RELATIONAL DBMS EXTENSIONS FOR DW

- ❑ SQL extensions

- ❑ Index and storage structures ⇦
  - ▪ Indexes: Inverted, Bitmap, Join, FC Join
  - ▪ Storage: Horizontal/Vertical Partitioning

- ❑ Star query physical plans ⇦

Hypothesis (valid for DW)

- Nonvolatile data!

INVERTED INDEX (for SELECTIVE ATTRIBUTES)

Table

RID	StudCode	City	BirthYear
1	100	MI	2002
2	101	PI	2000
3	102	PI	2001
4	104	FI	2000
5	106	MI	2000
6	107	PI	2002

```
CREATE INDEX idx_iv
ON Table (BirthYear);
```

Indexes

BirthYear	RID
2000	2
2000	4
2000	5
2001	3
2002	1
2002	6

Index IA on **BirthYear**

BirthYear	n	RID Lists		
2000	3	2	4	5
2001	1	3		
2002	2	1	6	

Inverted Index

INVERTED INDEX (for SELECTIVE ATTRIBUTES)

It is the solution in all relational DBMS

•Useful when the number of distinct values of an indexed attribute is high **(selective attribute)**.

•Standard **solution** in DWMS for primary key.

BITMAP INDEXES (for NOT-SELECTIVE ATTR.)

RID	StudCode	City	BirthYear
1	100	MI	2002
2	101	PI	2000
3	102	PI	2001
4	104	FI	2000
5	106	MI	2000
6	107	PI	2002

Table

CREATE BITMAP INDEX idx_bm
ON Table (BirthYear);

Inverted indexes

City	n	RID Lists		
FI	1	4		
MI	2	1	5	
PI	3	2	3	6

BirthYear	n	RID Lists		
2000	3	2	4	5
2001	1	3		
2002	2	1	6	

Bitmap indexes

City	Bitmaps					
FI	0	0	0	1	0	0
MI	1	0	0	0	1	0
PI	0	1	1	0	0	1

Index on **City**

BirthYear	Bitmaps					
2000	0	1	0	1	1	0
2001	0	0	1	0	0	0
2002	1	0	0	0	0	1

Index on **BirthYear**

BITMAP INDEXES (for NOT-SELECTIVE ATTRIBUTES)

Each value in the indexed attribute is associated to a bit vector (bitmaps)

The length of the bitmap is the number of records in the base table.

The i-th bit is set if the i-th record of the base table has the value for the indexed attribute.

A BM index is used in all DBMS for DW when the number of distinct values of an indexed attribute is small (i.e. the attribute is **not selective**). An inverted list index would be useless.

ADVANTAGES OF BITMAP INDEXES

- Efficient bit operations to answer some queries (and's, or's, bitcount)
- Bit vectors can be compressed
  - Oracle: Bitmap indexes are compressed and are suggested if Nkey < Nrec/2

City	Bitmap					
FI	0	0	0	1	0	0
MI	1	0	0	0	1	0
PI	0	1	1	0	0	1

BirthYear	Bitmap					
2000	0	1	0	1	1	0
2001	0	0	1	0	0	0
2002	1	0	0	0	0	1

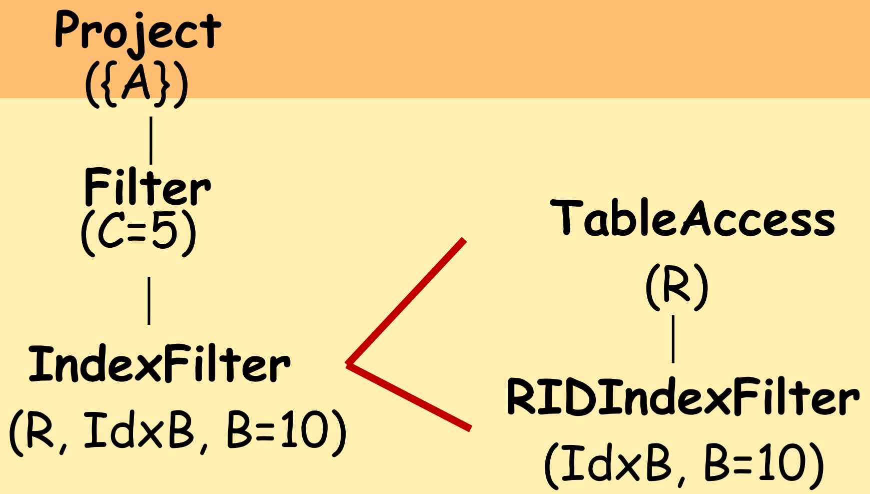
Q: How many students from Pisa were born in the 2002?

EXAMPLES OF ACCESS PLANS

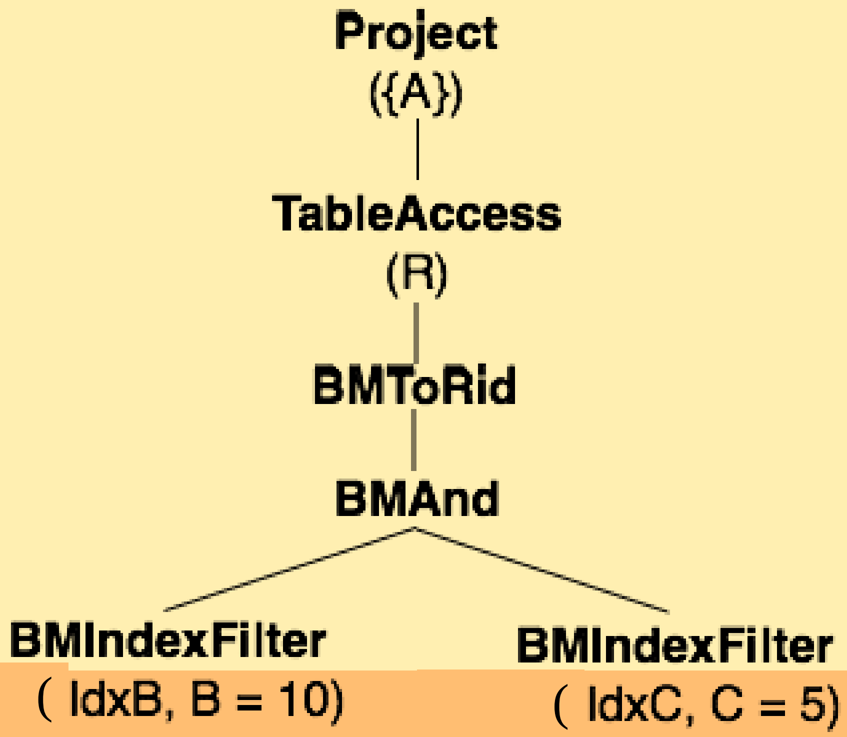
R(A, B, C)

```
SELECT   A
FROM     R
WHERE    B = 10 AND C = 5;
```

Inverted Indexes on B and C.



