Data Warehousing

Column versus Row Store

Peter Scheuermann

DW Performance Optimization Overview

- Maintaining Views
- Column Store Model
  - Bitmap Indices
  - Join Indices

Aggregate Use Example

- Consider a Sales fact table with 1 billion rows, with reference to 1000 products and 100 locations
- Consider the query

  SELECT p.category, SUM(s.sales)

  FROM Products p, Sales s
  WHERE p.pid=s.pid

  GROUP BY p.category

- To answer this query, we use 1 billion rows from Sales…

Sales

tid	pid	locid	sales
1	1	1	10
2	1	1	20
3	2	3	40
…	…	…	…

1 billion rows

Aggregate Use Example

- **Pre-compute** a view

- CREATE MATERIALIZED VIEW
  TotalSales (pid, locid, total) AS
  SELECT s.pid, s.locid, SUM(s.sales)
  FROM Sales s
  GROUP BY s.pid, s.locid

TotalSales

pid	locid	sales
1	1	30
2	3	40
…	…	…

100,000 rows

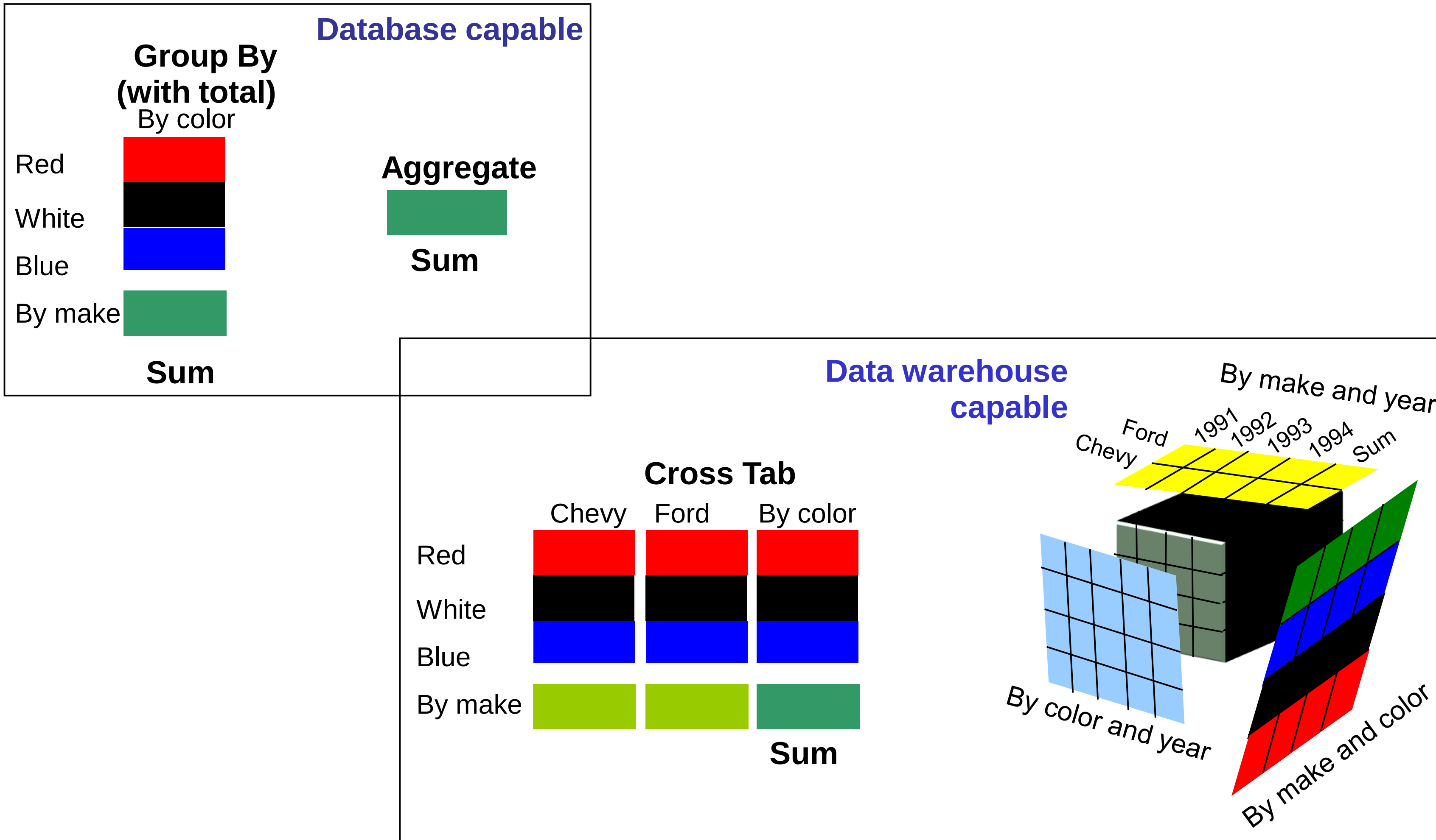
- Rewrite the query using the view:
  - SELECT p.category, SUM(v.total)
    FROM Products p, TotalSales v
    WHERE p.pid=v.pid
    GROUP BY p.category
  - This is 10,000 times faster!

