OLTP Compared With OLAP

- On Line Transaction Processing – *OLTP*
  - Maintains a database that is an accurate model of some real-world enterprise. Supports day-to-day operations. Characteristics:
    - Short simple transactions
    - Relatively frequent updates
    - Transactions access only a small fraction of the database
- On Line Analytic Processing – *OLAP*
  - Uses information in database to guide strategic decisions. Characteristics:
    - Complex queries
    - Infrequent updates
    - Transactions access a large fraction of the database
    - Data need not be up-to-date

OLAP: Traditional Compared with Newer Applications

- Traditional OLAP queries
  - Uses data the enterprise gathers in its usual activities, perhaps in its OLTP system
  - Queries are ad hoc, perhaps designed and carried out by non-professionals (managers)
- Newer Applications (e.g., Internet companies)
  - Enterprise actively gathers data it wants, perhaps purchasing it
  - Queries are sophisticated, designed by professionals, and used in more sophisticated ways

Data Mining

- *Data Mining* is an attempt at knowledge discovery – to extract knowledge from a database
- Comparison with OLAP
  - *OLAP*:
    - What percentage of people who make over $50,000 defaulted on their mortgage in the year 2000?
  - *Data Mining*:
    - How can information about salary, net worth, and other historical data be used to *predict* who will default on their mortgage?

Data Warehouses

- OLAP and data mining databases are frequently stored on special servers called *data warehouses*:
  - Can accommodate the huge amount of data generated by OLTP systems
  - Allow OLAP queries and data mining to be run off-line so as not to impact the performance of OLTP

Fact Tables

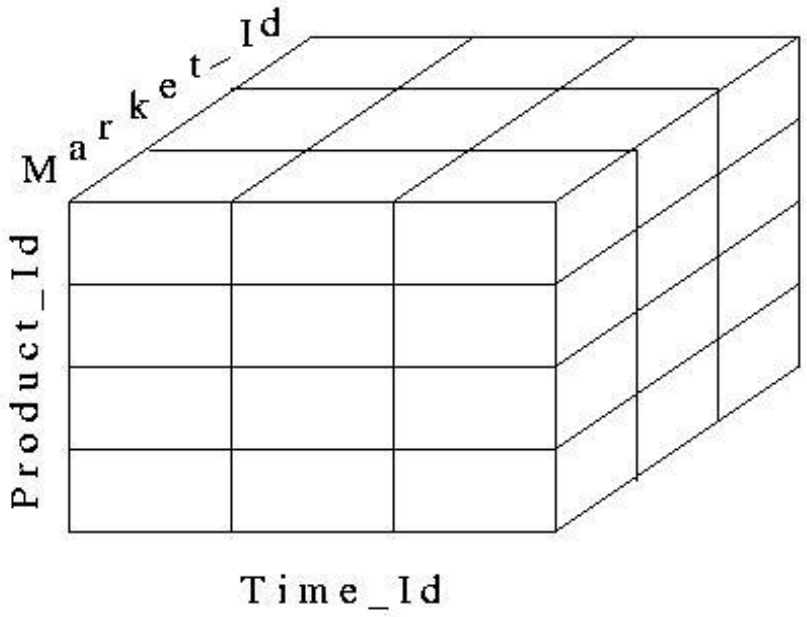
- Many OLAP applications are based on a *fact table*
- For example, a supermarket application might be based on a table

  Sales (*Market_Id*, *Product_Id*, *Time_Id*, *Sales_Amt*)

- The table can be viewed as *multidimensional*
  - *Market_Id*, *Product_Id*, *Time_Id* are the dimensions that represent specific supermarkets, products, and time intervals
  - *Sales_Amt* is a function of the other three

A Data Cube

- Fact tables can be viewed as an N-dimensional *data cube* (3-dimensional in our example)
  - The entries in the cube are the values for *Sales_Amts*



Dimension Tables

- The dimensions of the fact table are further described with *dimension tables*
- Fact table:

  Sales (*Market_id*, *Product_Id*, *Time_Id*, *Sales_Amt*)
- Dimension Tables:

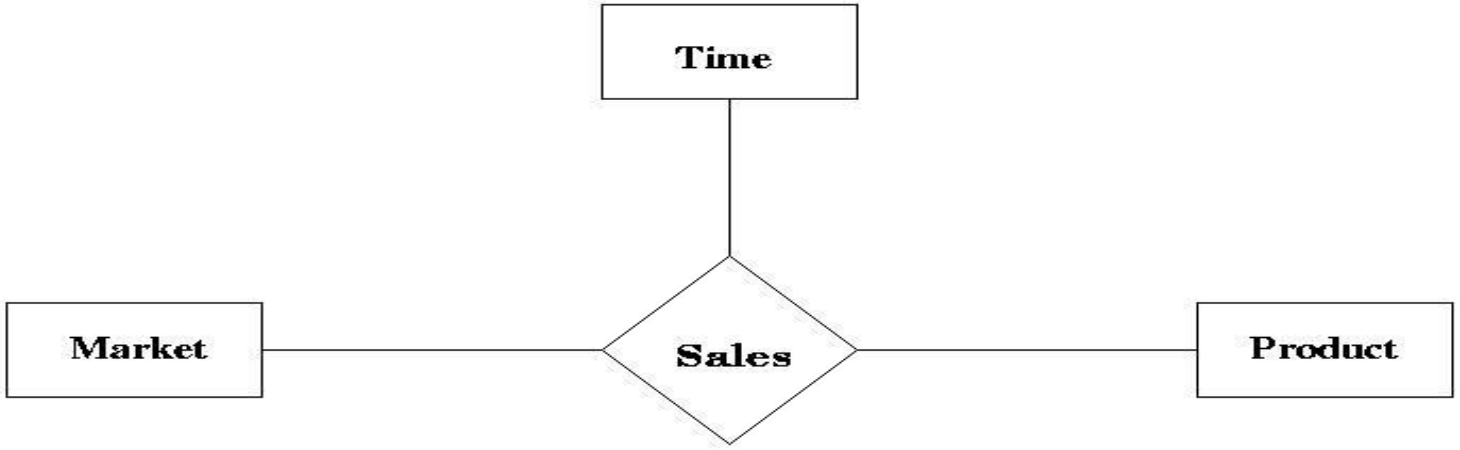
  Market (*Market_Id*, *City*, *State*, *Region*)

  Product (*Product_Id*, *Name*, *Category*, *Price*)

  Time (*Time_Id*, *Week*, *Month*, *Quarter*)

Star Schema

- The fact and dimension relations can be displayed in an E-R diagram, which looks like a star and is called a *star schema*



Aggregation

- Many OLAP queries involve *aggregation* of the data in the fact table
- For example, to find the total sales (over time) of each product in each market, we might use

```
SELECT    S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM      Sales S
GROUP BY  S.Market_Id, S.Product_Id
```

- The aggregation is over the entire time dimension and thus produces a two-dimensional view of the data. (Note: aggregation here is over time, not supermarkets or products.)

Aggregation over Time

- The output of the previous query

Market_Id

SUM(*Sales_Amt*)	M1	M2	M3	M4
P1	3003	1503	…	
P2	6003	2402	…	
P3	4503	3	…	
P4	7503	7000	…	
P5	…	…	…	

Product_Id

Drilling Down and Rolling Up

- Some dimension tables form an *aggregation hierarchy*

  Market_Id City State Region

- Executing a series of queries that moves down a hierarchy (*e.g.*, from aggregation over regions to that over states) is called *drilling down*
  - Requires the use of a dimension table or information more specific than the requested aggregation (*e.g.*, cities)
- Executing a series of queries that moves up the hierarchy (e.g., from states to regions) is called *rolling up*
  - Note: In a rollup, coarser aggregations can be computed using prior queries for finer aggregations

Drilling Down

- Drilling down on market: from *Region* to *State*

  Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)

  Market (*Market_Id, City, State, Region*)

1.
```
SELECT    S.Product_Id, M.Region, SUM (S.Sales_Amt)
FROM      Sales S, Market M
WHERE     M.Market_Id = S.Market_Id
GROUP BY  S.Product_Id, M.Region
```

2.
```
SELECT    S.Product_Id, M.State, SUM (S.Sales_Amt)
FROM      Sales S, Market M
WHERE     M.Market_Id = S.Market_Id
GROUP BY  S.Product_Id, M.State,
```

Rolling Up

- Rolling up on market, from *State* to *Region*
  - If we have already created a table, State_Sales, using

```
1.  SELECT   S.Product_Id, M.State, SUM (S.Sales_Amt)
    FROM     Sales S, Market M
    WHERE    M.Market_Id = S.Market_Id
    GROUP BY S.Product_Id, M.State
```

then we can roll up from there to:

```
2.  SELECT   T.Product_Id, M.Region, SUM (T.Sales_Amt)
    FROM     State_Sales T, Market M
    WHERE    M.State = T.State
    GROUP BY T.Product_Id, M.Region
```

Can reuse the results of query 1.

