BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal , Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica
Presented by Shreya Shankar for INFO 290T Fall 2023

The Next Step in Querying Large Datasets?

When tables can't fit in memory (e.g., 100s of millions of tuples satisfy a predicate in a SQL query), queries cannot have interactive latencies

```
SELECT AVG(SessionTime) FROM Sessions
WHERE City = 'New York'
```

How to execute such queries accurately *within seconds*?

Demo: https://youtu.be/6_IFUAJxm0U?si=yZVsxZyba_KC0cok (4:06)

Existing Ways to Query Large Datasets

General

Efficient

OLA: variable performance, can't provide error bars

Sketching & Sampling: low space & time complexity, but can't do queries outside the workload

Existing Ways to Query Large Datasets

- Approximation techniques rely on sampling
- Existing approximation techniques either:
  - OLA: Make no assumptions about workloads — this gives inaccurate answers for groups with little support

    ```
    SELECT AVG(SessionTime) FROM Sessions WHERE City =
    SOME_TINY_TOWN
    ```

  - Make very strict assumptions about workloads — this doesn't support queries that aren't encompassed by the workload

    ```
    SELECT AVG(SessionTime) FROM Sessions WHERE City = “New
    York City” OR City = “San Francisco” …
    ```

Target Workload

- Ad-hoc queries with real-time latency
- Columns queried together are pretty stable over time
- Skewed & high-dimensional data
- UDFs?

Blink DB

- Analytics system built on top of Hadoop
- Allows users to trade off accuracy for response time

```
SELECT COUNT(*) FROM Sessions WHERE
Genre = 'western' GROUP BY OS ERROR
WITHIN 10% AT CONFIDENCE 95%

SELECT COUNT(*) FROM Sessions WHERE
Genre = 'western' GROUP BY OS WITHIN 5
SECONDS
```

- Step 1: Offline *sample creation* step
- Step 2: Online *sample selection* step