Bitmap indexes on B and C.

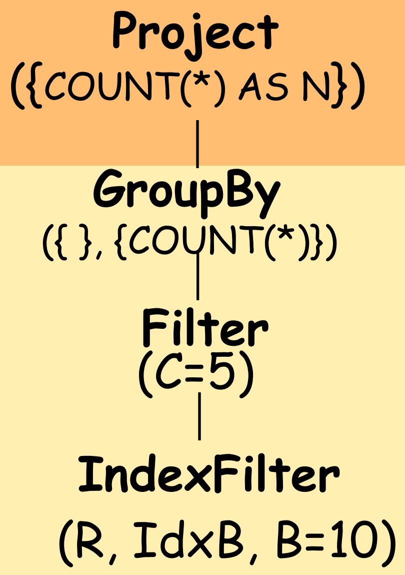


EXAMPLES OF ACCESS PLANS

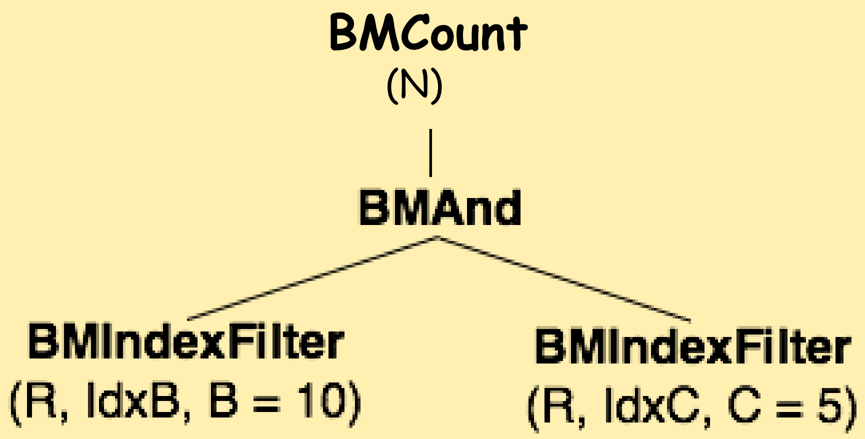
R(A, B, C)

```
SELECT  COUNT(*) AS N
FROM    R
WHERE   B = 10 AND C = 5;
```

Inverted Indexes on B and C.



Bitmap indexes on B and C.



A STAR SCHEMA EXAMPLE

D1

RID	pk1	A1	...
1	v1	a1	...
2	v2	a2	...
3	v3	a2	...

F

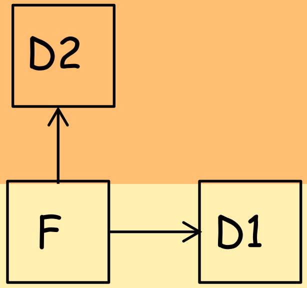
RID	fk1	fk2	M
1	v1	z1	150
2	v2	z1	300
3	v3	z2	400
4	v2	z2	200
5	v1	z2	90

```
SELECT …
FROM F, D1, D2
WHERE F.fk1 = D1.pk1 AND
        F.fk2 = D2.pk2 AND …
```

D2

RID	pk2	B1	...
1	z1	b1	...
2	z2	b2	...
3	z3	b2	...

