Course code	: **CSE3009**
Course title	: **No SQL Data Bases**
Module	: **5**
Topic	: **1**

Data warehousing schemas

Objectives

This session will give the knowledge about

- Data warehousing schemas
- Comparison of columnar and row-oriented storage,
- Column-store Architectures:
- C-Store and
- Vector-Wise

Data warehousing schemas

When RDBMS released, database workloads were dominated by record-based processing: so-called CRUD operations (Create, Read, Update, Delete)

RDBMS were iterated through entire tables and run in a background batch mode where query response time was not a critical issue.

In late 1980s and 1990s, RDBMS were tasked with supporting analytic and decision support applications that often demanded interactive response times. These systems became known as data warehouses and operated parallel to the OLTP system.

Data warehousing schemas

Separating OLTP(Online Transactional Processing) and OLAP (Online Analytic Processing) workloads was important for maintaining service-level response times for the OLTP systems.

The basic difference between OLTP and OLAP is that OLTP is an online database modifying system, whereas, OLAP is an online database query answering system.

But equally important, the OLAP system demanded a different schema from the OLTP system.

Data warehousing schemas

Star schemas were developed to create data warehouses in which aggregate queries could execute quickly and which would provide a predictable schema for Business Intelligence (BI) tools.

In a star schema, central large fact tables are associated with numerous smaller dimension tables.

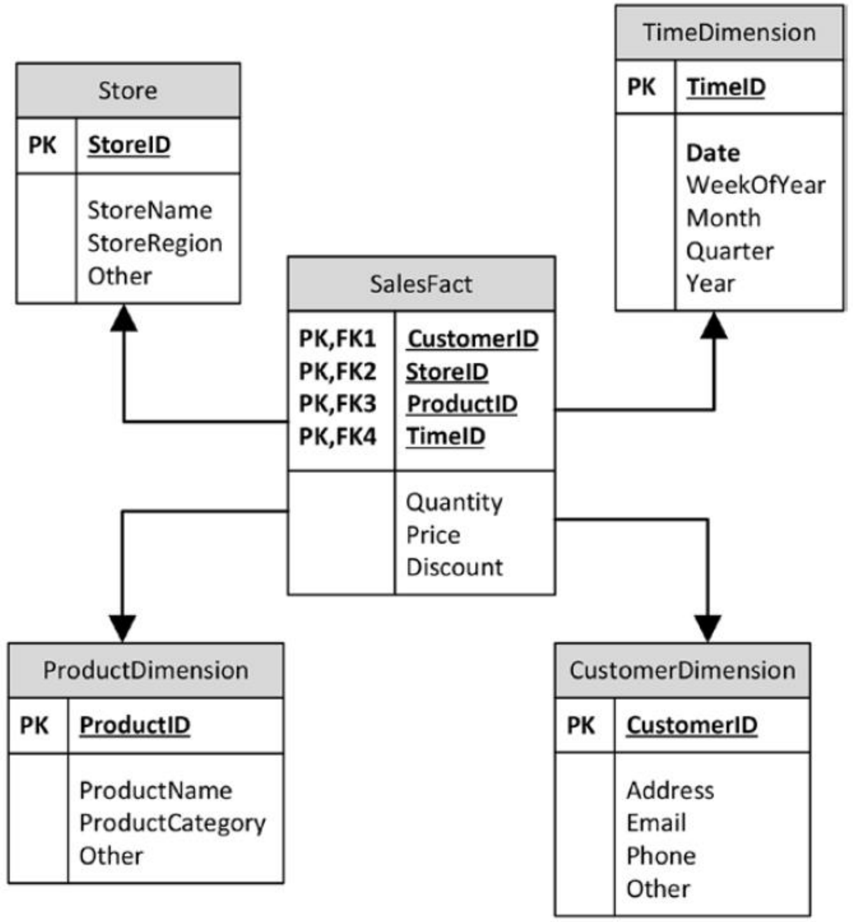
When the dimension tables implement a more complex set of foreign key relationships, then the schema is referred to as a snowflake schema.

Star schemas

Almost all data warehouses adopted some variation on the star schema paradigm, and almost all relational databases adopted indexing and SQL optimization schemes to accelerate queries against star schemas.

These optimizations allowed relational data warehouses to serve as the foundation for management dashboards and popular BI products.

However, despite these optimizations, star schema processing in data warehouses remained severely CPU and IO intensive.