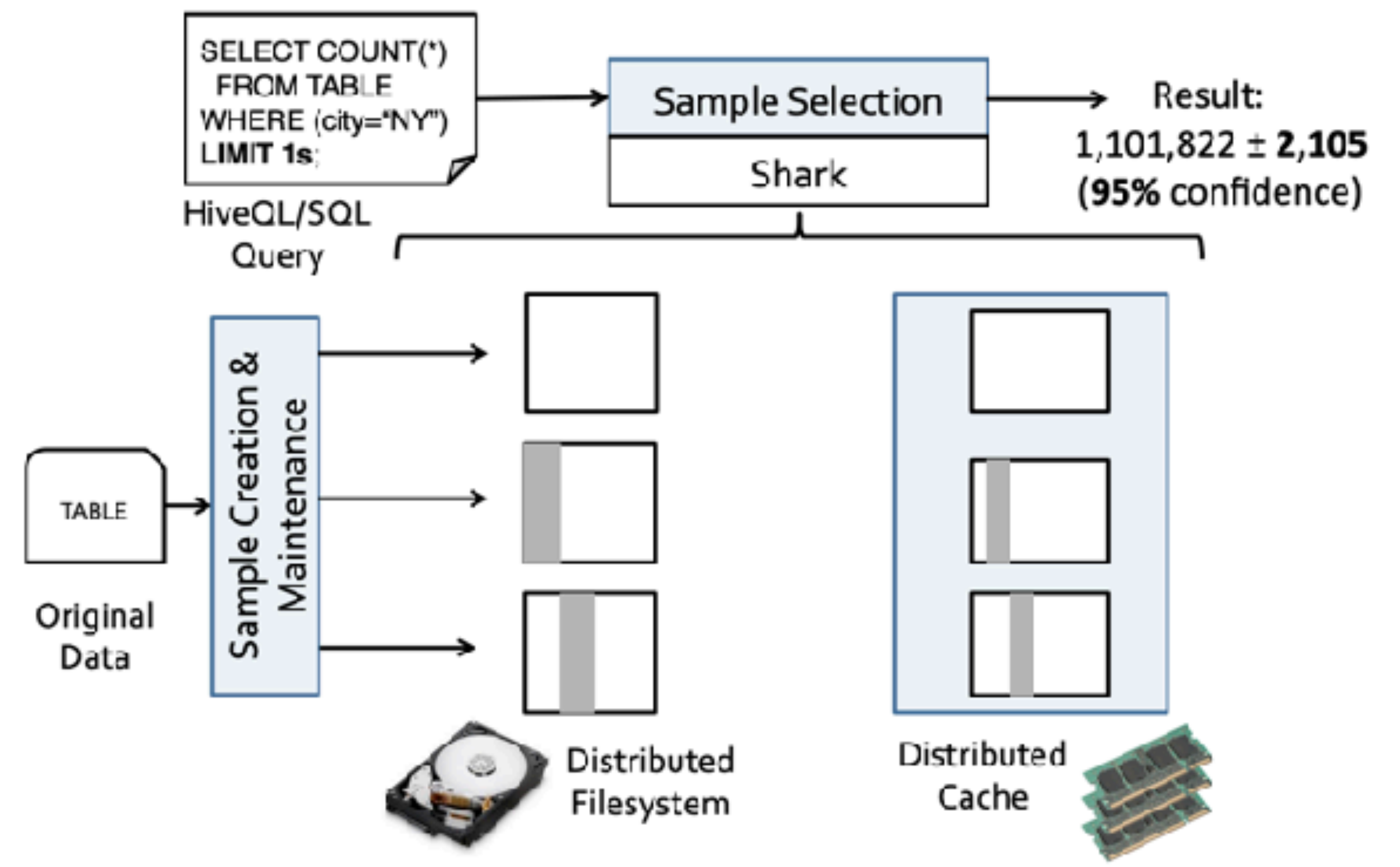


Figure 3. BlinkDB architecture.

Step 1: Sample Creation

- Given a workload of queries, how do we choose which samples of data to store?
    - Stratified sampling: sample from each subset or subgroup of data, to cover rare subgroups
    - Bounded storage overhead
    - Don’t overfit to historical queries
- Solve an optimization problem to find which *sets of columns* to build stratified samples on

Step 1: Sample Creation

- Predictable Queries, Predictable Query Predicates, Predictable Query Column Sets (QCS), Unpredictable Queries

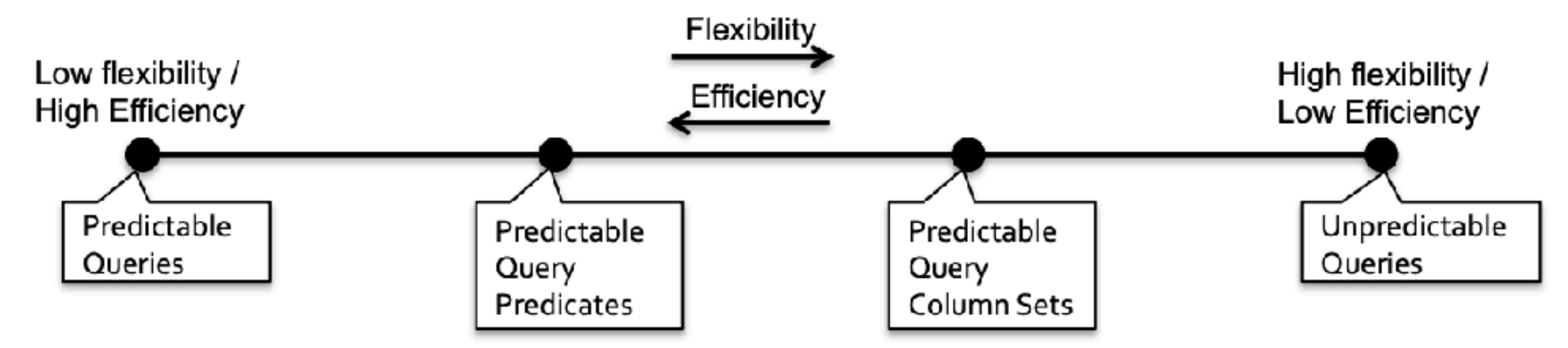


Figure 1. Taxonomy of workload models.

