Multi-dimensional index structures
Part I: motivation

Motivation: Data Warehouse

A definition

> *"A data warehouse is a repository of integrated enterprise data. A data warehouse is used specifically for decision support, i.e., there is (typically, or ideally) only one data warehouse in an enterprise. A data warehouse typically contains data collected from a large number of sources within, and sometimes also outside, the enterprise."*

Decision support (1/2)

'Traditional" relational databases were designed for online transaction processing (OLTP):

- flight reservations; bank terminal; student administration; . . .

OLTP characteristics:

- Operational setting (e.g., ticket sales)
- Up-to-date = critical (e.g., do not book the same seat twice)
- Simple data (e.g., [reservation, data, name])
- Simple queries that only access a small part of the database (e.g., "Give the flight details of X" or "List flights to Y")

Decision support systems have different requirements.

Decision support (2/2)

Decision support systems have different requirements:

- Offline setting (e.g., evaluate flight sales)
- Historical data (e.g., flights of last year)
- Summarized data (e.g., # passengers per carrier for destination X)
- Integrates different databases (e.g., passengers, fuel costs, maintenance information)
- Complex statistical queries (e.g., average percentage of seats sold per month and destination)

Decision support (2/2)

Decision support systems have different requirements:

- Offline setting (e.g., evaluate flight sales)
- Historical data (e.g., flights of last year)
- Summarized data (e.g., # passengers per carrier for destination X)
- Integrates different databases (e.g., passengers, fuel costs, maintenance information)
- Complex statistical queries (e.g., average percentage of seats sold per month and destination)

Taking these criteria into mind, data warehouses are tuned for online analytical processing (OLAP)

- Online = answers are immediately available, without delay.

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

- Example: sales June to August 2010

	Blue	Red	Orange	Total
June	51	25	128	**234**
July	58	20	120	**198**
August	65	22	51	**138**
Total	**174**	**67**	**329**	**570**

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

Data Cubes are extensions of cross-tabs to multiple dimensions

Dimension X

Dimension Y

	Blue	Red	Orange	Total
June	51	25	128	**234**
July	58	20	120	**198**
August	65	22	51	**138**
Total	**174**	**67**	**329**	**570**

Aggregated w.r.t Dimension X

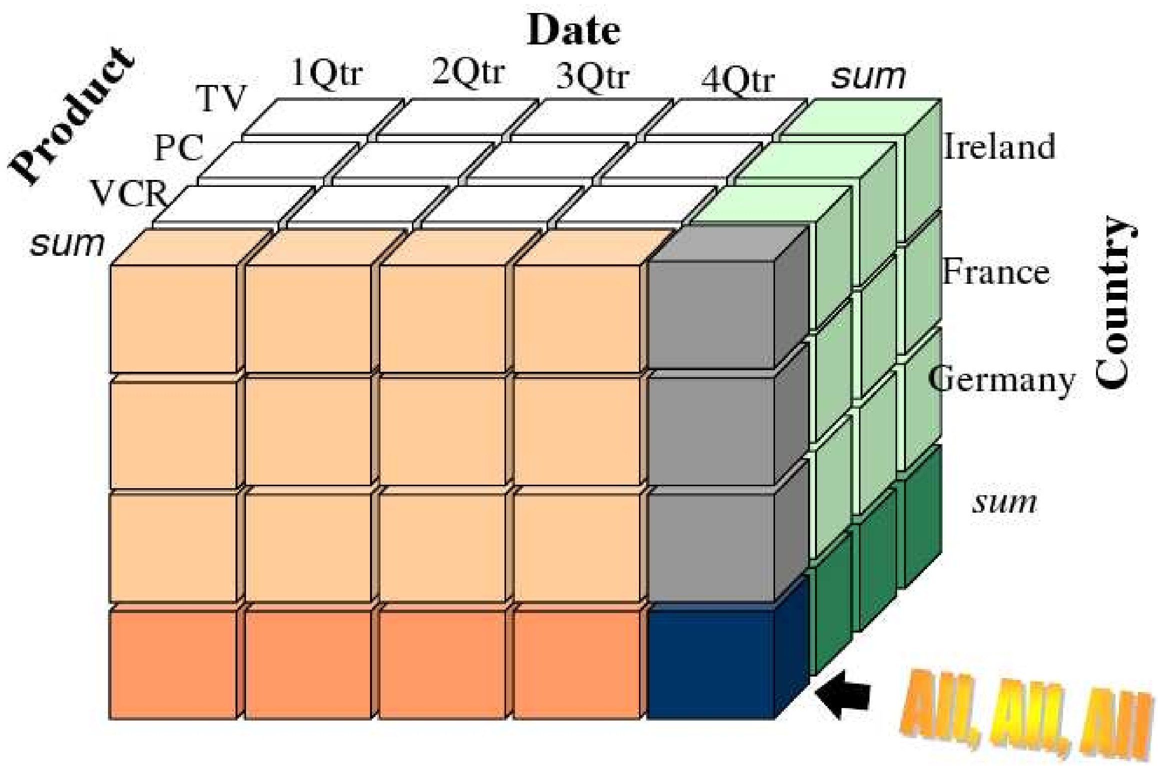
Aggregated w.r.t. Dimension Y

Aggregated w.r.t Dimension X and Y

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

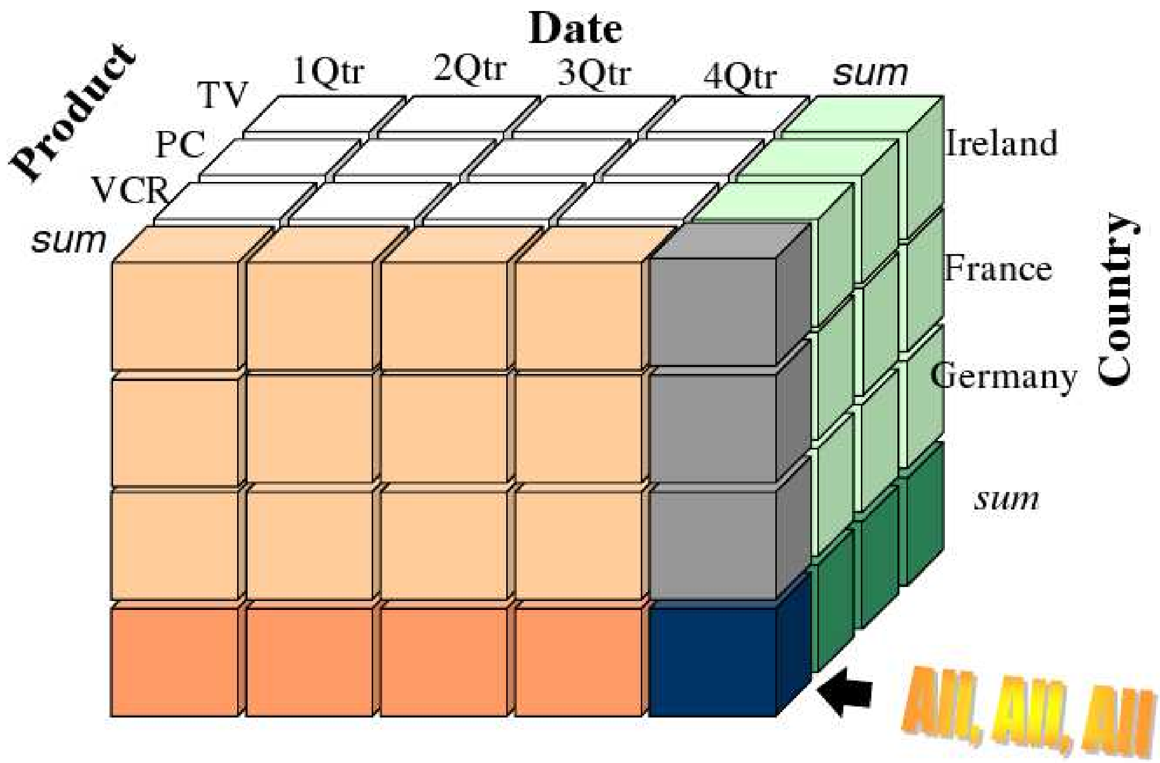
Data Cubes are extensions of cross-tabs to multiple dimensions



OLAP Operations on the CUBE

Roll-up

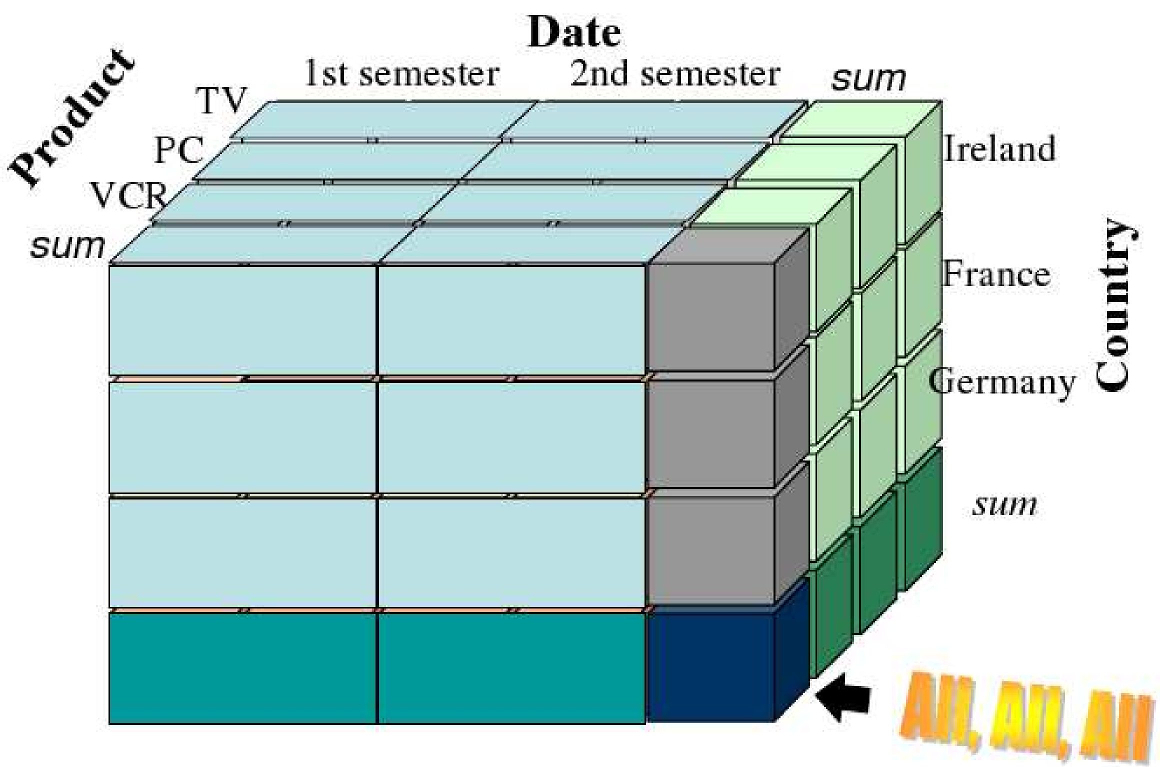
- Group per semester instead of per quarter