Pre-Aggregation Choices

- **Full** pre-aggregation: (all combinations of levels)
  - Fast query response
  - Takes a lot of space/update time (200-500 times raw data)
- **No** pre-aggregation
  - Slow query response (for terabytes…)
- **Practical** pre-aggregation: chosen combinations
  - A good compromise between response time and space use
  - Supported by (R)OLAP tools
    - IBM DB2
    - Oracle
    - MS Analysis Services

Data Cube

The data cube stores multidimensional GROUP BY relations of tables in data warehouses



A Data Cube Example

1. part, supplier, customer (6M rows)
2. part, customer (6M)
3. part, supplier (0.8M)
4. supplier, customer (6M)
5. part (0.2M)
6. supplier (0.01M)
7. customer (0.1M)
8. none (1)

19 M rows total

8 possible views for 3 dimensions. Each view gives the total sales for that grouping.

Scenario:

A query asks for the sales of a part.

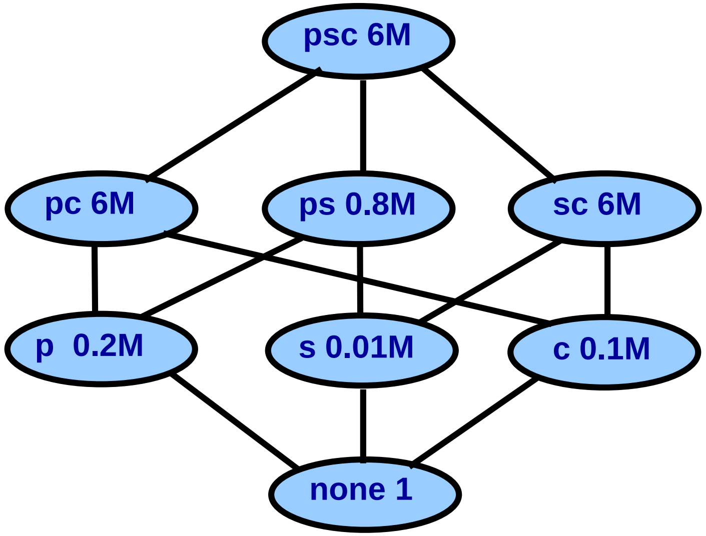
a) **If view pc is available, we need to process about 6M rows**

b) **If view p is available, we only need to process about 0.2M rows**

- Some immediate points:
  - In the example, the views (part, supplier) and (supplier, customer) are not needed – we avoid 12 M rows then (~60%)
  - Picking the right views to materialize will improve performance

Problem: Given that we have space S, what views to materialize for minimizing query costs?

