIN         |      |
|  4 |    TABLE ACCESS FULL| T1   |
|  5 |    TABLE ACCESS FULL| T2   |
|  6 |   TABLE ACCESS FULL | T3   |
|  7 |  TABLE ACCESS FULL  | T4   |
-----------------------------------
```

```
-----------------------------------
| Id | Operation           | Name |
-----------------------------------
|  1 | HASH JOIN           |      |
|  2 |  TABLE ACCESS FULL  | T1   |
|  3 |  HASH JOIN          |      |
|  4 |   TABLE ACCESS FULL | T2   |
|  5 |   HASH JOIN         |      |
|  6 |    TABLE ACCESS FULL| T3   |
|  7 |    TABLE ACCESS FULL| T4   |
-----------------------------------
```

Right Deep Tree



Join Trees – Zig-Zag and Bushy Trees

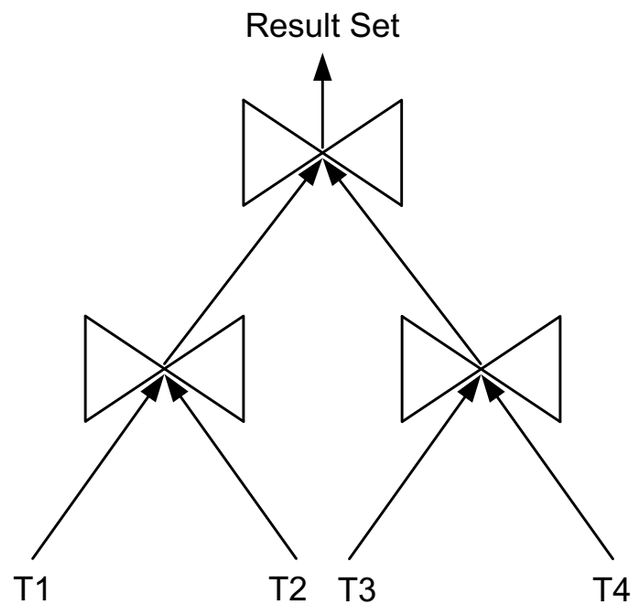


Zig-Zag Tree

```
-----------------------------------
| Id | Operation           | Name |
-----------------------------------
|  1 | HASH JOIN           |      |
|  2 |  HASH JOIN          |      |
|  3 |   TABLE ACCESS FULL | T1   |
|  4 |   HASH JOIN         |      |
|  5 |    TABLE ACCESS FULL| T2   |
|  6 |    TABLE ACCESS FULL| T3   |
|  7 |  TABLE ACCESS FULL  | T4   |
-----------------------------------
```

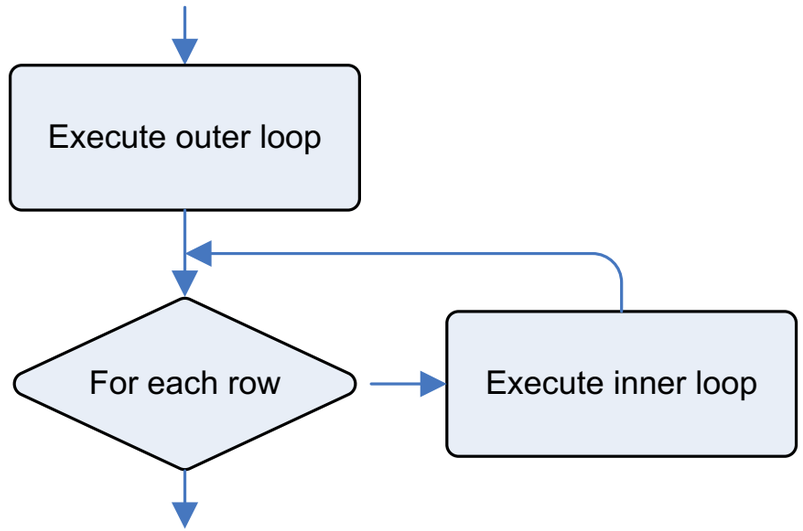
```
-----------------------------------
| Id | Operation           | Name |
-----------------------------------
|  1 | HASH JOIN           |      |
|  2 |  VIEW               |      |
|  3 |   HASH JOIN         |      |
|  4 |    TABLE ACCESS FULL| T1   |
|  5 |    TABLE ACCESS FULL| T2   |
|  6 |  VIEW               |      |
|  7 |   HASH JOIN         |      |
|  8 |    TABLE ACCESS FULL| T3   |
|  9 |    TABLE ACCESS FULL| T4   |
-----------------------------------
```

Bushy Tree



Nested Loops Join

- The left input (outer loop) is executed only once. The right input (inner loop) is potentially executed many times
- They are able to return the first row of the result set before completely processing all rows
- They can take advantage of indexes to apply both restrictions and join conditions
- They support all types of joins



Nested Loops Join – Beispiel

