Indexation: B-trees, bitmaps...

Chapter content:

- Indexes
- Clustering

Table of content

Indexation: B-trees, bitmaps...

Indexes in DBMS

Aim of an index: provide fast access to the data needed by the query.

Ex:

```
SELECT *
FROM Employee
WHERE EmployeeKey = 1234;
```

- With index on EmployeeKey: 1 disk access.
- Without: scan whole Employee table

Drawback: updates on indexed attribute require index update.

$\Longrightarrow$ too many indexes degrade performance.

Index taxonomy 分类

- *single-column* vs multicolumn
- *clustered* vs non-clustered: in a clustered index, records are physically ordered according to the index key.
  At most 1 clustered index, but several non-clustered indexes are possible.
- *unique* vs non-unique: in a unique index, only one record per key value whereas non-unique allow duplicates
- *dense* vs sparse: a dense index has one entry per data record.

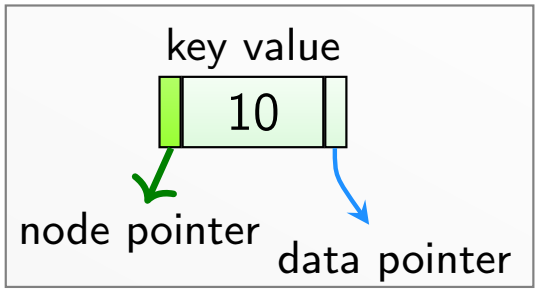
Indexes and partitions

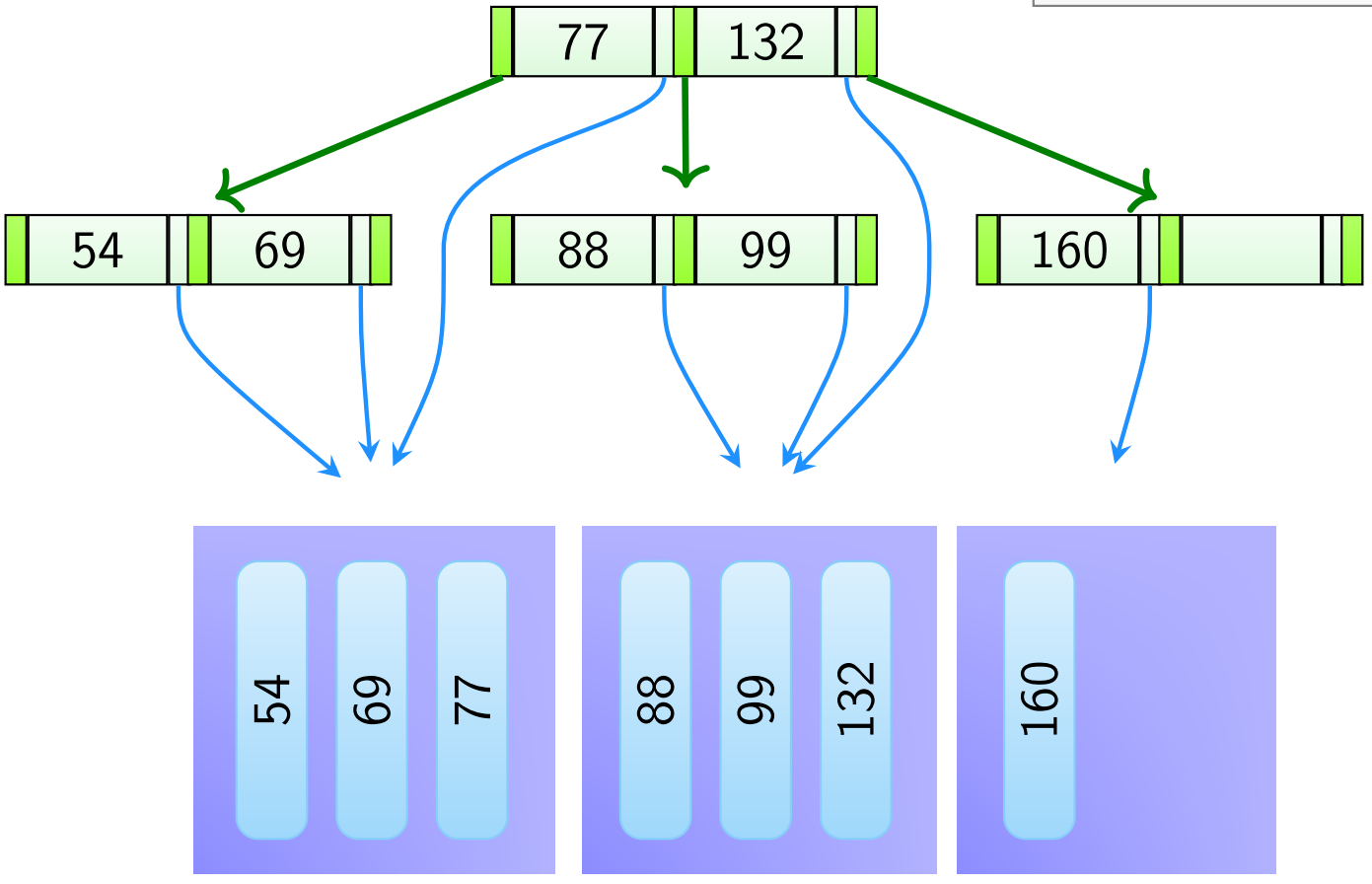
- *local index*: index partitions follow the partitions of the database: 1-1 correspondance between table and index partitions. Optimizes maintenance. Most common in DW.
- *global partitioned index* : index partitions do not correspond to table partitions. More common in OLTP.
- *global non-partitioned index*: index is not partitioned. Mostly used to enforce unique key constraints. But one can also rely on ETL for that.