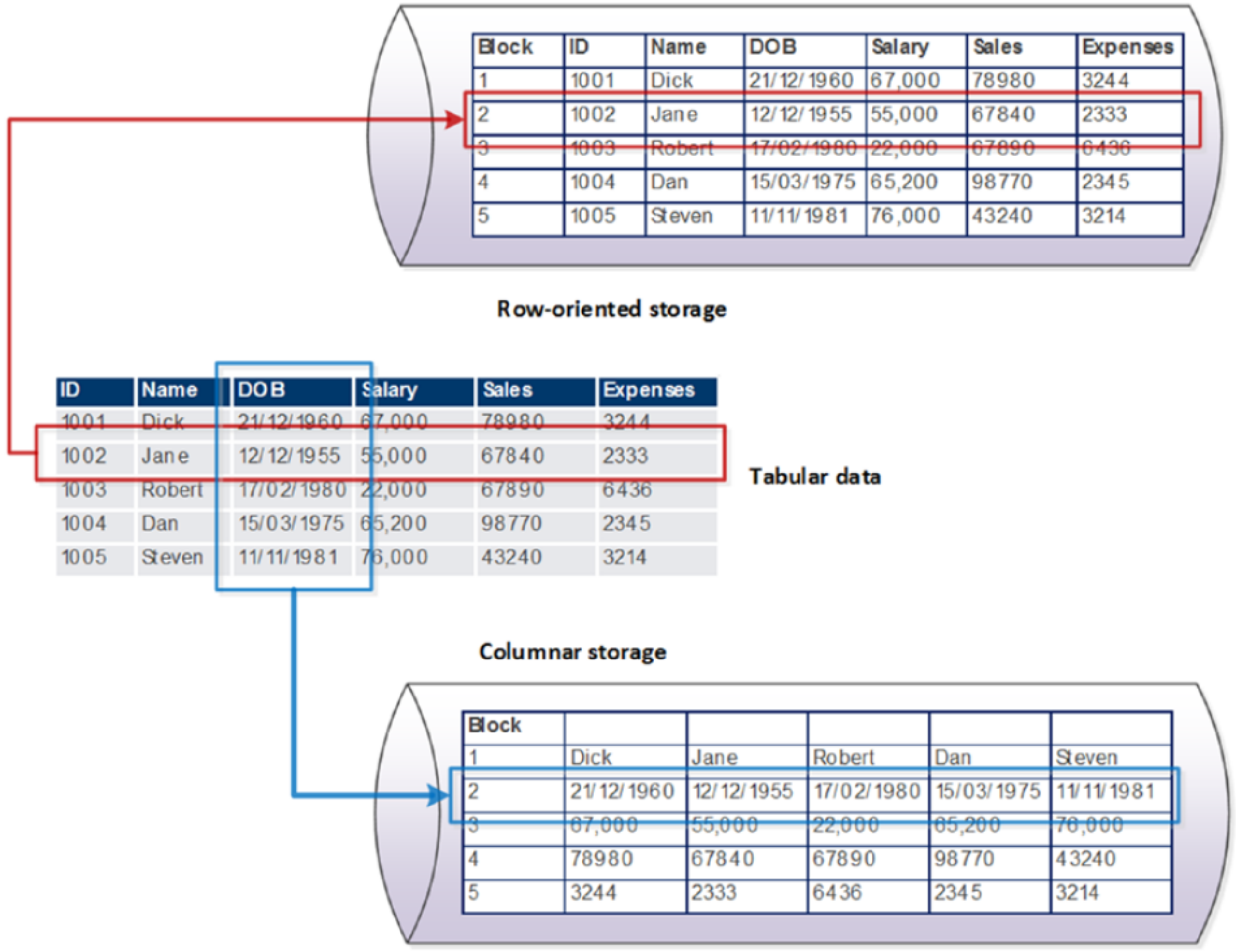
The Columnar Alternative

As data volumes grew and user demands for interactive response times continued, there was increasing restlessness with traditional data warehousing performance.

The idea that it might be better to store data in columnar format dates back to the 1970s, although commercial columnar databases did not appear until the mid-1990s.

Next Fig compares columnar and row-oriented storage for some simple data: in a columnar database, values for a specific column become co-located in the same disk blocks, while in the row-oriented model, all columns for each row are co-located.

The Columnar Alternative



Block	ID	Name	DOB	Salary	Sales	Expenses
1	1001	Dick	21/12/1960	67,000	78980	3244
2	1002	Jane	12/12/1955	55,000	67840	2333
3	1003	Robert	17/02/1980	22,000	67890	6436
4	1004	Dan	15/03/1975	65,200	98770	2345
5	1005	Steven	11/11/1981	76,000	43240	3214

Row-oriented storage

ID	Name	DOB	Salary	Sales	Expenses
1001	Dick	21/12/1960	67,000	78980	3244
1002	Jane	12/12/1955	55,000	67840	2333
1003	Robert	17/02/1980	22,000	67890	6436
1004	Dan	15/03/1975	65,200	98770	2345
1005	Steven	11/11/1981	76,000	43240	3214

Tabular data

Columnar storage

Block					
1	Dick	Jane	Robert	Dan	Steven
2	21/12/1960	12/12/1955	17/02/1980	15/03/1975	11/11/1981
3	67,000	55,000	22,000	65,200	76,000
4	78980	67840	67890	98770	43240
5	3244	2333	6436	2345	3214