Lattice of views

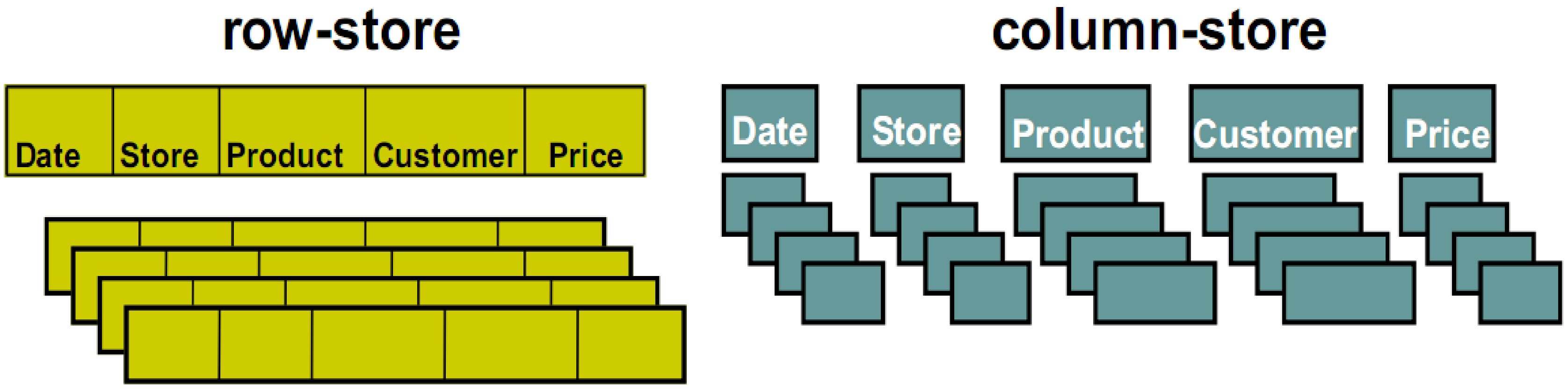


The 8 views from the previous cube example organized into a **Lattice**

To answer a query Q, choose an ancestor of Q, say Q_A, that has been materialized

We then need to process the table for Q_A to answer Q

Row Store and Column Store



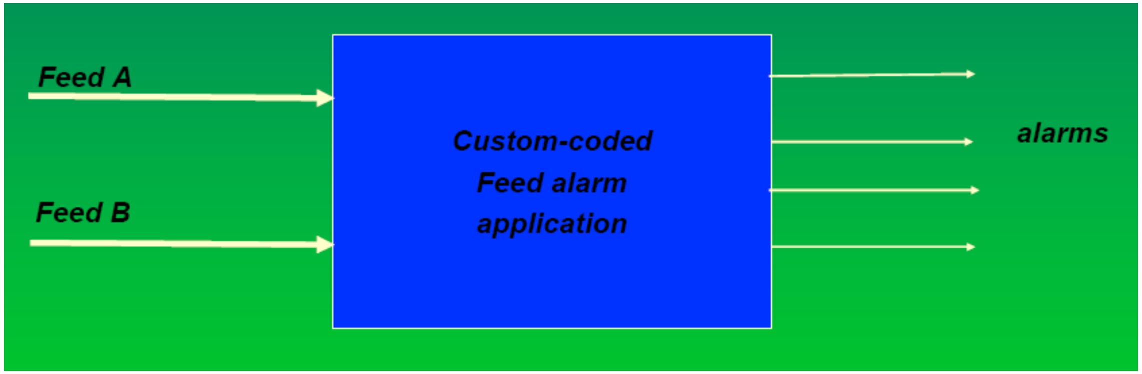
- In row store data are stored on the disk tuple by tuple.
- Where in column store data are stored in the disk column by column

Row Stores Are Write-Optimized

- Can insert and delete a record in one physical write
- Good for on-line transaction processing (OLTP)
- Efficient implementations exist in (almost) all commercial DBMS
- Standardized benchmarks help discovering performance gaps

