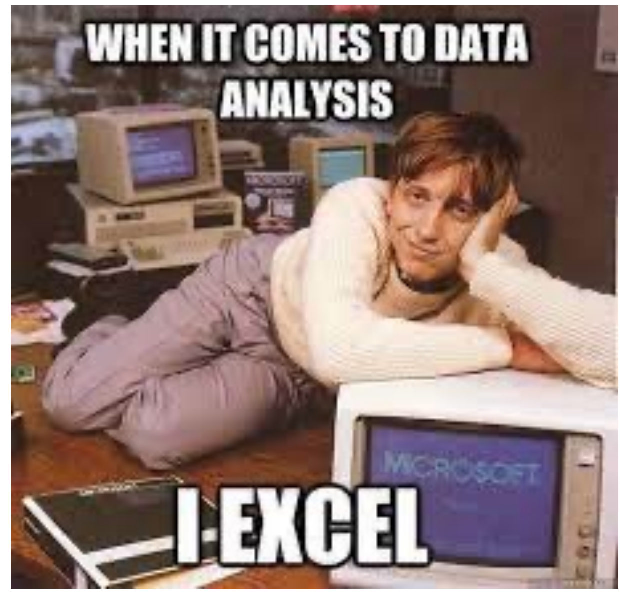
http://dsg.csail.mit.edu/6.S079/

6.S079 Lecture 4

Sam Madden



Key ideas:
Pandas
Parquet
FARS Example

Lab 2 Out

Recap: Last Two Lectures

- Relational Model
- SQL
- Database Tuning with Indexes

- Bands schema
  - **Bands**: bandid, name, genre
  - **Shows**: showid, show_bandid REFERENCES bands.bid, date, venue
  - **Fans**: fanid, name, birthday
  - **BandFans**: bf_bandid REFERENCES bands.bandid, bf_fanid REFERENCES fans.fanid

Bandfans Database Tuning Example

- Created a larger fake version of bandfans
  - 1M likes
  - 800 fans
  - 100K bands

Understanding Database Plans

- Most database systems provide an “explain” command that shows how it executes a query

  EXPLAIN SELECT count(*)

  FROM bandfans JOIN bands ON bf_bandid = bandid

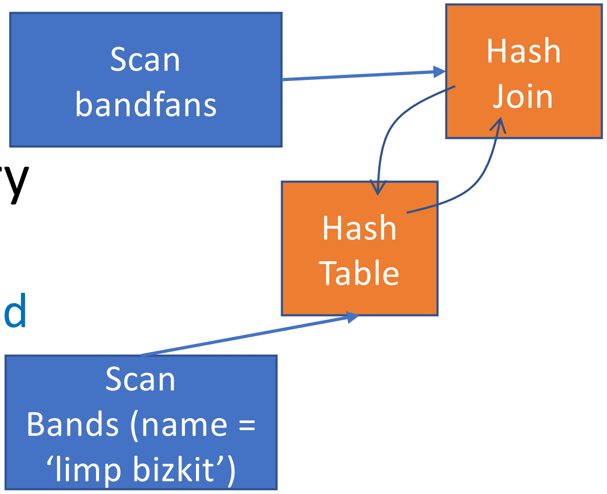
  WHERE name = 'limp bizkit'

This query takes 80ms to execute
Not slow, but this isn’t a large DB, and could be painful if we have to run many times

Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

Understanding Database Plans



- Most database systems provide an "explain" command that shows how it executes a query

  EXPLAIN SELECT count(*)
  FROM bandfans JOIN bands ON bf_bandid = bandid
  WHERE name = 'limp bizkit'

Example: POSTGRES

```
Aggregate  (cost=18210.82..18210.83 rows=1 width=8)
   ->  Hash Join  (cost=4.60..18204.60 rows=2489 width=0)
         Hash Cond: (bandfans.bf_bandid = bands.bandid)
         ->  Seq Scan on bandfans  (cost=0.00..14425.08 rows=1000008 width=4)
         ->  Hash  (cost=4.59..4.59 rows=1 width=4)
               ->  Seq Scan on bands  (cost=0.00..4.59 rows=1 width=4)
                     Filter: ((name)::text = 'limp bizkit'::text)
```

Parse tree
Read bottom up

How Can We Make This Faster?

- Goal: Reduce amount of data read
- What about just scanning bands rows that correspond to ‘limp bizkit’?
  - Index on bands.name
- Could we just scan the bandfans rows that correspond to ‘limp bizkit’?
  - Index on bandfans.bf_bandid

Creating An Index

- CREATE INDEX band_name ON bands(name);
- CREATE INDEX bf_index ON bandfans(bf_bandid);

B-Tree Index Example (B=2)