- "Surprisingly, over 90% of queries are covered by 10% and 20% of unique QCSs in the traces from Conviva and Facebook respectively"

Step 1: Sample Creation

- Predictable Queries, Predictable Query Predicates, Predictable Query Column Sets (QCS), Unpredictable Queries

```
SELECT AVG(Salary) WHERE City = “New York”
```

Q: whole query

QP: City = “New York”

QCS: City

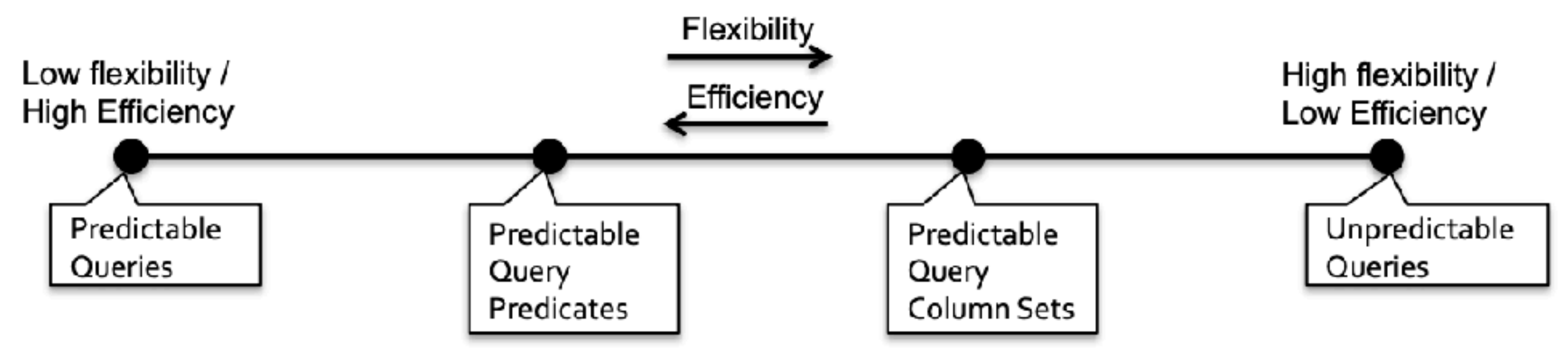


Figure 1. Taxonomy of workload models.

Step 1: Sample Creation

- Stratified sampling: allows us to ensure that rare groups are appropriately represented in the sample
- Approach idea
  - Find all distinct groups & compute their counts
  - Sample uniformly (with a cap) from within each group

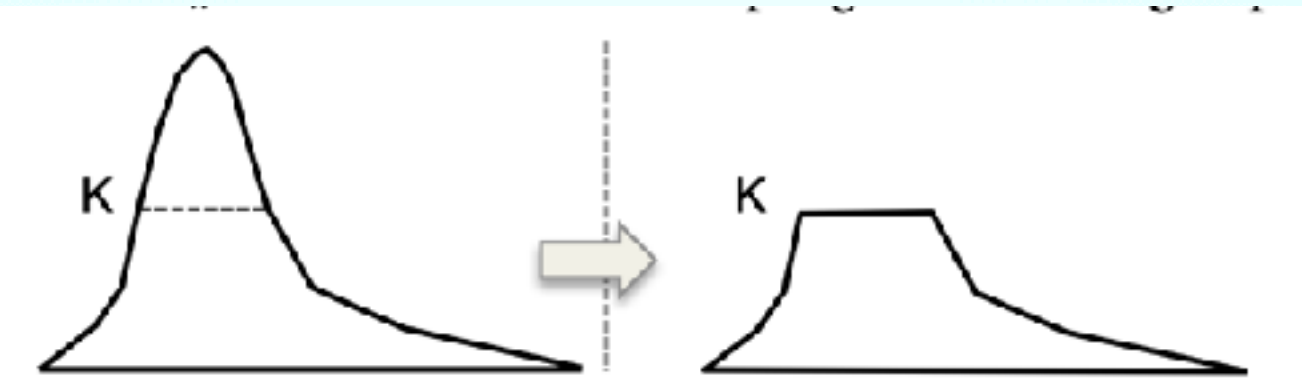


Figure 4. Example of a stratified sample associated with a set of columns, ϕ.

