title

wbg231

January 2023

1 Introduction

- the speed up from spark comes from delayed computing parallelism and caching
- it is not faster at every problem but in something like gradient descent it is way faster
- it is

column oriented storage

- it is all about speed (ie wall time)

history of column oriented storage

- idea goes back to the 80's
- had a resurgence in the 200s
- in the 80s it was not seen as that important because both cpu speed and storage speed were increasing
- starting in the mid 2000s storage speed stagnated so to make things faster speeds up were needed
- transferring from disk to memory is vary slow
- sequential memory reads are faster due to cache per fetching
- so we want to transfer fewer bites and use predictable and contiguous memory access patters to make reading data faster

row oriented storage csv

- if you had data stored in a row of text oriented way how would you go about getting the nth record, or just takes the kth column. you would have to go through each row until you find the kth value in that row since strings are of variable size
- rows and column are hard to predict
- basically requires a full serial scan

record oriented storage relational data

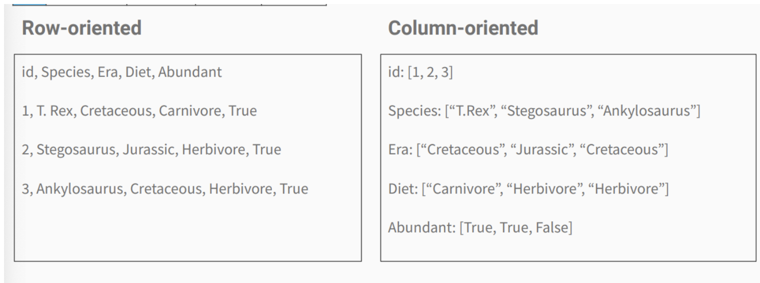
- relational data can be logically grouped by rows
- that is good if you want to process all records in a row at one time
- that is also nice for appending data
- it is human readable as well

queries row stores

- getting a col from a row oriented database is equivlent to a loop in the best case
- each row is loeaeded from storage
- atribute is inspected
- rows that pass are sent down stram
- an index can help local rows but that still involves pulling entire rows when we only want one column
- loading data from the disk is slow

column oriented storage

- each column is stored on its own
- values in each col have a constant type
- disk access patterns become much more regular
- this improves locality
- enables compresion and vectorized processing

- 

- i think the above picture shows why this works well as vectors.

speed is not everything

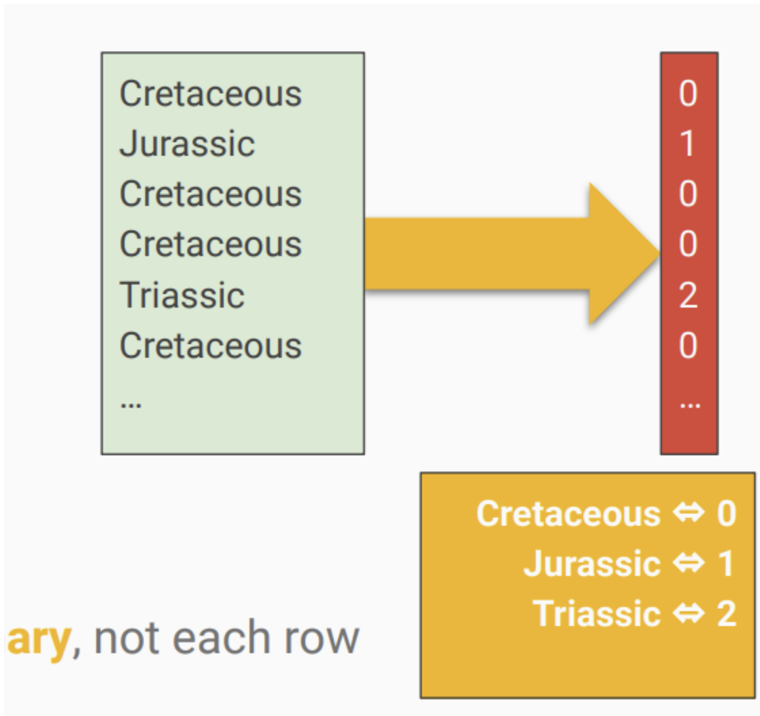
- storage space matters too
- mixed types are hard to compress
- once that data is arranged in a coloumnar fashion they all have the same type
- so tehy can be comresssed saving space meaning that the data can also be sent and recived much more easily

