BlinkDB

(some figures were poached from the Eurosys conference talk)

The Holy Grail

Support **interactive** SQL queries over **massive sets** of data

Individual queries should return within seconds

Petabytes of data

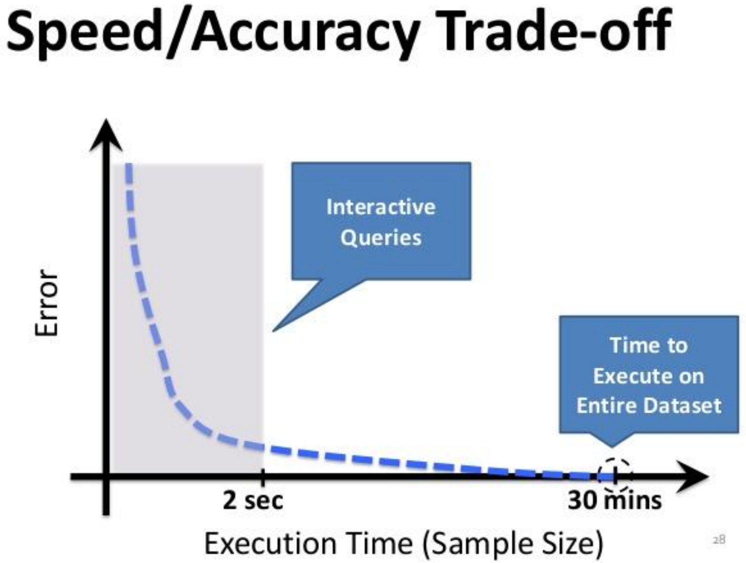
```
Select AVG(Salary) from Salaries
Where Gender= Women
GroupBy City
Left Outer Join Rent
On Salaries.City = Rent.City
```

Why is this hard?

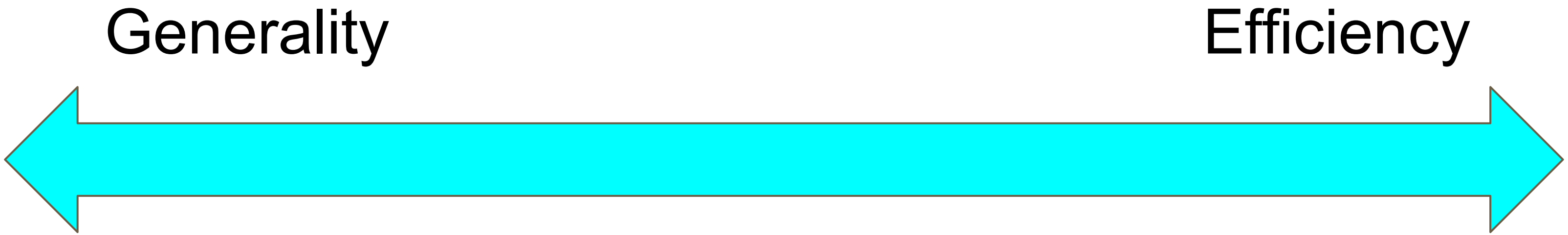
- Using Hadoop:
  - processing 10TB on 100 machines will take approx an hour
- Using In-Memory computing:
  - processing 10TB on 100 machines will take you 5 minutes
- Data is continuing to grow!
- So how can we get to **second-scale** latency?

An opportunity: approximate computing

- Key Observation
  - Most analytics workloads can deal with some amount of inaccuracy as these are often exploration queries
- This can buy you a lot!



Existing solutions



- OLA: General but …
  - Variable performance (faster for popular items)
  - Hard to provide error bars?
  - Inefficient IO Use

- Sketching, sampling.
  - Low space and time complexity
  - Strong assumptions about predictability of the workload and on queries that can be executed
  - Can’t do joins or subqueries

Arrive BlinkDB!

- Data warehouse analytics system built on top of Spark/Hive
- Allows users to trade-off accuracy for response time, and provide users with meaningful bounds on accuracy
- Support COUNT, AVG, SUM, QUANTILE

```
Select AVG(Salary) from Salaries
Where Gender= Women
GroupBy City
Left Outer Join Rent
On Salaries.City = Rent.City
ERROR WITHIN 10% AT CONFIDENCE 95%
```

```
Select AVG(Salary) from Salaries
Where Gender= Women
GroupBy City
Left Outer Join Rent
On Salaries.City = Rent.City
WITHIN 5 SECONDS
```

Goal: Better balance between efficiency and generality

- Key Idea 1: Sample creation
  - Optimisation framework that builds set of multi-dimensional stratified samples from original data using query column sets
- Key Idea 2: Sample selection
  - Runtime sample selection strategy that selects best sample size based on query's accuracy or response time requirements (uses an Error-Latency-Profile heuristic)
- Nice feature : Query execution
  - Returns fast responses to queries with error bars

Step 1: Sample Creation

- Three factors to consider
  - Workload taxonomy (how similar will future queries be to past queries)
  - The frequency of rare subgroups (sparsity) in the data (column entries are often long tail)
  - The store overhead of storing samples
- Design an optimization framework as a linear integer program to find out on which sets of columns should stratified samples be built.