Step 1: Sample Creation

- Stratified sampling: allows us to ensure that rare groups are appropriately represented in the sample
- *n* columns, 2^n samples

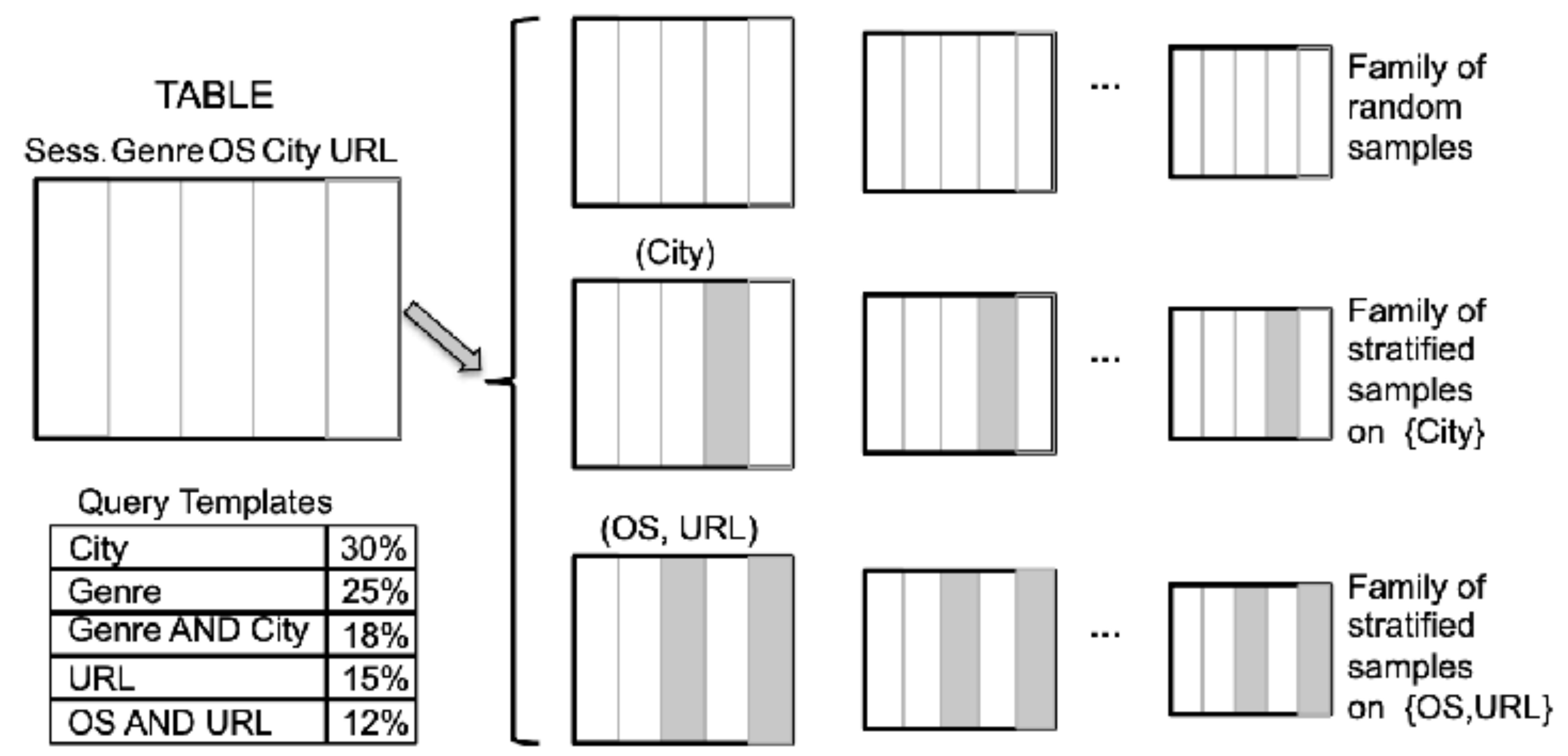


Figure 2: An example showing the samples for a table with five columns, and a given query workload.

Step 1: Sample Creation

- Optimization problem: create samples for a set of queries that share QCS
- Why is this hard?
  - For each query, we read a variable # rows (n) to satisfy user bounds
  - Thus we need access to a *family* of stratified samples, one for each possible value of *n*
- Solution: choose set of samples that prioritize sparsity, data distribution (QCS likely to appear in future), and storage costs

Step 2: Sample Selection

- Given our samples created from step 1, whenever we receive a query Q, which samples do we use to evaluate Q?
- Depends on:
  - Set of columns in Q: pick stratified sample that has a superset of columns in Q if possible, else
  - Selectivity of Q: pick sample(s) with high selectivity (i.e., number of rows in the selection / number of rows in the sample is high). This lowers the error margin.

