# CS422
# Database systems

## Today: Data streams & AQP

Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

Slides adapted from presentations of the Berkeley/MIT team

# Overview

## Previous weeks

- Big Data infrastructures & architectural choices

## This week

- Data stream processing
- Approximate query processing

# Data streams management

- Traditional DBMS – data stored in finite, persistent data sets
- Data Streams – distributed, continuous, unbounded, rapid, time varying, noisy, …
- Data-Stream Management – variety of applications

- Real-time network analytics
- Network security
- Traffic engineering
- Sensor networks
- Financial applications
- Telecom call-detail records
- Web logs and clickstreams
- Manufacturing processes
- …

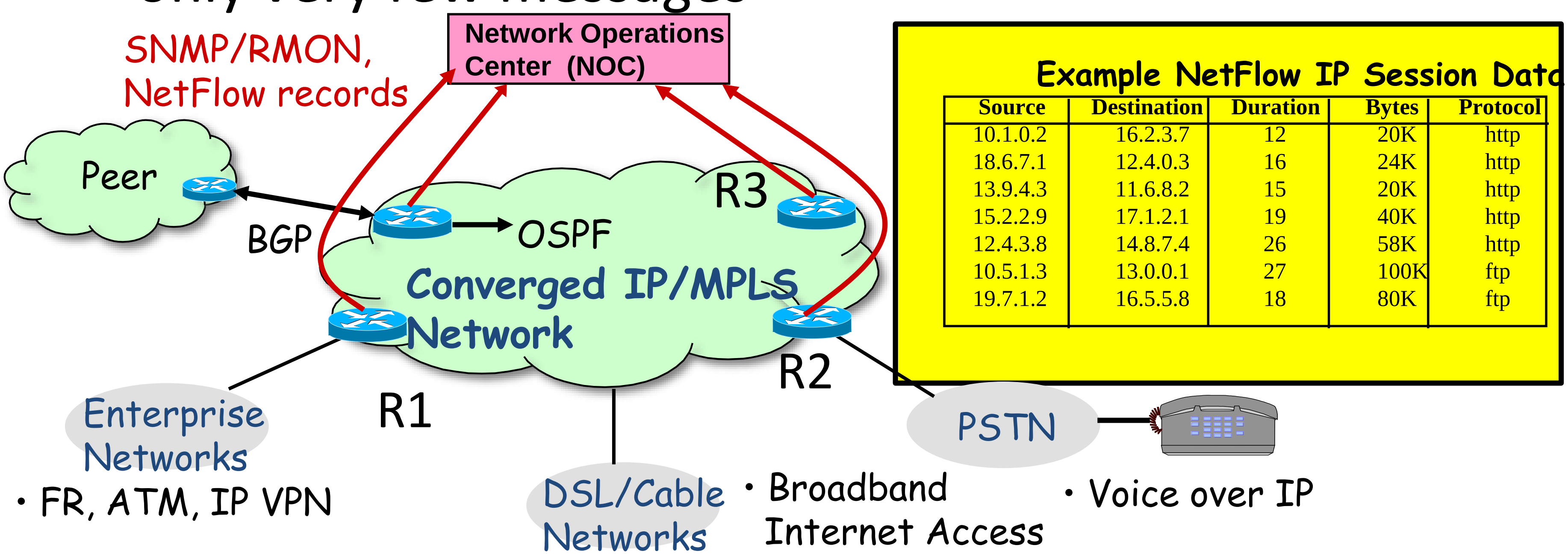
# Example: stock monitoring

- Notify me when some stock price increases by at least 5% in two seconds.
- Find the top (most-frequent) 10 traded stocks in the last 10 minutes
- Notify me when correlation of two stocks over the last 10 minutes exceeds 0.6

| Time (sec) | MSFT | APPL | WMT |
|---|---|---|---|
| 0 | 65.06 | 121.35 | 66.74 |
| 1 | 65.06 | 121.36 | 66.75 |
| 2 | 65.06 | 121.36 | 66.73 |
| 3 | 65.07 | 121.36 | 66.72 |
| 4 | 65.08 | 121.36 | 66.72 |
| 5 | 65.07 | 121.35 | 66.71 |
| … | … | … | … |

# Example: Real-time netw. analytics

- Which are the top (most frequent) 1000 (source, dest) ip pairs seen by R1 over the last month
- How many distinct (source,dest) pairs have been seen by R1 and R2 but not R3
- Which IP addresses receive a lot of data but send only very few messages




| Source | Destination | Duration | Bytes | Protocol |
|---|---|---|---|---|
| 10.1.0.2 | 16.2.3.7 | 12 | 20K | http |
| 18.6.7.1 | 12.4.0.3 | 16 | 24K | http |
| 13.9.4.3 | 11.6.8.2 | 15 | 20K | http |
| 15.2.2.9 | 17.1.2.1 | 19 | 40K | http |
| 12.4.3.8 | 14.8.7.4 | 26 | 58K | http |
| 10.5.1.3 | 13.0.0.1 | 27 | 100K | ftp |
| 19.7.1.2 | 16.5.5.8 | 18 | 80K | ftp |

# Example: Real-time netw. analytics (2)

Where is the challenge -- assume **packet-level** statistics

- Single 2Gb/sec link;  say avg packet size is 50 bytes
- Number of packets/sec = 5 million
- If we only capture header information per packet: src/dest IP, time, no. of bytes, etc. – at least 10 bytes.
  - Space per second is 50MB
  - Space per day is 4.5TB per link
  - ISPs typically have hundreds of links!

- Deep packet analysis – whole new ballgame!!

# Databases Vs Data Streams

| | |
|---|---|
| A relation is a set of tuples | A stream is a bag of tuples with partial order |
| Relations are persistent | Streams need to be processed in real time as tuples arrive |
| **Interactive** queries | **Continuous** queries |
| Random access to data, queries need to be processed as they arrive | Sequential access to data, random access to continuous queries |
| Physical database design does not change during query, queries can be unpredictable | Queries do not change, stream can be very unpredictable |

Slide based on material from Jennifer Widom.

# Overview

- Introduction & motivation
- **Data Stream Model**
- The data facet
- The time facet

# The Data Stream Model

- Underlying signal: One-dimensional array A[1…N] with values A[i] all initially zero
  - Multi-dimensional arrays also possible
- Signal implicitly represented via a stream of updates
  - j-th update is <k, c[j]> implying
    - A[k]=A[k]+c[j]   (count c[j] can be >0 or <0)
- Goal: Compute functions on A[] subject to
  - Small space
  - Fast processing of updates
  - Fast function computation

# The Data Stream Model – example

- Underlying signal: One-dimensional array A[1…N]
- Signal implicitly represented via a stream of updates

Signal A

| Position | #packets |
|---|---|
| … | … |
| 2159214850 (corresponds to 128.179.1.2) | 13 |
| 2159214851 | 5442 |
| 2159214852 | 0 |
| … | … |
| … | … |
| … | … |

2159214850

Map IP to integer

Stream of updates

| | | | |
|---|---|---|---|
| <128.179.1.2,+1> | … | … | …. |

# The Data Stream Model – special cases

- Cash-register model
  - c[j] is always >=0 (increment-only)
  - In many cases, c[j]=1
- Turnstile model
  - Most general streaming model
  - c[j] can be positive or negative (increment or decrement)
- Time-series model
  - j-th update updates A[j] (i.e., A[j]=c[j])

# The Data Stream Model – special cases

- Cash-register model
- Turnstile model
- Time-series model

- Difficulty varies depending on the model & problem
  - E.g., min/max in time-series Vs turnstile

# The two facets of the problem

**DATA**

- Too much data
  - Velocity
  - Dimensions

**TIME**

- Interested only for parts of the stream
- Need to expire old data

Solutions of small space/small computational complexity

# The facet of data

Stream of items:

| Source | Destination | Time | Protocol | Data |
|---|---|---|---|---|
| 10.1.0.2 | 16.2.3.7 | 1992191 | http | ….. |

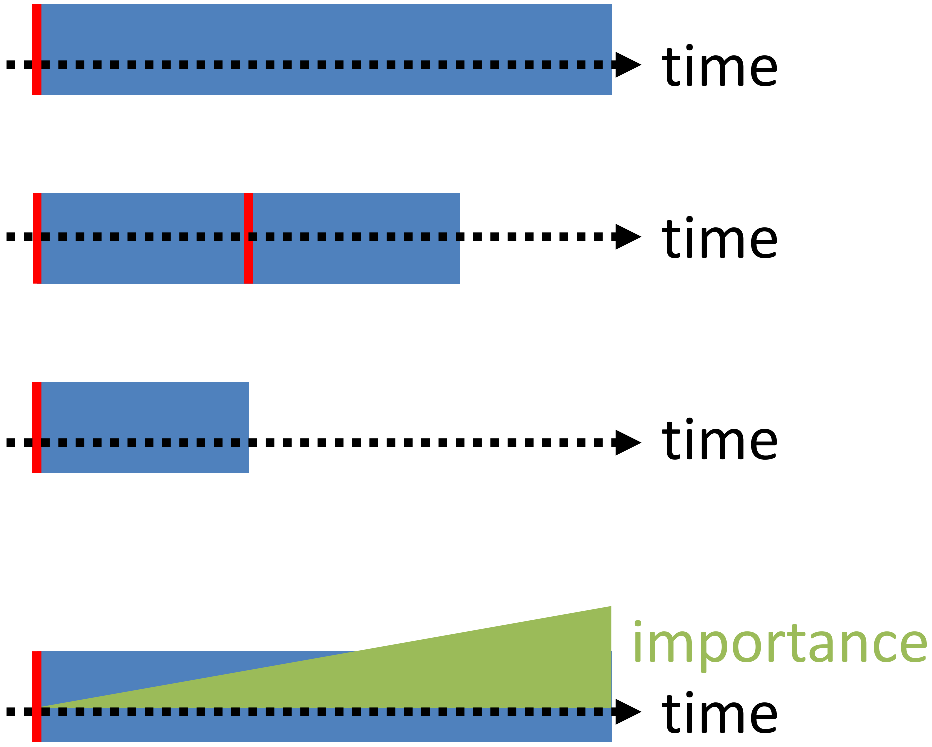
IP network signals

- Number of active flows per source-IP address
  - $2^{32}$ sized array: increment & decrement (~16 GB)
- Number of packets exchanged between any two IP addresses **during the day**
  - $2^{64}$ sized array: increment only (64 EB!)
- Number of packets sent by each IP **in the last hour** (???)
  - $2^{32}$ sized array, sliding window

# The facet of time

## Queries: Continuous

- Defining the query range
  - Landmark window
  - Fixed windows
  - Sliding window
  - Decay model



# The facet of time (2)

## Measuring time

- Wall-clock time (time-based definition)
  - All updates since 13:32:00
  - [current time – 100 seconds, current time]
  - Event time vs processing time

- Number of updates (event-based definition)
  - Last 1000 updates
  - Also called count-based, arrival-based

# Two choices to handle streams

- Scale-out to handle real-time requirements
  - Add more machines
  - Programming model to ease
    - Expressing user requirements
    - Distributing the stream and computations