```
SELECT /*+ ordered use_nl(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id AND t1.n = 19
```

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		1000
1	NESTED LOOPS		1		1000
2	NESTED LOOPS		1	1000	1000
3	NESTED LOOPS		1	100	100
4	NESTED LOOPS		1	10	10
5	TABLE ACCESS BY INDEX ROWID	T1	1	1	1
* 6	INDEX RANGE SCAN	T1_N	1	1	1
7	TABLE ACCESS BY INDEX ROWID	T2	1	10	10
* 8	INDEX RANGE SCAN	T2_T1_ID	1	10	10
9	TABLE ACCESS BY INDEX ROWID	T3	10	10	100
* 10	INDEX RANGE SCAN	T3_T2_ID	10	10	100
* 11	INDEX RANGE SCAN	T4_T3_ID	100	10	1000
12	TABLE ACCESS BY INDEX ROWID	T4	1000	10	1000

Block Prefetching

Nested Loops Join without Block Prefetching

```
-----------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  NESTED LOOPS                 |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| T1       |
|*  3 |    INDEX UNIQUE SCAN          | T1_N     |
|   4 |   TABLE ACCESS BY INDEX ROWID| T2       |
|*  5 |    INDEX RANGE SCAN           | T2_T1_ID |
-----------------------------------------------

   3 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

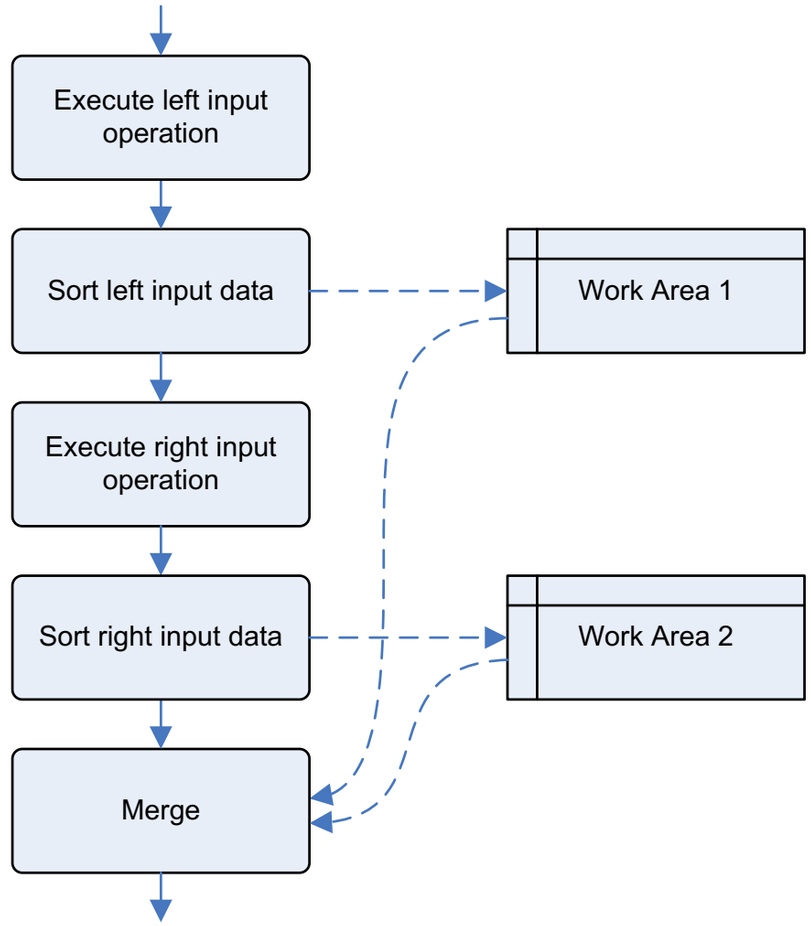
Nested Loops Join with Block Prefetching

```
-----------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  NESTED LOOPS                 |          |
|   2 |   NESTED LOOPS                |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| T1       |
|*  4 |     INDEX RANGE SCAN          | T1_N     |
|*  5 |    INDEX RANGE SCAN           | T2_T1_ID |
|   6 |   TABLE ACCESS BY INDEX ROWID | T2       |
-----------------------------------------------

   4 - access("T1"."N"=19)
   5 - access("T1"."ID"="T2"."T1_ID")
```

Merge Join

- Each input is executed only once
- Both inputs must be sorted according to the columns of the join condition before returning the first row of the result set
- All types of joins are supported
- Sorting can be a very "expensive" operation → useful if data sources are already sorted