Step 2: Sample Selection

- Given the samples, what's the *smallest* size we can read to meet user constraints on latency or accuracy?
- Error Latency Profile (ELP): estimates error and response time on each sample:
  - Using very small samples, collect data on query selectivity, variance, standard deviation, etc.
  - Assume latency scales linearly with size of input
  - Assume variance is proportional to 1/n (sample size)

Implementation and Evaluation

- Used Conviva (17TB) and TPC-H (1TB) datasets
- 100 EC2 instances, each with 8 cores, 68 GB RAM, & 800 GB disk
- Query log = 19k queries. Did offline sample creation on 200 queries

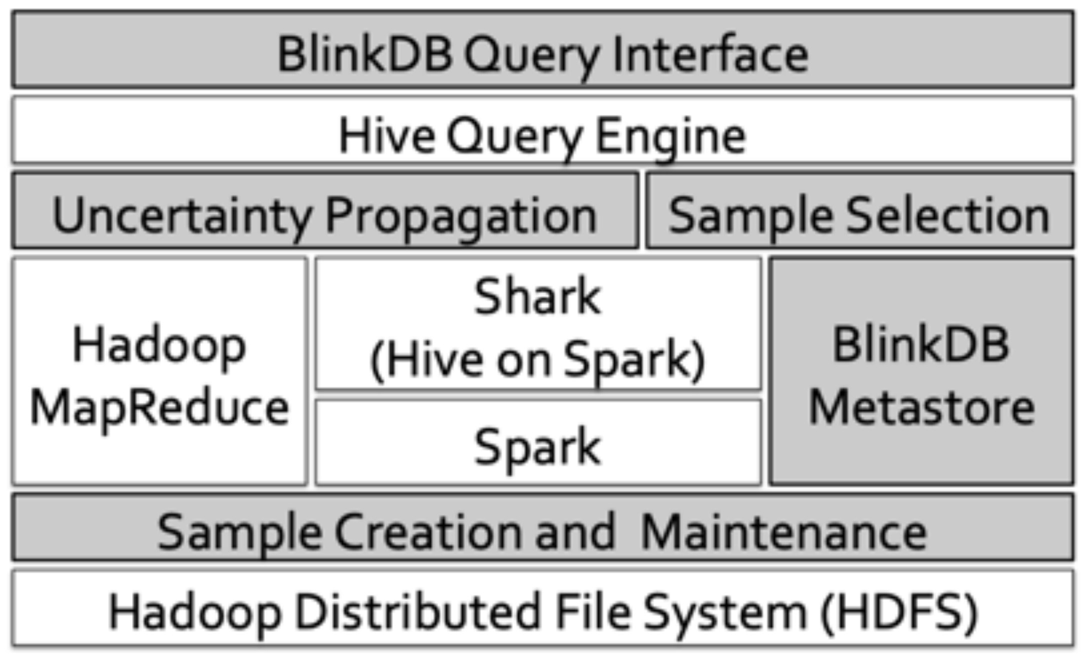
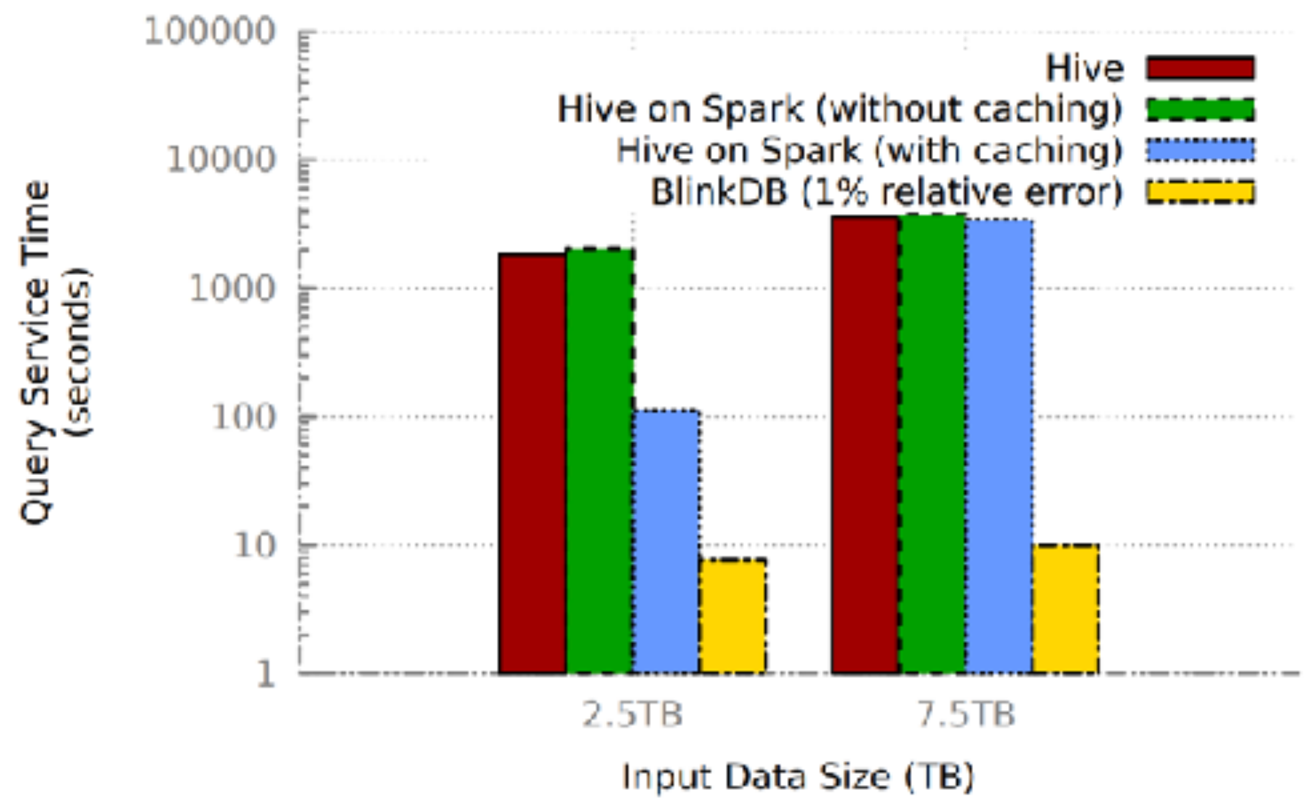


