SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER APPLICATIONS

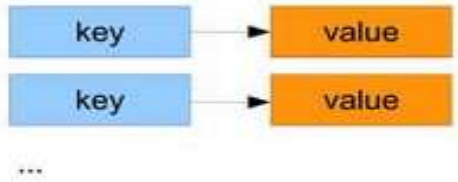
19CAE730 – Fundamentals of NOSQL database System

II YEAR III SEM

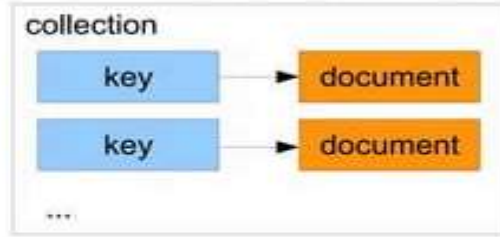
UNIT III – Comparison of columnar and row- oriented storage

Types of data stores

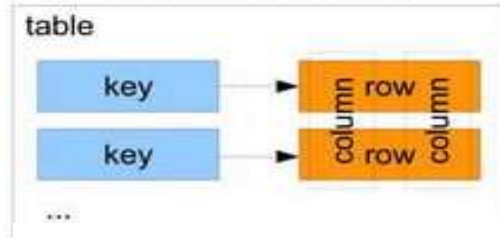
Key / value stores (opaque / typed)



Document stores (non-shaped / shaped)

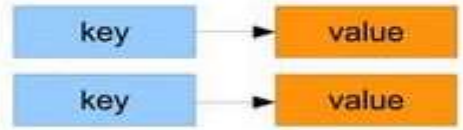


Relational databases



Key / value stores (opaque)

- Keys are mapped to values
- Values are treated as BLOBs (opaque data)
- No type information is stored
- Values can be heterogenous

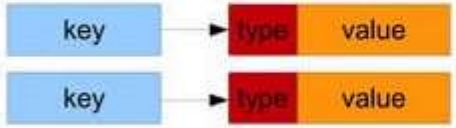


Example values:

- { name: „foo", age: 25, city: „bar" } => JSON, but store will not care about it
- \xde\xad\xb0\x0b => binary, but store will not care about it

Key / value stores (typed)

- Keys are mapped to values
- Values have simple type information attached
- Type information is stored per value
- Values can still be heterogenous



Example values:
- number: 25 => numeric, store can do something with
- list: [1, 2, 3] => list, store can do something with it

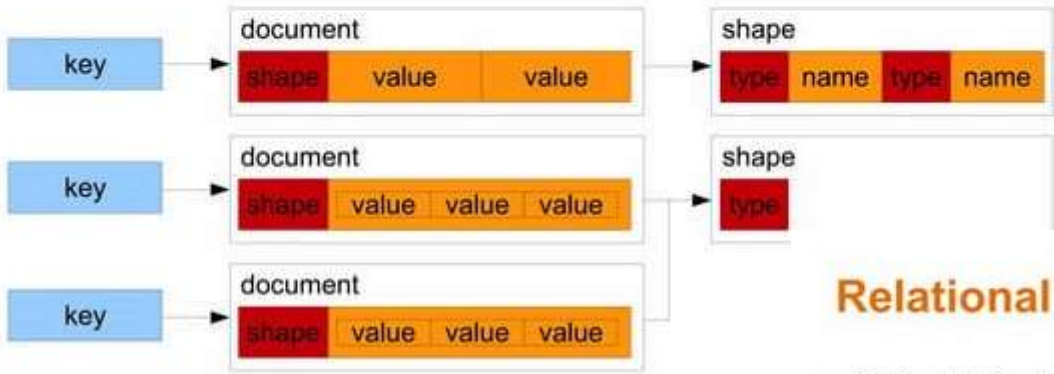
Document stores (non-shaped)

- Keys are mapped to documents
- Documents consist of attributes
- Attributes are name/typed value pairs, which may be nested
- Type information is stored per attribute
- Documents can be heterogenous
- Documents may be organised in collections or databases

Example documents:
- { name: „foo", age: 25, city: „bar" } => attributes and sub-attributes
- { name: { first: „foo", last: „bar" }, age: 25 } are typed and can be indexed