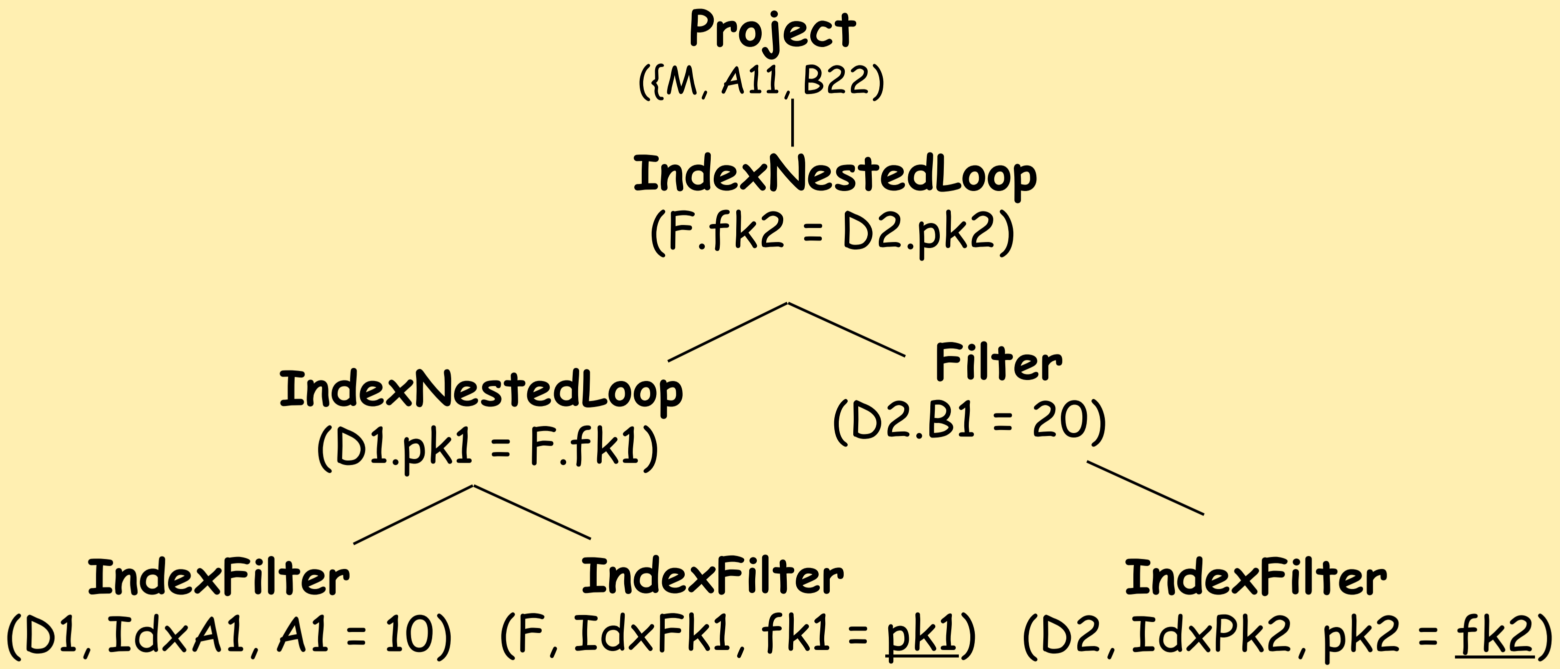
EXAMPLE WITH INVERTED INDEXES



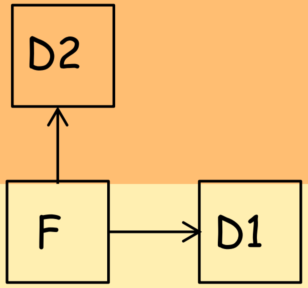
```
SELECT   F.M, D1.A11, D2.B22
FROM     F, D1, D2
WHERE    F.fk1 = D1.pk1 AND F.fk2 = D2.pk2
         AND D1.A1 = 10 AND D2.B2 = 20;
```

Hyp 0:

Inverted Indexes on F.fk1 and F.fk2,
Inverted indexes on primary keys of D1 and D2
Inverted indexes on D1.A1 and D2.B2



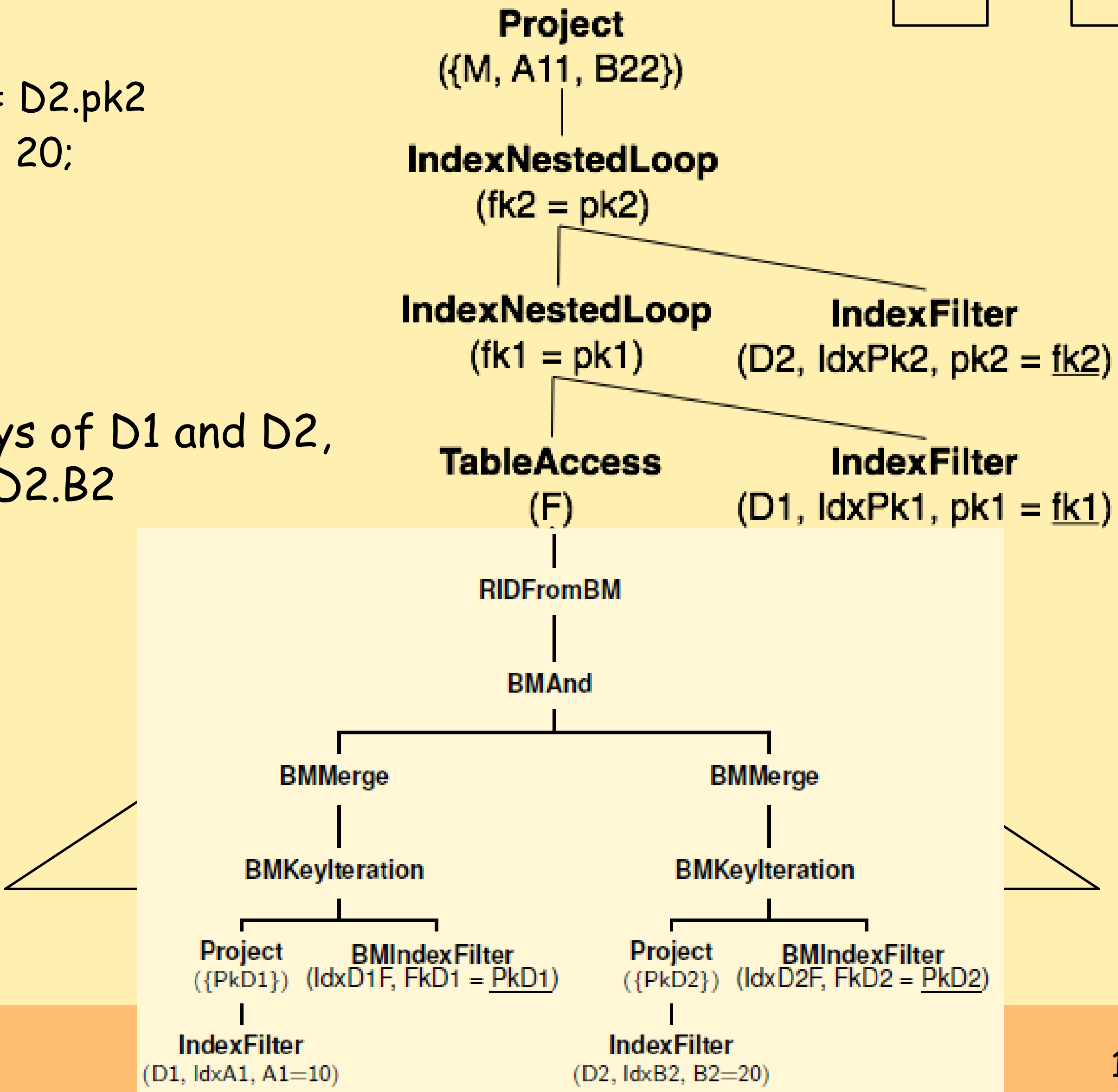
EXAMPLE WITH BM INDEXES (see Example 6.4 of the book)



```
SELECT   F.M, D1.A11, D2.B22
FROM     F, D1, D2
WHERE    F.fk1 = D1.pk1 AND F.fk2 = D2.pk2
         AND D1.A1 = 10 AND D2.B2 = 20;
```

Normal index

Hyp 1:
BM Indexes on F.fk1 and F.fk2,
Inverted indexes on primary keys of D1 and D2,
Inverted indexes on D1.A1 and D2.B2



(STAR) JOIN INDEX

F

RID	fk1	fk2	M
1	v1	z1	150
2	v2	z1	300
3	v3	z2	400
4	v2	z2	200
5	v1	z2	90

D1

RID	pk1	A1	...
1	v1	a1	...
2	v2	a2	...
3	v3	a2	...