Pivoting

- When we view the data as a multi-dimensional cube and group on a subset of the axes, we are said to be performing a *pivot* on those axes
  - Pivoting on dimensions $D_1,\ldots,D_k$ in a data cube $D_1,\ldots,D_k,D_{k+1},\ldots,D_n$ means that we use GROUP BY $A_1,\ldots,A_k$ and aggregate over $A_{k+1},\ldots A_n$, where A_i is an attribute of the dimension D_i
  - *Example*: Pivoting on Product and Time corresponds to grouping on *Product_id* and *Quarter* and aggregating *Sales_Amt* over *Market_id:*

```
SELECT    S.Product_Id, T.Quarter, SUM (S.Sales_Amt)
FROM      Sales S, Time T
WHERE     T.Time_Id = S.Time_Id
GROUP BY  S.Product_Id, T.Quarter
```

Pivot

Slicing-and-Dicing

- When we use WHERE to specify a particular value for an axis (or several axes), we are performing a *slice*
  - Slicing the data cube in the Time dimension (choosing sales only in week 12) then pivoting to *Product_id* (aggregating over *Market_id*)

```
SELECT   S.Product_Id,  SUM (Sales_Amt)
FROM     Sales S, Time T
WHERE    T.Time_Id = S.Time_Id  AND  T.Week = „W k-12“
GROUP BY  S. Product_Id
```

Slice

Pivot

Slicing-and-Dicing

- Typically slicing and dicing involves several queries to find the "right slice."

  For instance, change the slice & the axes (from the prev. example):

  - Slicing on Time and Market dimensions then pivoting to *Product_id* and *Week* (in the time dimension)

```
SELECT    S.Product_Id, T.Quarter, SUM (Sales_Amt)
FROM      Sales S, Time T
WHERE     T.Time_Id = S.Time_Id
              AND T.Quarter = 4
              AND S.Market_id = 12345
GROUP BY  S.Product_Id, T.Week
```

Slice

Pivot

The CUBE Operator

- To construct the following table, would take 4 queries (next slide)

Market_Id

SUM(*Sales_Amt*)	M1	M2	M3	*Total*
P1	3003	1503	…	…
P2	6003	2402	…	…
P3	4503	3	…	…
P4	7503	7000	…	…
Total	…	…	…	…

Product_Id

The Queries

- For the table entries, without the totals (aggregation on time)

```
SELECT    S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM      Sales S
GROUP BY  S.Market_Id, S.Product_Id
```

- For the row totals (aggregation on time and markets)

```
SELECT    S.Product_Id, SUM (S.Sales_Amt)
FROM      Sales S
GROUP BY  S.Product_Id
```

- For the column totals (aggregation on time and products)

```
SELECT    S.Market_Id, SUM (S.Sales)
FROM      Sales S
GROUP BY  S.Market_Id
```

- For the grand total (aggregation on time, markets, and products)

```
SELECT    SUM (S.Sales)
FROM      Sales S
```

Definition of the CUBE Operator

- Doing these queries is wasteful
  - The first does much of the work of the other two: if we could save that result and aggregate over *Market_Id* and *Product_Id*, we could compute the other queries more efficiently
- The CUBE clause is part of SQL:1999
  - GROUP BY CUBE (v1, v2, ... , vn)
  - Equivalent to a collection of GROUP BYs, one for each of the 2^n subsets of v1, v2, ... , vn

Example of CUBE Operator

- The following query returns all the information needed to make the previous products/markets table:

```
SELECT S.Market_Id, S.Product_Id, SUM (S.Sales_Amt)
FROM Sales S
GROUP BY CUBE (S.Market_Id, S.Product_Id)
```

ROLLUP

- ROLLUP is similar to CUBE except that instead of aggregating over all subsets of the arguments, it creates subsets moving from right to left
- GROUP BY ROLLUP (A_1,A_2,…,A_n) is a series of these aggregations:
  - GROUP BY A_1 ,… , A_{n-1} ,A_n
  - GROUP BY A_1 ,… , A_{n-1}
  - … … …
  - GROUP BY A_1, A_2
  - GROUP BY A_1
  - *No* GROUP BY
- ROLLUP is also in SQL:1999