OLAP Operations on the CUBE

Roll-up

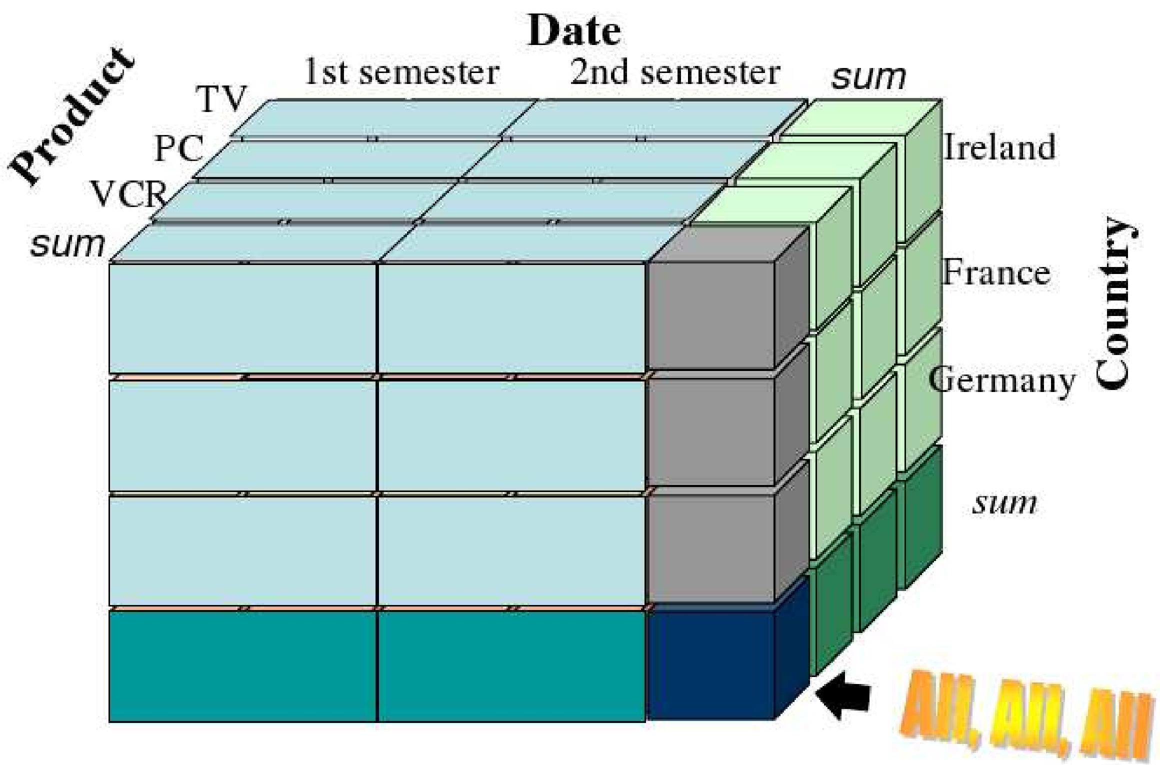
- Show me totals per semester instead of per quarter



OLAP Operations on the CUBE

Roll-up

- Show me totals per semester instead of per quarter

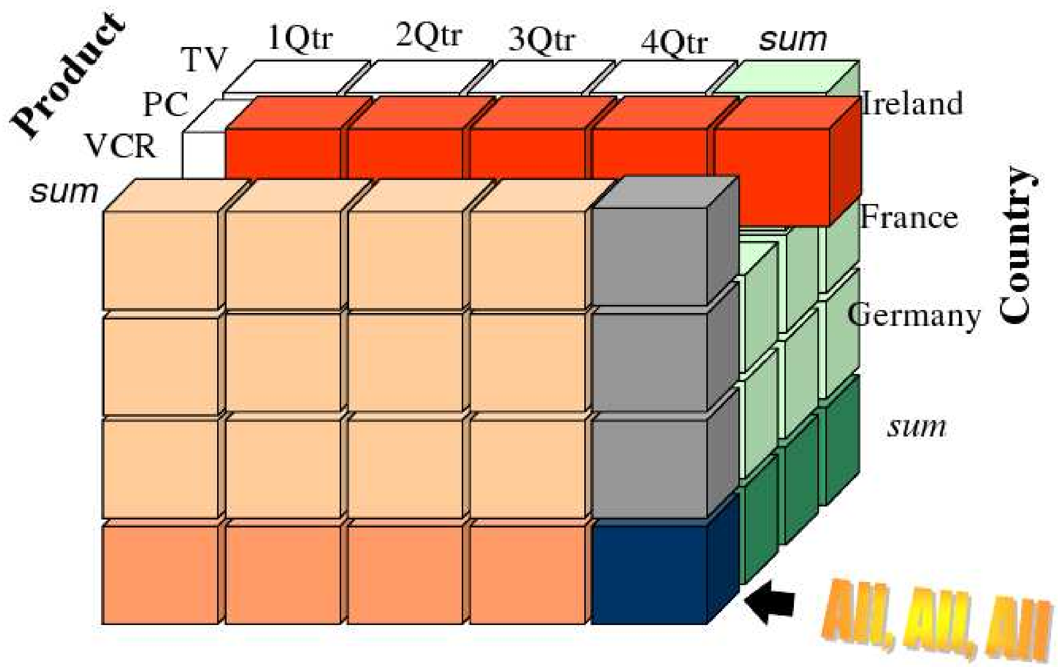


Inverse is drill-down

OLAP Operations on the CUBE

Slice and dice

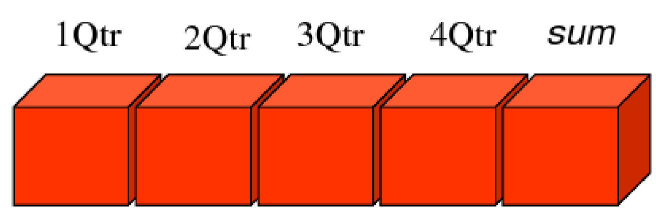
- Select part of the cube by restricting one or more dimensions
- E.g, restrict analysis to Ireland and VCR



OLAP Operations on the CUBE

Slice and dice

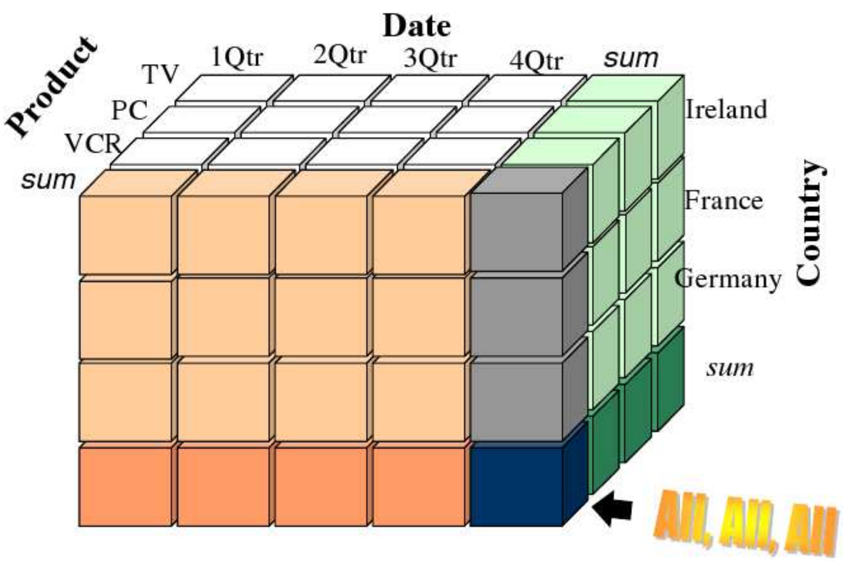
- Select part of the cube by restricting one or more dimensions
- E.g, restrict analysis to Ireland and VCR



Different OLAP systems

Multidimensional OLAP (MOLAP)

- Early implementations used a multidimensional array to store the cube completely:
- In particular: pre-compute and materialize all aggregations



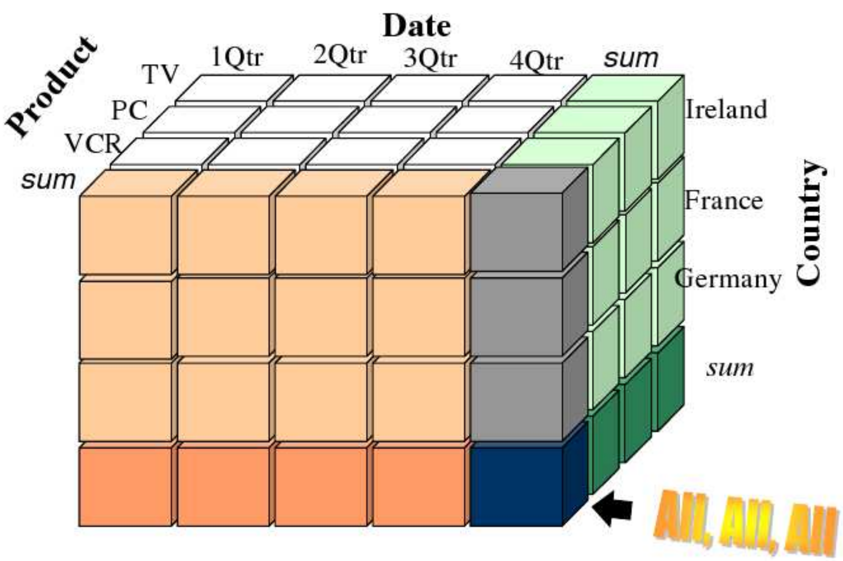
Array: **cell[product, date, country]**

- Fast lookup: to access cell[p,d,c] just use array indexation

Different OLAP systems

Multidimensional OLAP (MOLAP)

- Early implementations used a multidimensional array to store the cube completely:
- In particular: pre-compute and materialize all aggregations



Array: **cell[product, date, country]**

- Fast lookup: to access cell[p,d,c] just use array indexation
- Very quickly people realized that this is infeasible due to the data explosion problem

The data explosion problem

The problem:

- Data is not dense but sparse
- Hence, if we have n dimensions with each c possible values, then we do not actually have data for all the c^n cells in the cube.
- Nevertheless, the multidimensional array representation realizes space for all of these cells

The data explosion problem

The problem:

- Data is not dense but sparse
- Hence, if we have n dimensions with each c possible values, then we do not actually have data for all the c^n cells in the cube.
- Nevertheless, the multidimensional array representation realizes space for all of these cells

Example: 10 dimensions with 10 possible values each

- 10 000 000 000 cells in the cube
- suppose each cell is a 64-bit integer
- then the multidimensional-array representing the cube requires ≈ 74.5 gigabytes to store $\rightarrow$ does not fit in memory!
- yet if only 1 000 000 cells are present in the data, we actually only need to store ≈ 0.0074 gigabytes

Multidimensional OLAP (MOLAP)

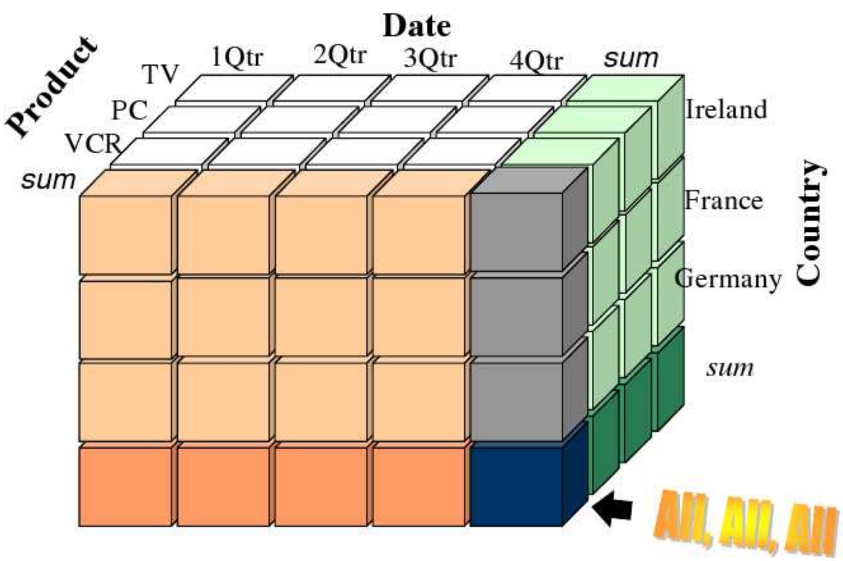
In conclusion

- Naively storing the entire cube does not work.
- Alternative representation strategies use sparse main memory index structures:
  - search trees
  - hash tables
  - ...
- And these can be specialized to also work in secondary memory
  $\rightarrow$ multidimensional indexes (the main technical content of this lecture).

