6.S079
Lecture 4
Sam Madden

Key ideas:
Pandas
Parquet
FARS Example

Piazza signup:
http://piazza.com/mit/spring2022/6s079


Lab 1 Next Weds
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'
```

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)
```

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
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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>korn</td>
<td>limp bizkit</td>
</tr>
<tr>
<td>slip knot</td>
<td>justin bieber</td>
</tr>
</tbody>
</table>
k.d. lang | lil nas x |
|beatles | mariah carey |
B-Tree Index Example (B=2)

<table>
<thead>
<tr>
<th>&lt;= korn</th>
<th>&gt; korn</th>
</tr>
</thead>
<tbody>
<tr>
<td>korn</td>
<td></td>
</tr>
</tbody>
</table>

"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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>korn</td>
</tr>
<tr>
<td>2</td>
<td>limp bizkit</td>
</tr>
<tr>
<td>3</td>
<td>slip knot</td>
</tr>
<tr>
<td>4</td>
<td>justin bieber</td>
</tr>
<tr>
<td>5</td>
<td>k.d. lang</td>
</tr>
<tr>
<td>6</td>
<td>lil nas x</td>
</tr>
<tr>
<td>7</td>
<td>beatles</td>
</tr>
<tr>
<td>8</td>
<td>mariah carey</td>
</tr>
</tbody>
</table>
B-Tree Index Example (B=2)

"Heap File"
Unordered records

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>korn</td>
<td>2</td>
<td>limp bizkit</td>
<td>3</td>
<td>slip knot</td>
</tr>
<tr>
<td>4</td>
<td>justin bieber</td>
<td>5</td>
<td>k.d. lang</td>
<td>6</td>
<td>lil nas x</td>
</tr>
<tr>
<td>7</td>
<td>beatles</td>
<td>8</td>
<td>mariah carey</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B-Tree Index Example (B=2)

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”

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

Next block pointers

“Heap File”
Unordered records

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>korn</td>
<td>limp bizkit</td>
<td>slip knot</td>
<td>justin bieber</td>
<td>k.d. lang</td>
<td>lil nas x</td>
<td>beatles</td>
<td>mariah carey</td>
</tr>
</tbody>
</table>
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
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)

*Do an index only scan to count the number of fans*
*Can do an index only scan because we just need the count of records – don’t need any other fields from bandfans*
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'

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)

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'
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
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

```python
import pandas as pd

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

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

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).

```
<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>rock</td>
</tr>
<tr>
<td>1</td>
<td>korn</td>
<td>rock</td>
</tr>
<tr>
<td>2</td>
<td>creed</td>
<td>rock</td>
</tr>
<tr>
<td>3</td>
<td>nickelback</td>
<td>rock</td>
</tr>
</tbody>
</table>
```
Accessing Columns

```
print(df.bandname)
0    limp bizkit
1      korn
2      creed
3   nickelback

print(df["genre"])  
Name: bandname, dtype: object
0     rock
1     rock
2     rock
3     rock
Name: genre, dtype: object
```

*Dots and brackets are equivalent*

*Can’t use dots if field names are reserved keywords (e.g., “type”, “class”)*
Accessing Rows

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

print(df_larc)

bandid   bandname   genre
0        limp bizkit rock

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

bandid   bandname  genre
2        korn      rock

```

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>rock</td>
</tr>
<tr>
<td>1</td>
<td>korn</td>
<td>rock</td>
</tr>
<tr>
<td>2</td>
<td>creed</td>
<td>rock</td>
</tr>
<tr>
<td>3</td>
<td>nickelback</td>
<td>rock</td>
</tr>
</tbody>
</table>
iloc vs loc

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

```
rock
```

- `iloc` uses the position in the dataframe and index into list of columns to access data.

```
Index column

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>limp</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>korn</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>creed</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>nickelback</td>
</tr>
</tbody>
</table>
```

- `loc` uses the dataframe index column to access rows and column names to access data.