B-trees

Bayer,McCreight'72

The typical index in operational DB is a B-tree.

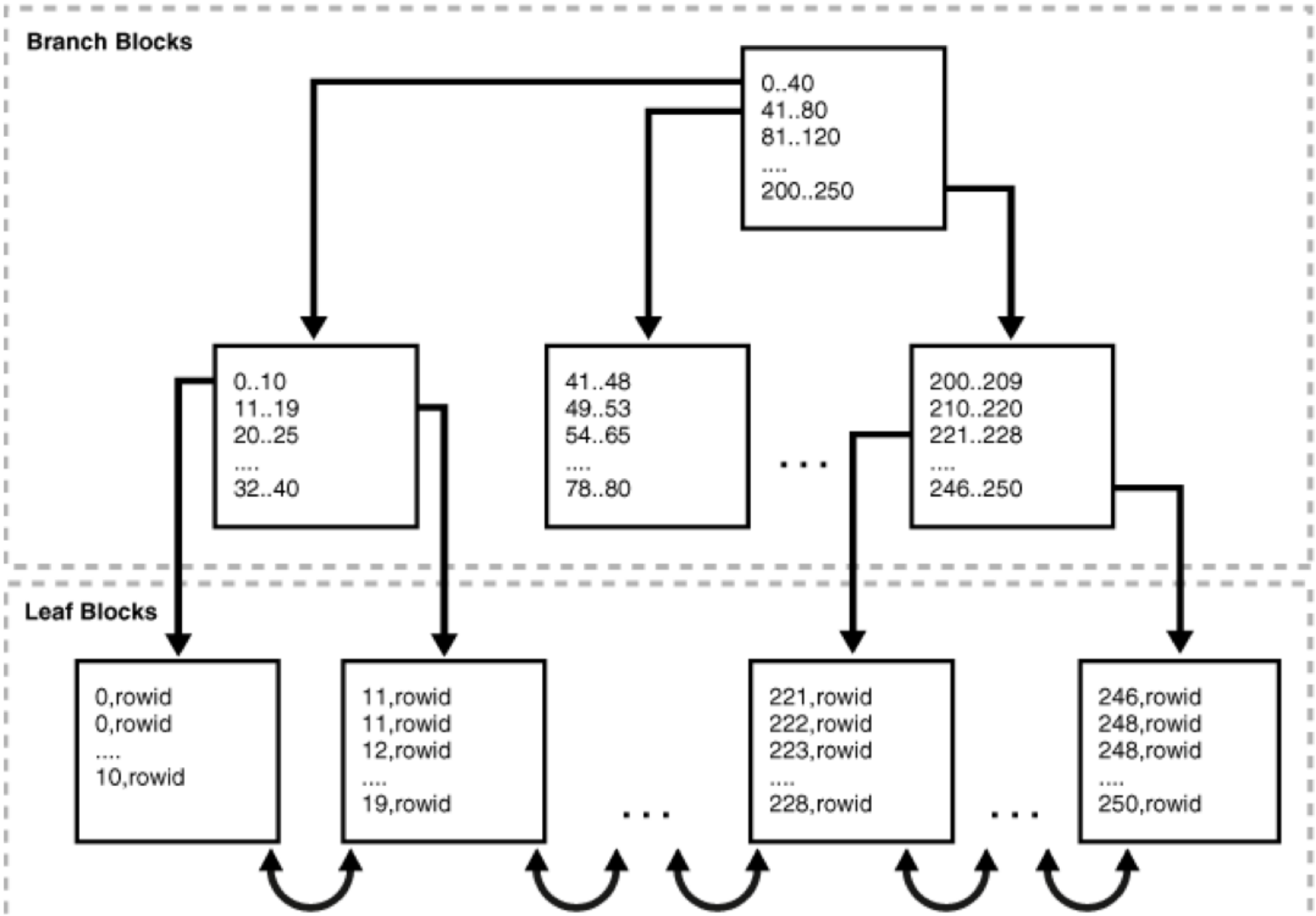




Between m and $2m$ keys per node (root may have less).
Leaves belong to same level (balanced tree).

B-trees: a "B-tree" index from Oracle

Figure 3-1 Internal Structure of a B-tree Index



What kind of index is that: B/B^+? particularities?

B-trees in oracle: properties

Generally, index key associated to ROWID.

For composite indexes : optimizer may perform *index skip-scan*: searches one subindex per distinct value of first attribute (useful if low cardinality)

Index clustering factor : measures correlation between index order and repartition of records among data blocks.

Creating a B-tree index

```
CREATE INDEX cust_gender_adr_idx ON Customers(gender,address);
SELECT * FROM Customers WHERE address='36 Quai des Orfèvres';
```

B-trees: properties

Assets:

- ✔ Inserts, update, delete in $O(\log_m(n))$.
- ✔ Data type-independent (only requires order)[1]
- ✔ Needs re-balancing after data modifications (not really an issue in DW)
- ✔ One-dimensional index structure: key= 1 attribute or sequence (not set).
- ✔ B^+-tree good for range queries, on primary index.

Weaknesses:

- ✘ hardly adequate for multidimensional queries
- ✘ useful only when selectivity is high
  (degenerate trees on low-cardinality attributes)

Ex: index on gender: 1.000.000 rows, $\approx$ 50% M/F
$\Longrightarrow$ 500.000 accesses. Table scan way faster.