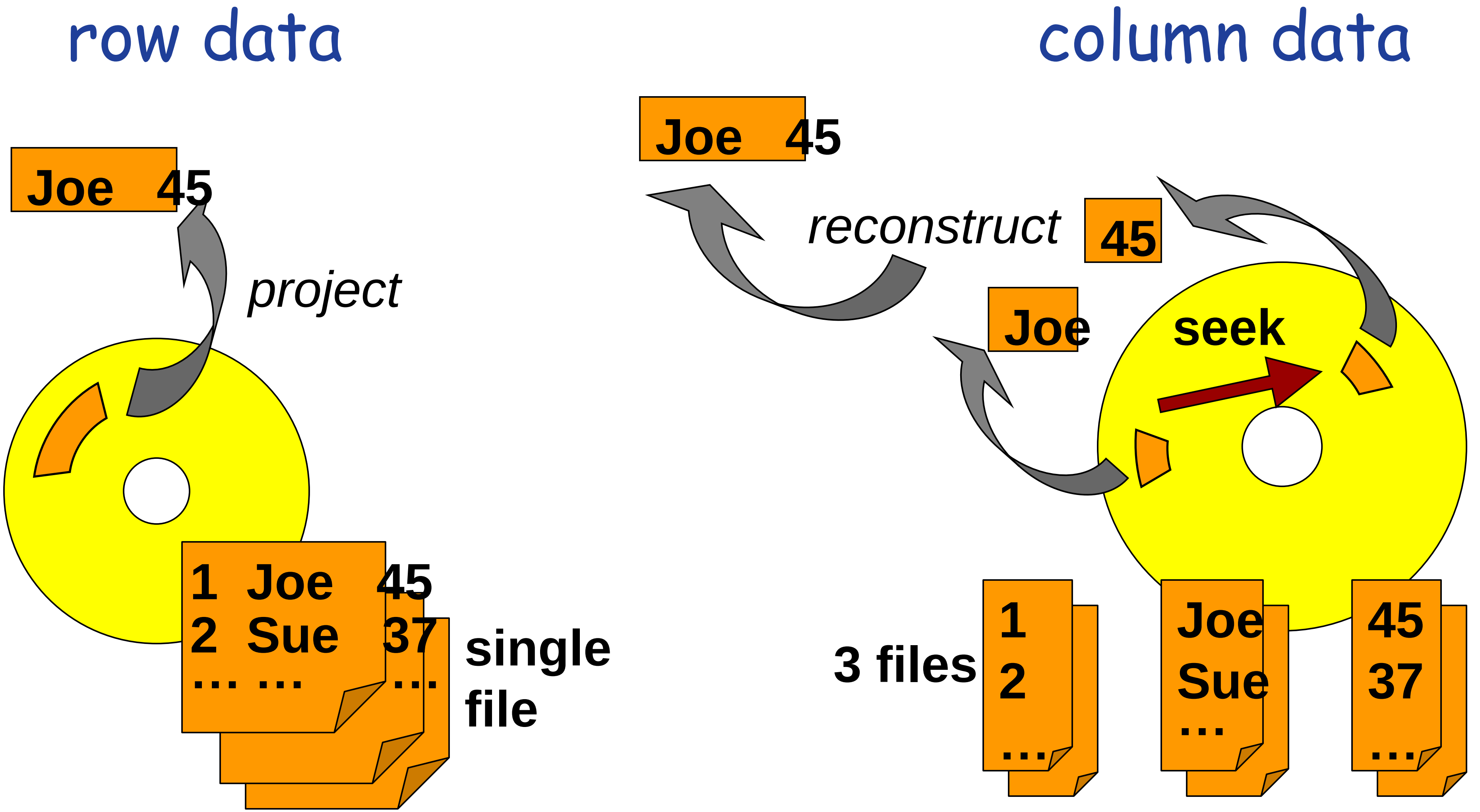
New Applications are often Read-Only

- Data warehouses
- CRM systems
- Text databases
- Streaming data
- Catalog Search
- Sensor networks
- Scientific data



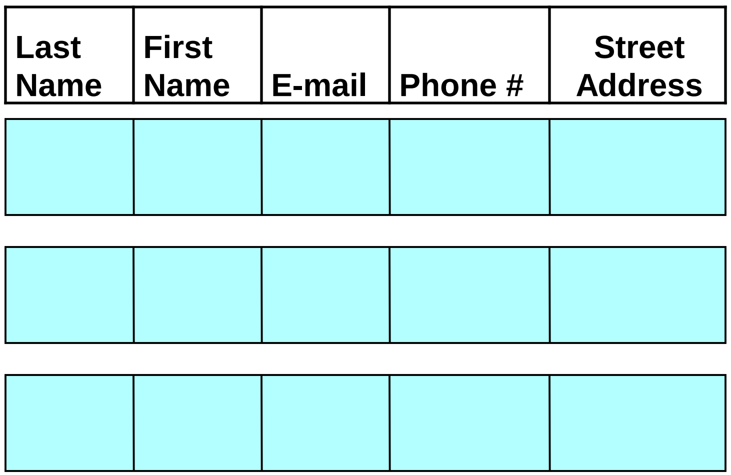
- Ad-hoc queries read 2 columns out of 20
- Column value space much smaller than domain
- In a very large warehouse, fact table is rarely clustered correctly

Rows vs. Columns



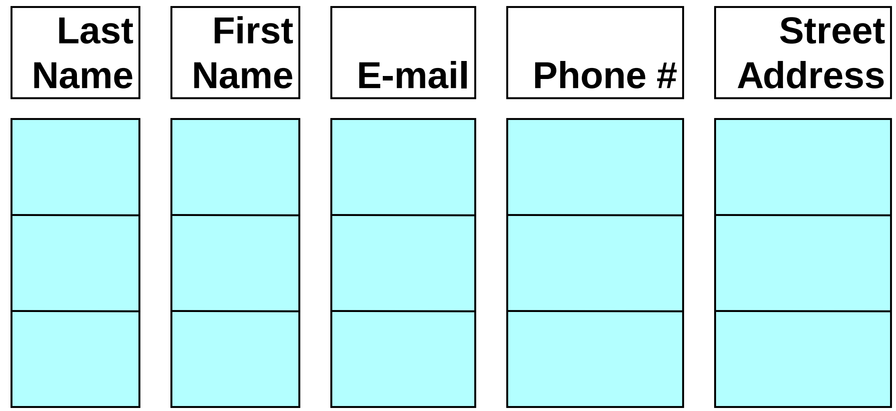
Rows vs. Columns Store

Row Store



+ Easy to add a new record
– Might read unnecessary data

Column Store



+ Fast aggregations (sum, min, max, avg, …), more flexibility for ad-hoc reporting
+ Each column can be compressed individually
– Insert might require multiple seeks