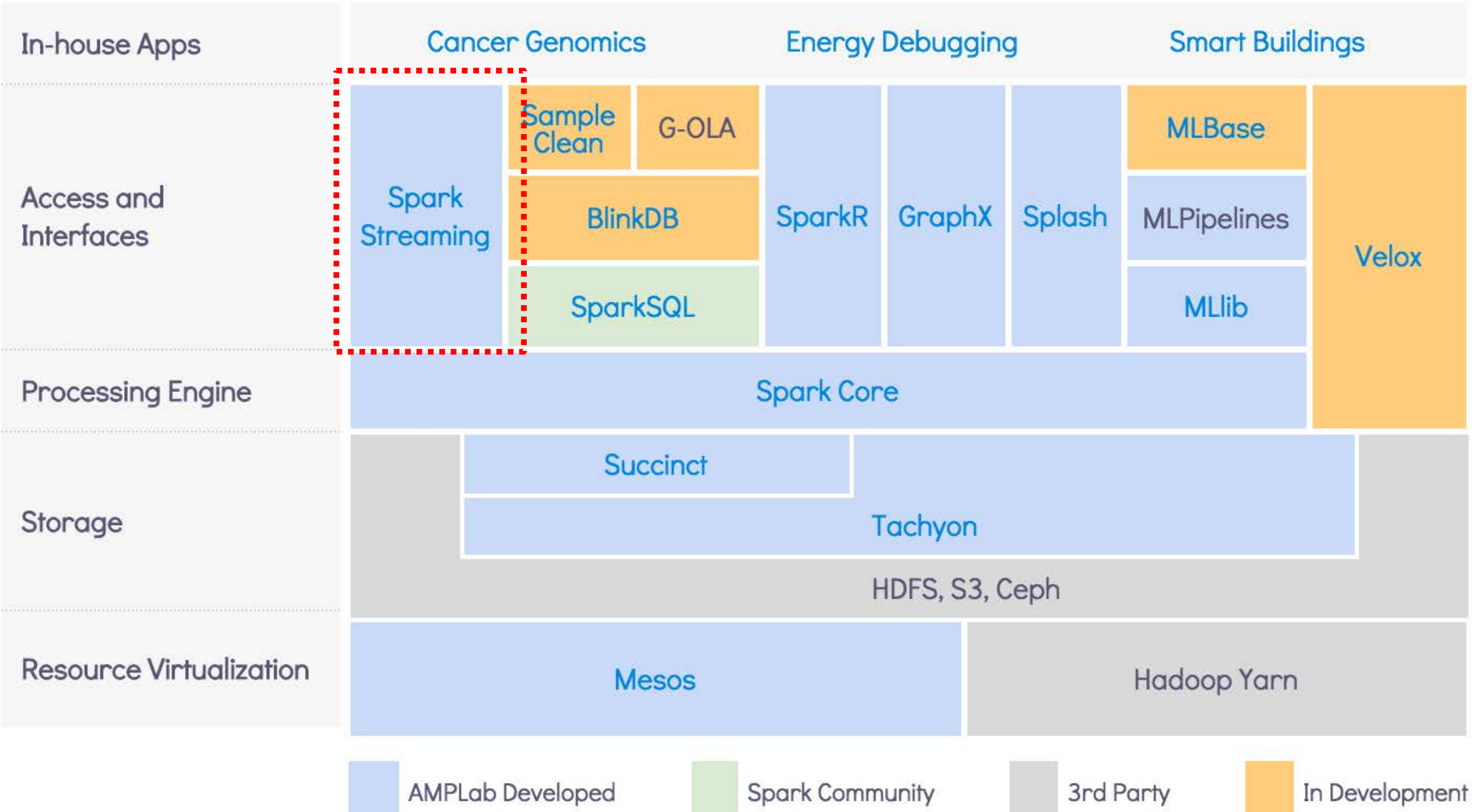
- The *poor man's* approach*
  - Summarize the stream
  - Approximate user requirements

Sometimes, combination of both!

# Scaling-out platforms

- Necessary when exact results are needed
  - Banks, medical sensors, Industry 4.0
- Typically comes with a hefty price tag
- Several platforms
  - Spark Streaming
  - Twitter: Storm → Heron
  - Apache Flink
  - Apache Kafka

# The Spark Unified Stack



# Spark Streaming

- Up to now
  - Data pre-existed in DFS
  - Analyzed in batch

One-shot queries over stored data

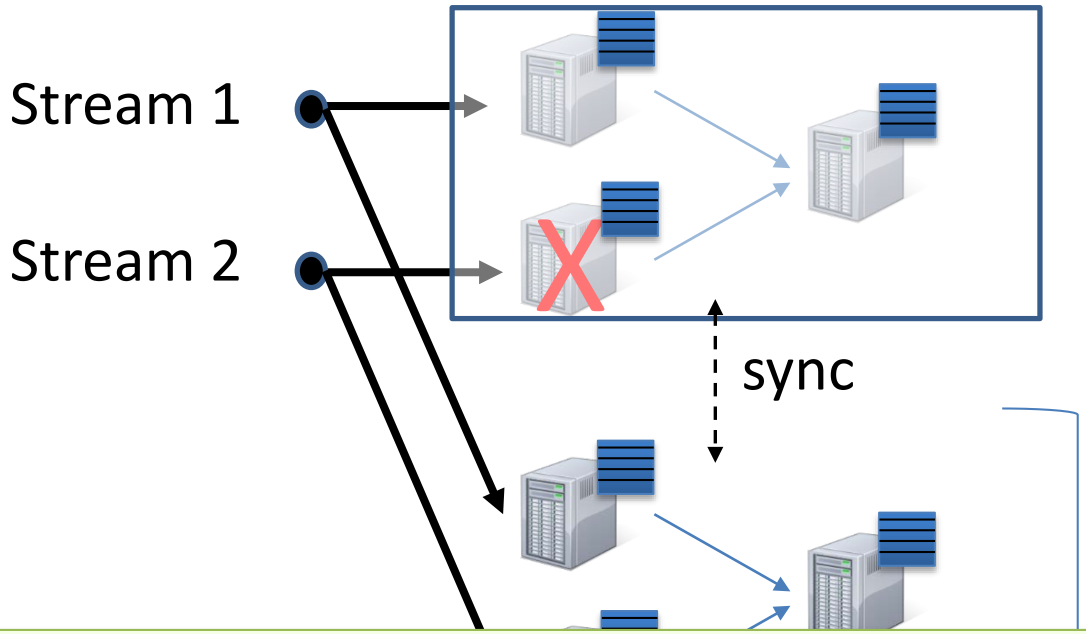
- Real-time analysis of big data
  - Process data *as soon as* it arrives, take action immediately
  - Continuous queries

# Expectations from the platform

- Programming model
  - Utilize commodity clusters for scalable processing
- Fault tolerance
  - Must recover from failures and stragglers quickly and efficiently
  - Imagine the VISA fraud detection system is down!
- It was *almost easy* in MapReduce and Spark! Why is it so difficult for streams?

# Fault tolerance for SP systems

- Replication
  - Replicate processing of each input to two or more nodes
  - When a node fails, the backup node takes over




Recover!

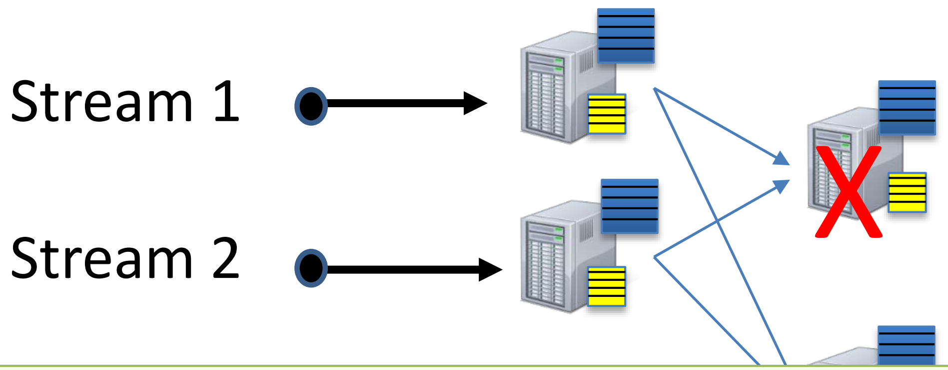
Hot failover nodes

**Fast recovery**

**Very expensive!**

# Fault tolerance for SP systems

- Upstream backup
  - Nodes maintain checkpoints (**safely**) and backups of updates forwarded after the checkpoints
  - When a node fails, the backup node takes over
    - Replay input stream after last surviving checkpoint

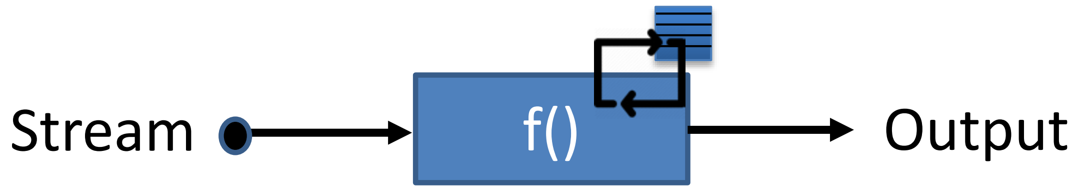



Fairly cheap – minimal additional hw

Slow recovery

# Fault tolerance for SP systems

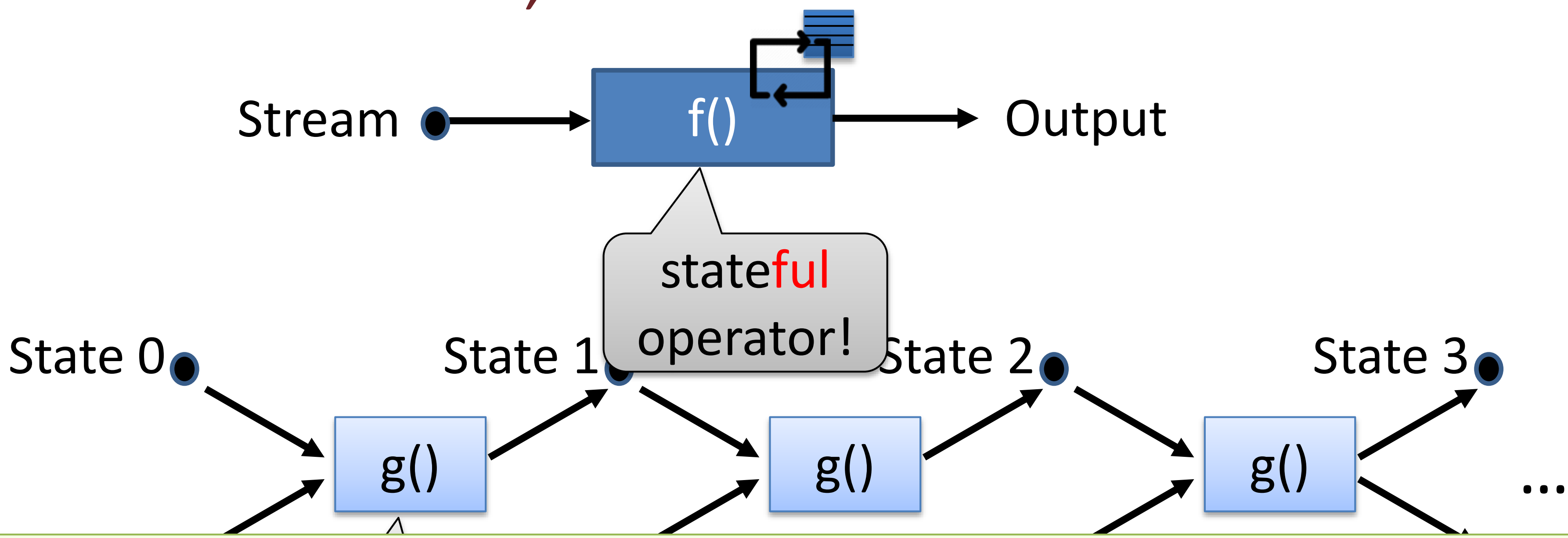
- Replication
  - Fast recovery
  - Expensive on resources – at least x2 nodes
- Upstream backup
  - Cheap – only a few additional nodes
  - Time-consuming recovery




Computation is closely coupled with state

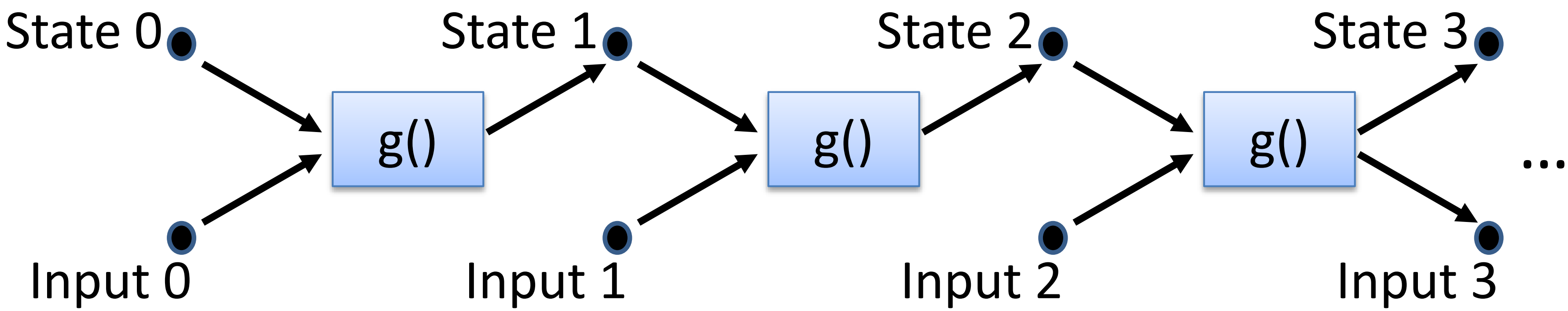
# A new fault tolerance technique for SP

- State: mutable → immutable
- Partition continuous computation into small, deterministic, stateless tasks




We have decoupled computation from state!