"Heap File"
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

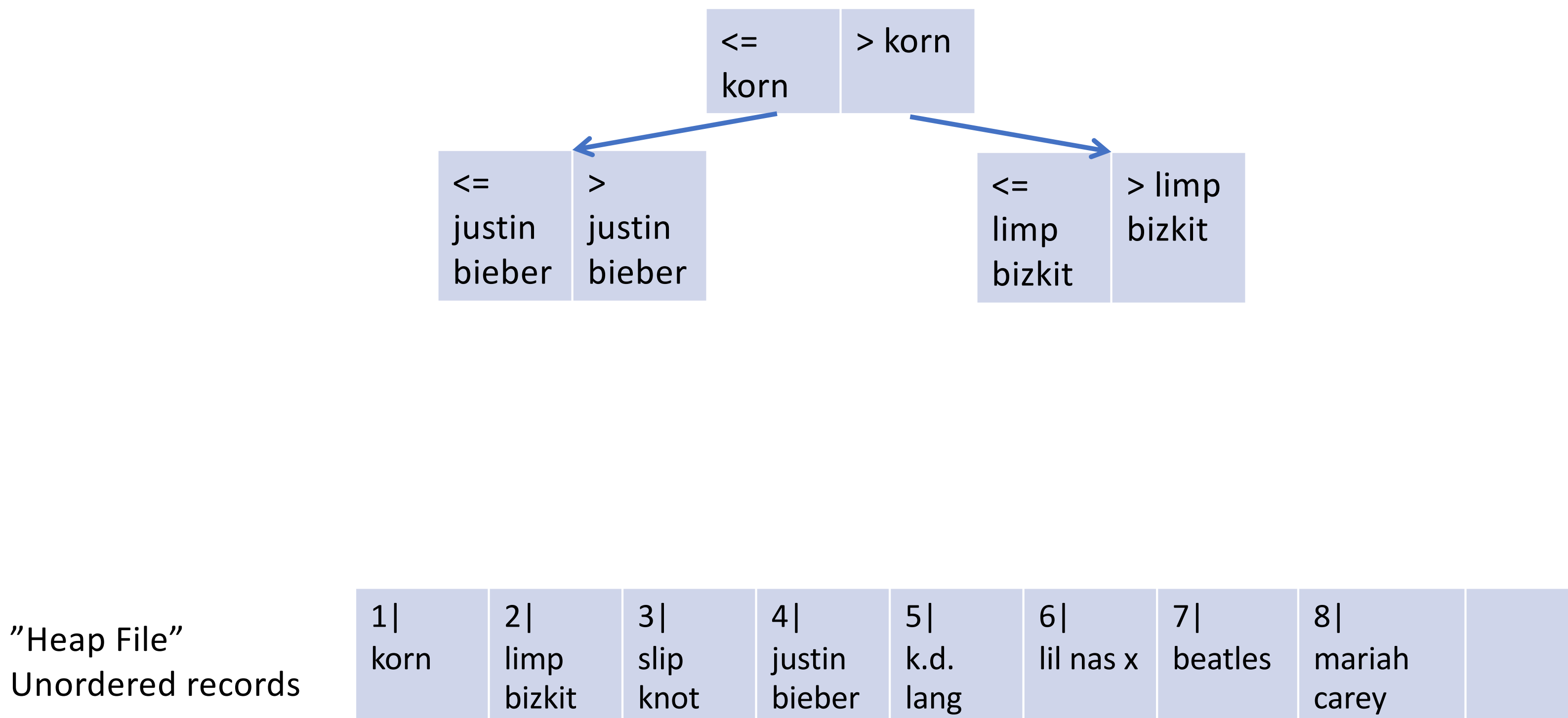
B-Tree Index Example (B=2)

<= korn	> korn

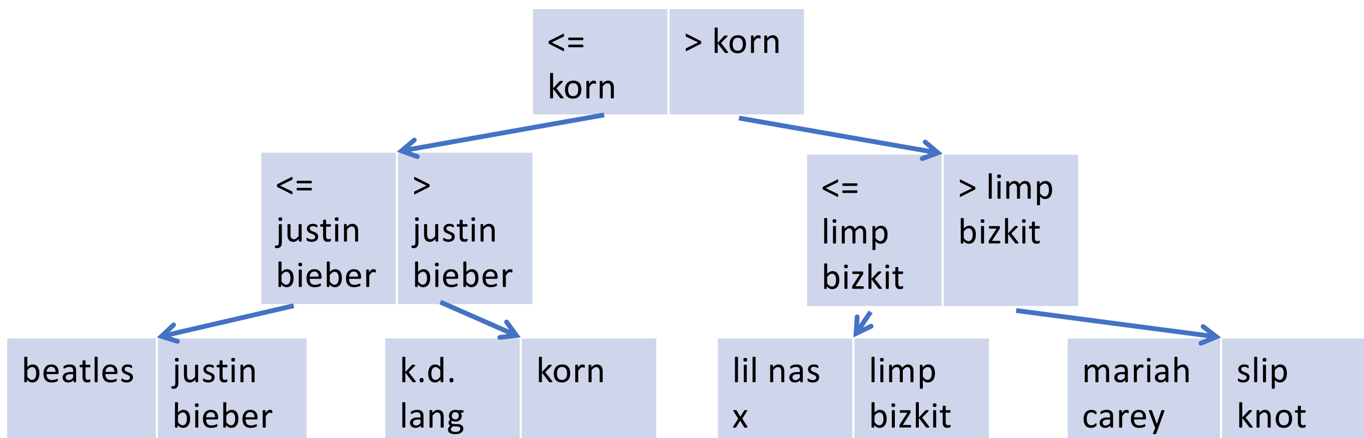
”Heap File”
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

B-Tree Index Example (B=2)