JoinIndex-F-D1

D1	F
1	1
1	5
2	2
2	4
3	3

```
SELECT …
FROM F, D1, D2
WHERE F.fk1 = D1.pk1 AND …
```

A join index relates a record **d** in a dimension **D1** of a star schema to a record **f** in the fact table **F**, such that (F.fk1 = D1.pk1).

NOT used in Operational DB due to high cost of updates

BITMAP (STAR) JOIN INDEXES

F

RID	fk1	fk2	M
1	v1	z1	150
2	v2	z1	300
3	v3	z2	400
4	v2	z2	200
5	v1	z2	90

D1

RID	pk1	A1	...
1	v1	a1	...
2	v2	a2	...
3	v3	a2	...

JoinIndex-F-D1

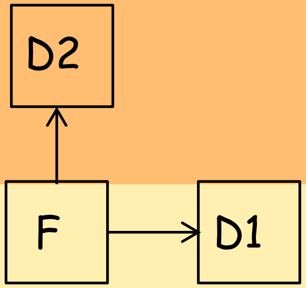
D1	F
1	1
1	5
2	2
2	4
3	3

```
SELECT ...
FROM F, D1, D2
WHERE F.fk1 = D1.pk1 AND ...
```

BMJoinIndex-F-D1

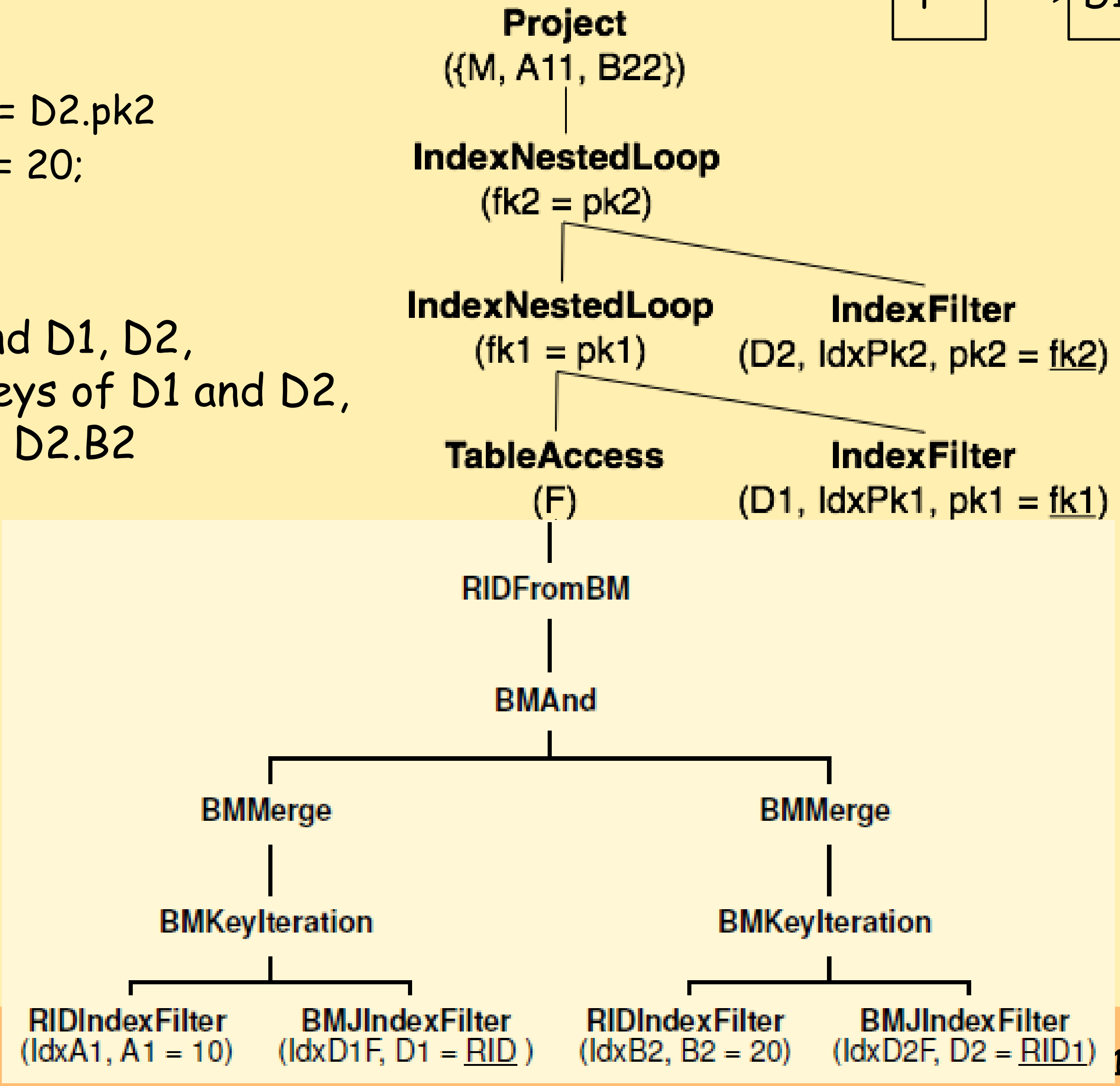
D1	Bitmap for F
1	10001...
2	01010...
3	00100...

EXAMPLE WITH BM JOIN (see Example 6.4 of the book)



```
SELECT   F.M, D1.A11, D2.B22
FROM     F, D1, D2
WHERE    F.fk1 = D1.pk1 AND F.fk2 = D2.pk2
         AND D1.A1 = 10 AND D2.B2 = 20;
```

Hyp 2:
BM Join Indexes between F and D1, D2,
Inverted indexes on primary keys of D1 and D2,
Inverted indexes on D1.A1 and D2.B2



FOREIGN COLUMN JOIN INDEX

F

RID	fk1	fk2	M
1	v1	z1	150
2	v2	z1	300
3	v3	z2	400
4	v2	z2	200
5	v1	z2	90

D1

RID	pk1	A1	...
1	v1	a1	...
2	v2	a2	...
3	v3	a2	...