Column-store Architectures

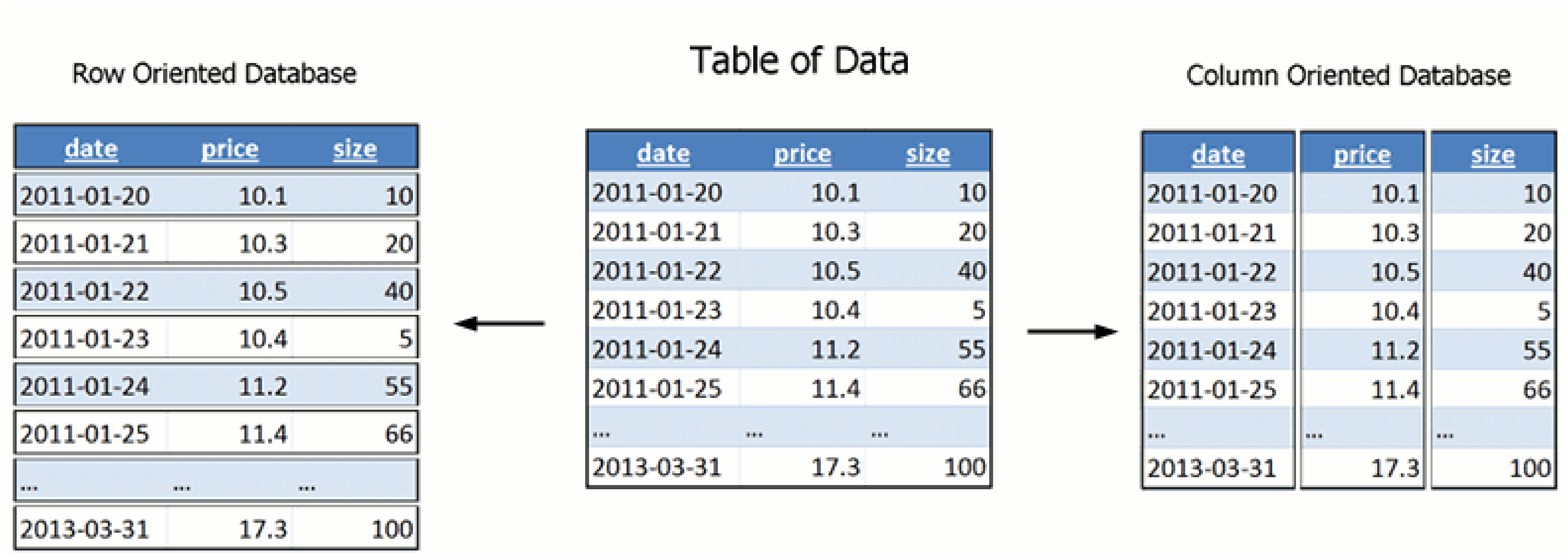
A column-oriented database stores each column continuously. i.e. on disk or in-memory each column on the left will be stored in sequential blocks.

For analytical queries that perform aggregate operations over a small number of columns retrieving data in this format is extremely fast.

Popular column family databases are Cassandra, HBase, AmazonSimpleDB

Column stores are suitable for read-mostly, read-intensive, large data repositories