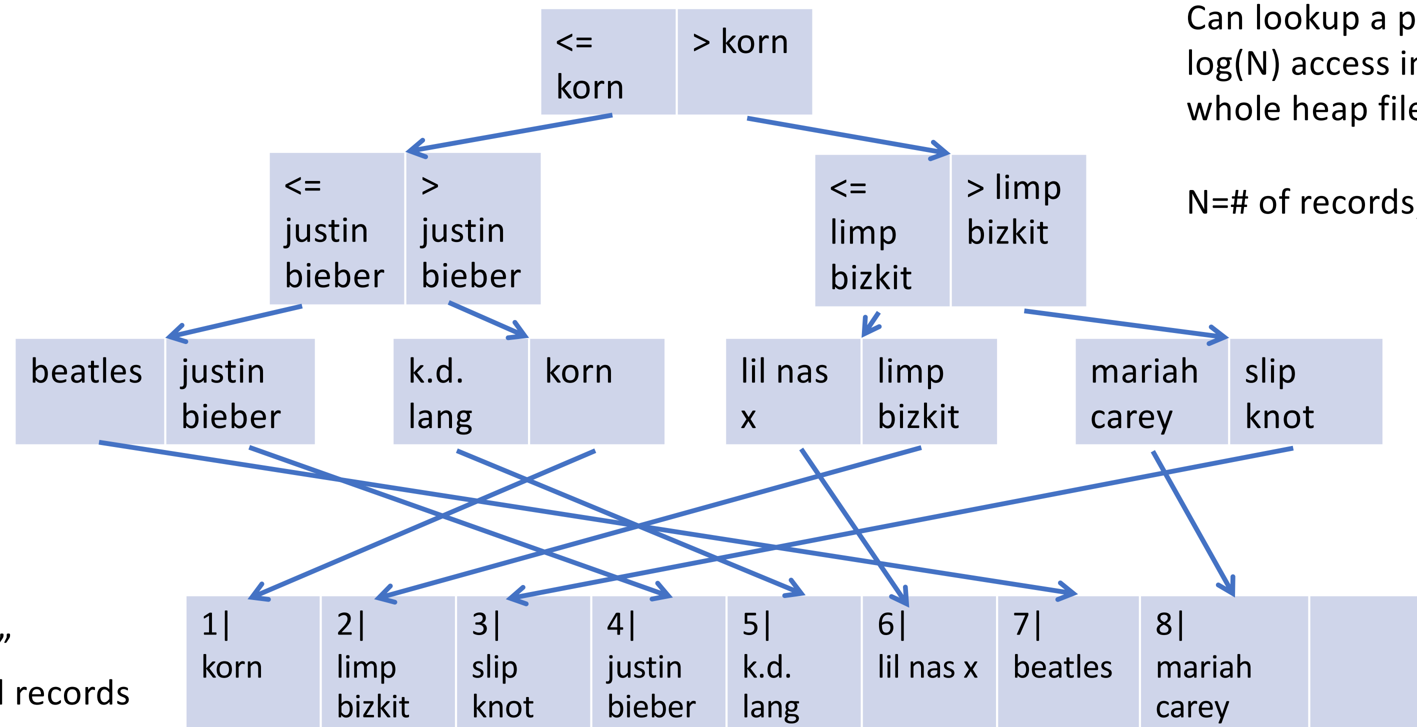
B-Tree Index Example (B=2)



”Heap File”
Unordered records

1| korn	2| limp bizkit	3| slip knot	4| justin bieber	5| k.d. lang	6| lil nas x	7| beatles	8| mariah carey	

B-Tree Index Example (B=2)