FCJI-F-D1-A1

D1.A1	RID-F
a1	1
a1	5
a2	2
a2	4
a2	3

SELECT ...
FROM F, D1, D2
WHERE F.fk1 = D1.pk1 AND
D1.A1=a1 AND ...

FOREIGN COLUMN JOIN INDEX

F

RID	fk1	fk2	M
1	v1	z1	150
2	v2	z1	300
3	v3	z2	400
4	v2	z2	200
5	v1	z2	90

D1

RID	pk1	A1	...
1	v1	a1	...
2	v2	a2	...
3	v3	a2	...

BMFCJoinIndex-F-D1-A1

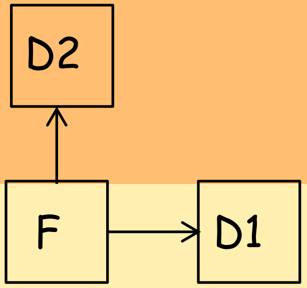
D1.A1	Bitmap per F
a1	10001...
a2	01110...

Bitmapped FC join index: The index elements **<p,q>** are represented as **<p, bitmap of the joined records>**

```
SELECT ...
FROM F, D1, D2
WHERE F.fk1 = D1.pk1 AND
      D1.A1=a1 AND ...
```

```
CREATE BITMAP INDEX idx_bmfcj
ON    F(D1.A1)
FROM  F, D1
WHERE F.fk1 = D1.pk1;
```

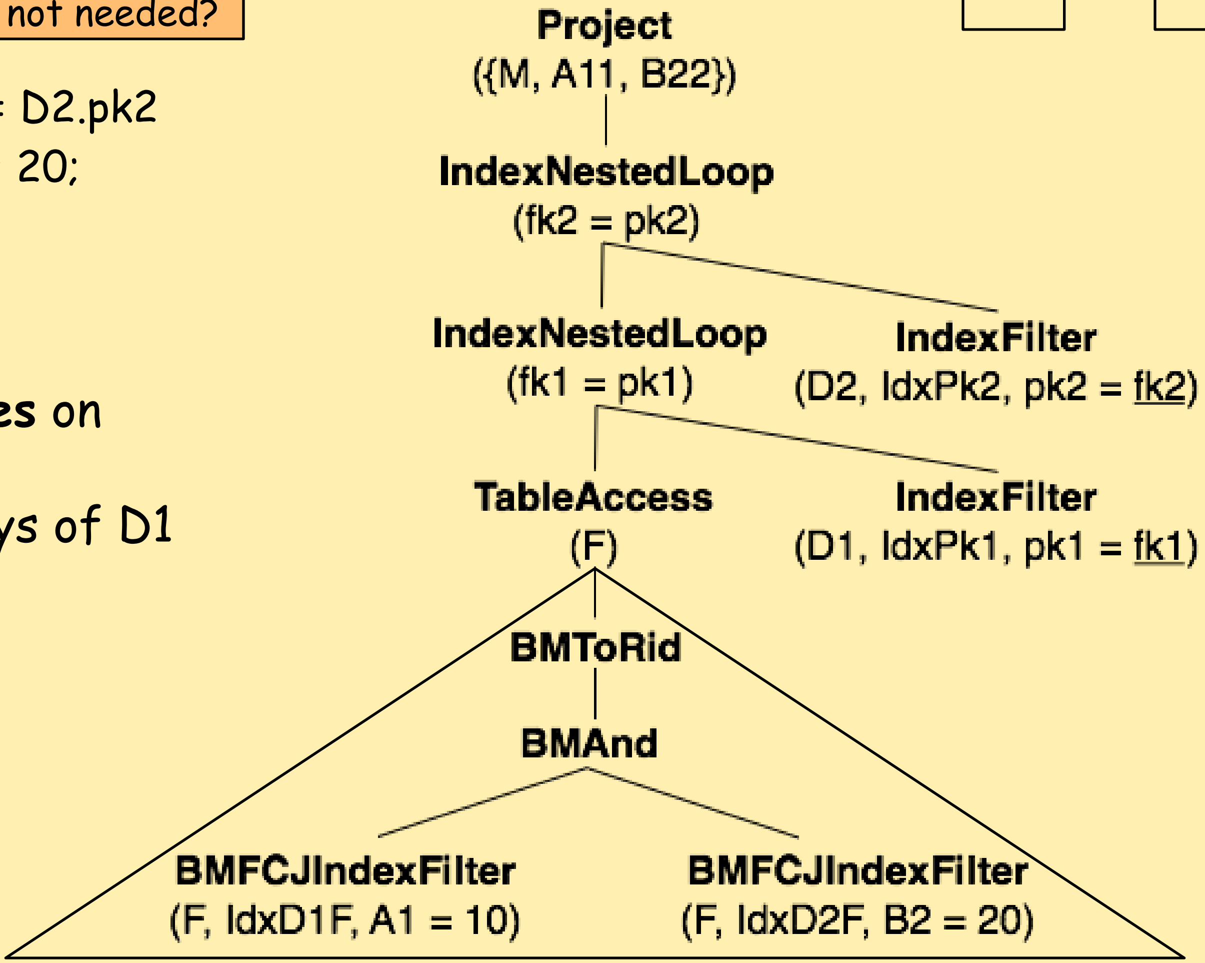
EXAMPLE WITH BMFCJ INDEX (see Example 6.4 of the book)



```
SELECT   F.M, D1.A11, D2.B22
FROM     F, D1, D2
WHERE    F.fk1 = D1.pk1 AND F.fk2 = D2.pk2
         AND D1.A1 = 10 AND D2.B2 = 20;
```