Relational OLAP (ROLAP)

Key Insight [Gray et al, Data Mining and Knowledge Discovery, 1997]

- The n-dimensional cube can be represented as a traditional relation with $n+1$ columns (1 column for each dimension, 1 column for the aggregate)
- Use special symbol `ALL` to represent grouping

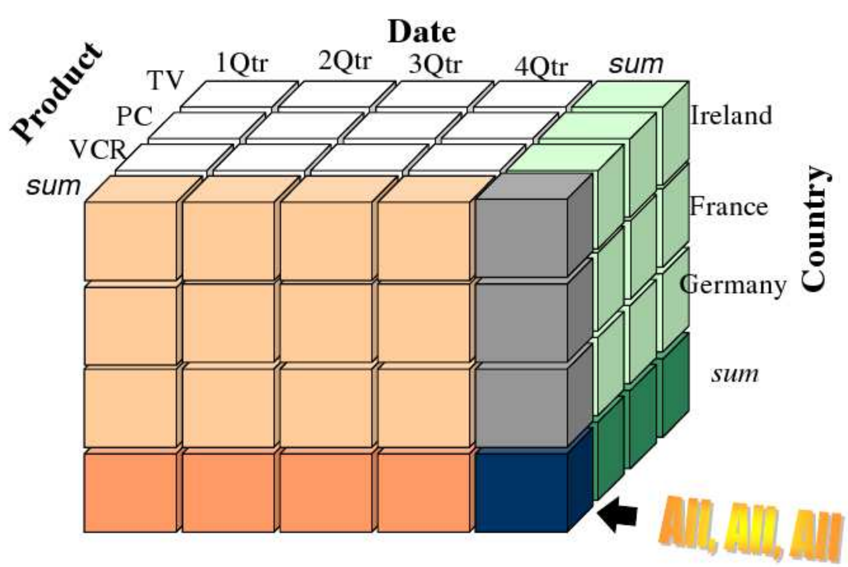


Product	Date	Country	Sales
TV	Q1	Ireland	100
TV	Q2	Ireland	80
TV	Q3	Ireland	35
...	...	...	...
PC	Q1	Ireland	100
...	...	...	...
TV	**ALL**	Ireland	215
TV	**ALL**	**ALL**	1459
...	...	...	...
ALL	**ALL**	**ALL**	109290

Relational OLAP (ROLAP)

Key benefits: space usage

- The non-aggregate cells that are not present in the original data are also not present in the relational cube representation.
- Moreover, it is straightforward to represent only aggregation tuples in which all dimension columns have values that already occur in the data



Product	Date	Country	Sales
TV	Q1	Ireland	100
TV	Q2	Ireland	80
TV	Q3	Ireland	35
...	...	...	...
PC	Q1	Ireland	100
...	...	...	...
TV	**ALL**	Ireland	215
TV	**ALL**	**ALL**	1459
...	...	...	...
ALL	**ALL**	**ALL**	109290

Relational OLAP (ROLAP)

Key benefits

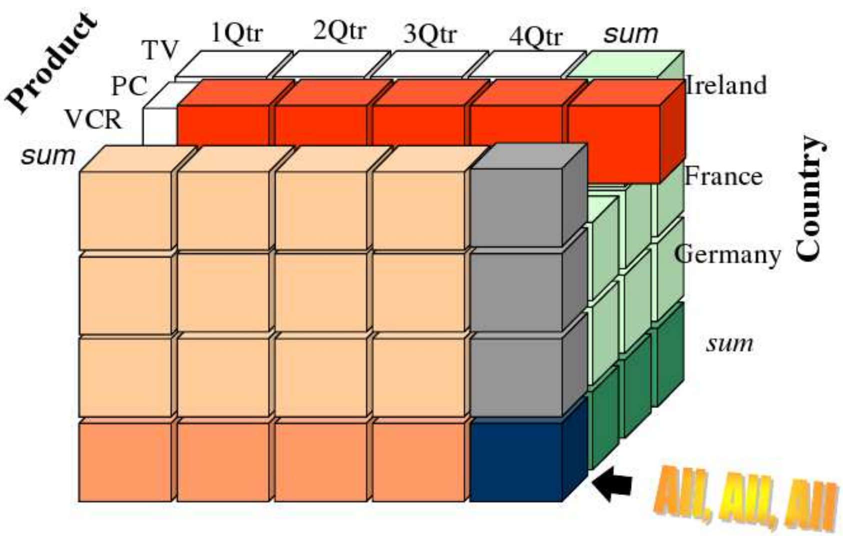
- By representing the cube as a relation it can be stored in a “traditional” relational DBMS ...
- ... which works in secondary memory by design (good for multi-terraby data warehouses) ...
- Hence one can re-use the rich literature on relational query storage and query evaluation techniques,

But, to be honest, much research was done to get this representation efficient in practice.

Relational OLAP (ROLAP)

Key benefits: use SQL

- Dice example: restrict analysis to Ireland and VCR



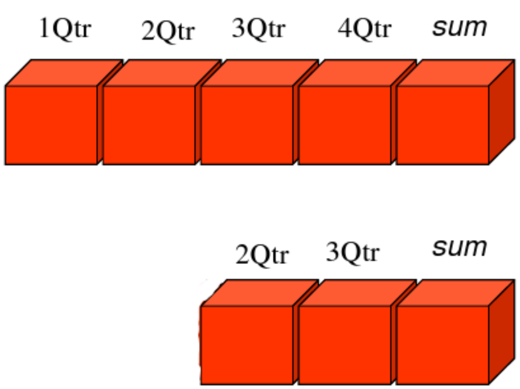

```
SELECT Date, Sales
FROM Cube_table
WHERE Product = "VCR"
  AND Country = "Ireland"
```

Date	Sales
Q1	100
Q2	80
Q3	35
ALL	215

Relational OLAP (ROLAP)

Key benefits: use SQL

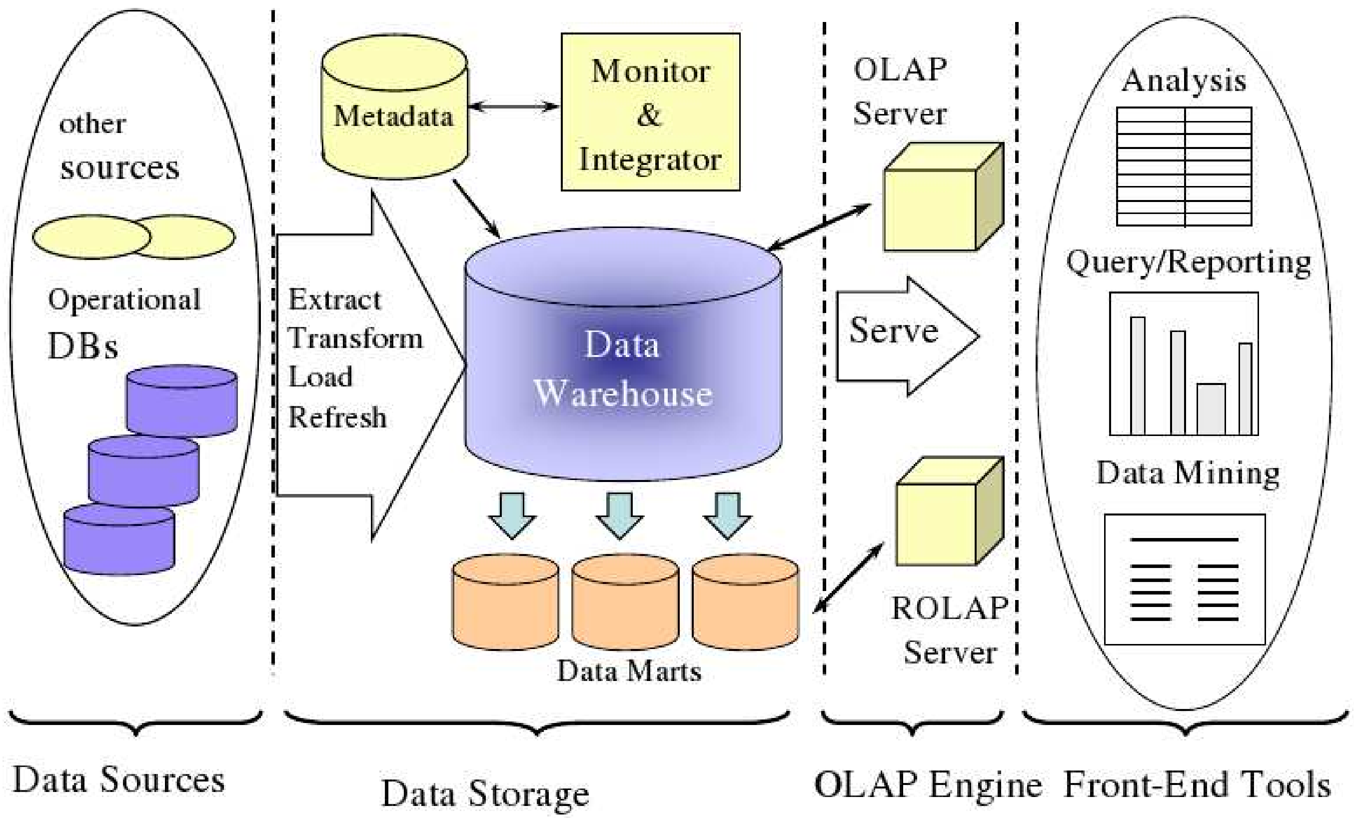
- Dice example: restrict analysis to Ireland and VCR, quarter 2 and quarter 3
  $\rightarrow$ need to compute a new total aggregate for this sub-cube




```
(SELECT Date, Sales
 FROM Cube_table
 WHERE Product = "VCR"
  AND Country = "Ireland"
  AND (Date = "Q2" OR Date = "Q3")
  AND SALES <> "ALL")
UNION
(SELECT "ALL" as DATE, SUM(T.Sales) as SALES
 FROM Cube_table t
 WHERE Product = "VCR"
  AND Country = "Ireland"
  AND (Date = "Q2" OR Date = "Q3")
  AND SALES <> "ALL"
 GROUP BY Product, Country)
```

This actually motivated the extension of SQL with CUBE-specific operators and keywords

Three-tier architecture



Multi-dimensional index structures
Part II: index structures

Multidimensional Indexes

Typical example of an application requiring multidimensional search keys:

Searching in the data cube and searching in a spatial database

Typical queries with multidimensional search keys:

- Point queries:
  - retrieve the Sales total for the product TV sold in Ireland, with an `ALL` value for date.
  - does there exist a star on coordinate $(10, 3, 5)$?
- Partial match queries: return the coordinates of all stars with $x = 5$ and $z = 3$.
- Dicing / Range queries:
  - return all cube cells with date $\geq$ Q1 and date $\leq$ Q3 and sales ≤ 100;
  - return the coordinates of all stars with $x >= 10$ and $20 \leq y \leq 35$.
- Nearest-neighbour queries: return the three stars closest to the star at coordinate $(10, 15, 20)$.

Multidimensional Indexes

Indexes for search keys comprising multiple attributes?

- BTree: assumes that the search keys can be ordered. What order can we put on multidimensional search keys?

  $\rightarrow$ Pick the lexicographical order:

$$\begin{aligned}(x,y,z) \leq (x',y',z') \ \Leftrightarrow \ & x < x' \\ & \vee(x = x' \wedge y < y') \\ & \vee(x = x' \wedge y = y' \wedge z \leq z')\end{aligned}$$

- Hash table: assumes a hash function h : *keys* $\rightarrow \mathbb{N}$. What hash function can we put on multidimensional search keys?