# Does this ring a bell?

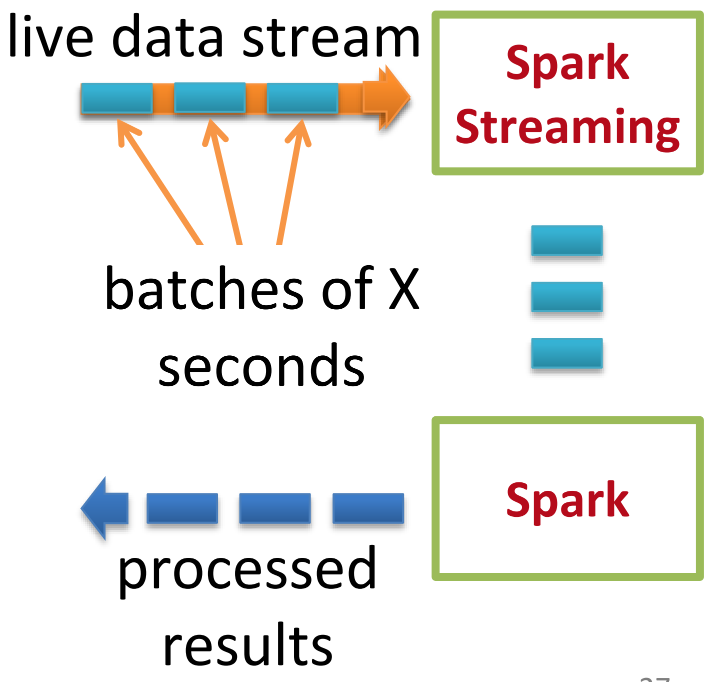



- Stateless operators → static batch model
  - MapReduce, Spark, …
  - Already advanced in fault tolerance!

# Discretized stream processing

- Consider stream in SMALL input batches
  - Micro-batches
- Streaming computation is a series of very small deterministic batch jobs

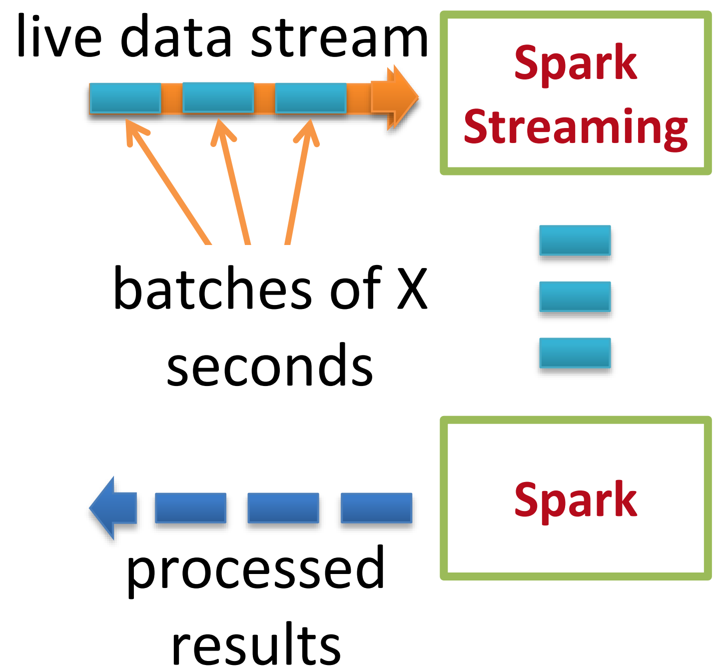
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



# Discretized stream processing

- Consider stream in SMALL input batches
  - Micro-batches
- Streaming computation is a series of very small deterministic batch jobs

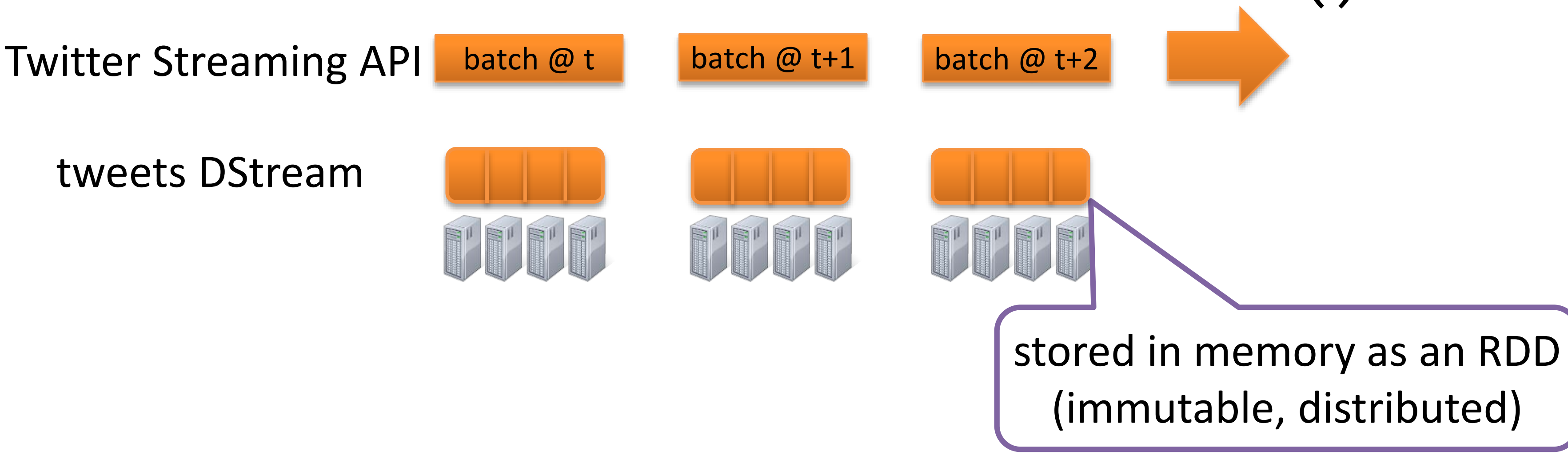
- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



# DStreams

- Discretized stream (DStream) is a sequence of immutable, partitioned datasets
  - a sequence of RDDs!
- Can be created from live data streams (e.g., twitter, network sockets, …) or by applying bulk parallel transformations on other DStreams

```
val tweets = ssc.twitterStream()
```



# Example – streaming word count

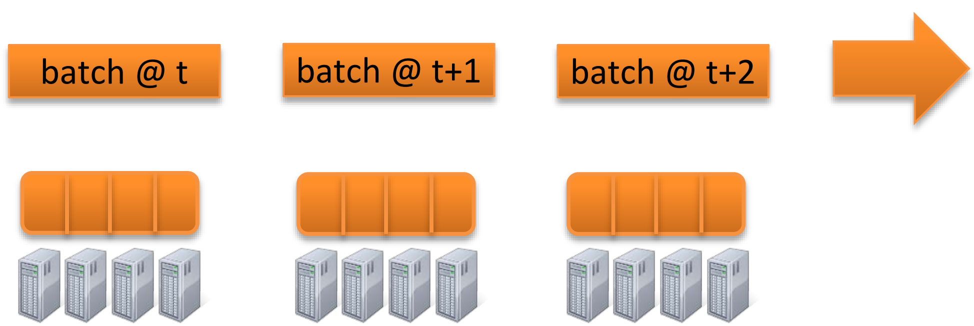
```
val ssc = new StreamingContext(…, Seconds(5))
```

**streaming** content

size of each batch

Twitter Streaming API

tweets DStream

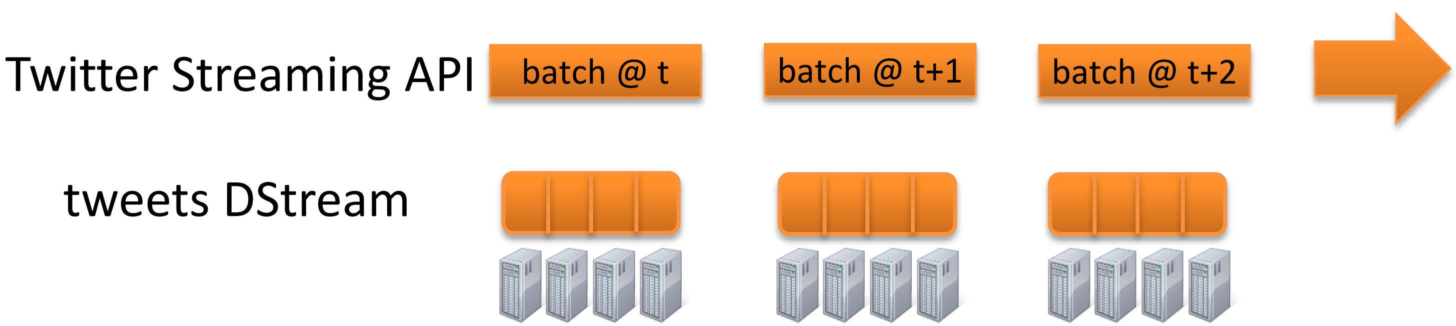


# Example – streaming word count

```
val ssc = new StreamingContext(…, Seconds(5))
val lines = ssc.socketTextStream(url,port)
```

As a DStream

read data from socket



# Example – streaming word count

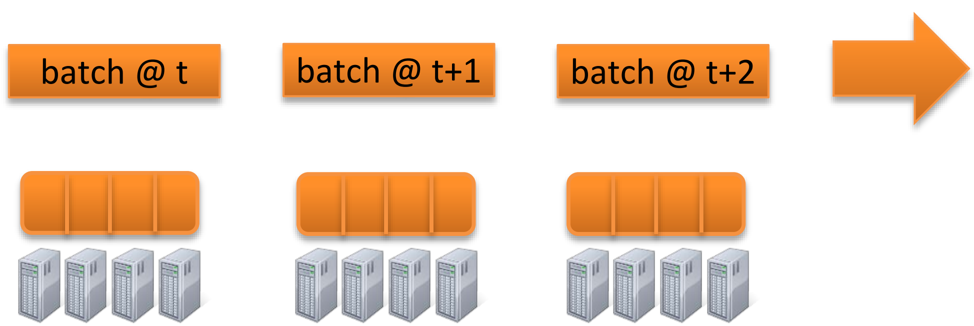
```
val ssc = new StreamingContext(…, Seconds(5))
val lines = ssc.socketTextStream(url,port)
val words = lines.flatMap(_.split(" "))
val wordC = words.map(x => (x, 1)).reduceByKey(_ + _)
```

DStreams

**transformation**: modify data in one DStream to create another DStream

Twitter Streaming API

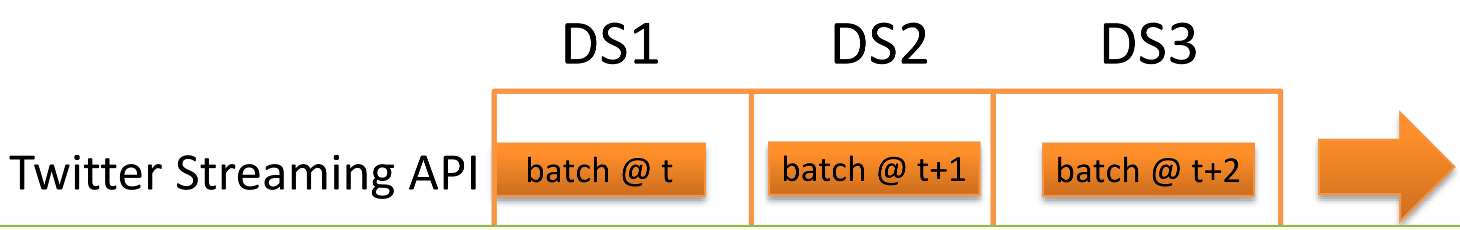
tweets DStream



# Example – SW streaming word count

```
val ssc = new StreamingContext(…, Seconds(5))
val lines = ssc.socketTextStream(url,port)
val words = lines.window(Seconds(10))
                 .flatMap(_.split(" "))
                 .map(x => (x, 1)).reduceByKey(_ + _)
words.print()
ssc.start()
ssc.awaitTermination()
```

operate **on sliding window** of 10 seconds → last 2 batches




Spark Streaming *magically* maintains sliding window operations

# Example – SW streaming word count

- Defining the sliding window

Using window

```
lines.window(Seconds(10))
                 .flatMap(_.split(" "))
                 .map(x => (x, 1)).reduceByKey(_ + _)
```

Using reduceByKeyAndWindow

```
lines.flatMap(_.split(" ")).map(x => (x,1))
     .reduceByKeyAndWindow( (a,b) => (a + b), Seconds(10) )
```

… and teaching Spark how to expire batches

```
lines.flatMap(_.split(" ")).map(x => (x,1))
```

What is the benefit of the last approach?