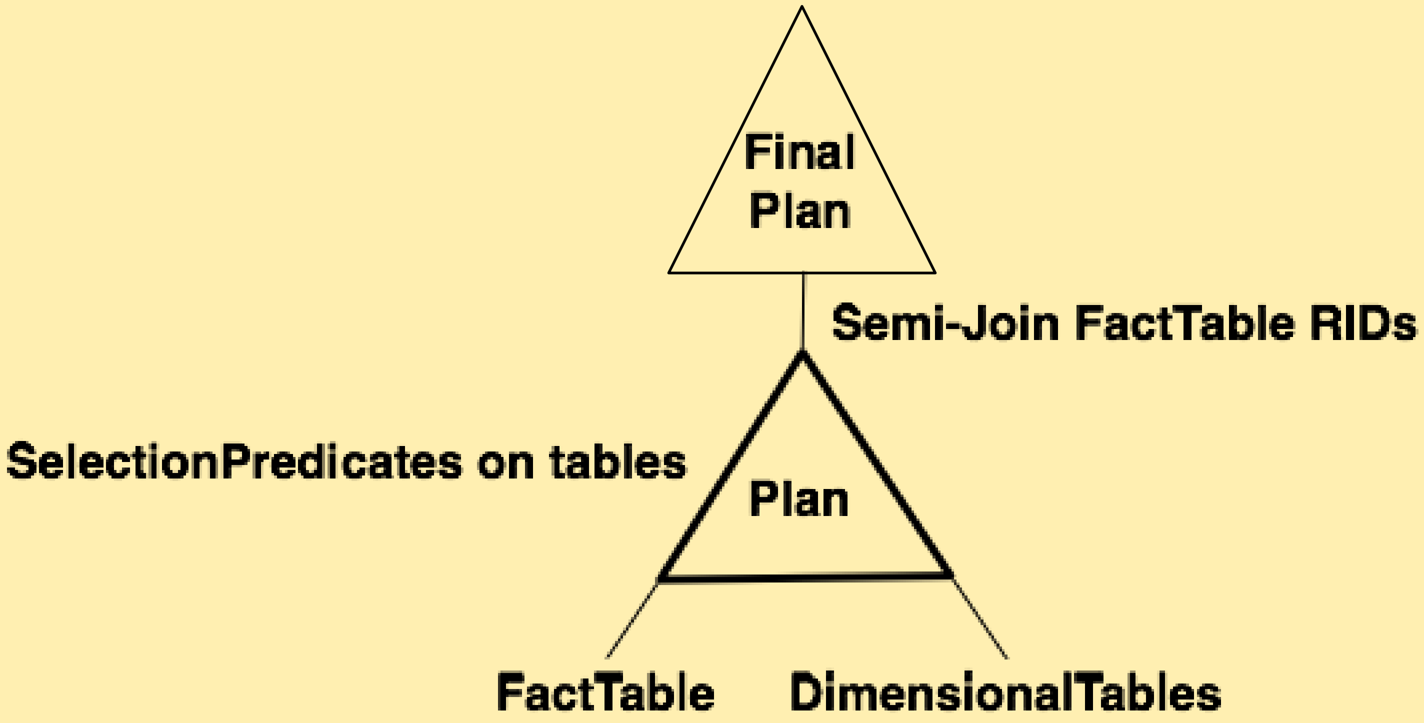
If they are not needed?

Hyp 1:
BM Foreign Column Join Indexes on D1.A1 and D2.B2,
Inverted indexes on primary keys of D1 and D2.