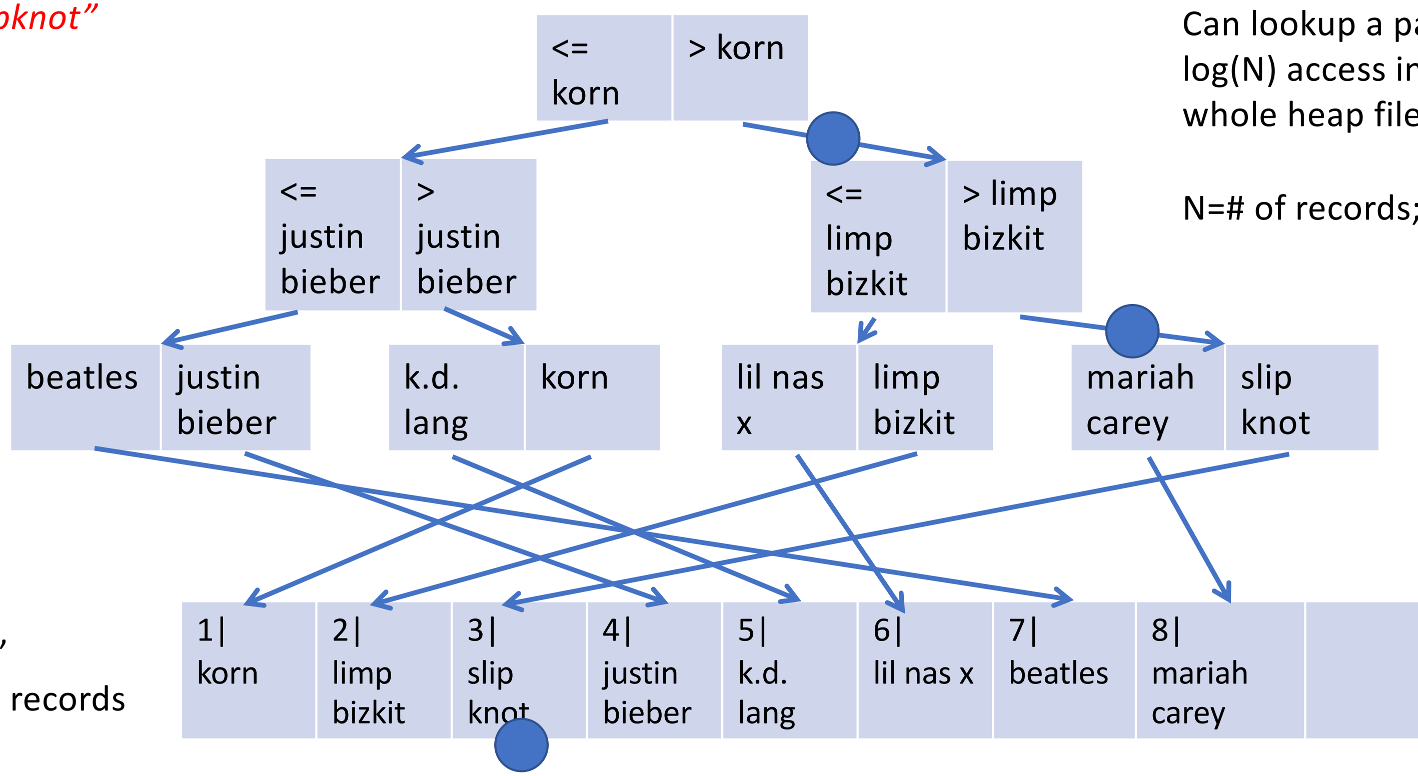
"Heap File"
Unordered records

Can lookup a particular record in log(N) access instead of scanning whole heap file

N=# of records; base of log is B

B-Tree Index Example (B=2)

Find "slipknot"



"Heap File"
Unordered records

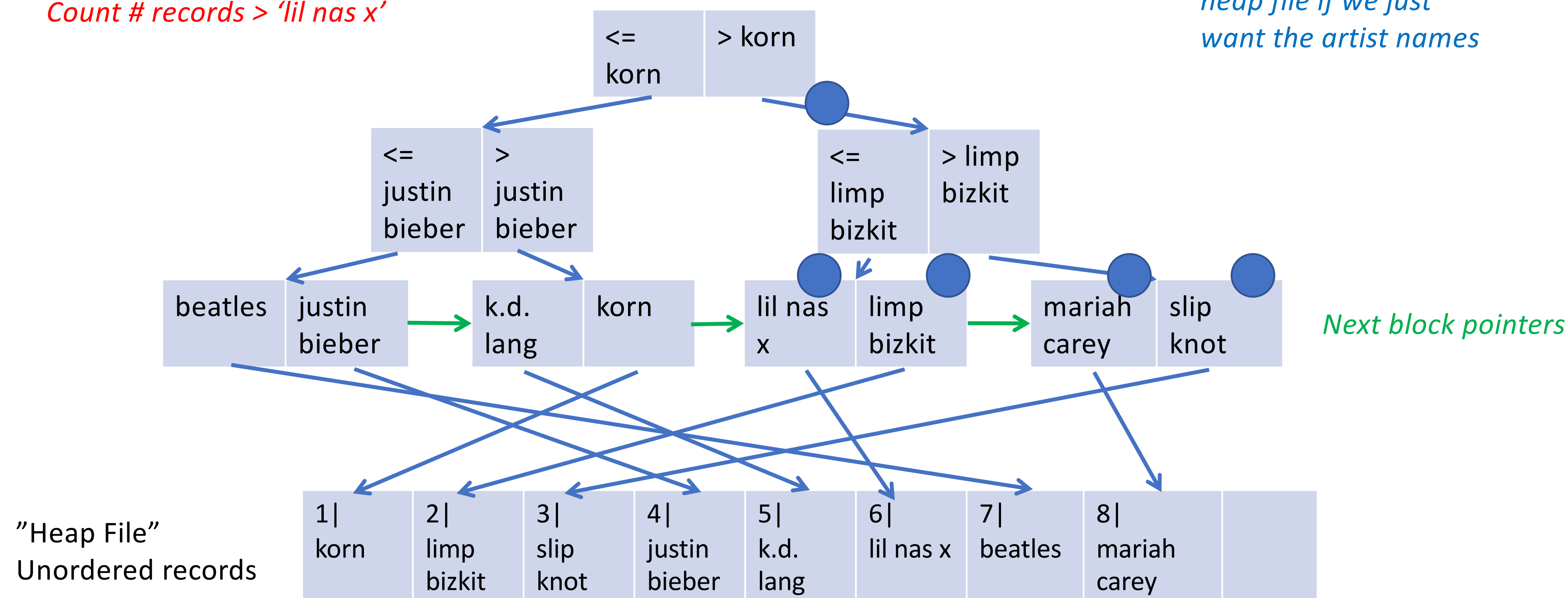
Can lookup a particular record in log(N) access instead of scanning whole heap file

N=# of records; base of log is B

Index-Only Scans

Count # records > 'lil nas x'

Don't need to go to heap file if we just want the artist names



Why Does an Index on Bandfans.bf_bandid Help?

```
SELECT count(*)
FROM bandfans
JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'
```

Given the bandid of limp bizkit (determined via a scan or index lookup), we can directly look up records in bandfans that match

Since there is only 1 record in bands for 'limp bizkit', this is a single index lookup instead of building a hash table on bandfans

Postgres

create index bf_index on bandfans(bf_bandid);

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

```
Aggregate  (cost=2162.44..2162.45 rows=1 width=8)
  ->  Nested Loop  (cost=0.42..2162.36 rows=30 width=0)
        ->  Seq Scan on bands  (cost=0.00..1918.84 rows=3 width=4)
              Filter: ((name)::text = 'limp bizkit'::text)
        ->  Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

Find limp bizkit record by scanning bands

Postgres

create index bf_index on bandfans(bf_bandid);

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

Estimated cost 2000 vs 12000
Actual 8ms vs 80ms