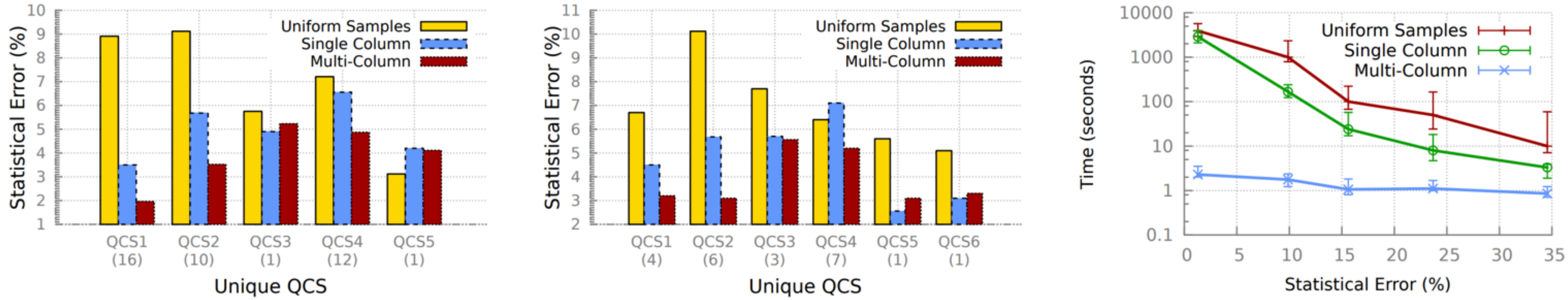
Figure 7. BlinkDB's Implementation Stack



(c) BlinkDB Vs. No Sampling

Multi-Column Stratifying

- Restricted sample selection optimization problem to stratify on no more than 3 columns (for solver speed)
- Using multi-column samples allows us to execute queries faster



(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.

Future Work

- How to support aggregation functions beyond COUNT, SUM, QUANTILE, AVG and complex queries like JOINs? UDFs?
- How to support this when the underlying data changes (i.e., new tuples are added to the dataset)?