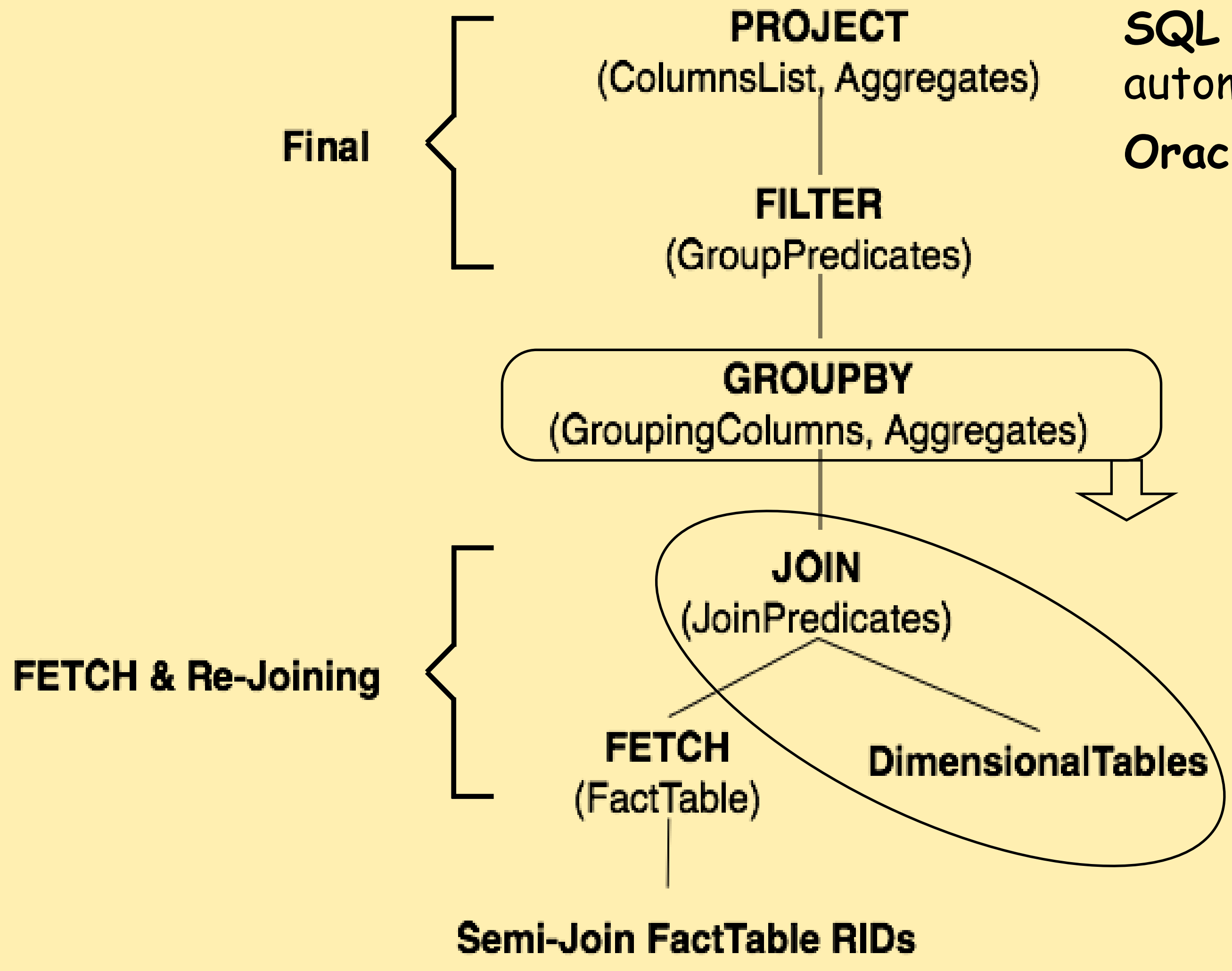


SUMMARY: STAR JOIN OPTIMIZATION

```
SELECT     ColumnsList, Aggregates
FROM       FactTable, DimensionTables
WHERE      JoinCond AND SelectionPredicates
GROUP BY   GroupingColumn
HAVING     GroupPredicates;
```



SUMMARY: STAR JOIN OPTIMIZATION



SQL Server: star-join optimization automatically detected

Oracle: requires setup configuration

STORAGE STRUCTURES: HEAP FILE

A table is stored in a Heap File
a file for each table, with tuples stored in the insertion order

```
SELECT *
FROM   Sales
WHERE  Date = '8 Jul 2016'
```

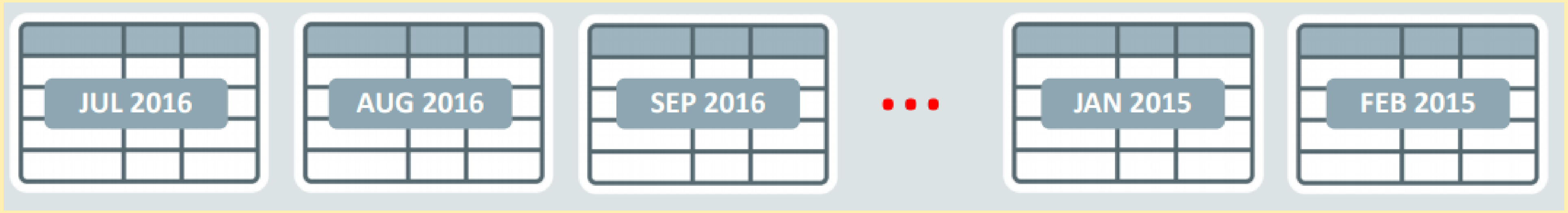
$$\begin{tikzcd}[row sep=large]
\textbf{Filter}\ (\text{Date = '8 Jul 2016'}) \arrow[d, dash] \\
\textbf{TableScan}\ (\text{Sales})
\end{tikzcd}$$

- Full table scan required! What about storing facts of July 2016 in a separate file?

STORAGE STRUCTURES: HORIZONTAL (FACT) TABLE PARTITIONING

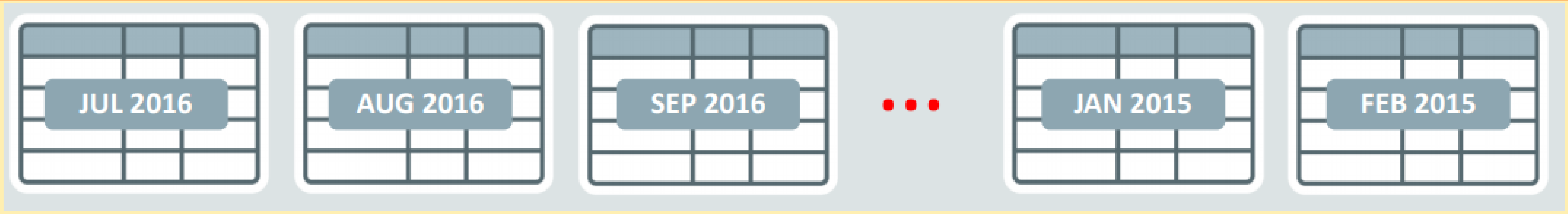
- **Partitioning** consists of storing table rows on the basis of a multi-way condition

- **Range partitioning**: condition based on membership to a range of values



```
CREATE TABLE Sales
(…,  sales_date    DATE, …)
PARTITION BY RANGE(sales_date)
(
     PARTITION jan2015 VALUES LESS THAN(TO_DATE('01/02/2015')),
     PARTITION feb2015 VALUES LESS THAN(TO_DATE('01/03/2015')),
     …
     PARTITION nov2016 VALUES LESS THAN(TO_DATE('01/12/2016')),
     PARTITION dec2016 VALUES LESS THAN(TO_DATE('01/0\/2017')),
);
```

TABLE PARTITIONING – PARTITION PRUNING



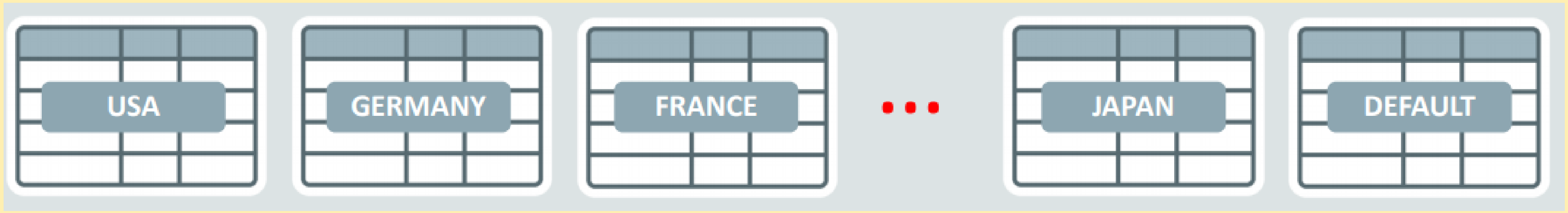
```
SELECT  *
FROM    Sales
WHERE   Date = '8 Jul 2016'
```

Filter
(Date = '8 Jul 2016')
|
TableScan
(Sales)

Filter
(Date = '8 Jul 2016')
|
PartitionedTableScan
(Sales, jul2016)