# Arbitrary Combinations of Batch and Streaming Computations

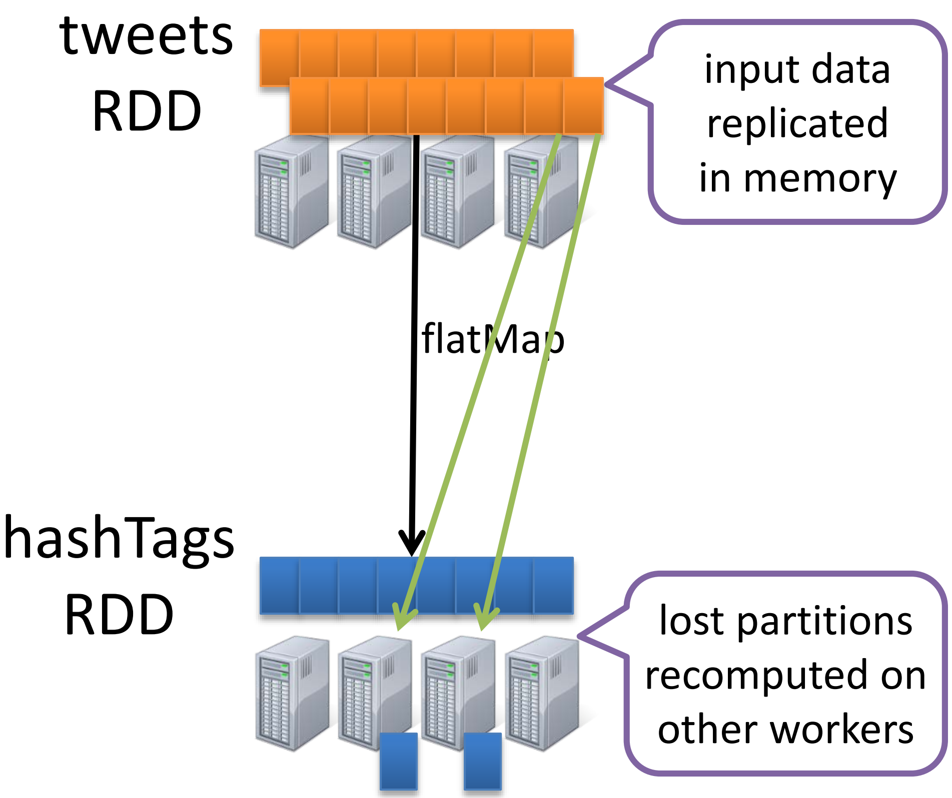
Inter-mix RDD and DStream operations!

Example: Join incoming tweets with a spam filter for data cleaning

```
tweets.transform(tweetsRDD => {
    tweetsRDD.join(spamHDFSFile).filter(...)
})
```

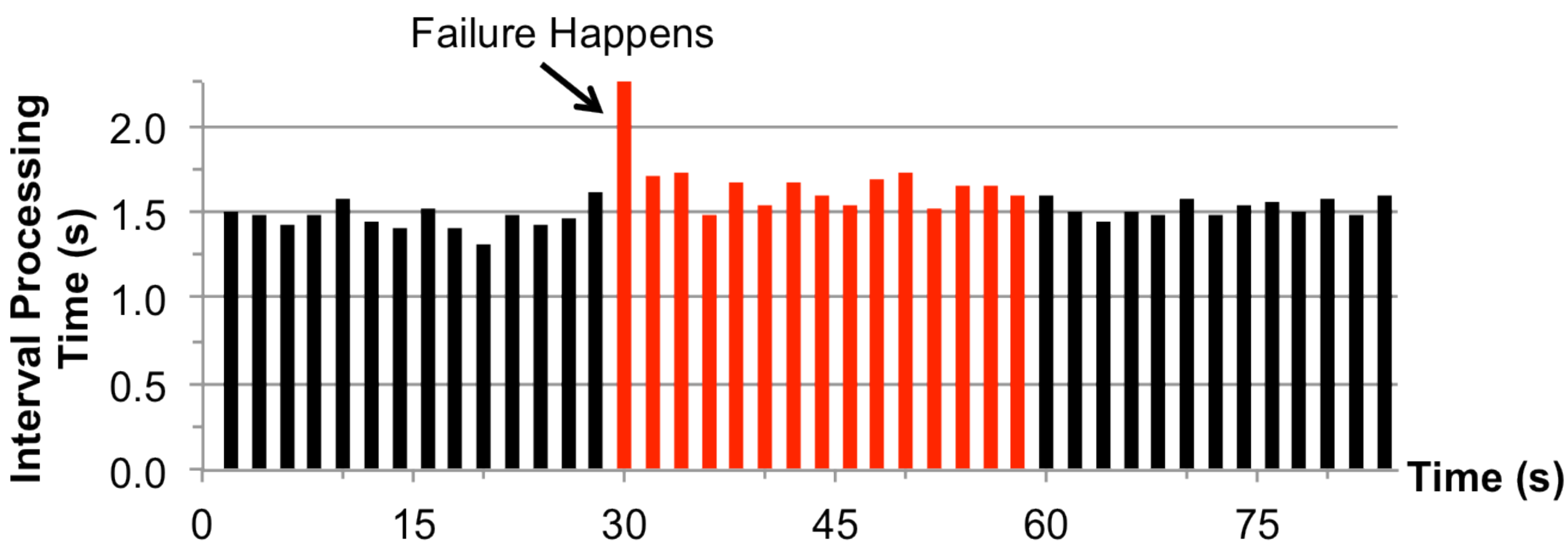
# Fault-tolerance

- RDD Lineage: RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Fault-tolerance and exactly-once transformations



# Fast Fault Recovery

Recovers from faults/stragglers within **1 sec**




Sliding WordCount on 10 nodes with 30s checkpoint interval

# **Unifying** Batch and Stream Processing Models

## Spark program on Twitter log file using RDDs

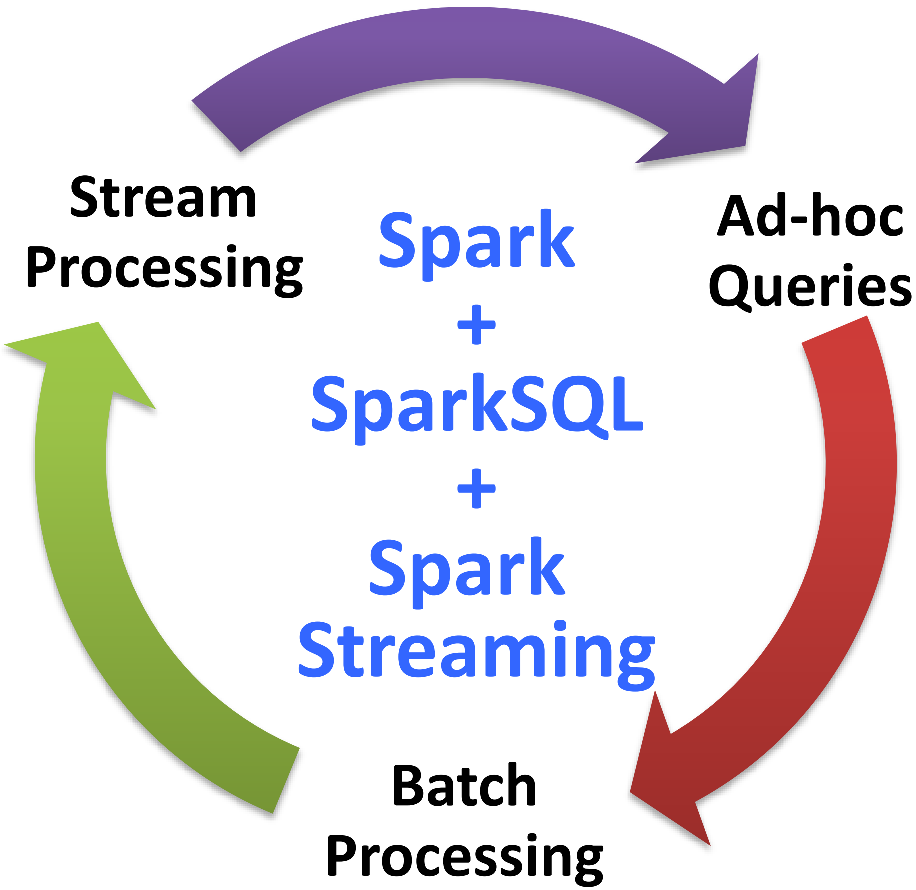
```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

ONLY NEED TO CHANGE THE INPUT!

## Spark Streaming program on Twitter stream using DStreams

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

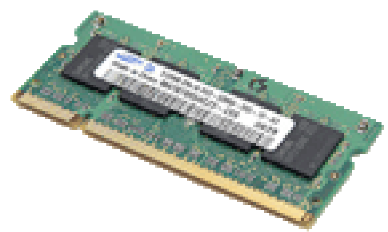
# Vision - *one stack to rule them all*




OTHERSIDE
RED HOT CHILI PEPPERS

# Our progress up to now!

## 100 TB on 1000 machines

| ½ - 1 Hour | 1 - 5 Minutes | 1-5 seconds |
|---|---|---|
|  |  | ? |
| Hard Disks | Memory | |
| |  |  |

# Approximate Query Processing

| | Exact | Approximate |
|---|---|---|
| **Offline** | hadoop Map Reduce<br>APACHE Spark<br>Spark SQL | BlinkDB |
| **Real-time** | Spark Streaming<br>Apache Flink<br>APACHE kafka A distributed streaming platform | Synopses & approx. alg. |

# An example…

- Server log files of several TB

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/Main/Double_...
142.22.55.12 - - [07/Mar/2004:16:06:51 -0800] "GET /twiki/bin/rdiff/TWiki/NewUs…
142.242.63.63 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision…
142.22.55.12 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/TWiki/WikiSy…
111.11.32.65 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/Main/DCCAndP...
112.13.45.99 - - [07/Mar/2004:16:23:12 -0800] "GET /twiki/bin/oops/TWiki/Append…
62.11.1.123 - - [07/Mar/2004:16:24:16 -0800] "GET /twiki/bin/view/Main/PeterTh…
```

…

- How to query these files
  - using good old SQL
    - and get near-interactive responses?
      - approximate results acceptable!!!

# BlinkDB: Blink and it's done

- Target: Support **interactive** SQL-like aggregate queries over massive sets of data by **approximation**
- Interactive: ~ 1-2 seconds
- Strategy: Approximate answers via sampling
- Supports
  - Aggregates & group-by [Paper 1, in exam]
  - Filters [Paper 1, in exam]
  - Joins [Paper 2, not in exam material]
  - User-defined functions (UDFs)! [Paper 2, not in exam material]

Paper 1 (in exam): https://sameeragarwal.github.io/blinkdb_eurosys13.pdf
Paper 2 (optional, will not be examined in midterm or final): https://sameeragarwal.github.io/mod282-agarwal.pdf

# BlinkDB (2)

Supports Aggregates

- blinkdb>

```
SELECT AVG(jobtime)
FROM very_big_log
```

AVG, COUNT, SUM, STDEV, PERCENTILE etc.

# BlinkDB (2)

Supports Aggregates Filters

- blinkdb> SELECT AVG(jobtime)

FROM very_big_log

WHERE src = 'hadoop'

FILTERS, GROUP BY clauses

# BlinkDB (2)

Supports Aggregates Filters

Joins

- blinkdb>

```
SELECT AVG(jobtime)
FROM very_big_log
WHERE src = 'hadoop'
LEFT OUTER JOIN logs2
ON very_big_log.id = logs.id
```

JOINS, Nested Queries etc.