Column-oriented databases



Row Oriented Database

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Table of Data

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

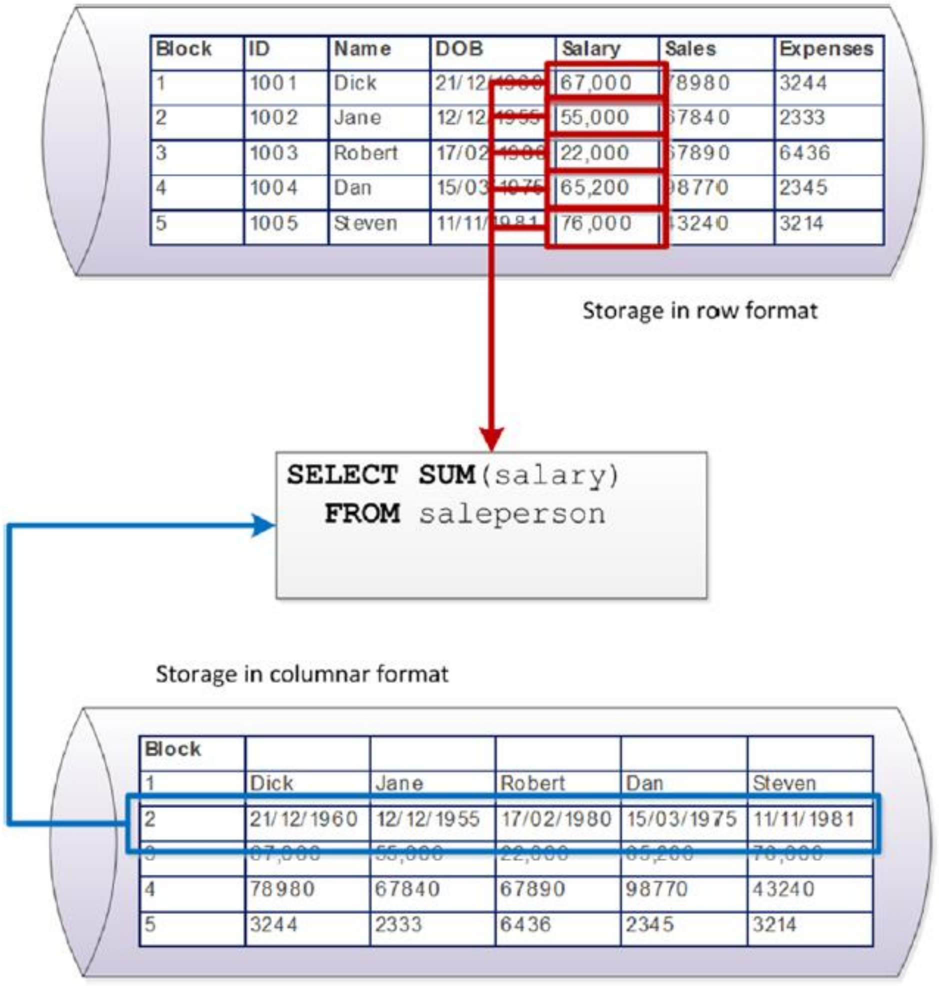
Column Oriented Database

date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

Column-store Advantages

The first key advantage:

In a columnar architecture, queries that seek to aggregate the values of specific columns are optimized, because all of the values to be aggregated exist within the same disk blocks.



Block	ID	Name	DOB	Salary	Sales	Expenses
1	1001	Dick	21/12/1960	67,000	78980	3244
2	1002	Jane	12/12/1955	55,000	67840	2333
3	1003	Robert	17/02/1980	22,000	67890	6436
4	1004	Dan	15/03/1975	65,200	98770	2345
5	1005	Steven	11/11/1981	76,000	43240	3214

```
SELECT SUM(salary)
  FROM saleperson
```

Block					
1	Dick	Jane	Robert	Dan	Steven
2	21/12/1960	12/12/1955	17/02/1980	15/03/1975	11/11/1981
3	67,000	55,000	22,000	65,200	76,000
4	78980	67840	67890	98770	43240
5	3244	2333	6436	2345	3214

Columnar Compression

The second key advantage: Compression.

Compression algorithms work primarily by removing redundancy within data values.

Data that is highly repetitious - especially if those repetitions are localized - achieve higher compression ratios than data with low repetition.

Although the total amount of repetition is the same across the entire database, regardless of row or column orientation, compression schemes usually try to work on localized subsets of the data; the CPU overhead of compression is far lower if it can work on isolated blocks of data.

Columnar Compression

Since in a columnar database the columns are stored together on disk, achieving a higher compression ratio is far less computationally expensive.

Furthermore, in many cases a columnar database will store column data in a sorted order. In this case, very high compression ratios can be achieved simply by representing each column value as a “delta” from the preceding column value.

The result is extremely high compression ratios achieved with very low computational overhead.

Disadvantage: Columnar Write Penalty

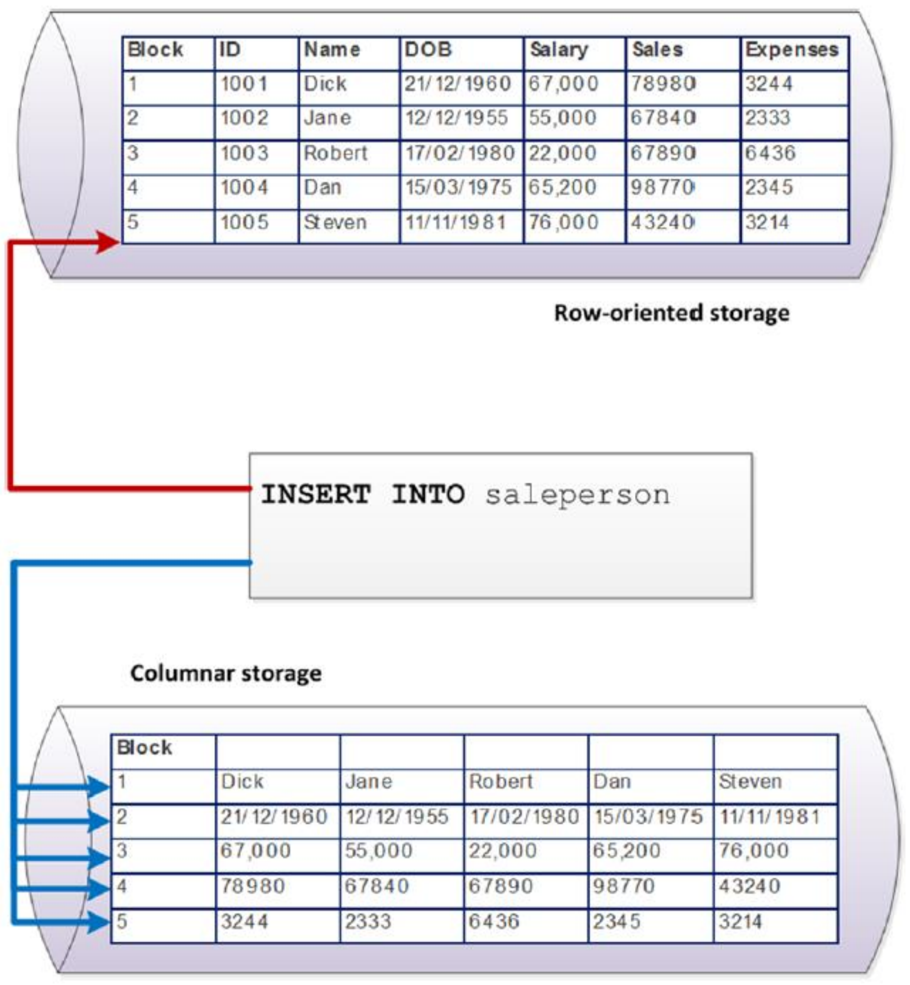
The key disadvantage of the columnar architecture - and the reason it is a poor choice for OLTP databases - is the overhead it imposes on single row operations.

