INFO 290T

Human-Centered Data Management

A Primer on Scalability & Performance

Announcements

- Project proposal due on the 19th
- Plenty of auxiliary slots for next week, feel free to sign up

A Primer on Scalability & Performance

We know that there's low-level data processing operations that are expensive:

- Sorting
- Joins
- Grouping/aggregation
- …

And also higher level: ML/AI, data extraction, cleaning, transformation, visualization, …

Sometimes hard to avoid these sorts of operations, but people demand low latencies.

Even 500ms latencies leads to less hypotheses explored/insights generated [Liu and Heer 2014]

A Primer on Scalability & Performance

- Stuff you can do to improve performance
- Aka read less data or do less processing, or both
- You should already know at least some of these techniques
- Meant primarily as a recap
  - Indexes
  - Materialization
  - Columnar Representation
  - Compression
  - Prefetching
  - Approximation
  - Parallelism
  - Multi-Query Optimization
  - Returning Early

Indexes

- Many types: B+Tree, Hash, …
- Auxiliary structure that lets you "skip ahead" to data of interest
  - Don't actually change how the data is organized
  - Only really helps when you're reading a small portion of the data
  - Or if your data is organized (sorted) according to the same indexing attribute(s)
- Drawbacks:
  - Have to update your index when the data changes

Materialized Views

- Precomputed results of certain queries stored separately
  - Doesn't actually change how the raw data is stored
  - If the same query is issued repeatedly, materialization gets you the result "for free"
  - Otherwise, additional computation required to produce results using views
    - E.g., total sales from the materialized view of sales by region
- Drawbacks:
  - Have to update your index when the data changes

Columnar Representation & Compression

- Traditional databases store data as rows, one row after the other
- Most "analytical" queries read a subset of columns rather than all the columns
- Why not store columns individually and read them as needed?
  - O(columns read) rather than O(total columns)
- Plus, if you're storing as a column, you can compress better
- Drawbacks:
  - Doesn't work as well with mixed (read/write) workloads
  - Decompression may be costly

Prefetching

- Predicting what the user will need next, and retrieving it or computing it preemptively
- Requires a model of user behavior
- Drawbacks
  - May lead to unnecessary computation/IO if the prediction is inaccurate

Approximation

- Read not all of your data, but a sample of it
- Can be a pre-materialized/offline sample
  - If you store it based on some "bucketing" scheme, e.g., 100 tuples per state, then it is known as *stratified sample*
- Or computed on the fly: online
- Drawbacks
  - Results are approximate
  - Pre-materialized samples need to be updated
  - Online samples, if truly random, may be very expensive to draw (random accesses)
    - Or require the data to be randomly sorted

Parallelism

- Run things in parallel!
- Most analytical queries are embarrassingly parallel
  - Thinking computing aggregates on partitions, then aggregating the aggregates
- Most database systems already automatically do this for you
- Drawbacks:
  - More parallelism for a single user query means less for others
  - Diminishing returns beyond a point

Multi-Query Optimization

- Group work together
- Again, database systems will do some of this for you, but most are not smart enough to do so
- Drawbacks:
  - Figuring out how to do this is not straightforward
  - Sometimes need to post-process results
  - May need to hold back a query's execution to time it with another one

Returning Results Early

- Sometimes you don't need to do all of the processing to provide enough results for the user to decide on the next step
  - E.g., returning the first-k tuples, or doing enough work until you know the results are not going to change much visually
- Drawbacks
  - Not applicable to all settings