  $\rightarrow$ Extend the hash function to tuples:

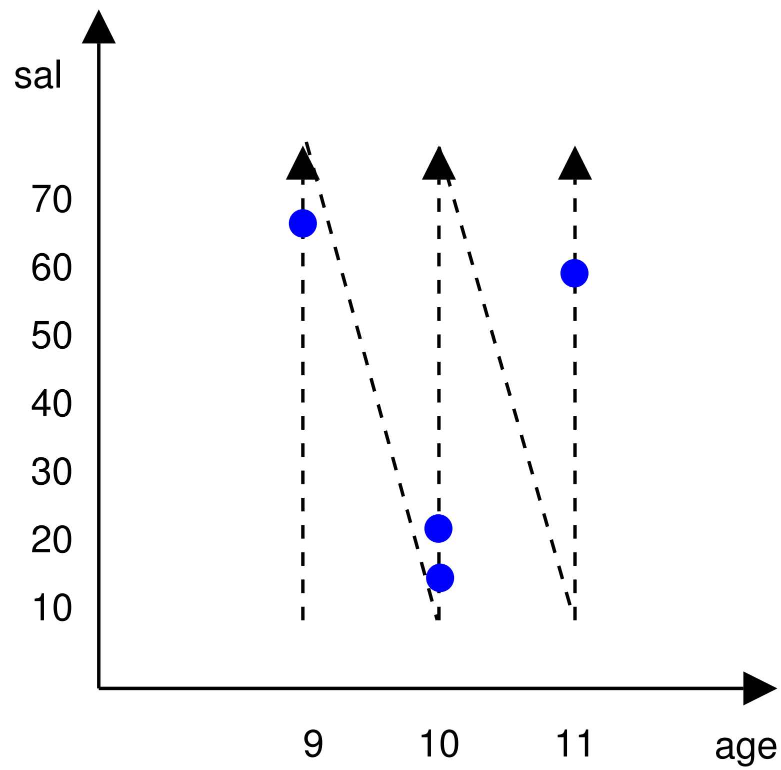
$$h(x,y,z) = h(x) + h(y) + h(z)$$

Multidimensional Indexes

Problem with the lexicographical order in BTrees:

Assume that we have a BTree index on (age, sal) pairs.

- age < 20: ok
- sal < 30: linear scan
- age $< 20 \wedge$ sal < 20



Multidimensional Indexes

Problem with hash tables:

Assume that we have a hash table on (age, sal) pairs.

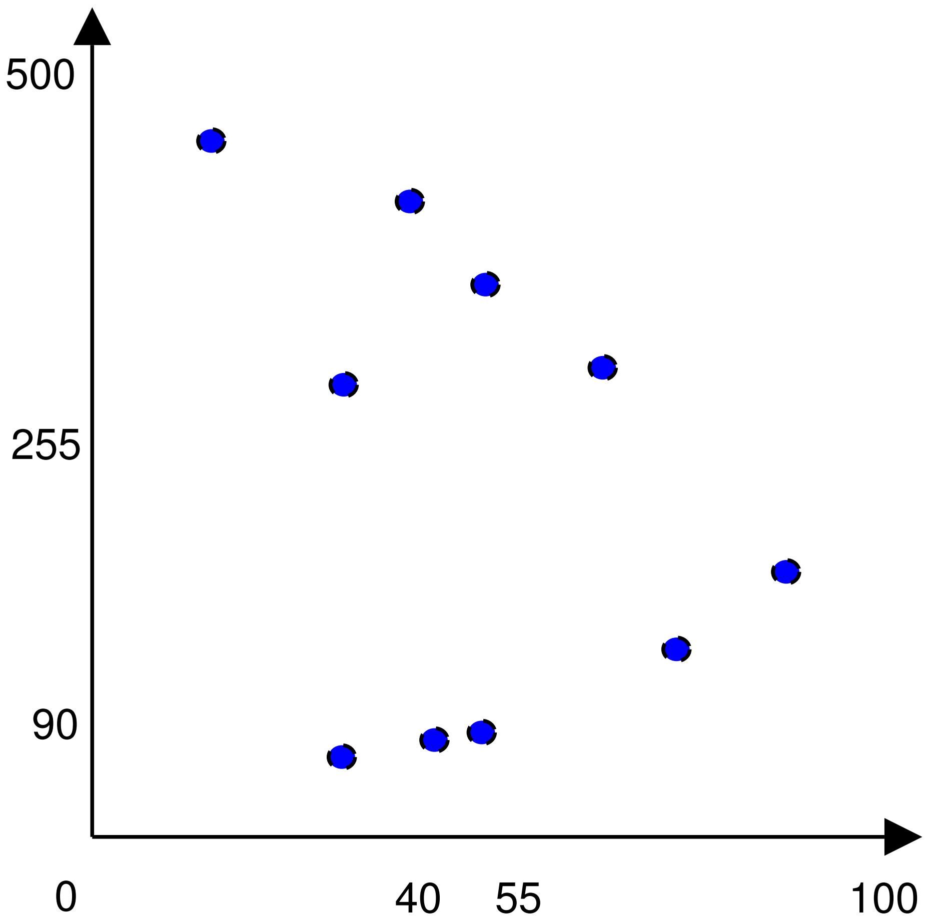
- age < 20: linear scan
- sal < 30: linear scan
- age $< 20 \land$ sal < 20: linear scan

Conclusion: for queries with multidimensional search keys we want to index points by spatial proximity

.

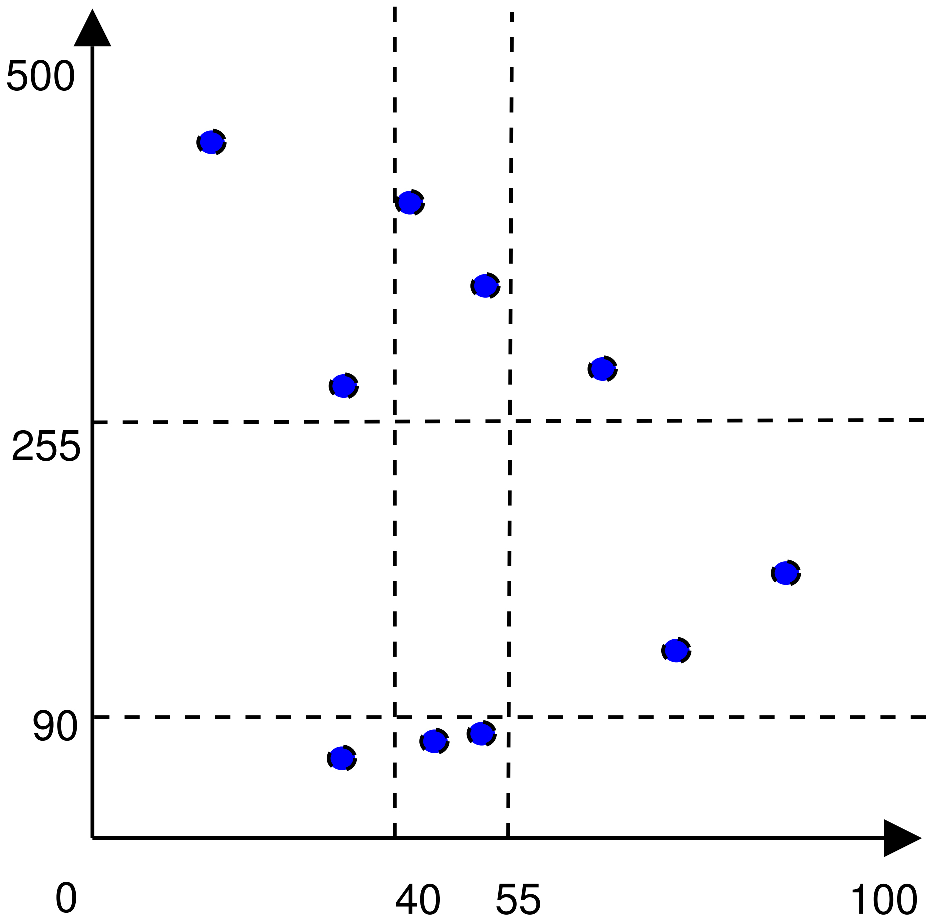
Multidimensional Indexes

Grid files: a variant on hashing



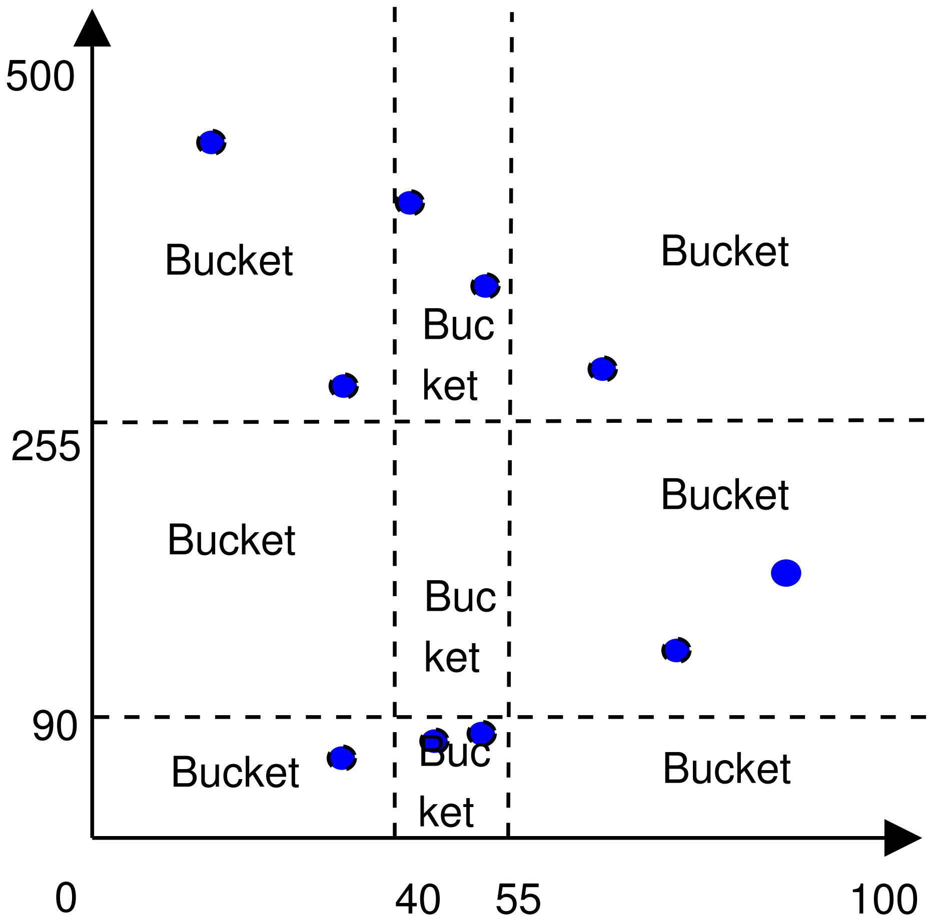
Multidimensional Indexes

Grid files: a variant on hashing



Multidimensional Indexes

Grid files: a variant on hashing



- Insert: find the corresponding bucket, and insert.