```
Aggregate  (cost=2162.44..2162.45 rows=1 width=8)
  -> Nested Loop  (cost=0.42..2162.36 rows=30 width=0)
        -> Seq Scan on bands  (cost=0.00..1918.84 rows=3 width=4)
              Filter: ((name)::text = 'limp bizkit'::text)
        -> Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

For each limp bizkit record (3 estimated)

Do an index only scan to count the number of fans

Don't need to go to records at all since index keys have bandid

Postgres

create index bf_index on bandfans(bf_bandid);
create index band_name on bands(name);

EXPLAIN SELECT count(*)
FROM bandfans JOIN bands ON bf_bandid = bandid
WHERE name = 'limp bizkit'

Estimated cost 260 vs 2000 vs 12000
Actual .5 ms vs 8 ms vs 80 ms

160x speedup!

Use index to directly lookup 'limp bizket'

```
Aggregate  (cost=259.94..259.95 rows=1 width=8)
  ->  Nested Loop  (cost=0.72..259.87 rows=30 width=0)
        ->  Index Scan using band_name on bands  (cost=0.29..16.34 rows=3 width=4)
              Index Cond: ((name)::text = 'limp bizkit'::text)
        ->  Index Only Scan using bf_index on bandfans  (cost=0.42..56.17 rows=2500 width=4)
              Index Cond: (bf_bandid = bands.bandid)
```

Monday's Reading

- Critique of SQL
- Some specific complaints about, e.g.,
  - json and windowing support
  - Verbose join syntax
  - Pitfalls around, e.g., subqueries
- More generally:
  - Lack of standards for extensions, e.g., new types or procedural support
  - New features, e.g., json and windows, are added via new syntax, rather than libraries as in most languages
    - Massive spec, very complex to support, huge burden on developers

Against SQL

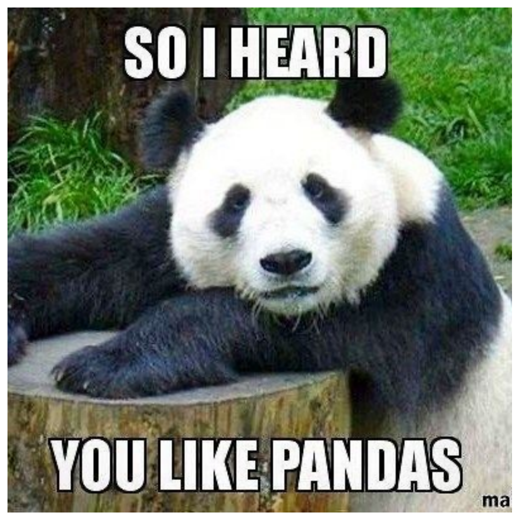
Published 2021-07-09

Recap: Some Common Data Access Themes

- SQL provides a powerful set-oriented way to get the data you want
- Joins are the crux of data access and primary performance concern
- To speed up queries, "read what you need"
  - Indexing & Index-only Scans
  - Predicate pushdown
    - E.g., using an index to find 'limp bizkit' records
  - Column-orientation
    - More on this later – we can physically organize data to avoid reading parts of records we don't need

Onto Pandas

- Pandas is a python library for working with tabular data
- Set-oriented thinking in Python
- Provides relation-algebra like ability to filter, join, and transform data



Loading a Data Set

```
import pandas as pd

df = pd.read_csv("bands.csv")
print(df)
```

Pandas tables are called “data frames”

As in SQL, columns are named and typed
Unlike SQL, they are also ordered (i.e., can access records by their position, and the notion of “next record” is well defined)

All dataframes have an “index” – by default, a monotonically increasing number

```
   bandid     bandname  genre
0       1  limp bizkit   rock
1       2         korn   rock
2       3        creed   rock
3       4   nickelback   rock
```

Accessing Columns

```
print(df.bandname)
```

```
0    limp bizkit
1           korn
2          creed
3     nickelback
```

Dots and brackets are equivalent
Can't use dots if field names are reserved keywords (e.g., "type", "class")

```
print(df["genre"])
```

```
Name: bandname, dtype: object
0     rock
1     rock
2     rock
3     rock
Name: genre, dtype: object
```

Accessing Rows

```
#limp bizkit rows
df_lb = df[df.bandname == 'limp bizkit']

print(df_lb)
```

```
   bandid     bandname genre
0       1  limp bizkit  rock
```

Array of Booleans with len(df) values in it

Indexing into a dataframe with a list of bools returns records where value in list is true

```
#get the record at position 1
print(df.iloc[1])
```

```
bandid         2
bandname    korn
genre       rock
Name: 1, dtype: object
```

```
   bandid     bandname genre
0       1  limp bizkit  rock
1       2         korn  rock
2       3        creed  rock
3       4   nickelback  rock
```

iloc vs loc

```
#get the genre of record with index attribute = 1
print(df.loc[1,"genre"])
```

```
rock
```

Index column

```
   bandid     bandname  genre
0       1  limp bizkit   rock
1       2         korn   rock
2       3        creed   rock
3       4   nickelback   rock
```

df.loc[1,'bandid']
df.iloc[1,0]

- loc uses the dataframe index column to access rows and column names to access data
- iloc uses the position in the dataframe and index into list of columns to access data
- By default index column and position are the same