[1] can even be defined on a function of attributes

Index requirements for datawarehouses

- index may rely on ETL to enforce UNIQUE constraints.
- queries often combine multiple conditions in WHERE clause
- queries often join fact table with dimensions

- other indexes considerations: pre-aggregated levels, dimensions play symmetric role, cube data is sparse, batch updates should be efficient.

$\Longrightarrow$ B-trees will be used almost exclusively for unique columns.

Typical ROLAP indexes:

- bitmap index
- bitmap join index

Bitmap Index

Idea: code value of attribute with bitmaps (bit-array).

Bitmap index on attribute a of relation R (with n records)

Data structure that stores one bitmap (n bits) per value of a:

$$i^{\text{th}} \text{ bit} = \begin{cases} 1 & \text{in the bitmap corresponding to value of } a \text{ in } i^{\text{th}} \text{ record} \\ 0 & \text{for other bitmaps} \end{cases}$$

Book

BookID	Title	Binding	Language
3240	Le Petit Prince	Hardcover	French
2211	Winnie the Pooh	Hardcover	English
9754	Paddington Bear	Paperback	NULL
4315	Pinocchio	Hardcover	Italian
2368	Les Vacances	Paperback	French

Bitmap index on *Binding*

Hardcover	Paperback
1	0
1	0
0	1
1	0
0	1

or rather:

```
Hardcover: 11010
Paperback: 00101
```

Bitmap compression

Bit-vectors sparse on high-cardinality attribute:

✔ compression opportunities

✘ compression may increase maintenance overhead

✘ *must be decompressed at query time*

In OLTP: using compressed bitmap results in lock on many records.