In a columnar database, retrieving a single row involves assembling the row from each of the column stores for that table. Read overhead can be partly reduced by caching and multicolumn projections (storing multiple columns together on disk).

However, when it comes to DML operations - particularly inserts - there is virtually no way to avoid having to modify all the columns for each row.

Columnar Write Penalty

Figure illustrates the insert overhead for a column store on our simple example database. The row store need only perform a single IO to insert a new value, while the column store must update as many disk blocks as there are columns.



Block	ID	Name	DOB	Salary	Sales	Expenses
1	1001	Dick	21/12/1960	67,000	78980	3244
2	1002	Jane	12/12/1955	55,000	67840	2333
3	1003	Robert	17/02/1980	22,000	67890	6436
4	1004	Dan	15/03/1975	65,200	98770	2345
5	1005	Steven	11/11/1981	76,000	43240	3214

Block					
1	Dick	Jane	Robert	Dan	Steven
2	21/12/1960	12/12/1955	17/02/1980	15/03/1975	11/11/1981
3	67,000	55,000	22,000	65,200	76,000
4	78980	67840	67890	98770	43240
5	3244	2333	6436	2345	3214

Column-Oriented Database vs Row Oriented Database

Operation	Column-Oriented Database	Row-Oriented Database
Aggregate Calculation of Single Column e.g. sum(price)	fast	slow
Compression	Higher. As stores similar data together	-
Retrieval of a few columns from a table with many columns	Faster	has to skip over unnecessary data
Insertion/Updating of single new record	Slow	Fast
Retrieval of a single record	Slow	Fast

C-Store

In C-Store, the primary representation of data on disk is as a set of column files.

Each column-file contains data from one column, compressed using a column-specific compression method, and sorted according to some attribute in the table that the column belongs to. This collection of files is known as the “read optimized store” (ROS).

Additionally, newly loaded data is stored in a write-optimized store (“WOS”), where data is uncompressed and not vertically partitioned.

Periodically, data is moved from the WOS into the ROS via a background “tuple mover” process, which sorts, compresses, and writes re-organized data to disk in a columnar form.

C-Store

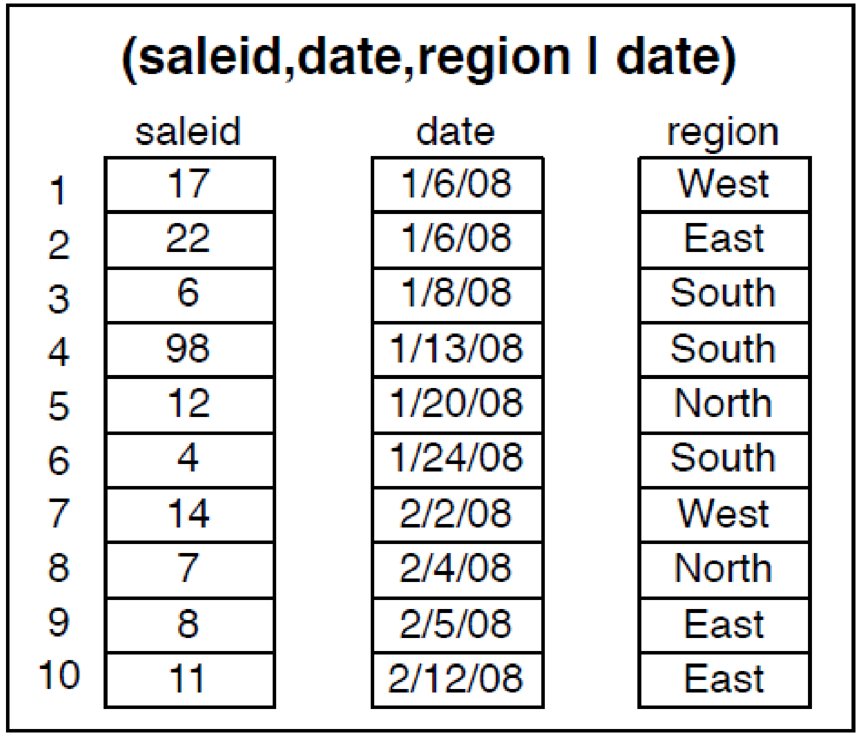
Each column in C-Store may be stored several times in several different sort orders. Groups of columns sorted on the same attribute are referred to as “projections”.