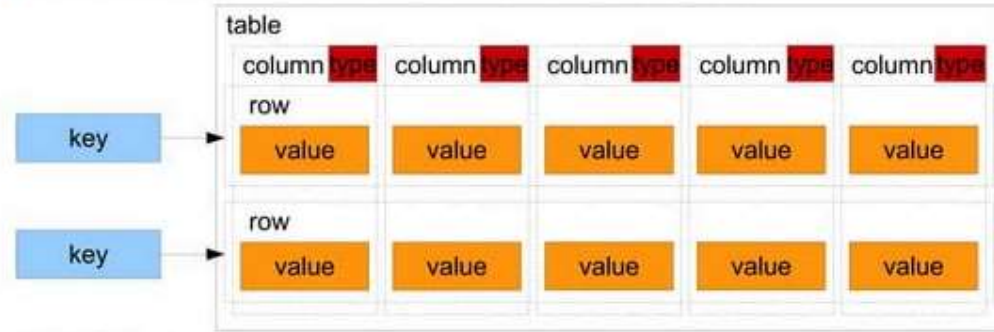
Document stores (shaped)

- Same as document stores, but...
- ...document type information is stored in shapes
- ...documents with similar structure (attribute names and types) point to the same shape



Relational databases

- Tables (relations) consist of rows and columns
- Columns have a type. Type information is stored once per column
- A rows contains just values for a record (no type information)
- All rows in a table have the same columns and are homogenous



Example rows:

- „foo", „bar", 25, 35.63
- „bar", „baz", 42, -673.342

Row vs. columnar relational databases

- All relational databases deal with tables, rows, and columns
- But there are sub-types:
  - row-oriented: they are internally organised around the handling of rows
  - columnar / column-oriented: these mainly work with columns
- Both types usually offer SQL interfaces and produce tables (with rows and columns) as their result sets
- Both types can generally solve the same queries
- Both types have specific use cases that they're good for (and use cases that they're not good for)

Row vs. columnar relational databases

- In practice, row-oriented databases are often optimised and particuarly good for OLTP workloads
- whereas column-oriented databases are often well-suited for OLAP workloads
- this is due to the different internal designs of row- and column-oriented databases

Row-oriented storage

- In row-oriented databases, row value data is usually stored contiguously:

row0 header	column0 value	column1 value	column2 value	column3 value
row1 header	column0 value	column1 value	column2 value	column3 value
row2 header	column0 value	column1 value	column2 value	column3 value

(the row headers contain record lengths, NULL bits etc.)

Row-oriented storage

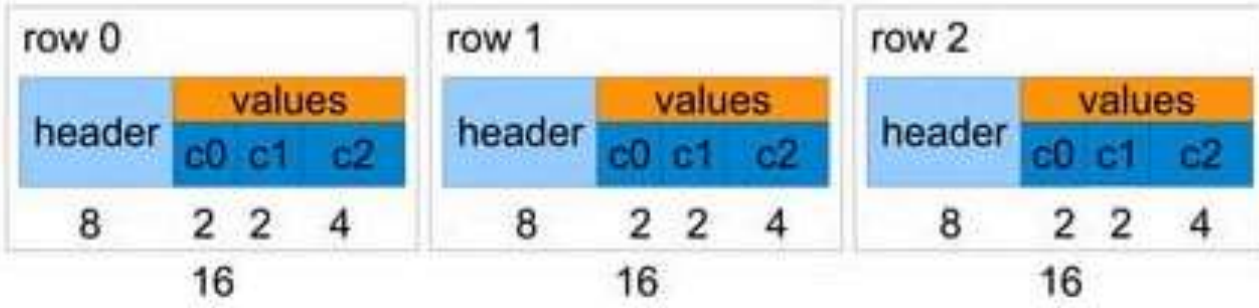
- When looking at a table's datafile, it could look like this:

- Actual row values are stored at specific offsets of the values struct:

- Offsets depend on column types, e.g. 4 for int32, 8 for int64 etc.

Row-oriented storage

- To read a specific row, we need to determine its position first
- This is easy if rows have a fixed length:
  - position = row number * row length + header length
  - header length = 8
  - row length = header length (8) + value length (8) = 16
  - value length = c0 length (2) + c1 length (2) + c2 length (4) = 8



Row-oriented storage

- Datafiles are often organised in pages
- To read data for a specific row, the full page needs to be read
- The smaller the rows, the more will fit onto a page