# BlinkDB (2)

Supports

Aggregates | Filters
Joins | UDFs

- blinkdb>

```
SELECT my_function(jobtime)
FROM very_big_log
WHERE src = 'hadoop'
LEFT OUTER JOIN logs2
ON very_big_log.id = logs.id
```

ML Primitives,
User Defined Functions

# BlinkDB (2)

Supports

Aggregates    Filters

Joins    UDFs

Accuracy reqs

- blinkdb>

```
SELECT AVG(jobtime)
FROM very_big_log
WHERE src = ‘hadoop’
LEFT OUTER JOIN logs2
ON very_big_log.id = logs.id
ERROR WITHIN 10% AT CONFIDENCE 95%
```

Desired accuracy

# BlinkDB (2)

Supports

Aggregates
Joins
Accuracy reqs

Filters
UDFs
Performance reqs

- blinkdb>

```
SELECT AVG(jobtime)
FROM very_big_log
WHERE src = ‘hadoop’
LEFT OUTER JOIN logs2
ON very_big_log.id = logs.id
WITHIN 5 SECONDS
```

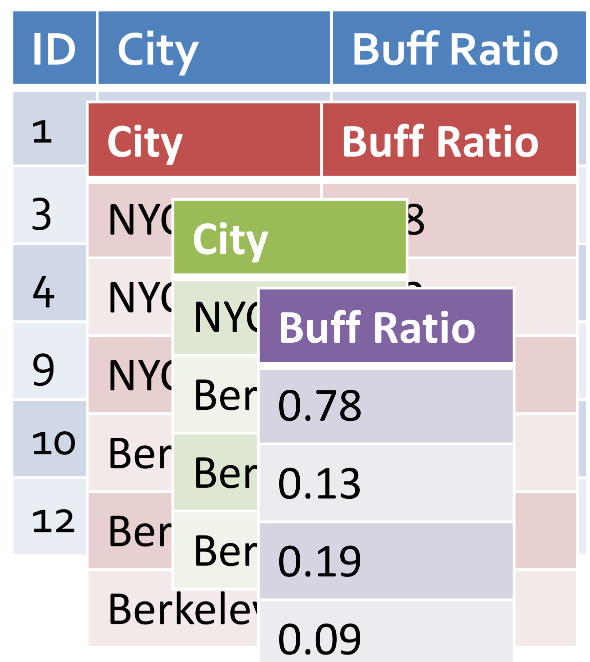
Desired performance

# Workflow of BlinkDB

| ID | City | Buff Ratio |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

Query log analysis:
Q1: SELECT count(*) FROM log WHERE city=NYC
Q2: SELECT AVG(bradio) FROM log WHERE city=NYC
Q3: SELECT AVG(bradio) FROM log GROUP BY city
...

Decide on the samples




...

# Query Execution on Samples

| ID | City | Buff Ratio |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

What is the average <u>buffering ratio</u> in the table?

0.2325

# Query Execution on Samples

| ID | City | Buff Ratio |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

What is the average buffering ratio in the table?

→ Uniform Sample

| ID | City | Buff Ratio | Sampling Rate |
|---|---|---|---|
| 2 | NYC | 0.13 | 1/4 |
| 6 | Berkeley | 0.25 | 1/4 |
| 8 | NYC | 0.19 | 1/4 |

~~0.2325~~

0.19 +/- 0.05

# Query Execution on Samples

| ID | City | Buff Ratio |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

$\longrightarrow$ Uniform Sample

What is the average buffering ratio in the table?

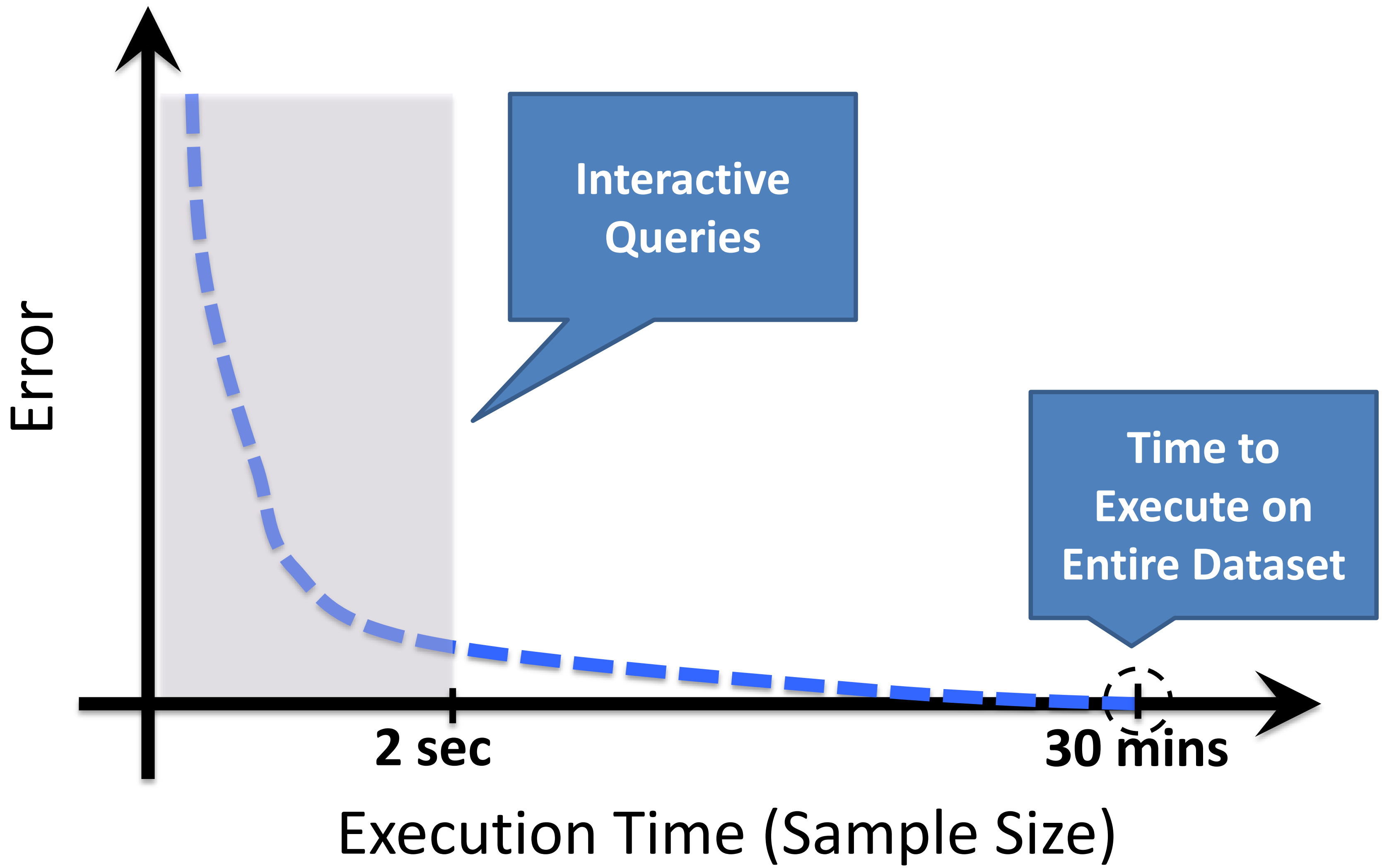
| ID | City | Buff Ratio | Sampling Rate |
|---|---|---|---|
| 2 | NYC | 0.13 | **1/2** |
| 3 | Berkeley | 0.25 | **1/2** |
| 5 | NYC | 0.19 | **1/2** |
| 6 | Berkeley | 0.09 | **1/2** |
| 8 | NYC | 0.18 | **1/2** |
| 12 | Berkeley | 0.49 | **1/2** |