Typically there is at least one projection containing all columns that can be used to answer any query.

Projections with fewer columns and different sort orders are used to optimize the performance of specific frequent queries.

C-Store

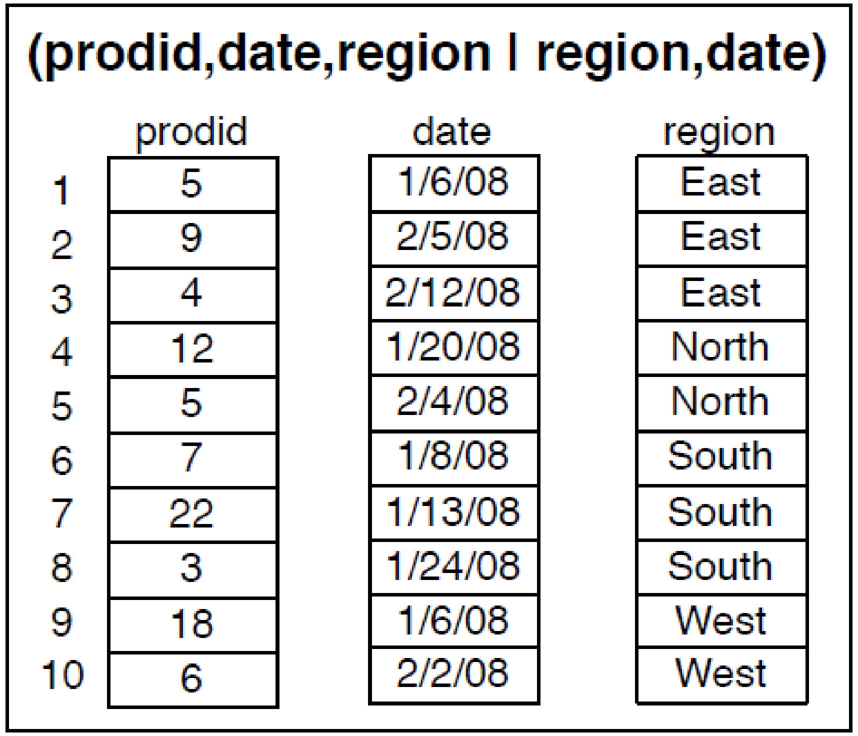
Two different projections of Sales table.



(saleid,date,region | date)

	saleid	date	region
1	17	1/6/08	West
2	22	1/6/08	East
3	6	1/8/08	South
4	98	1/13/08	South
5	12	1/20/08	North
6	4	1/24/08	South
7	14	2/2/08	West
8	7	2/4/08	North
9	8	2/5/08	East
10	11	2/12/08	East

(a) Sales Projection Sorted By Date



(prodid,date,region | region,date)

	prodid	date	region
1	5	1/6/08	East
2	9	2/5/08	East
3	4	2/12/08	East
4	12	1/20/08	North
5	5	2/4/08	North
6	7	1/8/08	South
7	22	1/13/08	South
8	3	1/24/08	South
9	18	1/6/08	West
10	6	2/2/08	West

(b) Sales Projection Sorted By Region, Date

C-Store

Each column in C-Store is compressed and for each column a different compression method may be used. The choice of a compression method for each column depends on

a) whether the column is sorted or not,

b) on the data type and

c) on the number of distinct values in the column.

For example, the sorted product class column is likely to have just a few distinct values; since these are represented in order, this column can be encoded very compactly using run-length encoding (RLE).

C-Store

C-Store does not support secondary indices on tables, but does support efficient indexing into sorted projections through the use of sparse indexes.

A sparse index is a small tree-based index that stores the first value contained on each physical page of a column. A typical page in C-Store would be a few megabytes in size.

Given a value in a sorted projection, a lookup in this tree returns the first page that contains that value. The page can then be scanned to find the actual value.

A similar sparse index is stored on tuple position, allowing C-store to efficiently find a given tuple offset in a column, even when the column is compressed or contains variable-sized attributes.