Bitmap compression: RLE

Run-Length Encoding

Principle: *coding length of each block of repeated numbers*: *run-length*

Actually many variants of RLE. Here is *one*:

- a block (run) represents $k \geq 0$ "0" followed by a "1"
- block length written in binary
- add prefix to delimit each block (code must be unambiguous)
- trailing "0"s can be omitted (# records is known)

RLE: encodes a block of k "0" and a "1" ($\underline{k}_2$=k in binary):

- $k \geq 1$: $(|\underline{k}_2| - 1)$ "1", then "0", then $\underline{k}_2$.
- $k = 0$: 00

Examples:

000101 is encoded as $10\overbrace{1}^{\underline{3}_2}01$

$\underbrace{1}_{7}\underbrace{0000000}_{}\underbrace{1}_{4}\underbrace{0000}_{}1000$ as: $00\underbrace{11}_{\underline{7}_2}\underbrace{0}_{}\underbrace{111}_{}\underbrace{10}_{\underline{4}_2}\underbrace{0}_{}$

Bitmap Indexes: benefits

Assets:

- ✔ no need to store ROWID: each bit-vector is much more compact than B-tree.
- ✔ because of small size, bitmaps usually fit into main memory.
- ✔ boolean operations on bitmaps very fast (AND, OR, XOR, NOT...)
- ✔ good on low-cardinality attributes
- ✔ not bad on high cardinality attributes:
- ✔ compression opportunities if sparse array.

Weaknesses:

- ✘ maintenance is costly: all indexes must be updated on insertion...
- ✘ storage space increases on high cardinality attributes
- ✘ decompression overhead (degenerate trees on low-cardinality attributes)

Bitmap indexes: queries with boolean operations

Book

BookID	Title	Binding	Language	Price
3240	Le Petit Prince	Stapled	French	35
2211	Winnie the Pooh	Hardcover	English	40
9754	Paddington Bear	Paperback	English	35
4315	Pinocchio	Stapled	Italian	15
2368	Les Vacances	Paperback	French	40

```
SELECT *
FROM Book
WHERE Binding != 'Stapled'
  AND Price BETWEEN 20 AND 40;
```

```
Hardcover:      01000        40:        01001
Paperback:      00101        35:        10100
Stapled:        10010        15:        00010

NOT Stapled:    01101        35 OR 40:  11101
```

```
NOT Stapled AND (35 OR 40):   01101
```

Bitmap join index

Aim: index on relation R over attribute a in relation R'.

Precomputes the join(s) (spares joins at query time).

Stores one bitmap on R for each value of n'.

Sales

ShopID	BookID	Count
1	3240	2
1	2368	1
2	3240	4
2	9754	8
2	2211	5
3	9754	3

Book

BookID	Title	Binding	Language
3240	Le Petit Prince	Hardcover	French
2211	Winnie the Pooh	Hardcover	English
9754	Paddington Bear	Paperback	NULL
4315	Pinocchio	Hardcover	Italian
2368	Les Vacances	Paperback	French

Bitmap (join) index on Sales over Binding:

```
Hardcover: 101010
Paperback: 010101
```

Bitmap/JOIN indexes in Oracle

Bitmap index

```
CREATE BITMAP INDEX binding_ix ON Book(binding);
```

Bitmap join index

```
CREATE BITMAP JOIN INDEX binding_bjix
ON Sales(Book.binding)
FROM Sales, Book
WHERE Sales.BookID=Book.BookID;
```

JOIN INDEX $\neq$ INDEX JOIN

Star queries

Queries in DW are typically *star queries*, i.e., join the (huge) fact table to (small) dimension tables, but do not join any pair of dimensions.

(Traditional) DBMS joins are *pairwise*. Query optimizer then orders joins to minimize size of intermediate results. Traditional heuristic: only join relations connected through an attribute (no cross product).