Example of ROLLUP Operator

SELECT S.*Market_Id*, S.*Product_Id*, SUM (S.*Sales_Amt*)

FROM Sales S

GROUP BY ROLLUP (S.*Market_Id*, S. *Product_Id*)

- first aggregates with the finest granularity:
  GROUP BY S.*Market_Id*, S.*Product_Id*
- then with the next level of granularity:
  GROUP BY S.*Market_Id*
- then the grand total is computed with *no* GROUP BY clause

Materialized Views

The CUBE operator is often used to precompute aggregations on all dimensions of a fact table and then save them as a *materialized views* to speed up future queries

Aggregate Maintenance

- The accounting department of a convenience store chain issues queries every twenty minutes to obtain:
  - The total dollar amount on order from a particular vendor
  - The total dollar amount on order by a particular store outlet.

- Original Schema:

  Ordernum(ordernum, itemnum, quantity, purchaser, vendor)

  Item(itemnum, price)

- Ordernum and Item have a clustering index on itemnum

- The total dollar queries are expensive. Can you see why?

Aggregate Maintenance

- Add:
  - VendorOutstanding(vendor, amount), where amount is the dollar value of goods on order to the vendor, with a clustering index on vendor
  - StoreOutstanding(purchaser, amount), where amount is the dollar value of goods on order by the purchaser store, with a clustering index on purchaser.
- Each update to order causes an update to these two redundant tables (triggers can be used to implement this explicitely, materialized views make these updates implicit)
- Trade-off between update overhead and look-up speed-up.

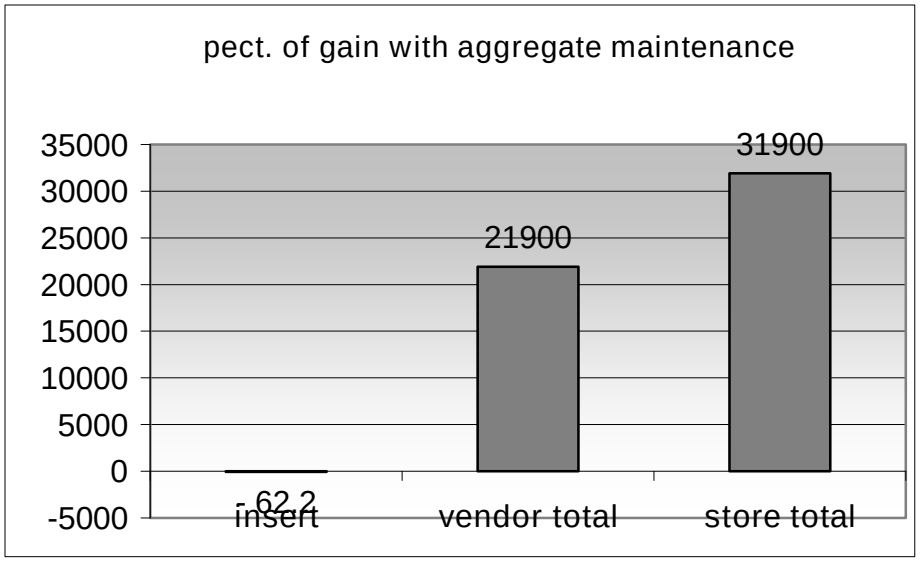
Materialized Views

- Oracle9i and above support materialized views:

```
CREATE MATERIALIZED
VIEW VendorOutstanding
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT orders.vendor,
sum(orders.quantity*item.price)
FROM orders,item
WHERE orders.itemnum =
item.itemnum
group by orders.vendor;
```

- Some Options:
  - BUILD immediate/deferred
  - REFRESH complete/fast
  - ENABLE QUERY REWRITE
- Key characteristics:
  - Transparent aggregate maintenance
  - Transparent expansion performed by the optimizer based on cost.
    - It is the optimizer and not the programmer that performs query rewriting

Aggregate Maintenance



- SQLServer on Windows2000
- accounting department schema and queries
- 1000000 orders, 1000 items
- Using triggers for view maintenance
- On this experiment, the trade-off is largely in favor of aggregate maintenance

Data Mining

- An attempt at knowledge discovery
- Searching for patterns and structure in a sea of data
- Uses techniques from many disciplines, such as statistical analysis and machine learning