STORAGE STRUCTURES: HORIZONTAL (FACT) TABLE PARTITIONING

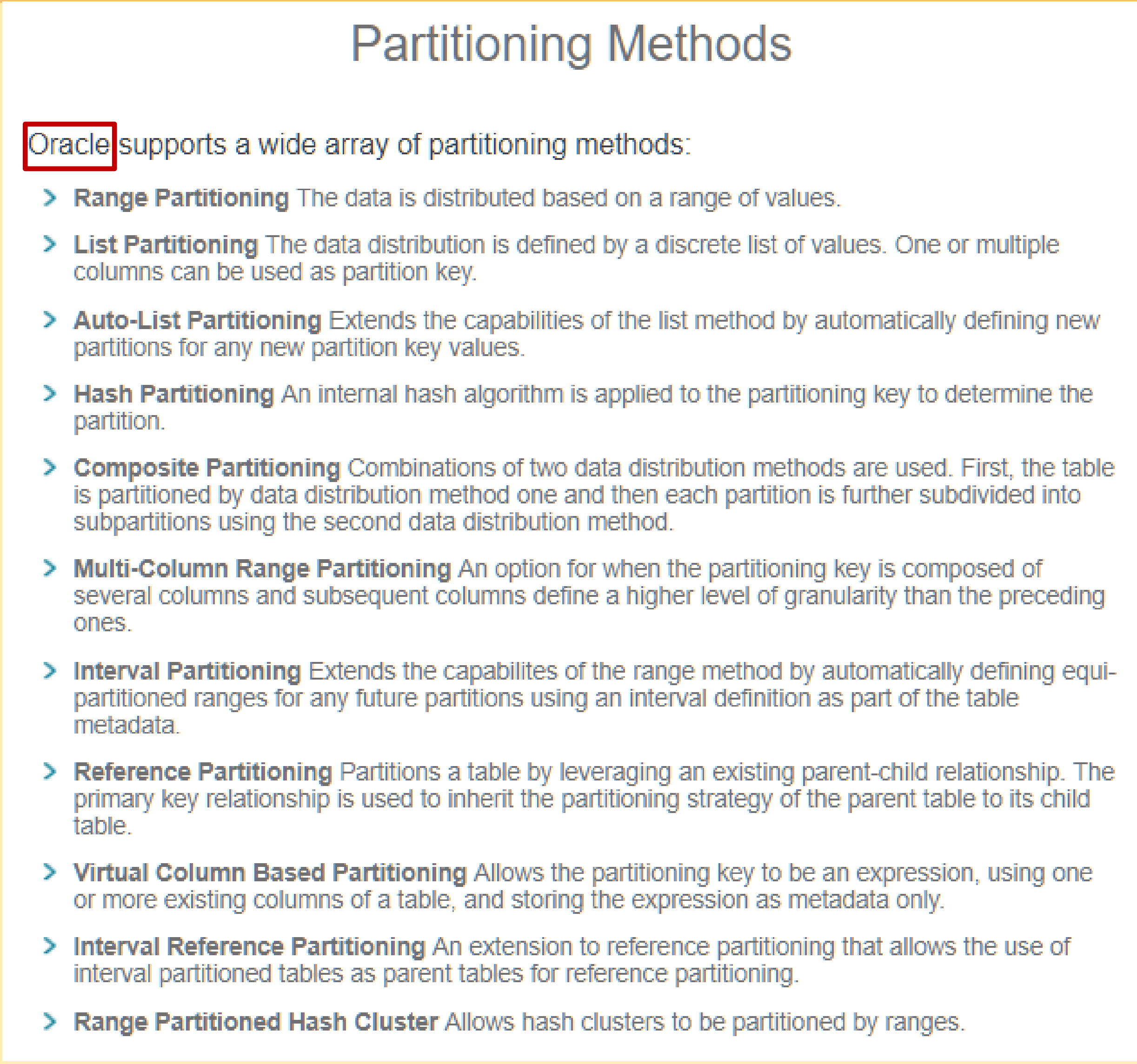
- **Partitioning** consists of storing table rows on the basis of a multi-way condition
- **List partitioning**: condition based on membership to a list of values



```
CREATE TABLE Sales
(…,  sales_country    VARCHAR(50), …)
PARTITION BY LIST(sales_country)
(
     PARTITION usa VALUES('USA')
     PARTITION germany VALUES('Germany'),
     …
     PARTITION japan VALUES('Japan')),
     PARTITION other VALUES(DEFAULT)),
);
```

TABLE PARTITIONING

- Increases **performance** by only working on the data that is relevant (partition pruning, partition-wise join and grouping)
- Increases **scalability** by parallelizing query execution that involves partitions in isolation.
- Improves **availability/load** through individual partition manageability.
- Is **easy** as to implement as it requires no changes to applications and queries.
- Applies to indexes as well (**index partitioning**)



Partitioning Methods

Oracle supports a wide array of partitioning methods:

- **Range Partitioning** The data is distributed based on a range of values.
- **List Partitioning** The data distribution is defined by a discrete list of values. One or multiple columns can be used as partition key.
- **Auto-List Partitioning** Extends the capabilities of the list method by automatically defining new partitions for any new partition key values.
- **Hash Partitioning** An internal hash algorithm is applied to the partitioning key to determine the partition.
- **Composite Partitioning** Combinations of two data distribution methods are used. First, the table is partitioned by data distribution method one and then each partition is further subdivided into subpartitions using the second data distribution method.
- **Multi-Column Range Partitioning** An option for when the partitioning key is composed of several columns and subsequent columns define a higher level of granularity than the preceding ones.
- **Interval Partitioning** Extends the capabilites of the range method by automatically defining equi-partitioned ranges for any future partitions using an interval definition as part of the table metadata.
- **Reference Partitioning** Partitions a table by leveraging an existing parent-child relationship. The primary key relationship is used to inherit the partitioning strategy of the parent table to its child table.
- **Virtual Column Based Partitioning** Allows the partitioning key to be an expression, using one or more existing columns of a table, and storing the expression as metadata only.
- **Interval Reference Partitioning** An extension to reference partitioning that allows the use of interval partitioned tables as parent tables for reference partitioning.
- **Range Partitioned Hash Cluster** Allows hash clusters to be partitioned by ranges.