Problem: only table related to others: Fact table.

Alternative approach: first compute cross product of (relevant) dimension rows, then join with fact table.

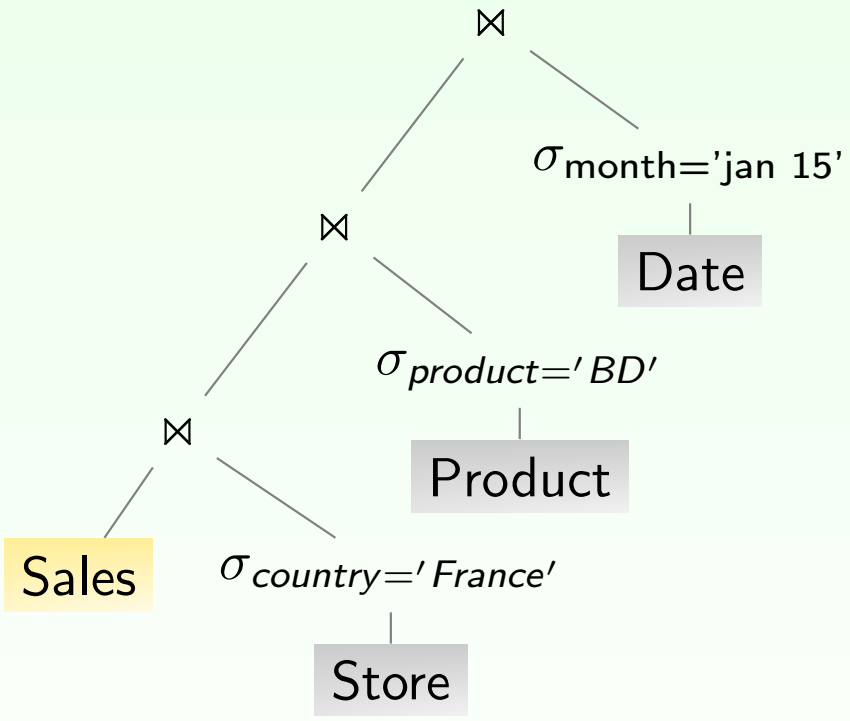
$\hookrightarrow$ alternative may be more efficient when predicates are highly selective!

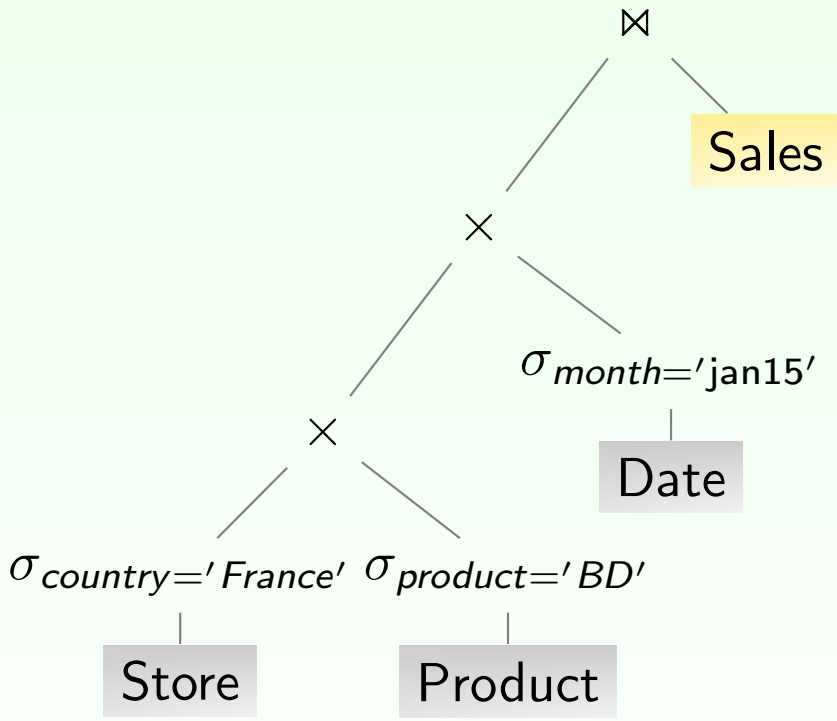
Bitmap indexes/ bitmap join indexes typically help optimize such queries.