~~0.2325~~

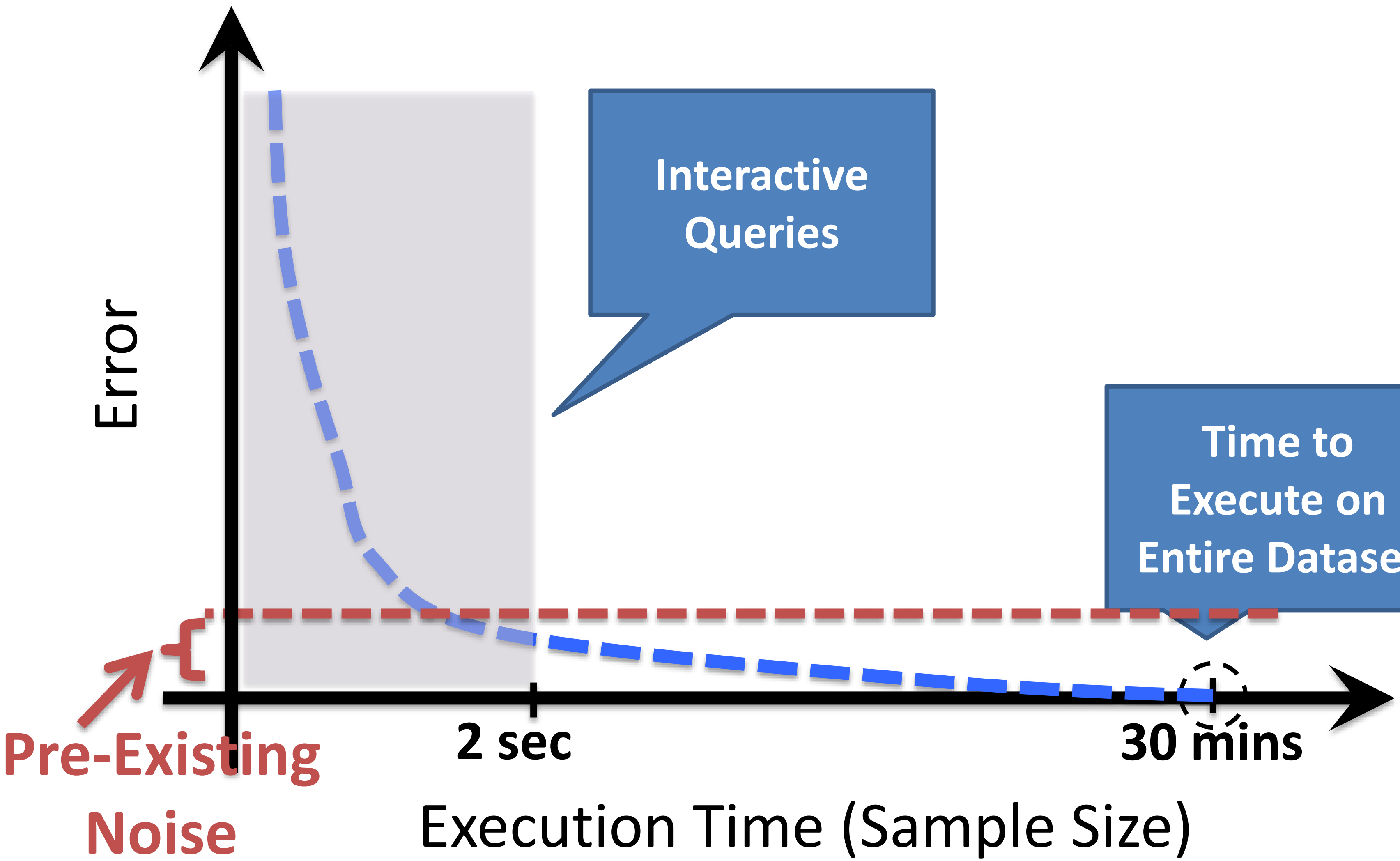
~~0.19 +/- 0.05~~

0.22 +/- 0.02

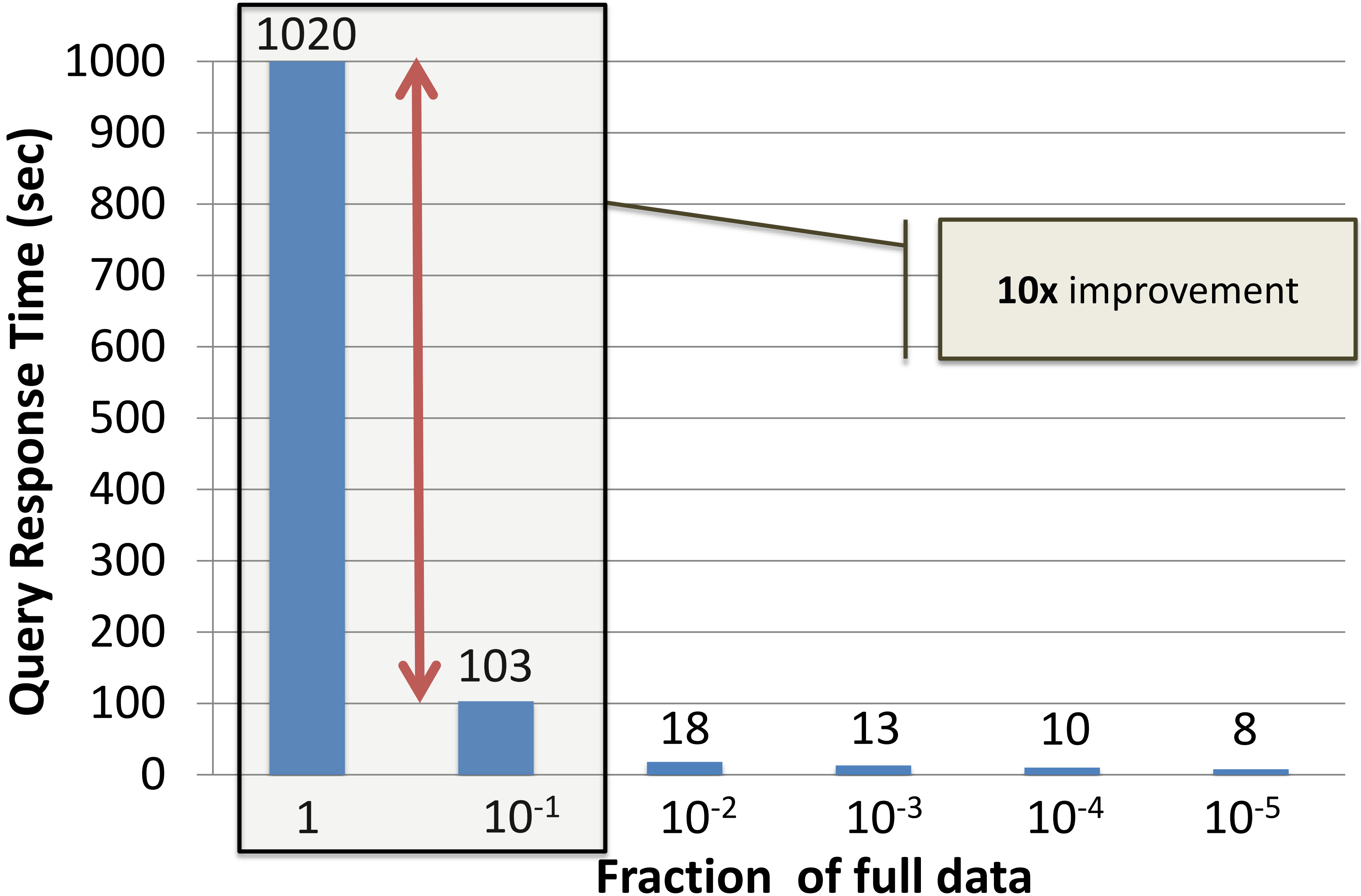
# Speed/Accuracy Trade-off



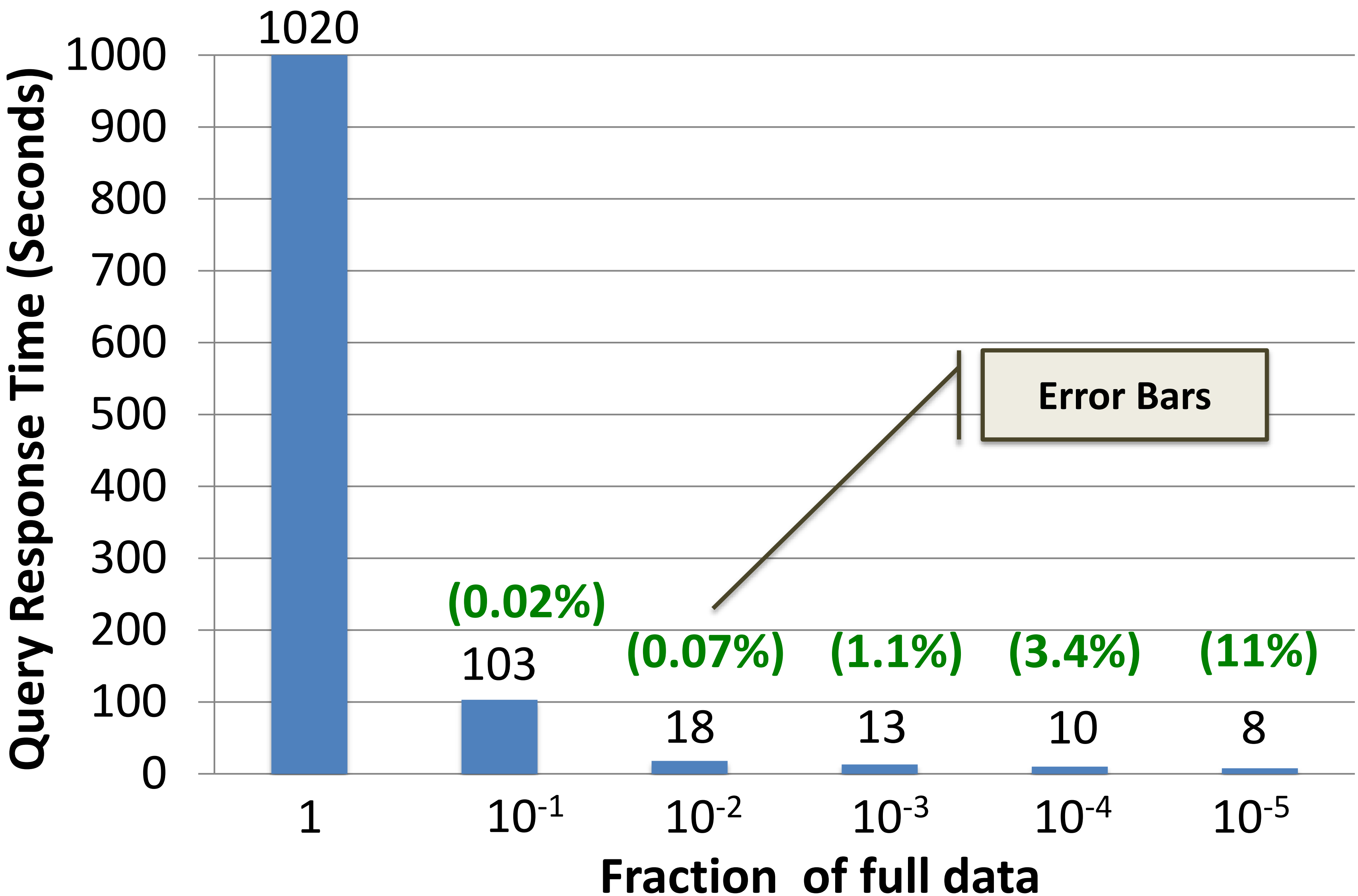
# Speed/Accuracy Trade-off



# Sampling Vs. No Sampling



# Sampling Vs. No Sampling



# What is BlinkDB?

- A framework that …

- creates and maintains a variety of uniform and stratified samples from underlying data
- returns fast, approximate answers with error bars by executing queries on samples of data
- verifies the correctness of the error bars that it returns at runtime

# What is BlinkDB?

- A framework that …
  - creates and maintains a variety of uniform and stratified samples from underlying data
  - returns fast, approximate answers with error bars by executing queries on samples of data
  - verifies the correctness of the error bars that it returns at runtime

# Learning to sample!

- Which types of sample to create?
  - On which columns
  - How many samples
- Typical assumption
  - Interests don't change → let the past queries guide you!

```
SELECT AVG(salary)
FROM tbl1
```

Uniform sample

```
SELECT AVG(salary)
FROM tbl1
WHERE city="London"
```

Uniform sample
for city="London"

☹

```
SELECT
AVG(salary)
FROM tbl1
GROUP BY city,
     profession
```

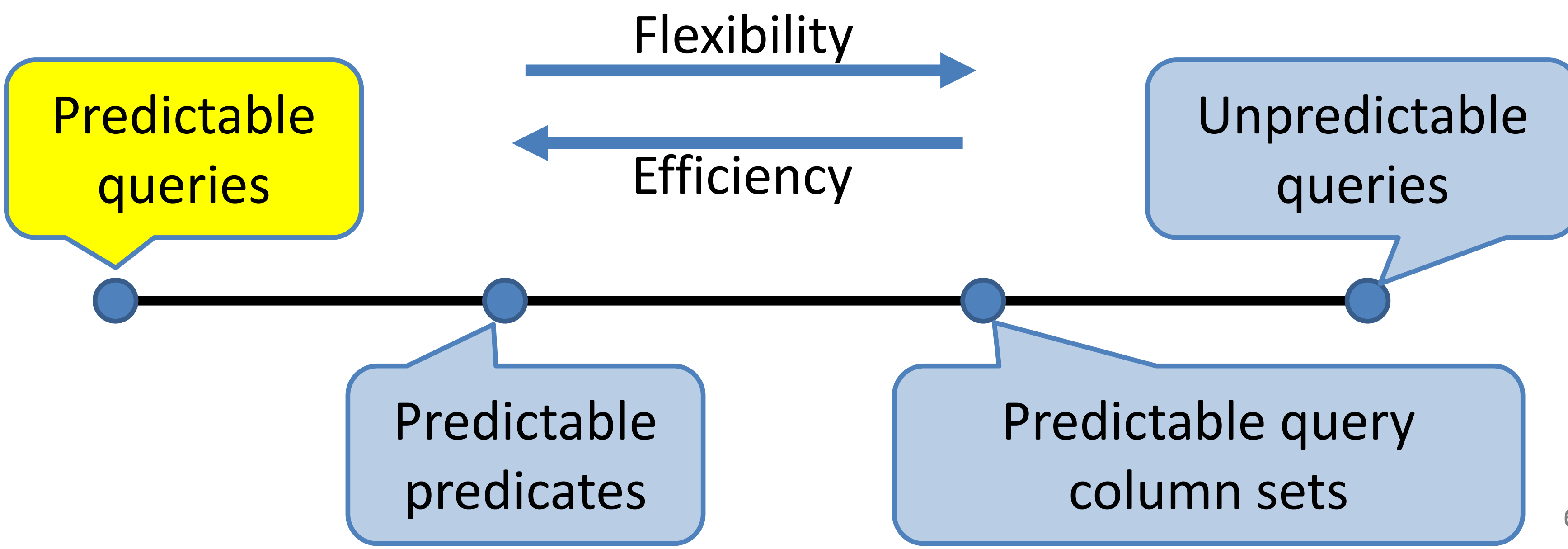
???
???

# Learning to sample (2)

- Predictable queries

```
SELECT AVG(salary)FROM tbl1
WHERE city="London"
```

- Can only answer this exact query!