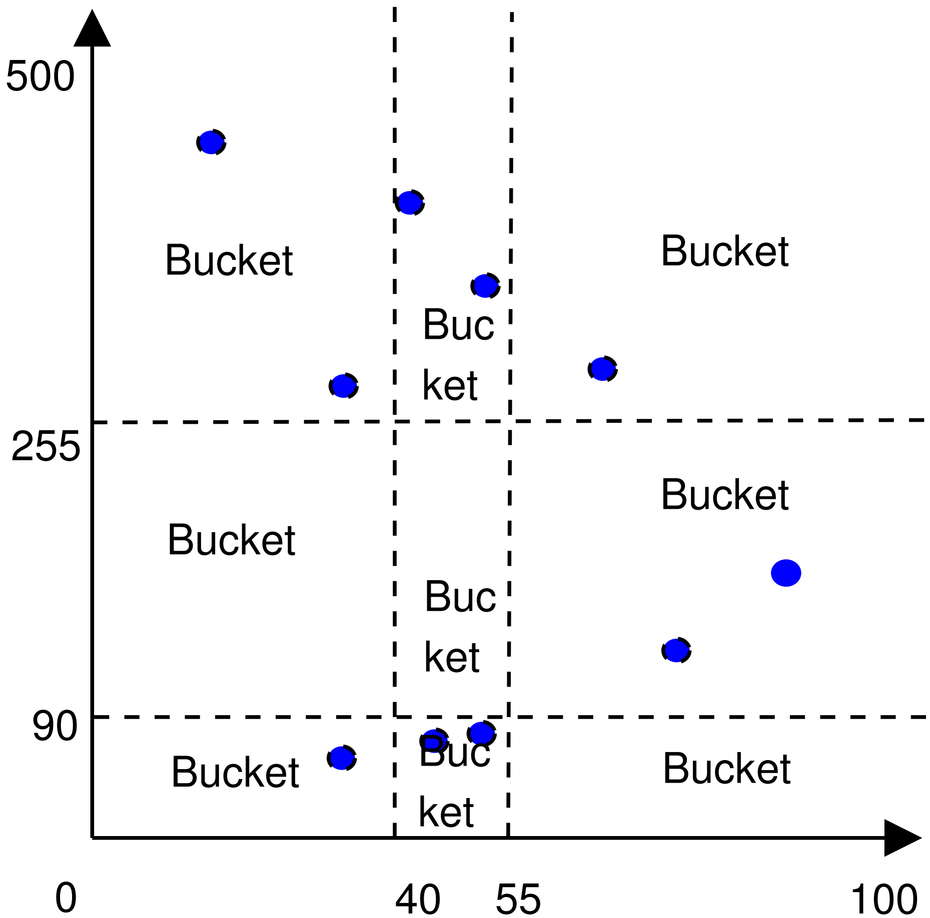
  If the block is full: create overflow blocks or split by creating new separator lines (difficult).

- Delete: find the corresponding bucket, and delete.

  Reorganize if desired

Multidimensional Indexes

Grid files: a variant on hashing



- Good support for point queries
- Good support for partial match queries
- Good support for range queries

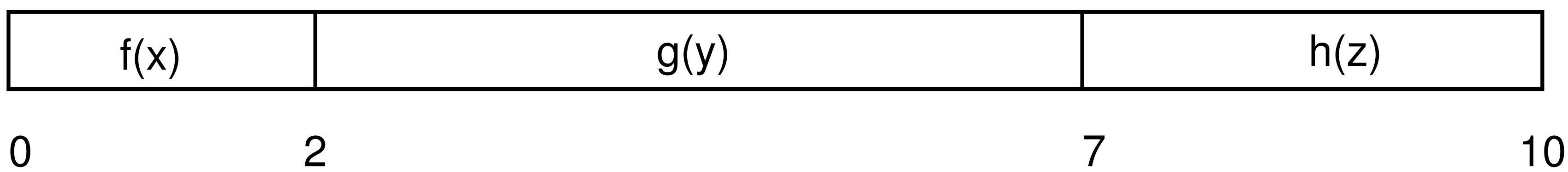
  → Lots of buckets to inspect, but also lots of answers
- Reasonable support for nearest-neighbour queries

  → By means of neighbourhood searching
- But: many empty buckets when the data is not uniformly distributed

Multidimensional Indexes

Partitioned Hash Functions

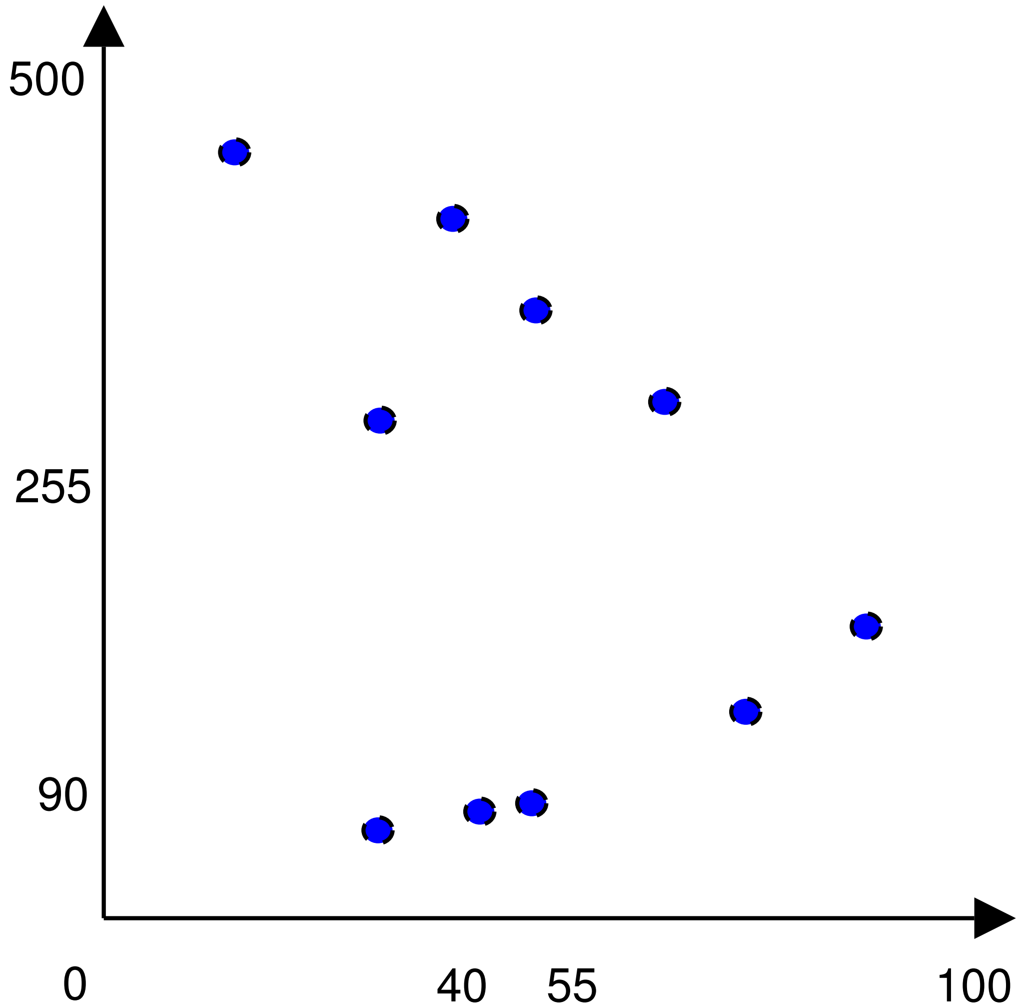
Assume that we have 1024 buckets available to build a hashing index for (x, y, z). We can hence represent each bucket number using 10 bits. Then we can determine the hash value for (x, y, z) as follows:



- Good support for point queries
- Good support for partial match queries
- No support for range queries
- No support for nearest-neighbour queries
- Less wasted space than grid files