Star queries (2)

Assume:

- 10.000.000 sales records
- query selects 10 stores (out of 100), 50 products (out of 1000), 20 days (out of 1000).
- uniform repartition of sales among stores, etc.





Star queries (Oracle)

If there is a bitmap index (or bitmap join index) on all necessary foreign keys in the fact table, Oracle can then consider among the possible query execution plans what they call the *star transformation*:

- first select relevant tuples in the fact table
  for this; select relevant IDs in each dimensions, then merge these IDs into a single bitmap per FK, then intersect the bitmaps
- then join the fact table to dimension tables

requires parameter STAR_TRANSFORMATION_ENABLED = true.

DB2 has similar approach, but uses one semijoin per dimension with B-tree on the FK instead of bitmaps.

Table of content

Indexation: B-trees, bitmaps...

- Indexes
- Clustering

"Clustered indexes" in oracle

Usual tables are *heap-organized*.

Index-organized-table (Oracle): a B^+-tree contains directly the data.
(index-only scan saves 1 indirection: the data block I/O)

Other indexes on table only store *logical* ROWIDs.

Creating an index-organized table

```
CREATE TABLE FrenchCities (
zip int PRIMARY KEY,
city_name VARCHAR2(20))
ORGANIZATION INDEX
MAPPING TABLE-- optional, creates a table mapping logical rowids
-- the mapping table is necessary to support bitmap indexes
;
```

"Clusters" in oracle: table clusters

When multiple tables share columns and are often queried together, they can be grouped in a *table cluster*:

- *Index cluster*: stores together (same block) rows from all tables with same value on cluster key.
- *Hash cluster*: stores together rows from all tables with same hash on cluster key.

Creating a clustered table

```
CREATE CLUSTER personnel(department NUMBER(4));

CREATE TABLE dept_10
  CLUSTER personnel (department_id)
  AS SELECT * FROM employees WHERE department_id = 10;

CREATE CLUSTER personnel(department NUMBER(4))
  HASHKEYS 10;
  --HASHKEYS should be approximately nb of unique values(performance)

--options may redefine hash function, storage size...
```

[https://docs.oracle.com/database/121/CNCPT/tablecls.htm]

"Clusters" in oracle: table clusters

Assets:

- ✔ speed-up on joins (access-time, # of block I/O)
- ✔ saves space (key repetitions)

Weaknesses:

- ✘ maintenance cost
- ✘ table scan slower

Multidimensional clustering

Multicol. index weakness: 1) costly 2) useless when accessing non-prefix of key.

Multidimensional clustering on a table:

stores rows according to the value of cluster columns (physical proximity).

In oracle ($\geq$12 only): supported through *attribute clustered* table. Store rows in an ordered way, where order can be:

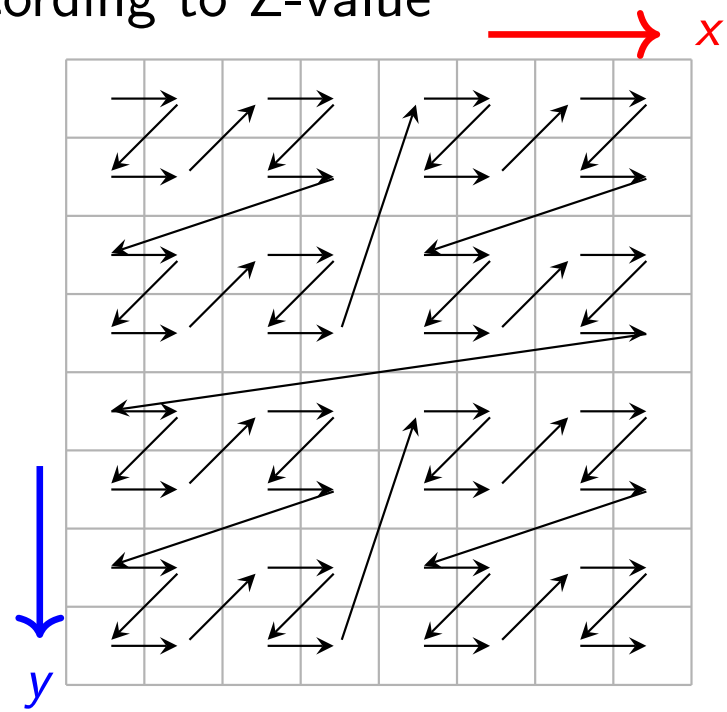
- *linear ordering*: usual order on cluster column(s)
- *interleaved ordering*: data stored according to Z-value