- By default, index column and position are the same.
Changing the Index

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

<table>
<thead>
<tr>
<th></th>
<th>bandid</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>bandname</td>
<td></td>
<td></td>
</tr>
<tr>
<td>limp bizkit</td>
<td>1</td>
<td>rock</td>
</tr>
<tr>
<td>korn</td>
<td>2</td>
<td>rock</td>
</tr>
<tr>
<td>creed</td>
<td>3</td>
<td>rock</td>
</tr>
<tr>
<td>nickelback</td>
<td>4</td>
<td>rock</td>
</tr>
</tbody>
</table>

print(df_new.loc["creed"])

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

• Given dataframe with bandname as index

<table>
<thead>
<tr>
<th>bandname</th>
<th>bandid</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>limp bizkit</td>
<td>1</td>
<td>rock</td>
</tr>
<tr>
<td>korn</td>
<td>2</td>
<td>rock</td>
</tr>
<tr>
<td>creed</td>
<td>3</td>
<td>rock</td>
</tr>
<tr>
<td>nickelback</td>
<td>4</td>
<td>rock</td>
</tr>
</tbody>
</table>

• What is does this statement output?

```python
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

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

```python
print(df)
```

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
<th>is_rock</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>rock</td>
<td>True</td>
</tr>
<tr>
<td>1</td>
<td>korn</td>
<td>rock</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>creed</td>
<td>rock</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>nickelback</td>
<td>rock</td>
<td>True</td>
</tr>
</tbody>
</table>

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

```python
print(df)
```

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>terrible</td>
</tr>
<tr>
<td>1</td>
<td>korn</td>
<td>rock</td>
</tr>
<tr>
<td>2</td>
<td>creed</td>
<td>rock</td>
</tr>
<tr>
<td>3</td>
<td>nickelback</td>
<td>rock</td>
</tr>
</tbody>
</table>
Must Use iloc/loc to Change Data

This works:

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

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

def[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

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

<table>
<thead>
<tr>
<th>genre</th>
<th>bandid</th>
<th>bandname</th>
</tr>
</thead>
<tbody>
<tr>
<td>rock</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>terrible</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Apply “count” to all non-grouping columns

Creates a “GroupByObject” which supports a variety of aggregation functions

Resulting data frame is indexed by the grouping column
Multiple Aggregates

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

<table>
<thead>
<tr>
<th>genre</th>
<th>max_band</th>
<th>num_bands</th>
</tr>
</thead>
<tbody>
<tr>
<td>rock</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>terrible</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Name of column in output data frame
Note funky syntax
Joining (Merge)

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

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

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
<th>bf_bandid</th>
<th>bf_fanid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>terrible</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>limp bizkit</td>
<td>terrible</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>korn</td>
<td>rock</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>creed</td>
<td>rock</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Bands that don’t join are missing
## Left/Right/Outer Join

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

<table>
<thead>
<tr>
<th>bandid</th>
<th>bandname</th>
<th>genre</th>
<th>bf_bandid</th>
<th>bf_fanid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>limp bizkit</td>
<td>terrible</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>limp bizkit</td>
<td>terrible</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>korn</td>
<td>rock</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>creed</td>
<td>rock</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>nickelback</td>
<td>rock</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>bandname</th>
<th>num_fans</th>
</tr>
</thead>
<tbody>
<tr>
<td>creed</td>
<td>1</td>
</tr>
<tr>
<td>korn</td>
<td>1</td>
</tr>
<tr>
<td>limp bizkit</td>
<td>2</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Row Group 1</th>
<th>Row Group 2</th>
<th>Row Group N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col 1 Block 1</td>
<td>Col 2 Block 1</td>
<td>Col 3 Block 1</td>
</tr>
<tr>
<td>Col 1 Block 2</td>
<td>Col 2 Block 2</td>
<td>Col 3 Block 2</td>
</tr>
<tr>
<td>Col 1 Block 3</td>
<td>Col 2 Block 3</td>
<td>Col 3 Block 3</td>
</tr>
<tr>
<td>Col 1 Block 4</td>
<td>Col 2 Block 4</td>
<td>Col 3 Block 3</td>
</tr>
<tr>
<td>Col 1 Block 5</td>
<td>Col 2 Block 5</td>
<td>Col 3 Block 4</td>
</tr>
<tr>
<td>Col 1 Block 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Col 1 Block i</td>
<td>Col 2 Block j</td>
<td>Col 3 Block k</td>
</tr>
<tr>
<td>Col 1 Block i+1</td>
<td>Col 2 Block j+1</td>
<td>Col 3 Block k+1</td>
</tr>
<tr>
<td>Col 1 Block i+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```python
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

```python
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
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
Next Time

• Guest Lecture
• Anant Bharwaj
• Former Ph.D. student in our group
• Founded Instabase, a platform transforming unstructured (e.g., text & images) to structured (e.g., tabular) data