Changing the Index

```
df_new = df.set_index("bandname")
print(df_new)
```

```
             bandid genre
bandname
limp bizkit       1  rock
korn              2  rock
creed             3  rock
nickelback        4  rock
```

```
print(df_new.loc["creed"])
```

```
bandid       3
genre     rock
Name: creed, dtype: object
```

Clicker

bandname	bandid	genre
limp bizkit	1	rock
korn	2	rock
creed	3	rock
nickelback	4	rock

- Given dataframe with bandname as index
- What is does this statement output?

```
print(df.iloc[1,1],df.loc[‘korn’,’bandid’])
```

A. rock 2
B. 2 2
C. 2 rock
D. 1 2

https://clicker.mit.edu/6.S079/

Transforming Data

```
df["is_rock"] = df.genre == "rock"

print(df)
```

```
   bandid     bandname genre  is_rock
0       1  limp bizkit  rock     True
1       2         korn  rock     True
2       3        creed  rock     True
3       4   nickelback  rock     True
```

```
df.loc[df.bandname == 'limp bizkit','genre'] = 'terrible'

print(df)
```

```
   bandid     bandname     genre
0       1  limp bizkit  terrible
1       2         korn      rock
2       3        creed      rock
3       4   nickelback      rock
```

Must Use iloc/loc to Change Data

This works:

```
df.loc[df.bandname == 'limp bizkit','genre'] = 'terrible'
```

This does not (even though it is a legal way to read data):

```
df[df.bandname == 'limp bizkit']['genre'] = 'terrible'
```

```
/Users/madden/6.s079/lec4-code/code.py:14: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

Grouping

Apply "count" to all non-grouping columns

```
df_grouped = df.groupby("genre").count()
print(df_grouped)
```

Creates a "GroupByObject" which supports a variety of aggregation functions

```
          bandid  bandname
genre
rock           3         3
terrible       1         1
```

Resulting data frame is indexed by the grouping column

Multiple Aggregates

```
df_grouped = df.groupby("genre").agg(max_band=("bandid","max"),
    num_bands=("bandname","count"))
print(df_grouped)
```

Name of column in output data frame
Note funky syntax

```
          max_band  num_bands
genre
rock             4          3
terrible         1          1
```

Joining (Merge)

```
df_bandfans = pd.read_csv("bandfans.csv")

df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid")
print(df_merged)
```

Join attributes
"left" data frame is the one we are calling merge on
"right" data frame is the one we pass in

```
   bandid     bandname     genre  bf_bandid  bf_fanid
0       1  limp bizkit  terrible          1         1
1       1  limp bizkit  terrible          1         2
2       2         korn      rock          2         1
3       3        creed      rock          3         1
```

Bands that don't join (e.g., Nickelback) are missing

Left/Right/Outer Join

```
df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid", how="left")
print(df_merged)
```

```
   bandid      bandname     genre  bf_bandid  bf_fanid
0       1  limp bizkit  terrible        1.0       1.0
1       1  limp bizkit  terrible        1.0       2.0
2       2         korn      rock        2.0       1.0
3       3        creed      rock        3.0       1.0
4       4   nickelback      rock        NaN       NaN
```

Chained Expressions

- All Pandas operations make a copy of their input and return it (unless you specify inplace=True)
- This makes long chained expressions common
  - Inefficient, but syntactically compact

```
df_merged = df.merge(df_bandfans, left_on="bandid", right_on="bf_bandid")\
            .groupby("bandname")\
            .agg(num_fans=("bf_fanid","count"))
print(df_merged)
```