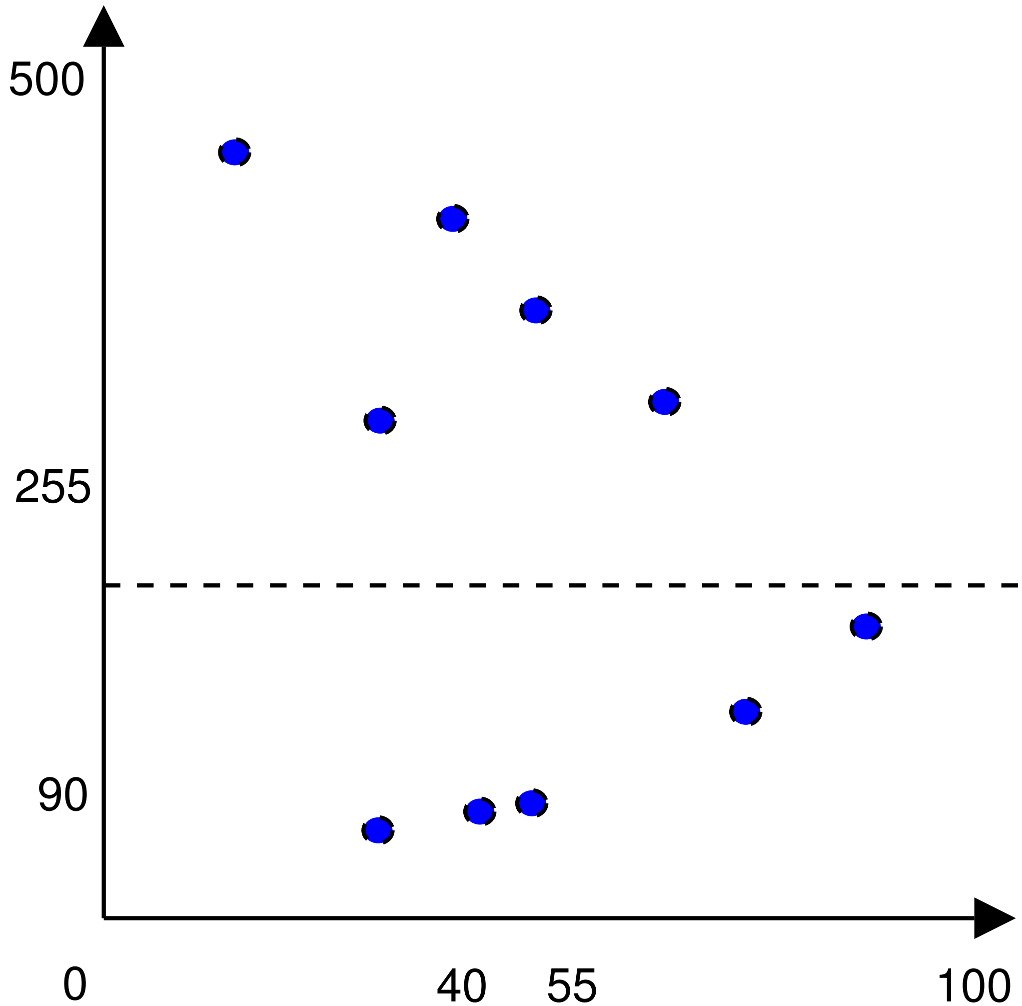
Multidimensional Indexes

kd-Trees



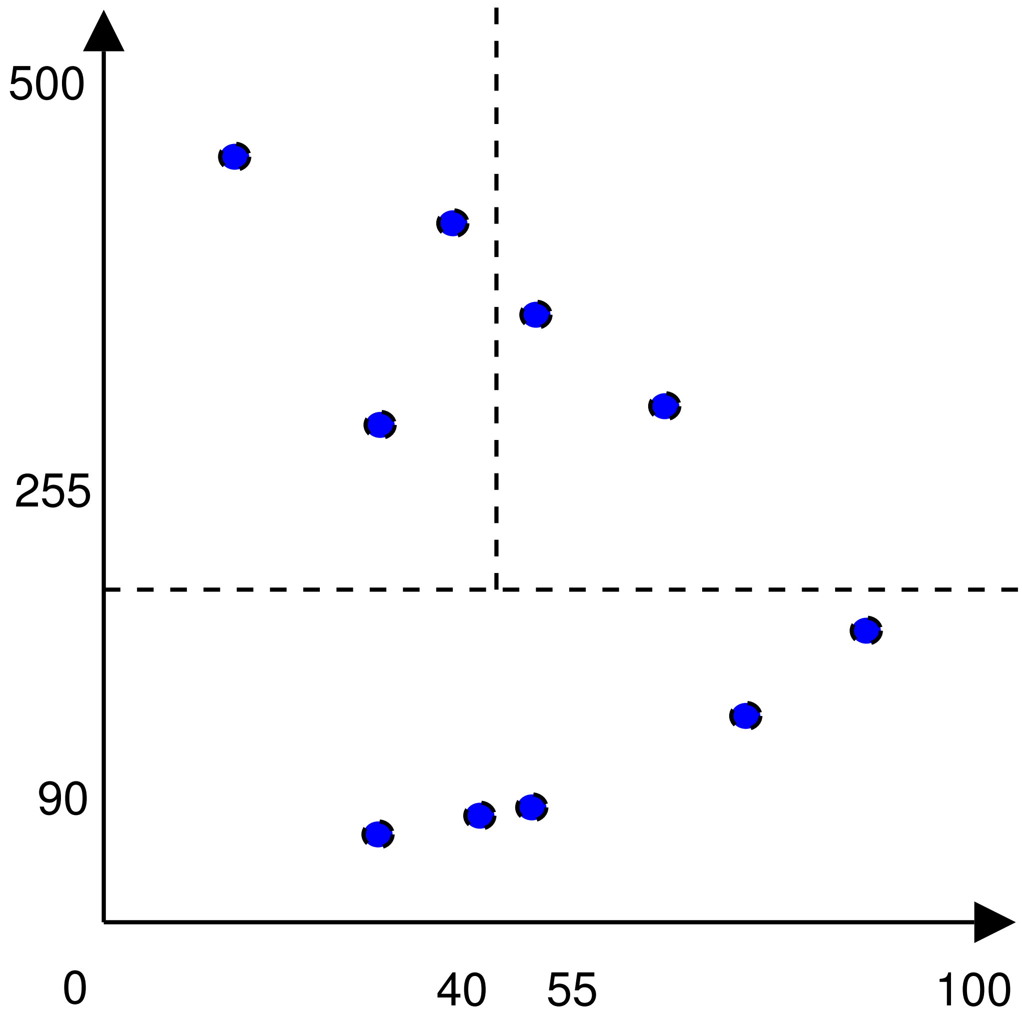
Multidimensional Indexes

kd-Trees



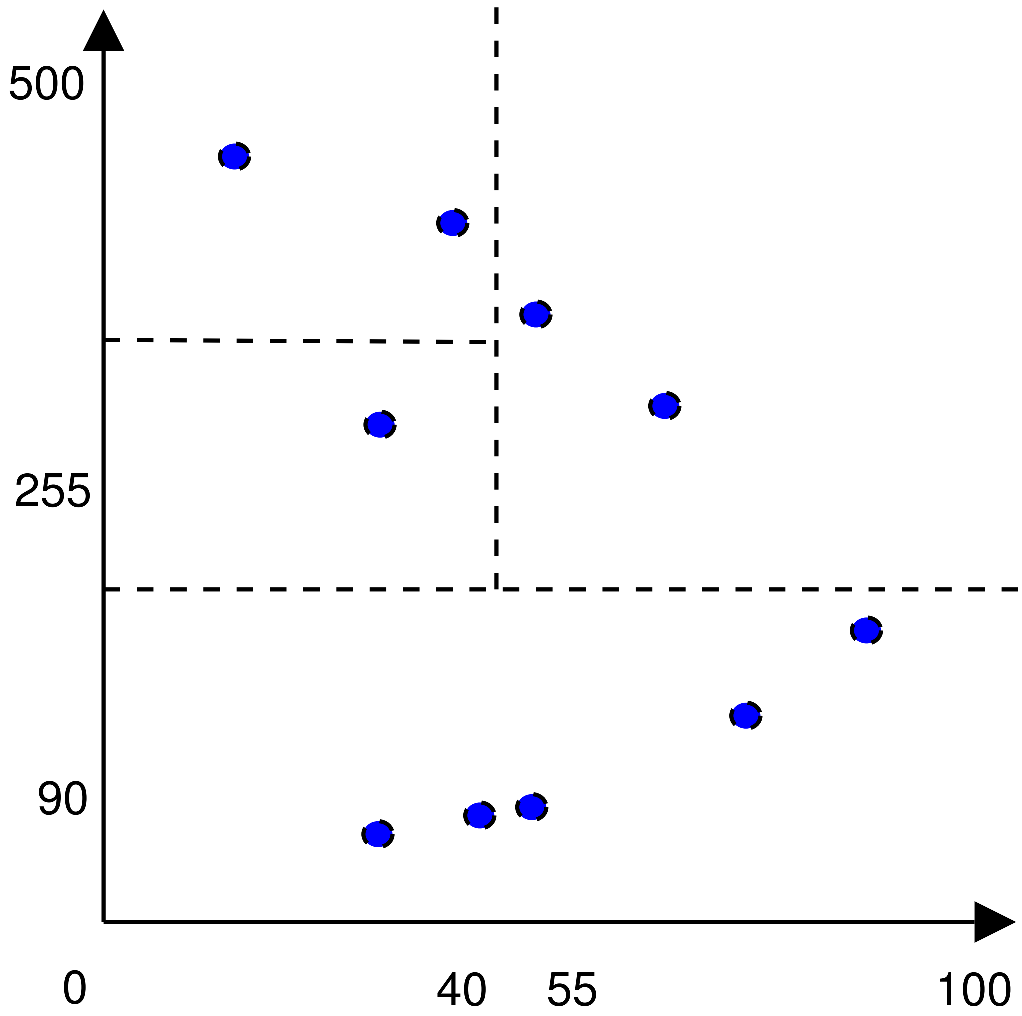
Multidimensional Indexes

kd-**Trees**



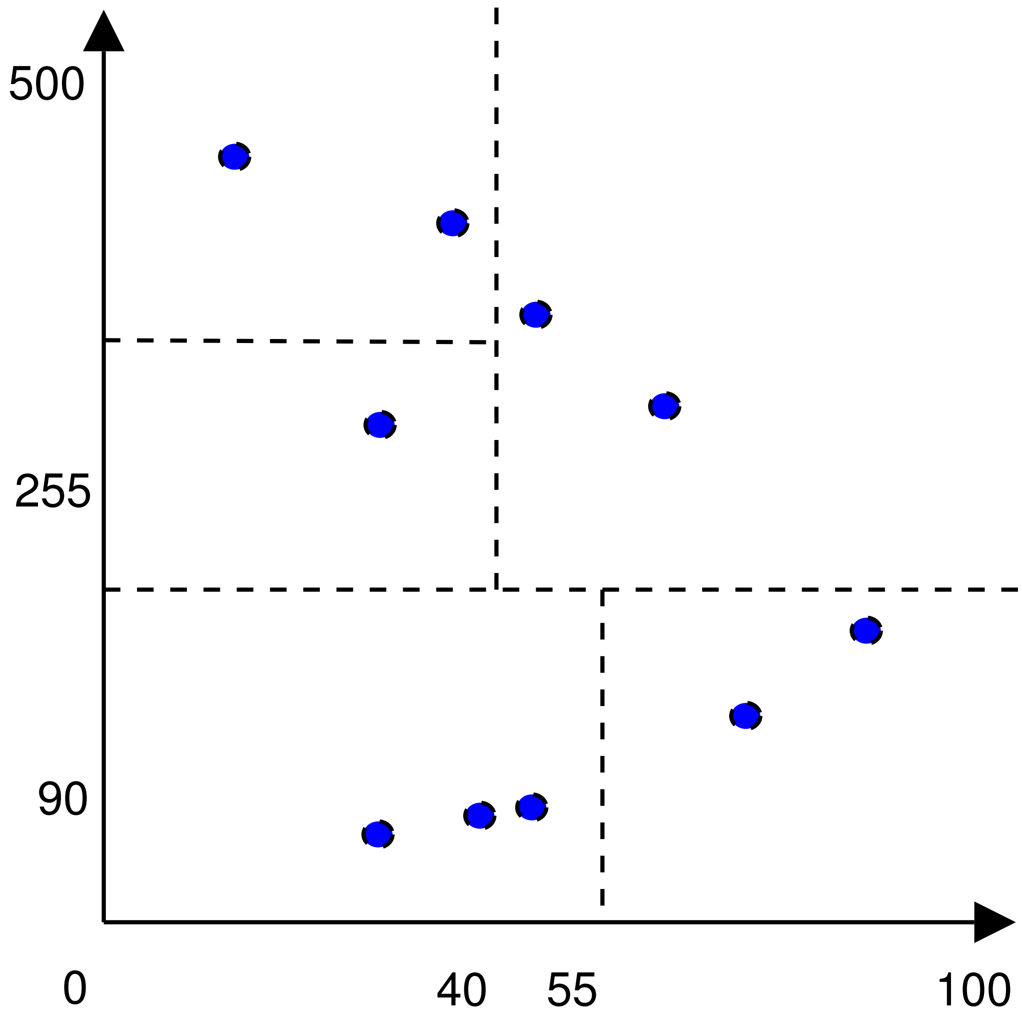
Multidimensional Indexes

kd-Trees



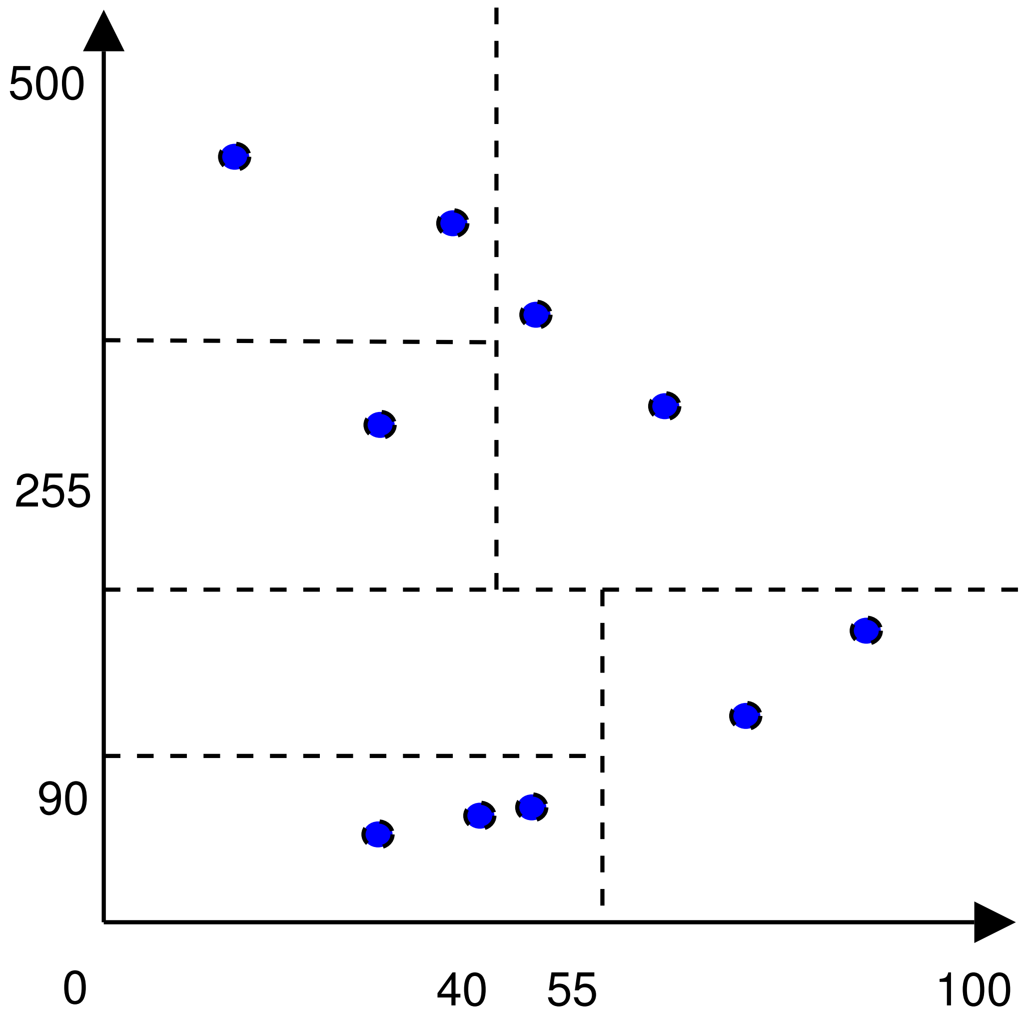
Multidimensional Indexes

kd-Trees