rank of (i,j) in Z-curve:
interleave $\underline{i}_2$ and $\underline{j}_2$

y x

ex: $(1,1) \rightarrow 11 = \underline{3}_2$

$(2,0) \rightarrow 1000 = \underline{8}_2$



Creating attribute-clustered table in Oracle

(or reorganizing a table into a clustered one)

Creating an attribute-clustered table

```
CREATE TABLE sales (
  prod_id         NUMBER(6) NOT NULL,
  cust_id         NUMBER NOT NULL,
  time_id         DATE NOT NULL,
  amount_sold     NUMBER(10,2) NOT NULL
)
CLUSTERING BY INTERLEAVED ORDER (time_id, prod_id,cust_id);
-- CLUSTERING BY LINEAR ORDER (time_id, prod_id,cust_id);
```

Join attribute clustering

```
ALTER TABLE sales
  ADD CLUSTERING
    sales JOIN product ON (sales.prod_id=products.product_id)
    BY INTERLEAVED ORDER (time_id, product_cat);
```

[https://docs.oracle.com/database/121/DWHSG/attcluster.htm]

"Clusters" in oracle: attribute clustering

Assets:

- ✔ no huge storage costs like indexes
- ✔ I/O reduction: allows direct access to relevant regions.
  To achieve such savings in oracle; use in conjunction with
  - zone maps
  - in-memory min/max pruning
  - exadata storage indexes
  - interleaved ordering not affected by columns order
- ✔ improves compression ratio

Weaknesses:

- ✘ costly
- ✘ therefore oracle will not maintain the clustering after updates

"Clusters" in various DBMS

Some form of clustering supported in:

- MySQL(InnoDB):
  clustered index
- Postgresql:
  CLUSTER table: 1-time reordering, linear.
- DB2 (8,9...):
  clustered index, MDC: creates one block index per column.
- SQL Server:
  clustered index, columnstores for multidimensional clustering.

[1] Z-curve also used by Oracle, SQL Server, DB2... for spatial objects.

References

Oracle Database Concepts
https://docs.oracle.com/database/121/CNCPT/tablecls.htm

Microsoft: clustered indexes and columstores
https://msdn.microsoft.com/en-us/library/ms189051

https://msdn.microsoft.com/fr-fr/library/gg492088

DB2: indexes and MDC
https://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0020180.html

http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.admin.partition.doc/doc/c0007201.html

http://www.dbisoftware.com/db2nightshow/episode37slides.pdf

[Padmanabhan and Cranston, SIGMOD'03]: available on Citeseer

PostgreSQL: Cluster
http://www.postgresql.org/docs/9.1/static/sql-cluster.html

(MultiDimensional) indexes: T. Grust's lecture
http://db.inf.uni-tuebingen.de/teaching/DatenbanksystemeIISS2014.html

Other physical storage structures

For multidimensional or spatial information:

- UB-tree: a B-tree on Z-order
- R-tree (and variants)
- kd-tree
- grid files
- multidimensional hashing

...

Other physical storage structures: Oracle's zone maps

Similar feature in IBM DB2.

Zone Map:

access structure that can be built on a table. Allows to prune blocks or partitions based on predicates during scans.

- stores min/max value of the relevant column(s) for each zone
- at most one per table, can include multiple columns
- implementation similar to materialized views
- can also store min/max value of another table through join

Properties:

- works best when data is clustered w.r.t. the zone map attributes
- more compact than indexes (granularity=1 zone instead of 1 row)
- the ranges for some zones may become stale