Sample creation: workload taxonomy (1)

- Most queries have some similarity with past queries. Challenge is to quantify that similarity to minimise overfitting while adapting to the data.

- Multiple approaches: predictable queries, predictable query predicates, predictable query column sets, unpredictable queries.

```
Select AVG(Salary) where City = "New York"
```

- Use predictable query column sets (QCS)
    - 90% of queries are covered by 10% of unique GCSs in Conviva workload

Sample creation: uniform vs stratified (2)

- There might be huge variations in the number of tuples that satisfy a particular column set.

- Uniform sampling doesn’t work well for aggregates in this case:
  - Miss rare groups entirely
  - Groups with few entries would have significantly lower confidence bounds than popular data (=> assumption that we care equally)

- Use stratified sampling: rare subgroups are over-represented relative to a uniform sample

- Achieve this by computing group counts/buckets on all distinct entries in each column set, and sampling uniformly within that bucket (smaller samples can be generated from larger samples)

Sample creation: optimization problem (3)

- Goal: maximise the weighted sum of the coverage of the GCSs of the queries

- Coverage is defined as the probability that a given value x of columns q_j is also present among the rows of the sample S where:
  - Priority is given to sparser column sets (sparsity is the number of groups whose size in the data set is smaller than some number M)
  - Priority is given to column sets that are more likely to appear in the future
  - Storage remains under a certain budget

Sample Selection

- Goal: Select one or more samples (either uniform or stratified) at runtime to meet time/error constraints for query Q of the appropriate size
  - Uniform or stratified: depends on set of columns in Q, selectivity of Q, and data placement, complexity

- Two steps:
  - Select sample type
  - Select sample size

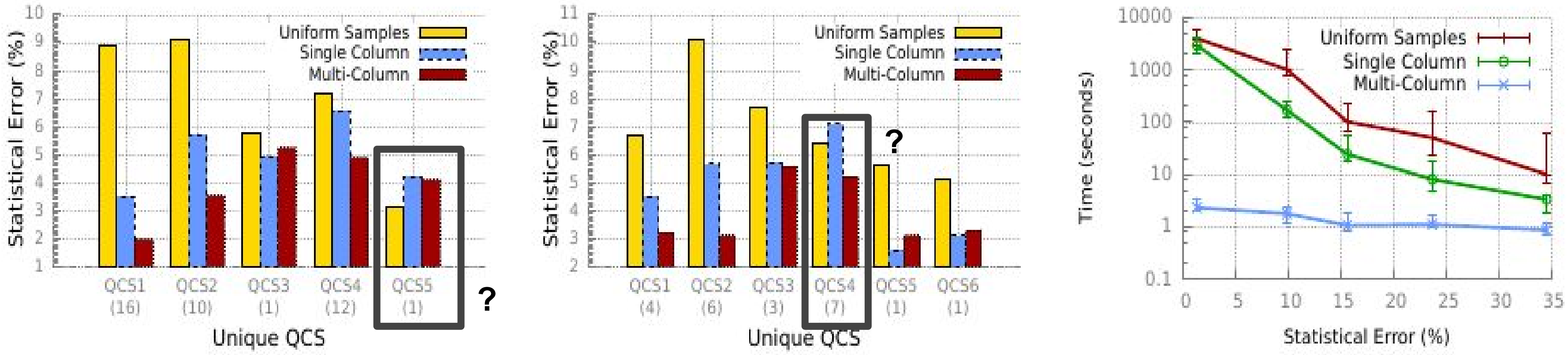
Sample Selection: Sample Type (1)

- Pick stratified sample that contains the necessary QSC if possible
- If no stratified sample contains the necessary QSC, compute Q in parallel on in-memory subsets of all computed samples. Pick samples that have high selectivity (ratio of columns selected to columns read)
    - High selectivity means better lower error margins

Sample Selection: Sample Size (2)

- ELP captures rate at which error/sample rate decreases/increases with increasing sample sizes
- Error Profile: Determine smallest sample size such that the error constraints specified are met
    - Collect data on query selectivity, variance, standard deviation by running query on small samples. Extrapolate variance/standard deviation for aggregate functions using closed form formulas (ex: variance proportional to 1/n where n is sampling size). Calculate the minimum number of rows needed to satisfy error constraint.
- Latency Profile: Determine smallest sample size such that the latency constraints specified are met
    - Run on small sample size. Assume that latency scales linearly with size of input

Evaluation sneak-peek



(a) Error Comparison (Conviva) (b) Error Comparison (TPC-H) (c) Error Convergence (Conviva)

Figure 9. 9(a) and 9(b) compare the average statistical error per QCS when running a query with fixed time budget of 10 seconds for various sets of samples. 9(c) compares the rates of error convergence with respect to time for various sets of samples.

Limitations & Future Work

- Query set seems actually quite limited (in the paper). What about joins and UDFs? How do you get error estimates in this case?
- What exactly is the importance of those rare tuples for applications?
- Is there a way to account for the initial variance in the data itself and “bias” sampling in that way?
- Pre-computed samples are all of the same size
- What is the effect of sampling on the results of more complex queries (ex: joins)?
- What happens when data changes? Consistency?