```
             num_fans
bandname
creed               1
korn                1
limp bizkit         2
```

Break

Example: Driving Fatalities in the US

- Motor vehicle crashes are the leading cause of death for people ages 1-54
  - 38,000 people die each year
  - ~30% of fatal crashes involve alcohol

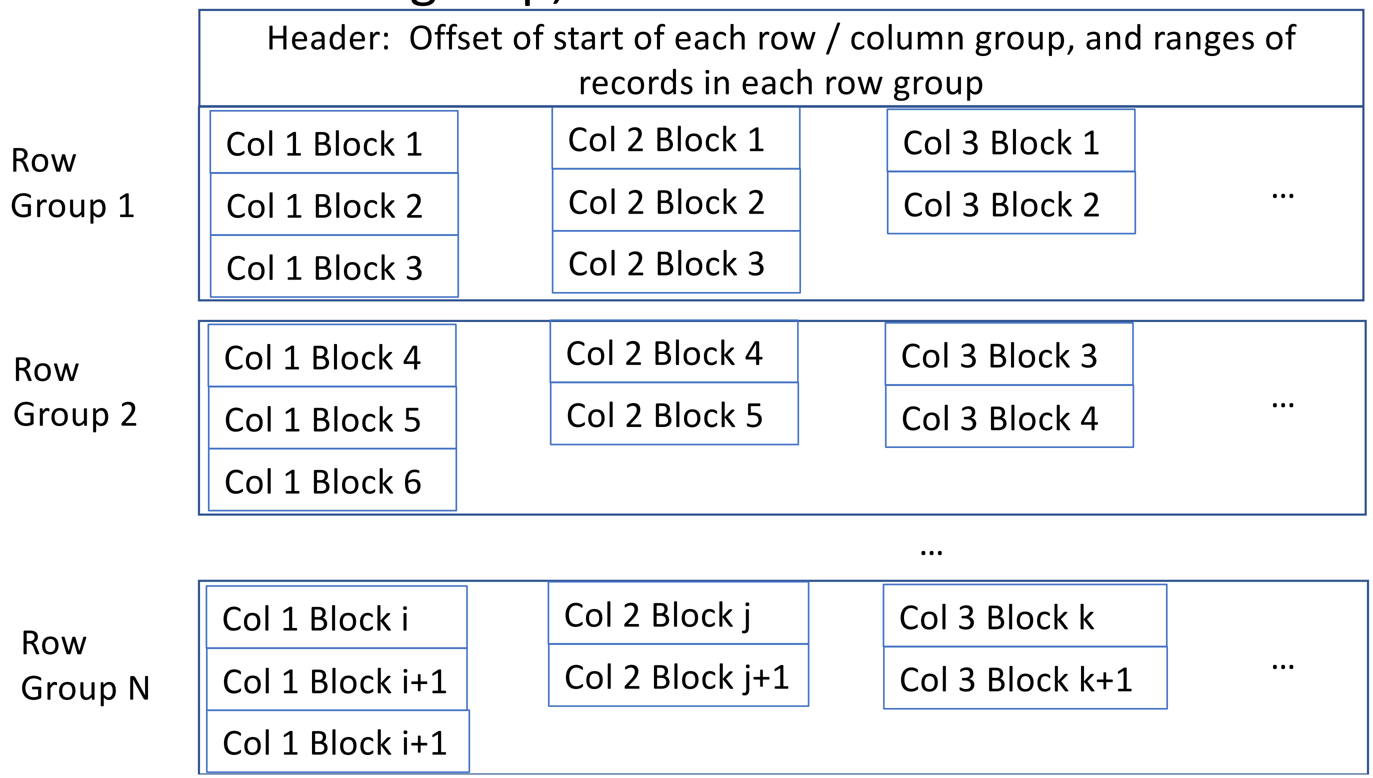
- The National Highway Traffic Safety Administration publishes detailed data about every fatal crash (FARS)

Efficient Data Loading: Parquet

- Parquet is a file format that is MUCH more efficient than CSV for storing tabular data
- Data is stored in binary representation
  - Uses less space
  - Doesn't require conversion from strings to internal types
  - Doesn't require parsing or error detection
  - Column-oriented, making access to subsets of columns much faster

Parquet Format

- Data is partitioned sets of rows, called “row groups”
- Within each row group, data from different columns is stored separately



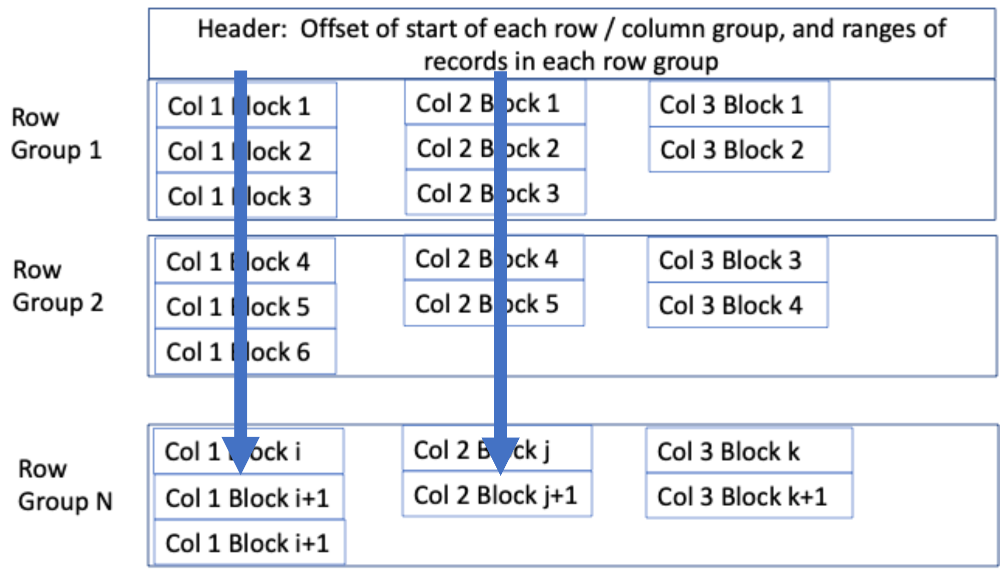
Using header, can efficiently read any subset of columns or rows without scanning whole file (unlike CSV)

Within a row group, data for each column is stored together

Predicate Pushdown w/ Parquet & Pandas

```
pd.read_parquet('file.pq', columns=['Col 1', 'Col 2'])
```

- Only reads col1 and col2 from disk
- For a wide dataset (e.g., our vehicle dataset w/ 93 columns), saves a ton of I/O



Performance Measurement

- Compare reading CSV to parquet to just columns we need

```
t = time.perf_counter()
df = pd.read_csv("FARS2019NationalCSV/Person.CSV", encoding = "ISO-8859-1")
print(f"csv elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq")
print(f"parquet elapsed = {time.perf_counter() - t:.3} seconds")

t = time.perf_counter()
df = pd.read_parquet("2019.pq", columns = ['STATE','ST_CASE','DRINKING','PER_TYP'])
print(f"parquet subset elapsed = {time.perf_counter() - t:.3} seconds")
```