Goals of Data Mining

- Association
  - Finding patterns in data that associate instances of that data to related instances
    - Example: what types of books does a customer buy
- Classification
  - Finding patterns in data that can be used to classify that data (and possibly the people it describes)
    - Example "high-end buyers" and "low-end" buyers
  - This classification might then be used for Prediction
    - Which bank customers will default on their mortgages?
  - Categories for classification are known in advance

Goals of Data Mining

- Clustering
  - Finding patterns in data that can be used to classify that data (and possibly the people it describes) into categories determined by a similarity measure
    - Example: Are cancer patients clustered in any geographic area (possibly around certain power plants)?
  - Categories are *not* known in advance, unlike is the classification problem

Associations

- An *association* is a correlation between certain values in a database (in the same or different columns)
  - *In a convenience store in the early evening, a large percentage of customers who bought diapers also bought beer*
- This association can be described using the notation

  Purchase_diapers => Purchase_beer

Confidence and Support

- To determine whether an association exists, the system computes the *confidence* and *support* for that association
- *Confidence* in A => B
  - The percentage of transactions (recorded in the database) that contain B among those that contain A
    - Diapers => Beer:

      The percentage of customers who bought beer among those who bought diapers
- *Support*
  - The percentage of transactions that contain both items among all transactions
    - 100* (customers who bought both Diapers and Beer)/(all customers)

Ascertain an Association

- To ascertain that an association exists, both the confidence and the support must be above a certain threshold
  - Confidence states that there is a high probability, given the data, that someone who purchased diapers also bought beer
  - Support states that the data shows a large percentage of people who purchased both diapers and beer (so that the confidence measure is not an accident)

A Priori Algorithm for Computing Associations

- Based on this observation:
  - If the support for A => B is larger than *T*, then the support for *A* and *B* must separately be larger than *T*
- Find all items whose support is larger than *T*
  - Requires checking *n* items
  - If there are *m* items with support > T (presumably, m<<n), find all pairs of such items whose support is larger than *T*
  - Requires checking *m*(*m-1*) pairs
- If there are *p* pairs with support > T, compute the confidence for each pair
  - Requires checking *p* pairs

Classification

- *Classification* involves finding patterns in data items that can be used to place those items in certain categories.That classification can then be used to predict future outcomes.
  - *A bank might gather data from the application forms of past customers who applied for a mortgage and classify them as defaulters or non-defaulters.*
  - *Then when new customers apply, they might use the information on their application forms to predict whether or not they would default*

Example: Loan Risk Evaluation

- Suppose the bank used only three types of information to do the classification
  - Whether or not the applicant was married
  - Whether or not the applicant had previously defaulted
  - The applicants current income
- The data about previous applicants might be stored in a table called the *training table*