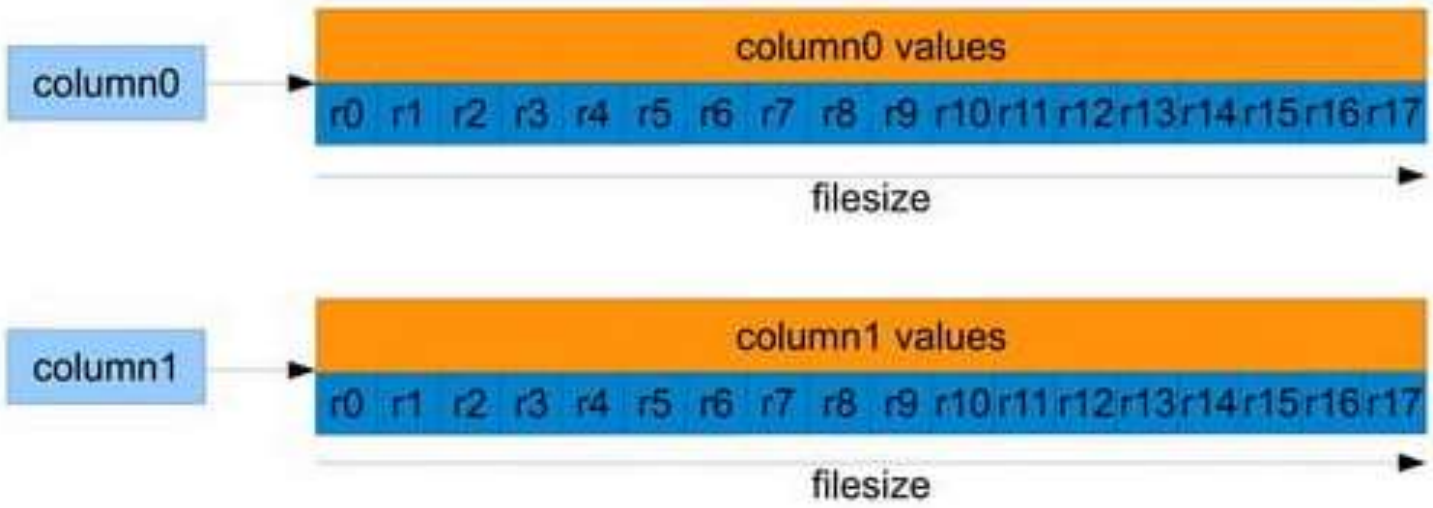
- In reality, pages are often not fully filled up entirely
- This allows later update-in-place without page splits or movements (these are expensive operations) but may waste a lot of space
- Furthermore, rows must fit onto pages, leaving parts unused

Column-oriented storage

- Column-oriented databases primarily work on columns
- All columns are treated individually
- Values of a single column are stored contiguously
- This allows array-processing the values of a column
- Rows may be constructed from column values later if required
- This means column stores can still produce row output (tables)
- Values from multiple columns need to be retrieved and assembled for that, making implementation of bit more complex
- Query processors in columnar databases work on columns, too

Column-oriented storage

- Column stores store data in column-specific files
- Simplest case: one datafile per column
- Row values for each column are stored contiguously



Column-oriented storage

- All data within each column datafile have the same type, making it ideal for compression
- Usually a much better compression factor can be achieved for single columns than for entire rows
- Compression allows reducing disk I/O when reading/writing column data but has some CPU cost
- For data sets bigger than the memory size compression is often beneficial because disk access is slower than decompression

Column-oriented storage

- A good use case for compression in column stores is dictionary compression for variable length string values
- Each unique string is assigned an integer number
- The dictionary, consisting of integer number and string value, is saved as column meta data
- Column values are then integers only, making them small and fixed width
- This can save much space if string values are non-unique
- With dictionaries sorted by column value, this will also allow range queries

Column-oriented storage, segments

- Column data in column stores is ofted grouped into segments/packets of a specific size (e.g. 64 K values)
- Meta data is calculated and stored seperately per segment, e.g.:
  - min value in segment
  - max value in segment
  - number of NOT NULL values in segment
  - histograms
  - compression meta data

Column-oriented processing

- Column values are not processed row-at-a-time, but block-at-a-time
- This reduces the number of function calls (function call per block of values, but not per row)
- Operating in blocks allows compiler optimisations, e.g. loop unrolling, parallelisation, pipelining
- Column values are normally positioned in contiguous memory locations, also allowing SIMD operations (vectorisation)
- Working on many subsequent memory positions also improves cache usage (multiple values are in the same cache line) and reduces pipeline stalls
- All of the above make column stores ideal for batch processing

Column-oriented processing

- Reading all columns of a row is an expensive operation in a column store, so full row tuple construction is avoided or delayed as much as possible internally
- Updating or inserting rows may also be very expensive and may cost much more time than in a row store
- Some column stores are hybrids, with read-optimised (column) storage and write-optimised OLTP storage
- Still, column stores are not really made for OLTP workloads, and if you need to work with many columns at once, you'll pay a price in a column store