C-Store

Additionally, C-Store uses a “no-overwrite” storage representation, where updates are treated as deletes followed by inserts, and deletes are processed by storing a special “delete column” that records the time every tuple was deleted (if ever).

Query execution in C-Store involves accessing data from both the ROS and WOS and unioning the results together.

Queries are run as of a specific time, which is used to filter out deleted tuples from the delete column. This allows queries to be run as of some time in the past.

C-Store

Finally, in addition to complete vertical partitioning, C-Store was conceived as a shared-nothing massively parallel distributed database system, although the academic prototype never included these features (the commercial version, Vertica, does).

The idea behind the parallel design of C-Store is that projections are horizontally partitioned across multiple nodes using hash- or range-partitioning, and queries are pushed down and executed as much as possible on each node, with partial answers merged to produce a final answer at the output node.

Vector Wise

Vector-wise is a next generation database management system from the Actian family of database products.

The main innovation in VectorWise is its vectorized execution model which strikes a balance between full materialization of intermediate results in MonetDB and the high functional overhead of tuple-at-a-time iterators in traditional systems.

Essentially, VectorWise processes one block/vector of a column at a time as opposed to one column-at-a-time or one tuple-at-a-time.

Vector Wise

The "vectorised“ method is used for evaluating queries.

Rather than operating on single values from single table records at a time, Vector-wise makes the CPU operate on "vectors," which are arrays of values from many different records.

Such vectorised execution brings out the best in modern CPU technology. It brings to the world of databases the high performance that modern computers exhibit for scientific calculation, gaming, and multimedia applications.

Vector Wise

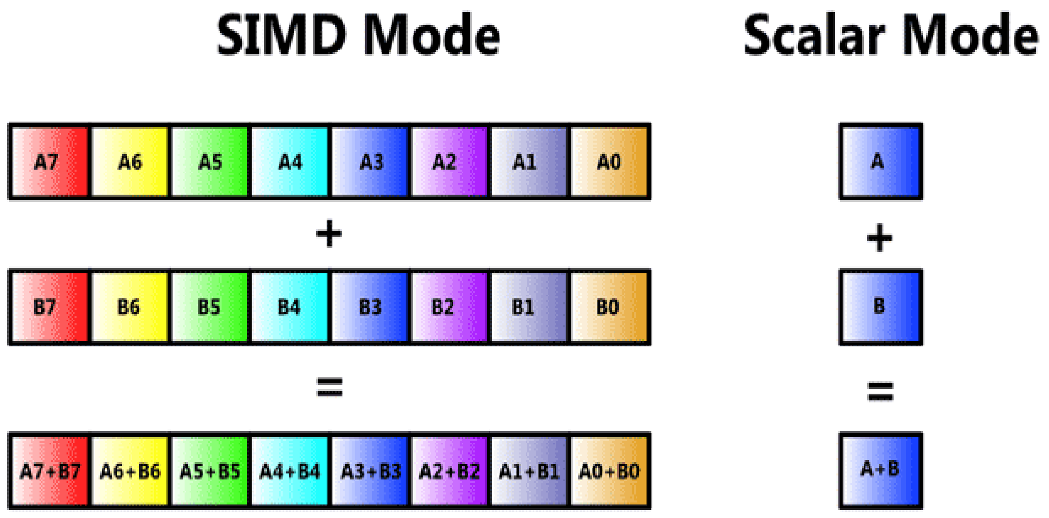
Vectorization is the term for converting a scalar program to a vector program.

Vectorized programs can run multiple operations from a single instruction, whereas scalar can only operate on pairs of operands at once.

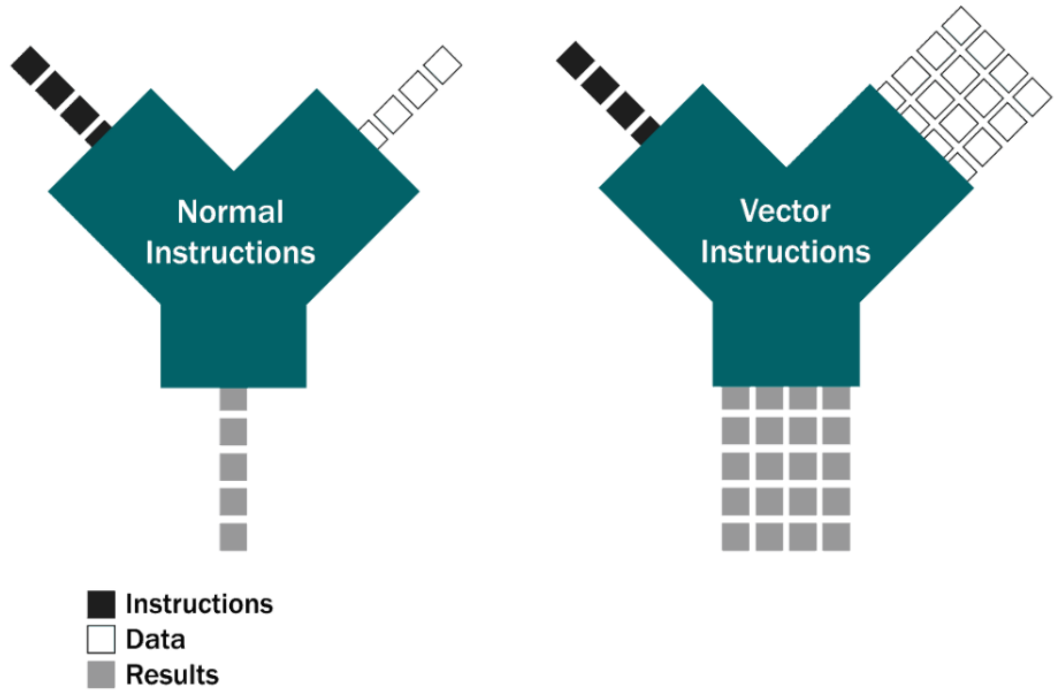
Vectorization exploits SIMD (Single Instruction Multiple Data) instructions which allows executing a single instruction on multiple data.