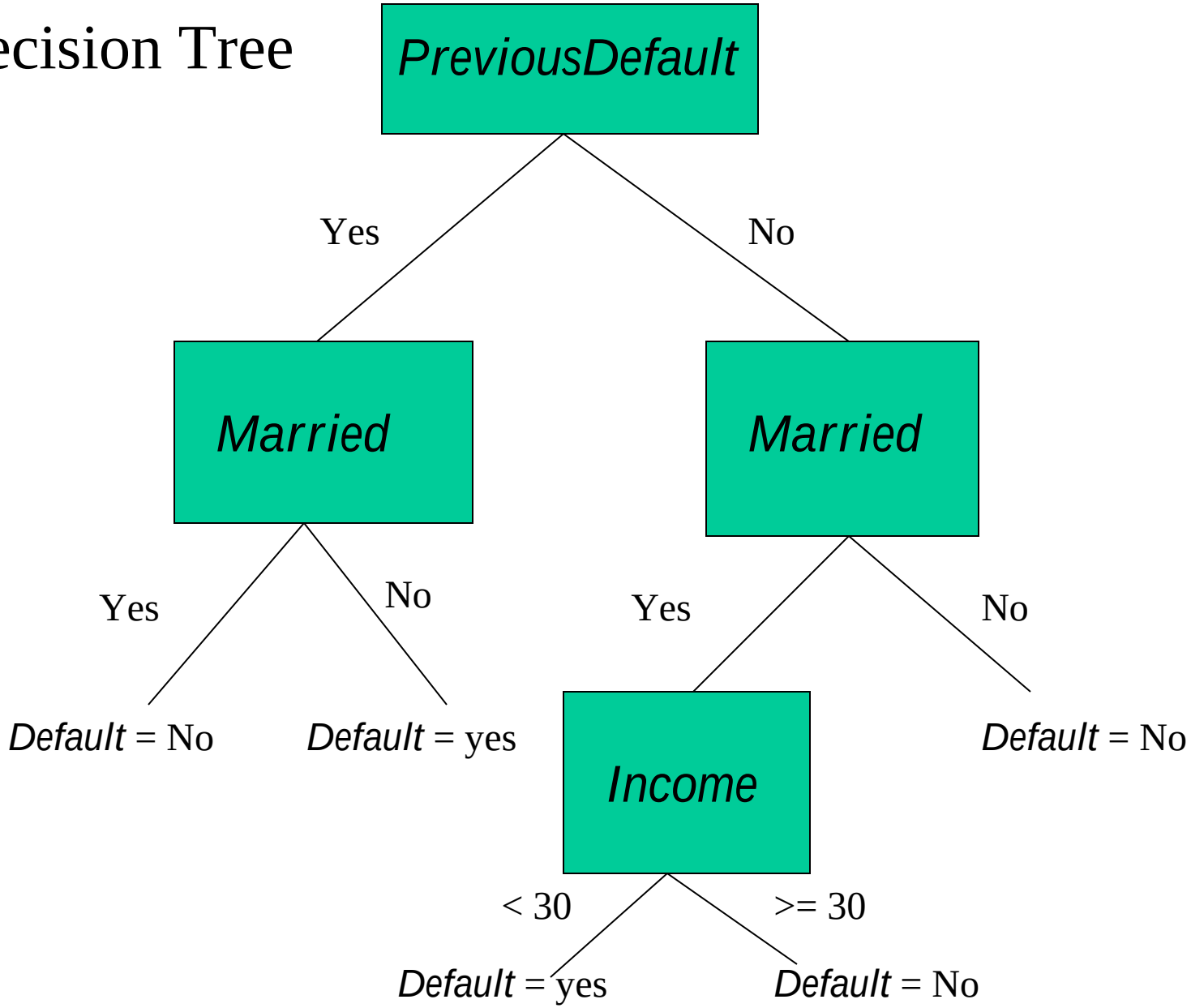
Training Table

Id	*Married*	*PreviousDefault*	*Income*	*Default* (outcome)
C1	Yes	No	50	No
C2	Yes	No	100	No
C3	No	Yes	135	Yes
C4	Yes	No	125	No
C5	Yes	No	50	No
C6	No	No	30	No
C7	Yes	Yes	10	No
C8	Yes	No	10	Yes
C9	Yes	No	75	No
C10	Yes	Yes	45	No

Classification Using Decision Trees

- The goal is to use the information in this table to classify new applicants into defaulters or non defaulters
- One approach is to use the training table to make a decision tree

A Decision Tree



Decision Trees Imply Classification Rules

- Each classification rule implied by the tree corresponds to a path from the root to a leaf
- For example, one such rule is

If

PreviousDefault = No AND *Married* = Yes AND *Income* < 30

Then

Default = Yes

Clustering

- Given:
  - a set of items
  - characteristic attributes for the items
  - a similarity measure based on those attributes
- *Clustering* involves placing those items into *clusters*, such that items in the same cluster are close according to the similarity measure
  - Different from Classification: there the categories are known in advance

Example: Clustering Students by Age

Student Id	Age	GPA
S1	17	3.9
S2	17	3.5
S3	18	3.1
S4	20	3.0
S5	23	3.5
S6	26	2.6

K-Means Algorithm

- To cluster a set of items into *k* categories
  1. Pick *k* items at random to be the (initial) centers of the clusters (so each selected item is in its own cluster)
  2. Place each item *in the training set* in the cluster to which it is closest to the center
  3. Recalculate the centers of each cluster as the mean of the items in that cluster
  4. Repeat the procedure starting at Step 2 until there is no change in the membership of any cluster

The Student Example

- Suppose we want 2 clusters based on *Age*
  - Randomly pick S1 (age 17) and S4 (age 20) as the centers of the initial centers
  - The initial clusters are

    17 17 18 20 23 26
  - The centers of these clusters are

    17.333 and 23
  - Redistribute items among the clusters based on the new centers:

    17 17 18 20 23 26
  - If we repeat the procedure, the clusters remain the same