```
csv elapsed = 1.18 seconds
parquet elapsed = 0.338 seconds
parquet subset elapsed = 0.025 seconds
```

47x speedup

When to Use Parquet?

- Will always be more efficient than CSV
- Converting from Parquet to CSV takes time, so only makes sense to do so if working repeatedly with a file
- Parquet requires a library to access/read it, whereas many tools can work with CSV
- Because CSV is text, it can have mixed types in columns, or other inconsistencies
  - May be useful sometimes, but also very annoying!
  - Parquet does not support mixed types in a column

Back to FARS Example

- Let’s look at how drunk driving has changed over the years

Example: Driving Fatalities in the US

- Motor vehicle crashes are the leading cause of death for people ages 1-54
  - 38,000 people die each year
  - ~30% of fatal crashes involve alcohol

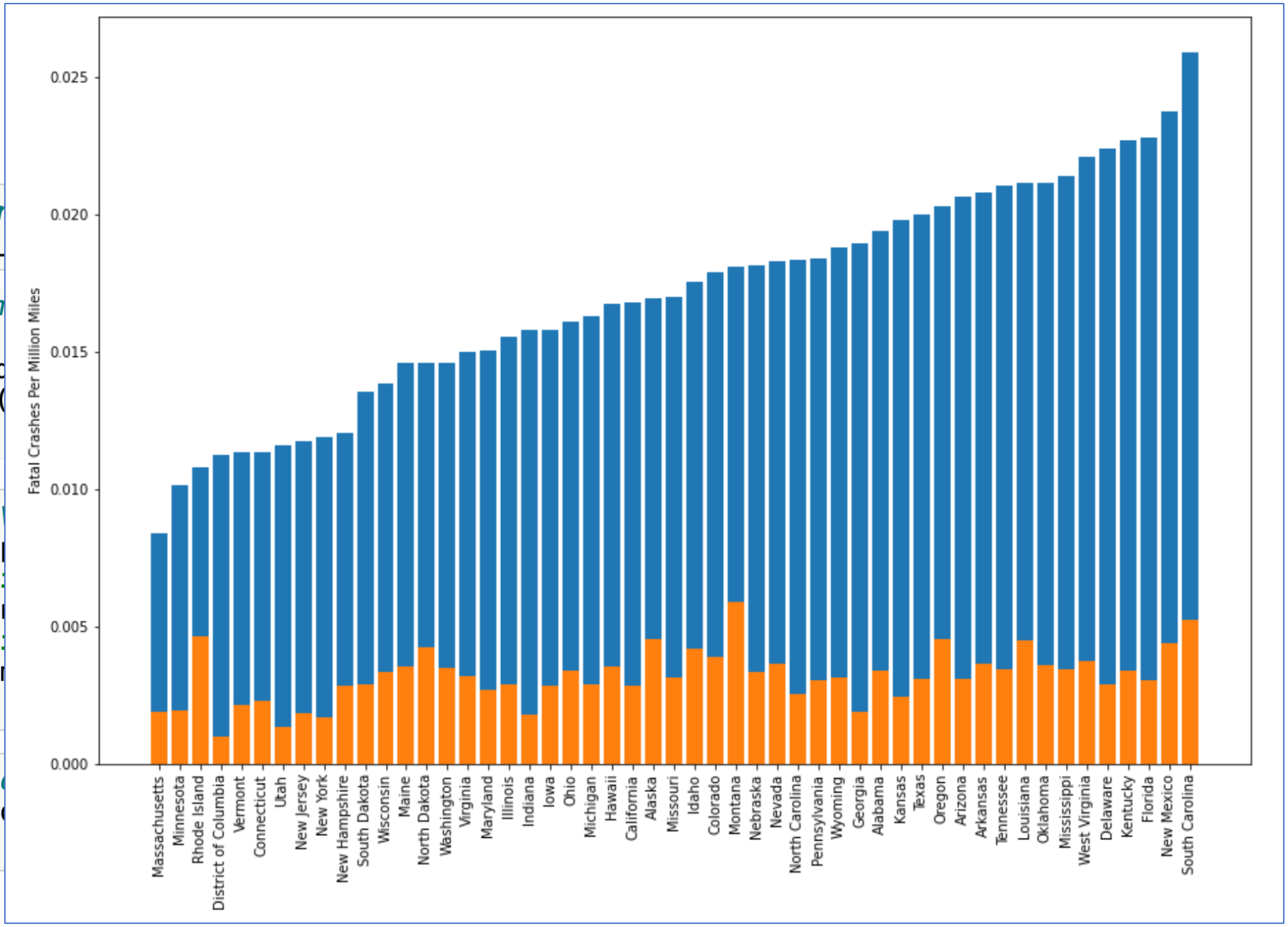
- The National Highway Traffic Safety Administration publishes detailed data about every fatal crash (FARS)

```
#load data fr
df = pd.read_
```

```
#need to norm
#get vehicle
vmt = pd.read
vmt.describe(
print(vmt)
```

```
#numbers in
vmt = vmt.re
for  column i
    if colum
        cont
    vmt[colu
print(vmt)
```

```
# now merge
df_merged =
```



Pandas vs SQL

- Could we have done this analysis in SQL?
- Probably…
- But not the plotting, or data cleaning, or data downloads
  - So would need Python to clean up data, reload into SQL, run queries
  - Declaring schemas, importing data, etc all somewhat painful in SQL
- So usual workflow is to use SQL to get to the data in the database, and then python for merging, cleaning and plotting
- Generally, databases will be faster for things SQL does well, and they can handle data that is much larger than RAM, unlike Python