Column Stores

- Really good for read-mostly data-warehouses
  - Lots of column scans and aggregations
  - Writes tend to be in batch

  - Yahoo's world largest data warehouse is a column store

- Often read only 10% of what a row store reads
- This is even more striking when the tables encode a different representation (e.g., RDF)

Vertical Partitioning of Tables

ID	Day	Discount
10	4/4/98	0.195
11	9/4/98	0.065
12	1/2/98	0.175
13	7/2/98	0

Note: tuple identification must be preserved.

OID	ID
100	10
101	11
102	12
103	13
104	14

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98
104	1/2/99

OID	Discount
100	0.195
101	0.065
102	0.175
103	0
104	0.065

Column Stores - Data Model

- To answer queries, projections are joined using

  - surrogate keys
  - join indexes
  - bit arrays (vectors)

Bitmap Indices

- A B^+-tree index stores a list of RowIDs for each value
  - A RowID takes ~8 bytes
  - **Large** space use for columns with **low cardinality** (gender, color)
  - Example: Index for 1 billion rows with gender takes 8 GB
  - Not efficient to do "index intersection" for these columns
- Idea: make a "position bitmap" for each value (only two)
  - Female: 01110010101010…
  - Male: 10001101010101…
  - Takes only (num. of values)*(num. of rows)*1 bit
  - Example: bitmap index on gender (as before) takes only 256 MB
  - **Very** efficient to do "index intersection" (AND/OR) on bitmaps
    - Intersection of 64 bits done in a single CPU instruction (word length=64)

Using Bitmap Indices

- Query example (assume two hair colors, three cities)
  - Find customers in Aalborg with black hair
  - Aalborg: 00000011111
  - Black: 10110110110
  - Result: 00000010110 – use AND, only 3 such customers
- Numeric attributes can also be handled
  - Use the **binning** technique, i.e., group every C values into a bin
    - E.g., group every 5000 values into a bin, and assign a bitmap for it
    - Bitmap for [20000-25000): 001001001
    - Bitmap for [25000-30000): 010010010
  - Find … Salary BETWEEN 22000 AND 29000
    - OR together: 011011011 (Why is it OR instead of AND?)
    - Refinement step: follow those records and check their actual salaries
  - Tradeoff between storage size and index effectiveness

Other applications: Mining Association Rules

- Discovering patterns from a large database (generally a data warehouse) is computationally expensive

- Goal is to find **all** rules of the form $X \rightarrow Y$ that satisfy *minsupport* and *minconf*

- Interpretation: Transactions in the database contain the *items* in X tend also contain the items in Y.

Definitions

- Let *I={i1, i2, …, id}* be set of all items in a market basket data
- Let *T={t1, t2, ..tN}* be the set of all transactions
- A collection of items is termed *itemset*

TID	i1	i2	…	id
t1	1	0		1
t2	1	0		0
…	0	0		1
tN	1	1		1

Definitions

Let X and Y be two disjoint Itemsets (N is the number of transactions)

- Support Count of X $\quad \sigma(X) = |\{t_i \mid X \subseteq t_i, t_i \in T\}|$
- Support of $X \rightarrow Y$ $\quad s(X \rightarrow Y) = \dfrac{\sigma(X \cup Y)}{N}$
- Confidence of $X \rightarrow Y$ $\quad c(X \rightarrow Y) = \dfrac{\sigma(X \cup Y)}{\sigma(X)}$

Mining Association Rules

TID	A	B	C	D	E
100	1	0	1	1	0
200	0	1	1	0	1
300	1	1	1	0	1
400	0	1	0	0	1

Create Bit Vectors

BV_1	BV_2	BV_3	BV_4	BV_5
1	0	1	1	0
0	1	1	0	1
1	1	1	0	1
0	1	0	0	1

Number the attributes

A:1 D:4

B:2 E:5

C:3

Bit Vectors (Column Store)

- This can be seen as a column store
- Read efficient
- For an item or itemset a 64-bit processor can count the support count of 64 rows in one instruction only
- Logical AND, OR

minsupport = 50% (2 transactions)

Is item 1 (column A) frequent ? Yes

BV_1
1
0
1
0

Is the itemset {1, 3} frequent? Yes

BV_1		BV_3		{1,3}
1	$\wedge$	1	=	1
0		1		0
1		1		1
0		0		0