compression

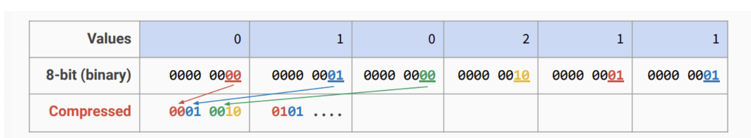
- records have heterogenou types
- a single coloumn has one type
- that means there is low entropy in a coloumn so can be easily compressed
- compressed data takes less space, is cheapter to load and sometimes we can compute directly on compressed data
- but what compression should we use

dictionary encoding

- this is where we encode each value in a coloumn with a unique key
- this works well when we have a few distinct values
- replace string value by string identfiers thi alows the column to have uniform data with and better cahc locality
- so string matching can be done on the dictionary not each row (since they are stored in difrent dictionarts)

- 

bit packing

- integers ussualy take 4-8 btes to store (32 bits or 64 bits)
- bit packing squeezes small integers togther
- 

Values	0	1	0	2	1	1
8-bit (binary)	0000 0000	0000 0001	0000 0000	0000 0010	0000 0001	0000 0001
Compressed	0001 0010	0101				

- matching and comparing can be done on the compressed data
- this will only work well when there are a lot of small integers

run length encoding

- usefull when we have long runs of constant values
- we convert a seuqnce of value to tuples of the type (value , #repeitions)
- sum average counts and other aggreagations can all be done on compressed values

other compressions

- frame of refrence encoding 1000 ,1004 , 1005 1002 $\rightarrow 1000|0, 4, 5, 2$
- delta coding $1004, 1005, 1006, \rightarrow 1004| + 0, +1, +1$
- many others
- copression can be combined
- the main trade of is space efficiency vs complexity of querying
- **slido** You work as a data scientist for Netflix and need to compress a movie to stream it efficiently. What is the most suitable compression scheme? Hint: A motion picture consists of many successive frames
- delta encoding, maybe run length

coloumn oriented storage take away

- pros
  1. can be faster if we only want a subset of atributes
  2. higher storage efficiency and throughput
  3. collecting dat of the same type allows for compression
- cons
  1. reconstructing full tuples from compressions can be slow
  2. writes and deletions an be slow
  3. handeling non tabular data is tricky

2 when data is not tabular

dremel and parquet

dremel

- dremel is a low latency query system for read only structured data
- devloped at google
- lots of cool ideas in the paper but lets talk about data format
- core ideas were quickly taken and reused in parquet

nested and structured data

- not everything fits nicely in relations
- varible lengths and depths are hard to deal with
- record oriented storage is more natural here
- how can we get all the benfits of column stores but for sturcutres data

trees

- we use the hierachal data strucutre tree

example web documnets

- there are required and not required tags
- need a doc id
- dont need links need at least 1 name and 1 language ecxeution

what specs would we like to see in a system that flattens hierachical records

- we want lossless representations of the hierachical records in coloumnar format
- it needs to be possible to recreate the heirchal record from the columnar format
- the key challange is being able tp parse records unambiouously
- we need to be able to keep track of the record strucutre, ie if a vllue abears in a table riwce we need to udnerstand wether ti is the same peice of data or a unque record
- to make this eficent it needs to be able to hand sparse datasets
- we need to be able to represnet mising fields efficently like null values

implementing records flattening with dremel

- the key idea is keeping track of rpetions of fields within a record to parse
- the repretiton level (which level repreated most recently) r
- the defention level d how many optimal diles are present
- the required fields same are the ame level as the partens
- otpinal fields the same r level as parents d lvel incprements
- then tehre are repeated fields
- go back through the flattneing example i dont have the life force for this right now
- dremel can easily rebuilt partial views
- unsued atributes can be ignoree
- but decoding the data is sequential so dremel data is hard to paralelleize

after flattening

- repertion adn defention cols are hgihyl compreasble
- value field are a new column of hte ame type
- cols broken into blocks and compressed incompedenitly

parquet

parquet

- devloped at twitter in 2013
- defualt storage for spark
- based on dremel flattening but without the analysis engenin or query machine

parquet format

- pages for a coloumn are compressed inpendntly
- small pages make it easier to read records but incur more overhead
- row grops should be large nut fit into one hdfs blocks

nice things about parquet

- there is cross language support
- allows for partial decoding ie only look at a few cols
- it works well with spark and hdfs
- preserved rdd dand data frmae directly

using parquet in practice with spark

- colum efficnet depend on row order it larely rellies on how compressable the dataa is
- data frame partions can be wrten out epeattly
- most frameworks we use in coding are already column oriented