One SIMD instruction can process 16 integers at a time, thereby achieving Data Level Parallelism.

Vector Wise



SIMD Mode
Scalar Mode
A7 A6 A5 A4 A3 A2 A1 A0
+
B7 B6 B5 B4 B3 B2 B1 B0
=
A7+B7 A6+B6 A5+B5 A4+B4 A3+B3 A2+B2 A1+B1 A0+B0
A
+
B
=
A+B



Normal Instructions
Vector Instructions
Instructions
Data
Results

Vectorization and Columnar Formats

Using columnar formats load only the columns required by the query. Thus, less I/O on reading from disk and more data could be loaded into the cache.

Applying an operator on a vector of a column is performant using the tight loops.

Data availability increases as the unused columns are not present in the cache. Having the data which is actually required for the query laid side by side like members of an array reduces cache misses.

Vector Wise

Vector-Wise does perform explicit I/O, in an advanced way, adaptively finding collaboration in the I/O needs of concurrent queries through its Active Buffer Manager (ABM) and Cooperative Scans.

Vector-Wise also provides a novel way of handling updates (Positional Delta Trees), and new high-speed compression algorithms.

Vector-wise is targeted at analytical database applications - applications that need to process large volumes of data and perform complex operations on it to derive useful information.

Vectorwise is optimized to work with both memory- and disk-resident datasets, allowing it to efficiently process large amounts of data (hundreds of gigabytes).

Vector Wise - Storage innovations

Any database system with such a high computational speed runs the risk of becoming I/O bound.

For this reason, the second major component of Vector-wise consists of storage innovations designed for high I/O throughput. These innovations include:

- Columnar data layout
- Advanced compression
- Storage indexes

Vector-wise introduces a new storage mechanism that uses columnar data layout, allowing analytical queries to avoid disk access for columns not involved in a query.

Vector Wise - Storage innovations

While you can generally think of Vector-wise storage as a column store, Vector-wise can mix columnar and row-based storage so that certain columns that are always accessed together get stored in the same disk block.

Layout decisions are largely handled automatically by the system, but can also be controlled by the user.

To further avoid I/O becoming a performance bottleneck, Vector-wise introduces a number of advanced compression schemes. These schemes are designed for fast decompression.

Finally, Vector-wise uses storage indexes. The storage indexes are small and store the minimum and maximum value per data block. The storage index, which is automatically created and maintained, enables the execution engine to rapidly identify candidate data blocks.

Vector-wise table structure

In broad terms, Vector-wise uses columnar storage.

While data is stored and retrieved in familiar relational rows, the internal storage is different.

Instead of storing all column values for a single row next to each other, all rows for a single column are stored together.

This storage format has benefits for data warehouse applications, such as reduced I/O and improved efficiency.

Vector-wise table structure

A relational table may have dozens or hundreds of columns. In traditional row-oriented relational storage, the entire row must be read even if a particular query requests only a few columns.

With column-oriented storage, a query that needs only a few columns will not read the remaining columns from disk and will not waste memory storing unnecessary values.

While Vector-wise is fundamentally a column store, it is not strictly so. For certain types of tables, Vector-wise stores data from more than one column together in a data block on disk.

Vector Wise

Typical examples include data warehousing, data mining, and reporting

The main benefit of Vector-wise over traditional Ingres is significantly higher performance on data analysis tasks. Vectorwise is built upon an Ingres framework and uses many Ingres facilities and utilities.

Limitations: Although it is fast for data analysis, Vectorwise is not meant to be used for traditional transaction processing. For OLTP, you can open a session to a traditional Ingres database from your Vector-wise client.

Summary

This session will give the knowledge about

- Data warehousing schemas
- Comparison of columnar and row-oriented storage,
- Column-store Architectures:
- C-Store and
- Vector-Wise