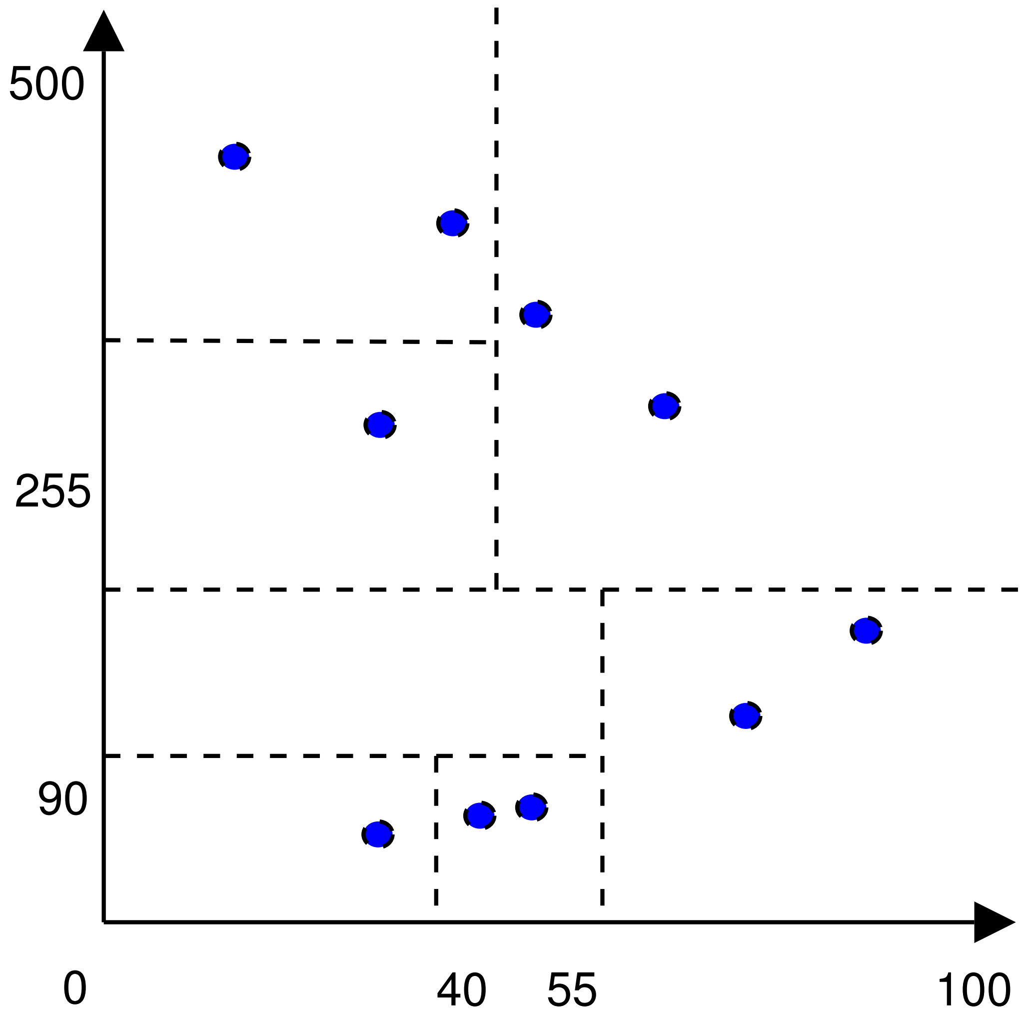
Multidimensional Indexes

kd-**Trees**



Multidimensional Indexes

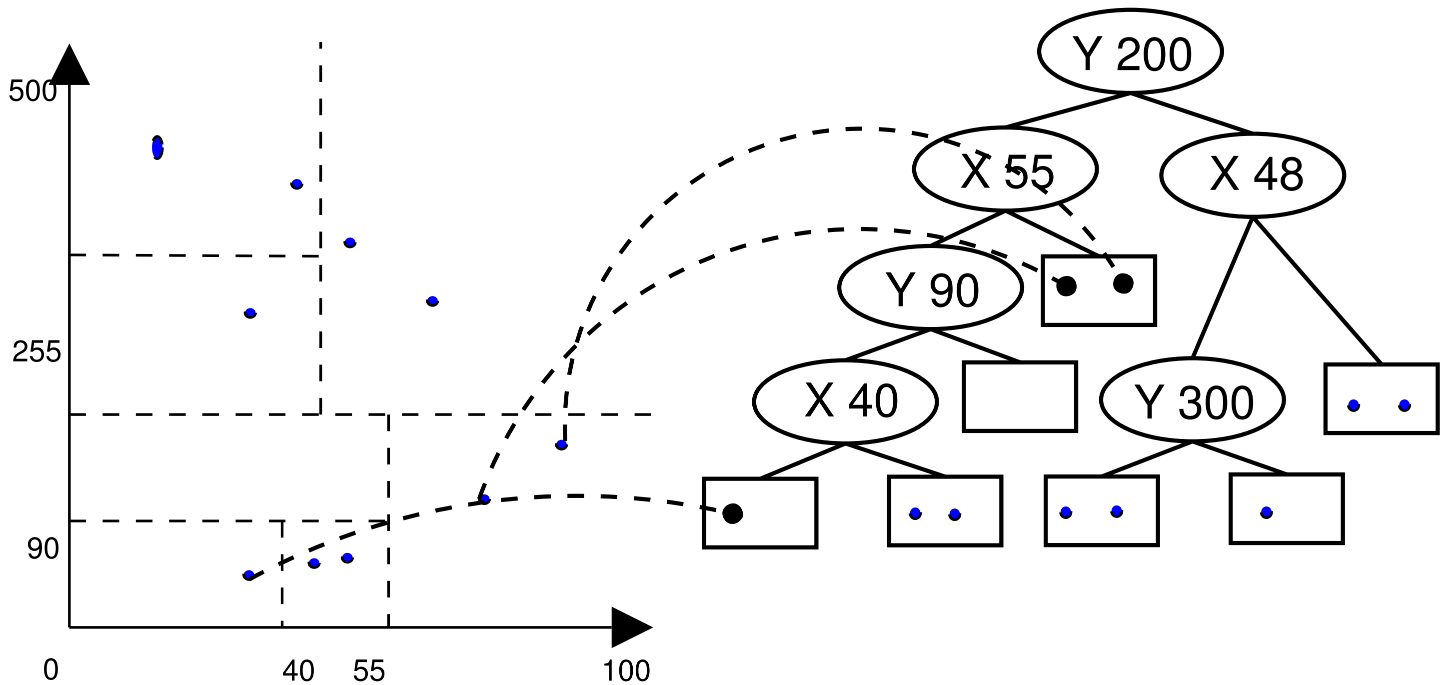
kd-Trees



Multidimensional Indexes

kd-Trees

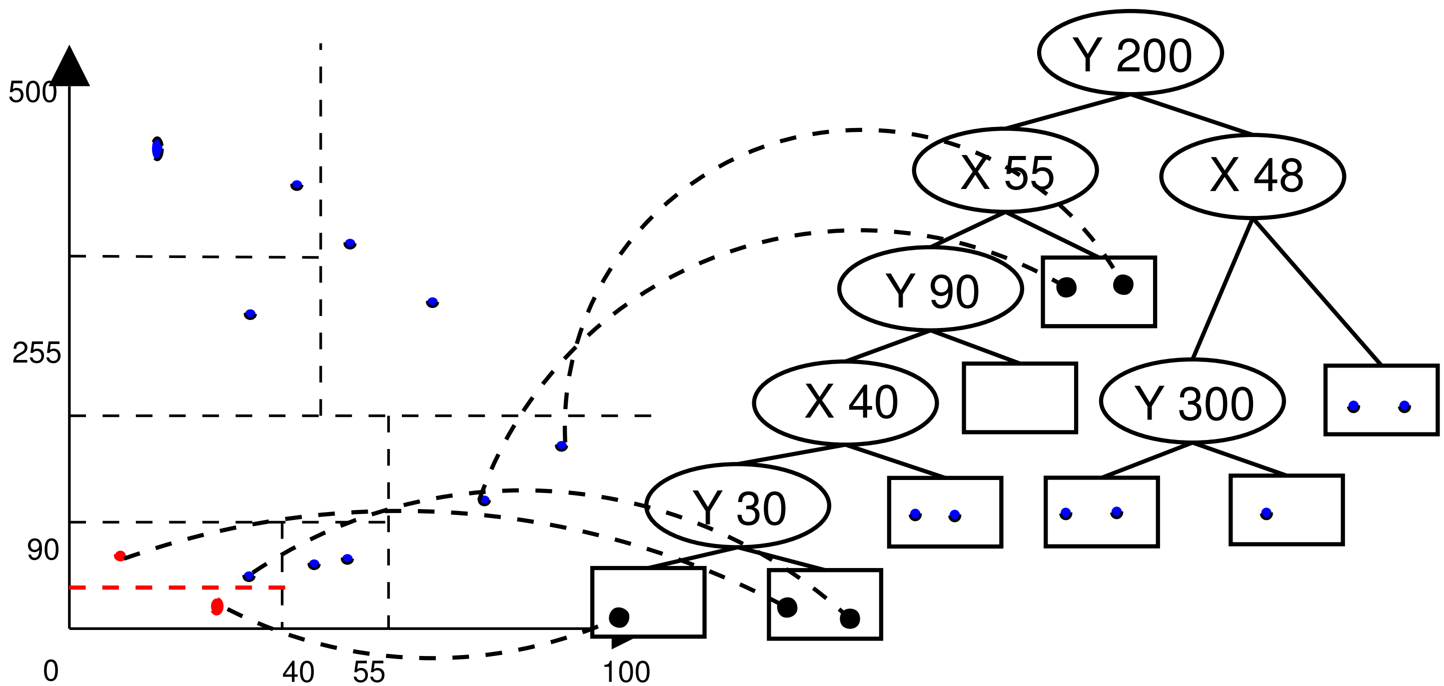
We can look at this as a tree as follows:



Multidimensional Indexes

kd-Trees

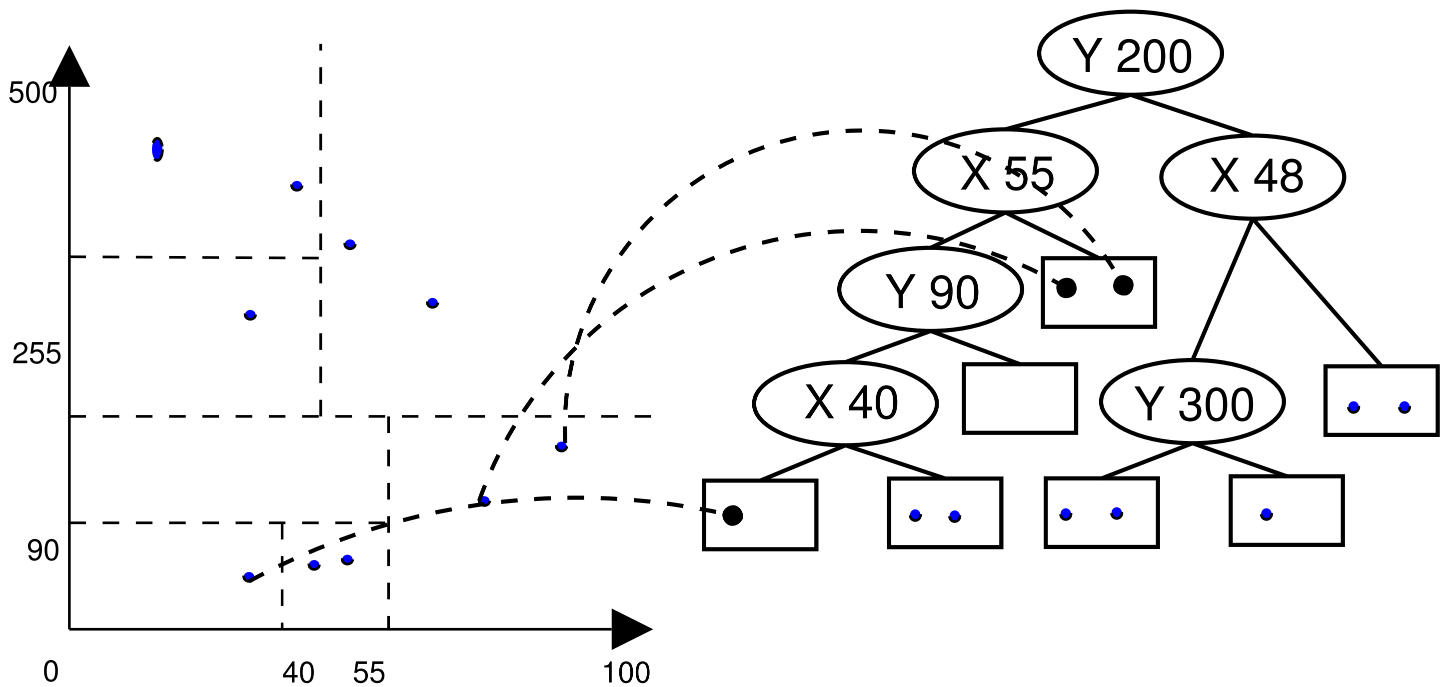
We continue splitting after new insertions:



Multidimensional Indexes

kd-**Trees**

- Good support for point queries
- Good support for partial match queries: e.g., $(40 \leq x \leq 45)$
- Good support for range queries $(40 \leq x \leq 45 \wedge y < 80)$
- Reasonable support for nearest neighbour



Multidimensional Indexes

kd-Trees for secondary storage

- Generalization to n children for each interal node (cf. BTree).

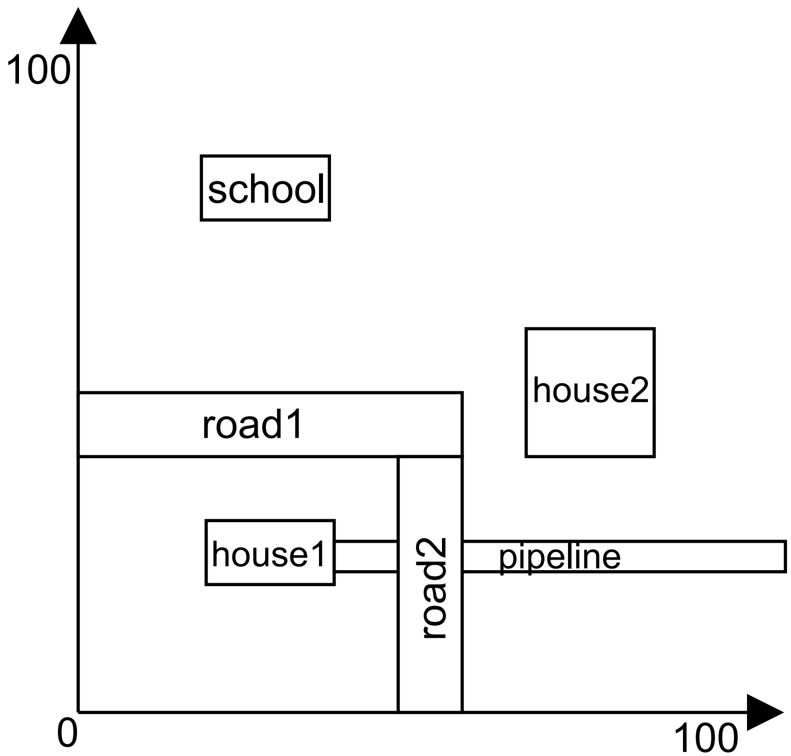
  But it is difficult to keep this tree balanced since we cannot merge the children

- We limit ourselves to two children per node (as before), but store multiple nodes in a single block.

Multidimensional Indexes

R-Trees: generalization of BTrees

Designed to index regions (where a single point is also viewed as a region). Assume that the following regions fit on a single block:

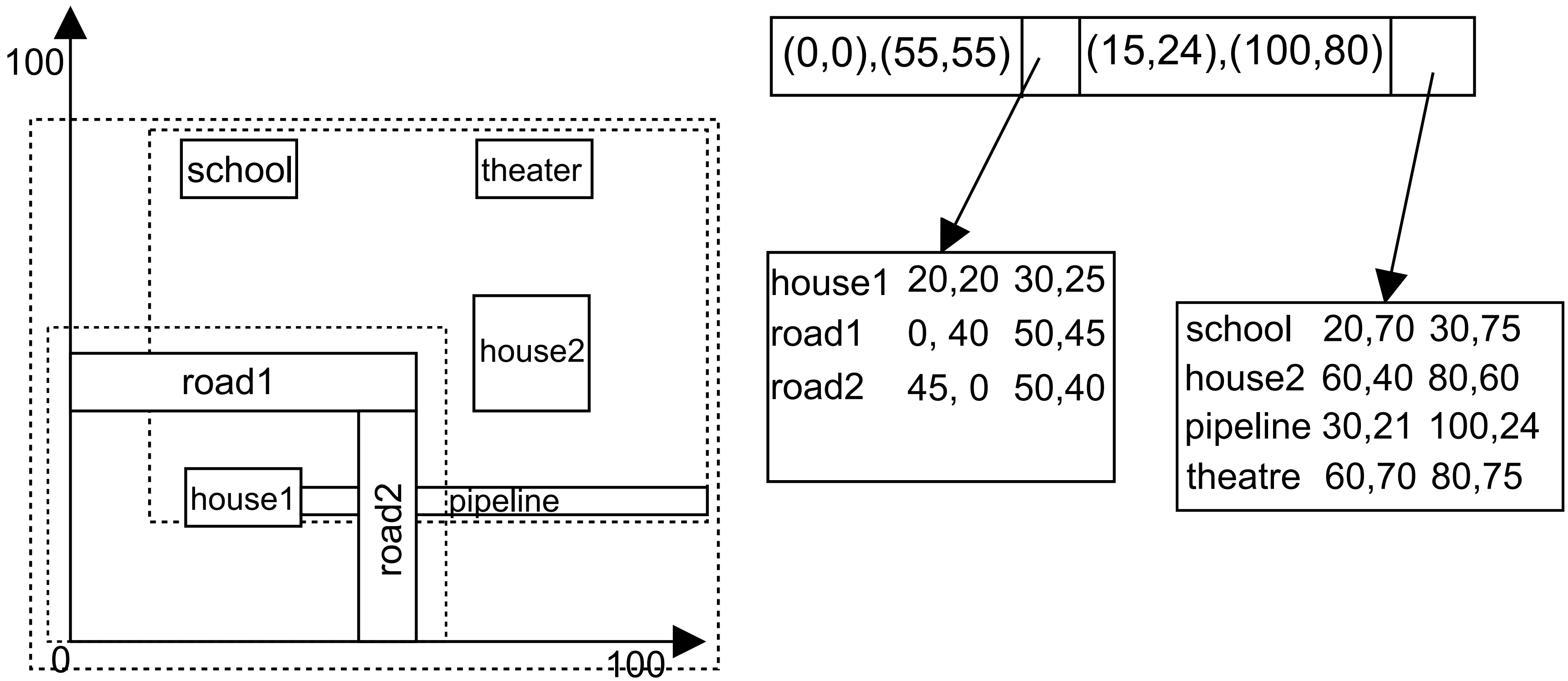


house1	20,20	30,25
road1	0, 40	50,45
road2	45, 0	50,40
school	20,70	30,75
house2	60,40	80,60
pipeline	30,21	100,24

Multidimensional Indexes

R-Trees: generalization of BTrees

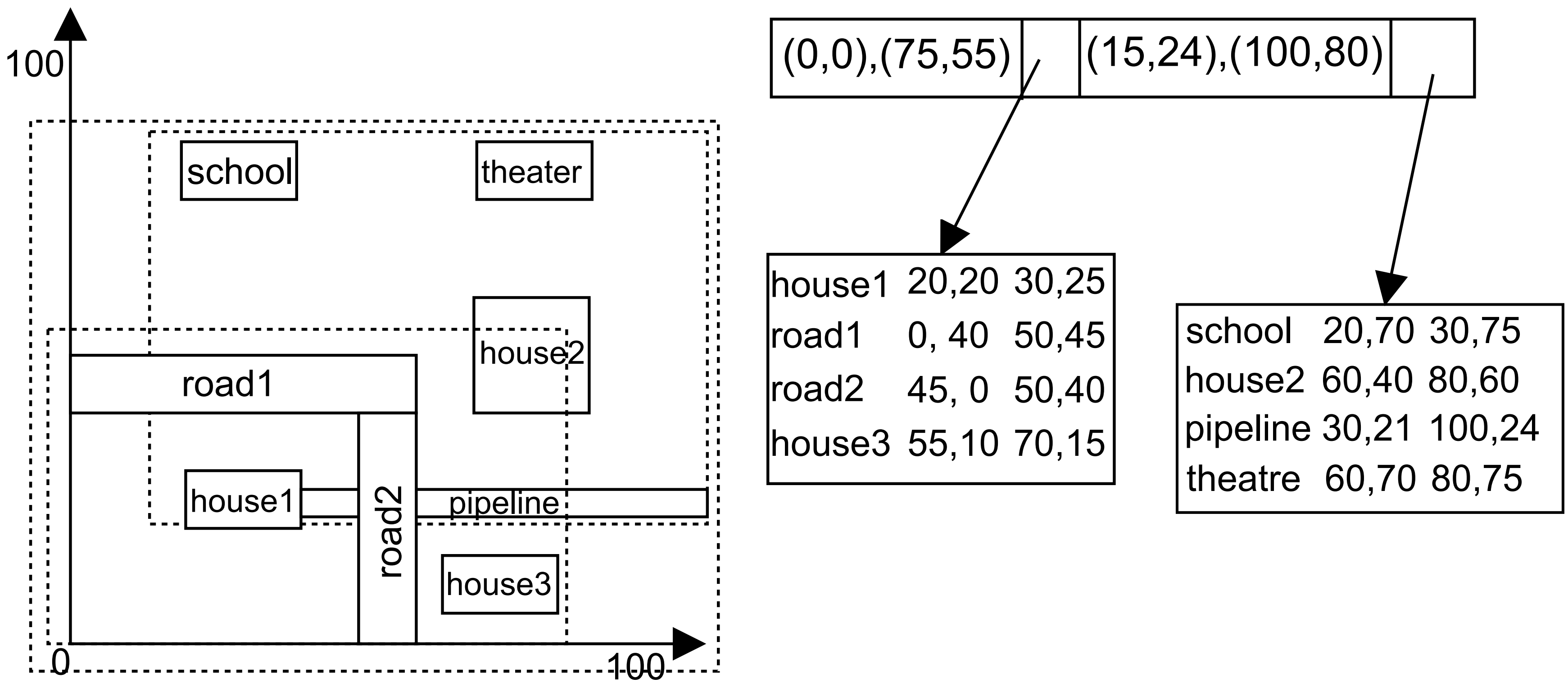
A new region is inserted and we need to split the block into two. We create a tree structure:



Multidimensional Indexes

R-Trees: generalization of BTrees

Inserting again can be done by extending the “bounding regions”:



Multidimensional Indexes

R-Trees: generalization of BTrees

- Ideal for "where-am-I" queries
- Ideal for finding intersecting regions

  e.g., when a user highlights an area of interest on a map
- Reasonable support for point queries
- Good support for partial match queries: e.g., $(40 \leq x \leq 45)$
- Good support for range queries
- Reasonable support for nearest neighbour
- Is balanced
- Often used in practice