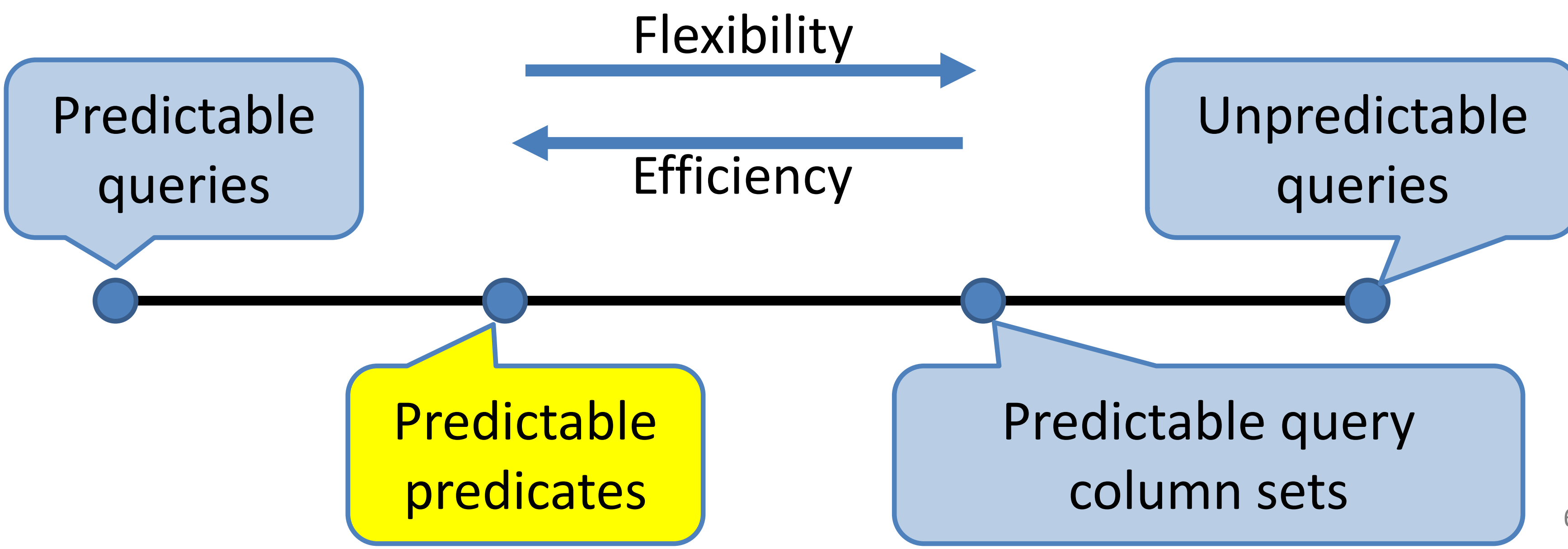


# Learning to sample (3)

- Predictable predicates

```
SELECT AVG($X)FROM tbl1
WHERE city="London"
```

- Can only answer this query, for different $X

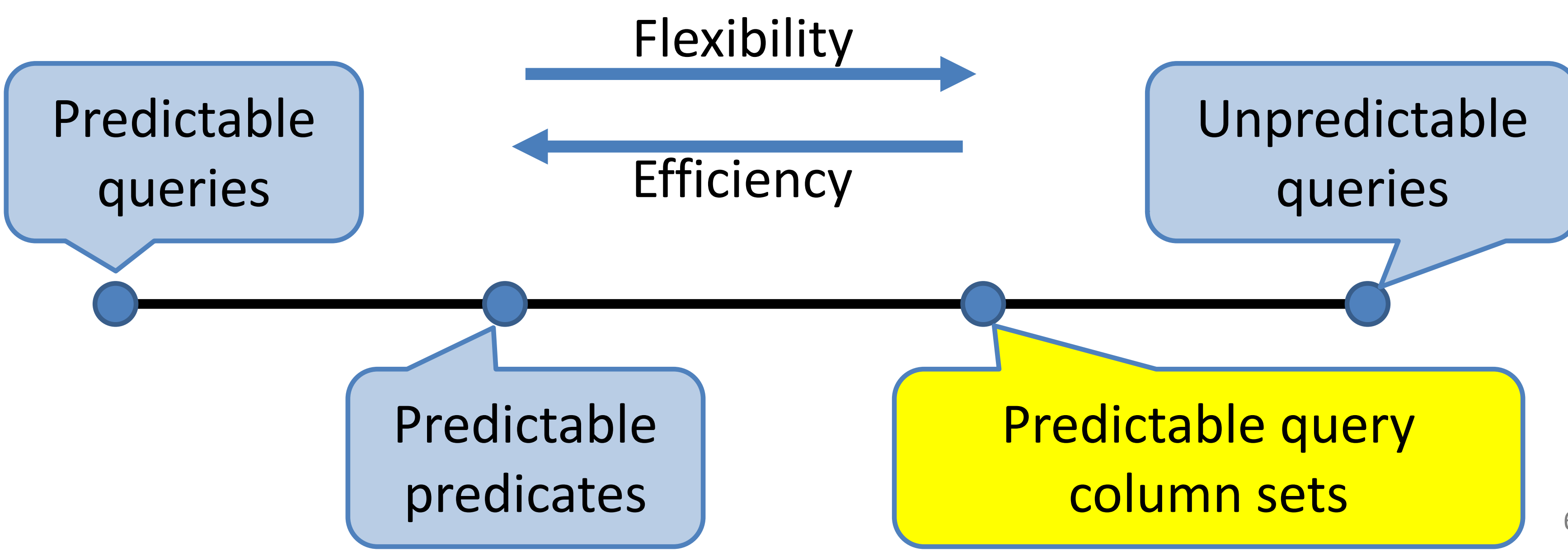


# Learning to sample (4)

- Key notion: QCS - Query Column Sets
  - All columns contained in the query that affect sampling
    - GROUP BY, HAVING, WHERE

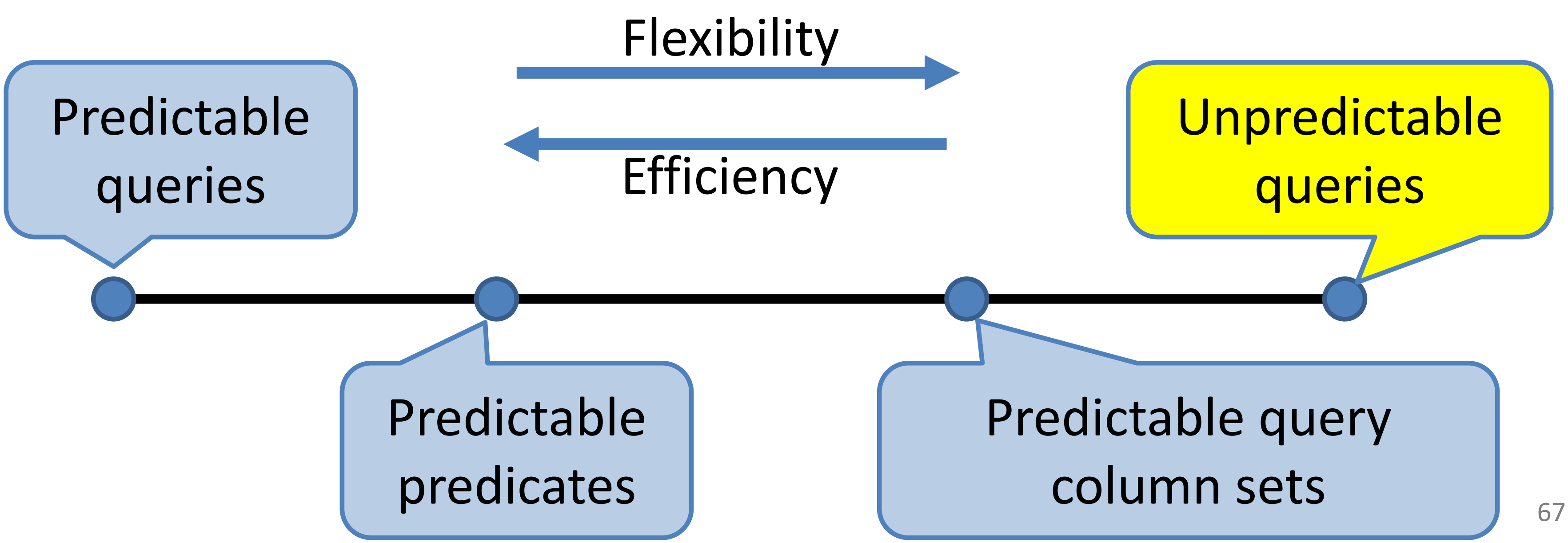
```
SELECT AVG($X)FROM tbl1
WHERE $Y1=$val1 AND $Y2=$val2 … GROUP BY $Z
```

  - Construct samples for each QCS



# Learning to sample (5)

- Unpredictable queries
- Fully flexible, but cannot sample efficiently!
  - Best effort approach



# Query column sets

- BlinkDB uses QCS to sample
  - Over 90% of queries covered by 10% - 20% of QCS →
  - Keeping samples for the most frequently used 10%-20% of QCS can help in 90% of the queries!
  - Use query logs to find the most frequent QCS
- Maintain a different sample for each frequent QCS!
  - For each query q with $QCS_q$
    - if there exists already a suitable sample S with $QCS_S$, use it
      - Suitable: $QCS_q$ is a subset of $QCS_S$
    - If there is no suitable sample, trial-and-error!

# Query column sets (2)

- Maintain a different sample for each frequent QCS!

**Pair the queries with the samples**

- Samples on the following QCS
  - <city>
  - <city, profession>
  - <city, age>
  - <city, profession, age>
  - <city, profession, education>
  - <education>

Queries

SELECT avg(salary)
FROM tbl1
GROUP BY
city, profession

# Query column sets (2)

- Maintain a different sample for each frequent QCS!

**Pair the queries with the samples**

- Samples on the following QCS
  - <city>
  - <city, profession>
  - <city, age>
  - <city, profession, age>
  - <city, profession, education>
  - <education>

Queries

SELECT avg(salary)
FROM tbl1
WHERE age=20
GROUP BY
city

# Query column sets (2)

- Maintain a different sample for each frequent QCS!

**Pair the queries with the samples**

- Samples on the following QCS
  - <city>
  - <city, profession>
  - <city, age>
  - <city, profession, age>
  - <city, profession, education>
  - <education>

<u>Queries</u>

SELECT avg(salary)
FROM tbl1
WHERE city=London
GROUP BY
education

# Query column sets (2)

- Maintain a different sample for each frequent QCS!

**Pair the queries with the samples**

- Samples on the following QCS
  - <city>
  - <city, profession>
  - <city, age>
  - <city, profession, age>
  - <city, profession, education>
  - <education>

Queries

SELECT avg(salary)
FROM tbl1
GROUP BY
city, education

# Query column sets (2)

- Maintain a different sample for each frequent QCS!

**Pair the queries with the samples**

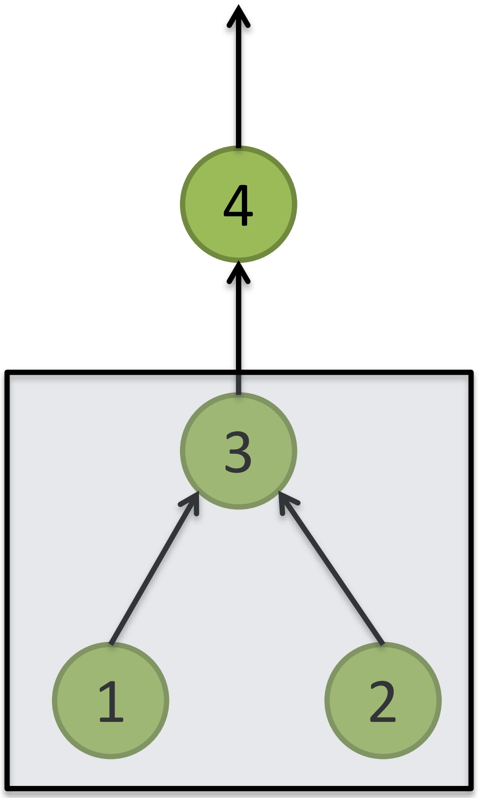
- Samples on the following QCS
  - <city>
  - <city, profession>
  - <city, age>
  - <city, profession, age>
  - <city, profession, education>
  - <education>

<u>Queries</u>

SELECT avg(salary)
FROM tbl1
GROUP BY
name

# Uniform samples

| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |



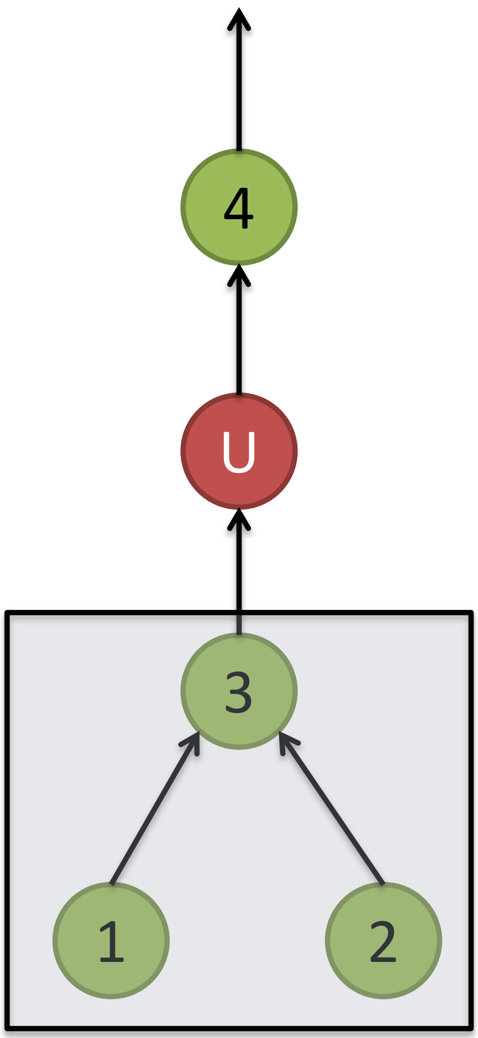

**Simplification**

SELECT AVG(data)
FROM tbl
GROUP BY city

SELECT AVG(data)
FROM tbl1, tbl2
WHERE tbl1.x=tbl2.y
GROUP BY city

# Uniform samples

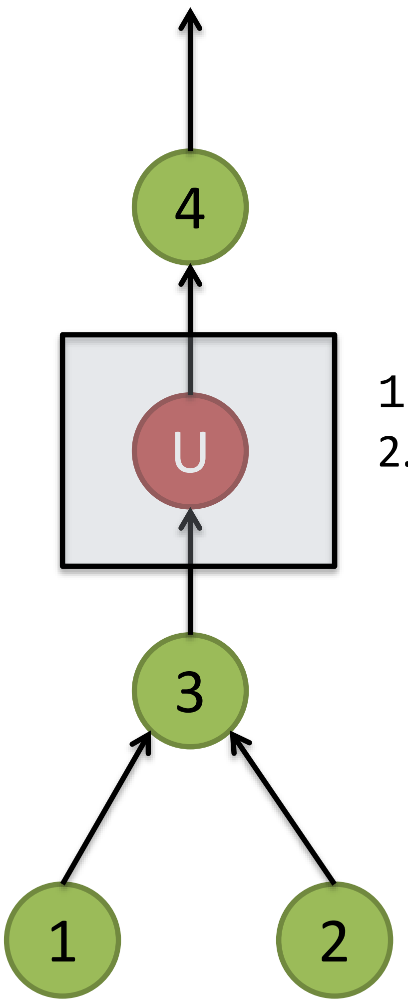
| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |