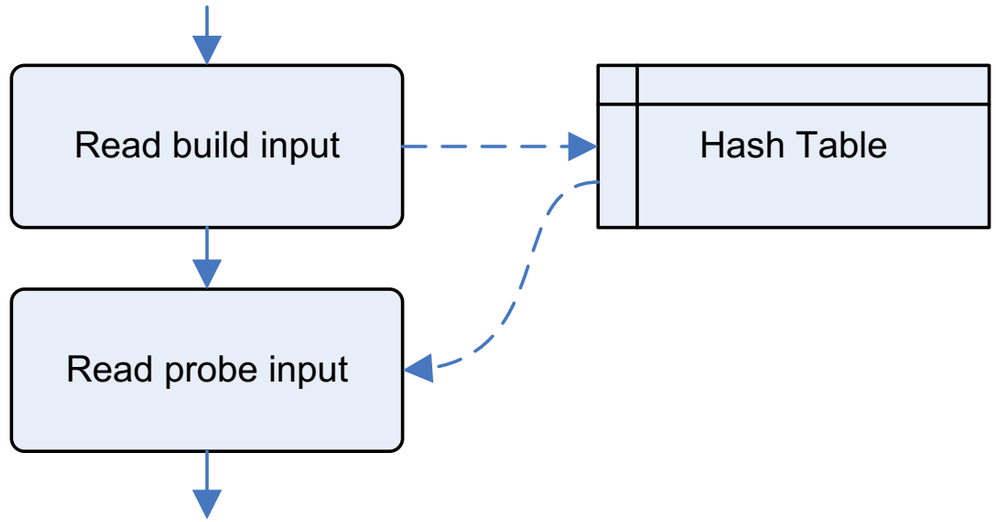
Merge Join – Beispiel

```
SELECT /*+ leading(t1 t2 t3) use_merge(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

-------------------------------------------------------------------------
| Id  | Operation                    | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |      |      1 |        |   1000 |
|   1 |  MERGE JOIN                  |      |      1 |   1000 |   1000 |
|   2 |   SORT JOIN                  |      |      1 |    100 |    100 |
|   3 |    MERGE JOIN                |      |      1 |    100 |    100 |
|   4 |     SORT JOIN                |      |      1 |     10 |     10 |
|   5 |      MERGE JOIN              |      |      1 |     10 |     10 |
|   6 |       SORT JOIN              |      |      1 |      1 |      1 |
|*  7 |        TABLE ACCESS FULL     | T1   |      1 |      1 |      1 |
|*  8 |       SORT JOIN              |      |      1 |    100 |     10 |
|   9 |        TABLE ACCESS FULL     | T2   |      1 |    100 |    100 |
|* 10 |     SORT JOIN                |      |     10 |   1000 |    100 |
|  11 |      TABLE ACCESS FULL       | T3   |      1 |   1000 |   1000 |
|* 12 |   SORT JOIN                  |      |    100 |  10000 |   1000 |
|  13 |    TABLE ACCESS FULL         | T4   |      1 |  10000 |  10000 |
-------------------------------------------------------------------------
```

Hash Join

- Each input is executed only once
- The hash table is built on the left input only. Consequently, it is usually built on the smallest input
- Before returning the first row, only the left input must be fully processed
- Cross joins, theta joins, and partitioned outer joins are not supported



Hash Join – Beispiel

Left Deep Tree

```
SELECT /*+ leading(t1 t2 t3) use_hash(t2 t3 t4) */ t1.*, t2.*, t3.*, t4.*
FROM t1, t2, t3, t4
WHERE t1.id = t2.t1_id
AND t2.id = t3.t2_id
AND t3.id = t4.t3_id
AND t1.n = 19

-------------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |   1000 |
|*  1 |  HASH JOIN            |      |      1 |   1000 |   1000 |
|*  2 |   HASH JOIN           |      |      1 |    100 |    100 |
|*  3 |    HASH JOIN          |      |      1 |     10 |     10 |
|*  4 |     TABLE ACCESS FULL| T1   |      1 |      1 |      1 |
|   5 |     TABLE ACCESS FULL| T2   |      1 |    100 |    100 |
|   6 |    TABLE ACCESS FULL | T3   |      1 |   1000 |   1000 |
|   7 |   TABLE ACCESS FULL  | T4   |      1 |  10000 |  10000 |
-------------------------------------------------------------------
```

Hash Join – Beispiel

Right Deep Tree

```
-------------------------------------------------------------------
| Id  | Operation             | Name | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |      1 |        |   1000 |
|*  1 |  HASH JOIN            |      |      1 |   1000 |   1000 |
|*  2 |   TABLE ACCESS FULL   | T1   |      1 |      1 |      1 |
|*  3 |   HASH JOIN           |      |      1 |  10000 |  10000 |
|   4 |    TABLE ACCESS FULL  | T2   |      1 |    100 |    100 |
|*  5 |    HASH JOIN          |      |      1 |  10000 |  10000 |
|   6 |     TABLE ACCESS FULL | T3   |      1 |   1000 |   1000 |
|   7 |     TABLE ACCESS FULL | T4   |      1 |  10000 |  10000 |
-------------------------------------------------------------------
```

Vergleich der Join-Methoden