SELECT AVG(data)
FROM tbl
GROUP BY city

# Uniform samples

| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

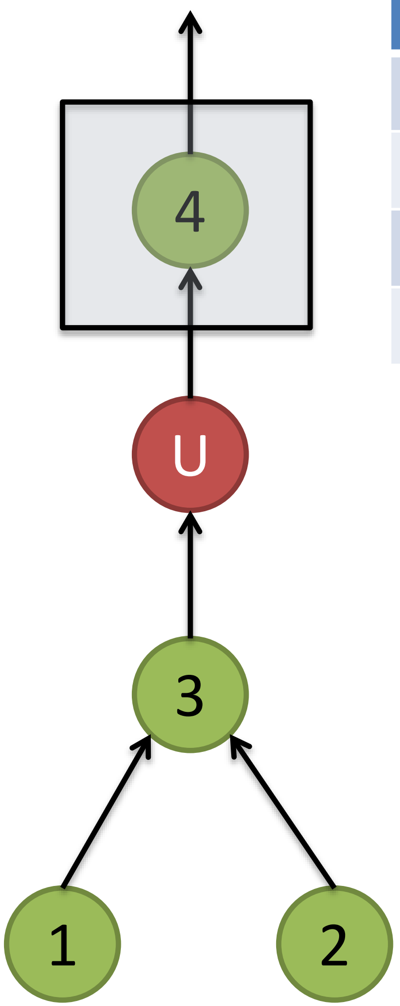



SELECT AVG(data)
FROM tbl
GROUP BY city

1. FILTER `rand() < 1/3`
2. Adds per-row sampling rates

# Uniform samples

| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |



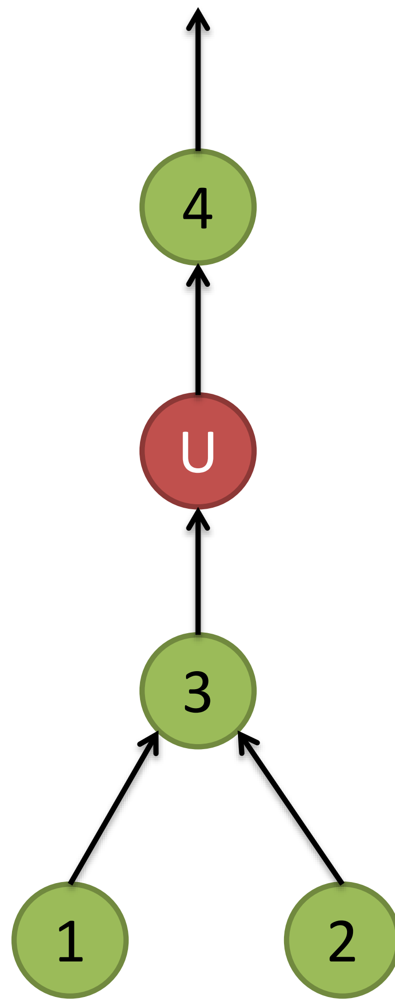

| ID | City | Data | Rate |
|---|---|---|---|
| 2 | NYC | 0.13 | **1/3** |
| 8 | NYC | 0.25 | **1/3** |
| 6 | Berkeley | 0.09 | **1/3** |
| 11 | NYC | 0.19 | **1/3** |

Does not change query semantics

SELECT AVG(data)
FROM tbl
GROUP BY city

# Uniform samples

| ID | City | Data |
|---|---|---|
| 1 | Lausanne | 0.91 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |



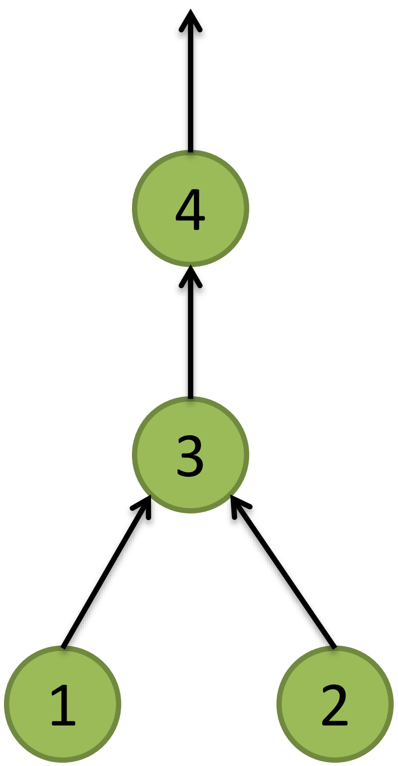

| ID | City | Data | Rate |
|---|---|---|---|
| 2 | NYC | 0.13 | **1/3** |
| 8 | NYC | 0.25 | **1/3** |
| 6 | Berkeley | 0.09 | **1/3** |
| 11 | NYC | 0.19 | **1/3** |

Uniform samples fail to include rare values

```
SELECT AVG(data)
FROM tbl
GROUP BY city
```

# Stratified samples

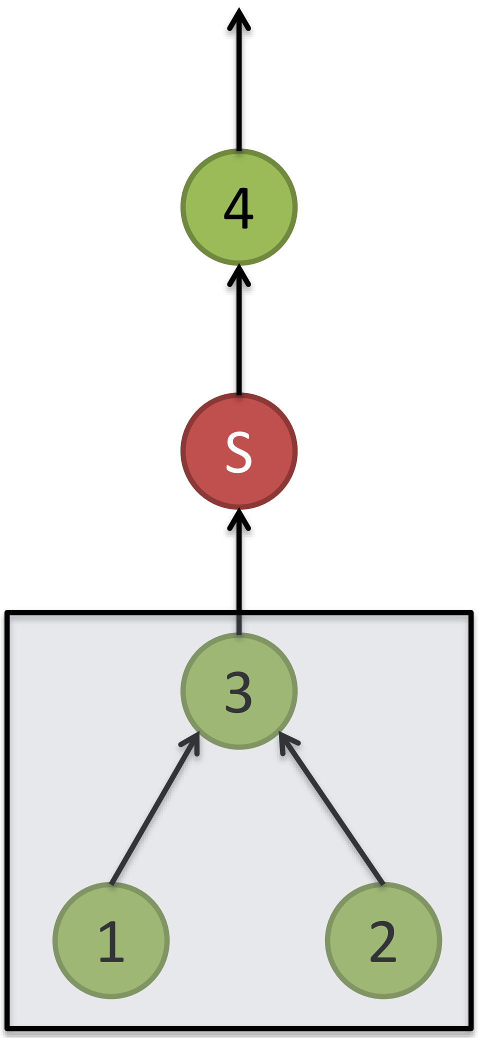
| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |




SELECT AVG(data)
FROM tbl
GROUP BY city

# Stratified samples

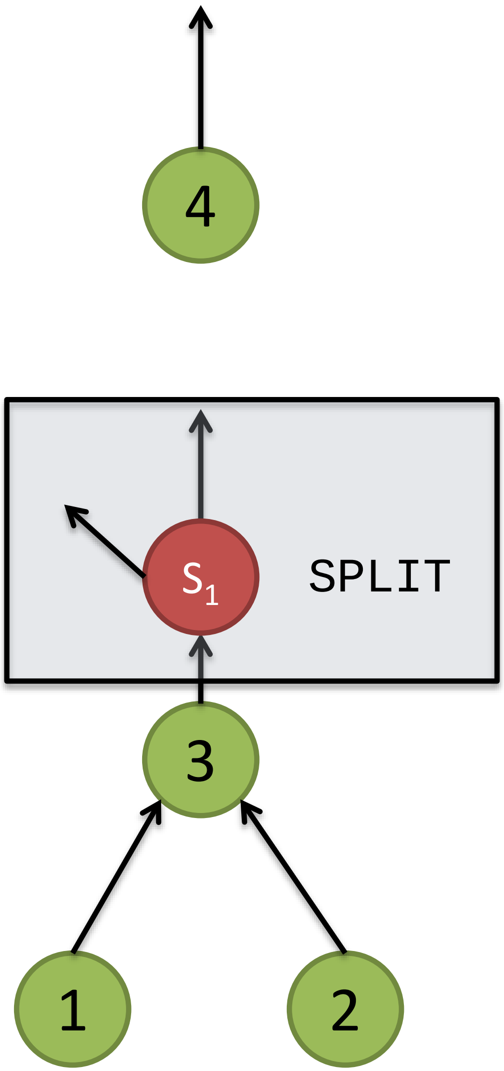
| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |




SELECT AVG(data)
FROM tbl
GROUP BY city

# Stratified samples

```
SELECT AVG(data)
FROM tbl
GROUP BY city
```

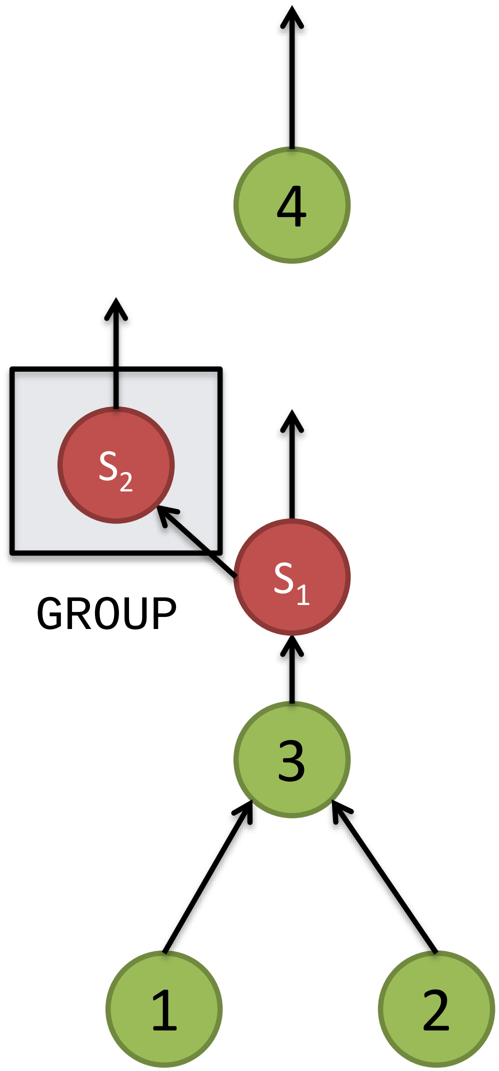



| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

# Stratified samples

```
SELECT AVG(data)
FROM tbl
GROUP BY city
```

| City | Count |
|---|---|
| NYC | 7 |
| Berkeley | 5 |

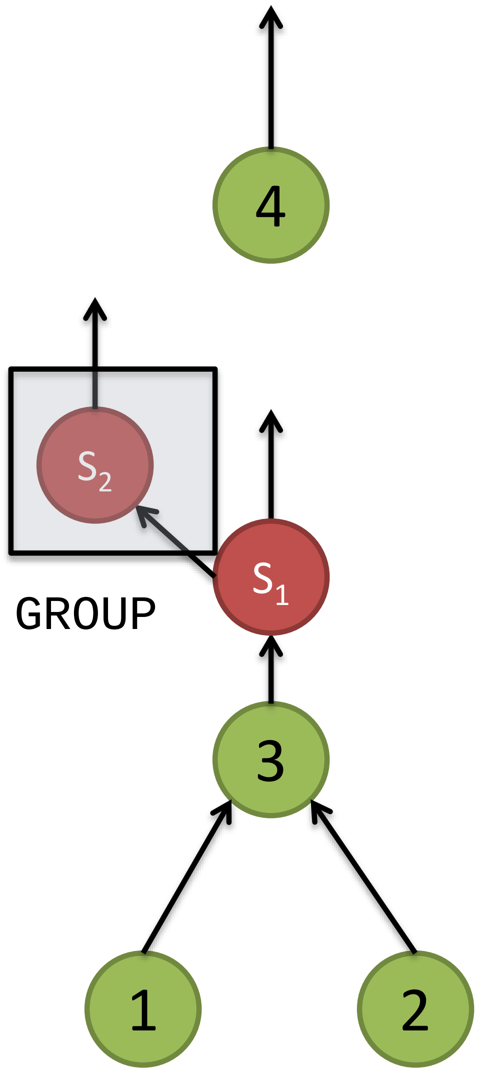



| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

# Stratified samples

```
SELECT AVG(data)
FROM tbl
GROUP BY city
```

| City | Count | Rate |
|---|---|---|
| NYC | **7** | **2/7** |
| Berkeley | **5** | **2/5** |

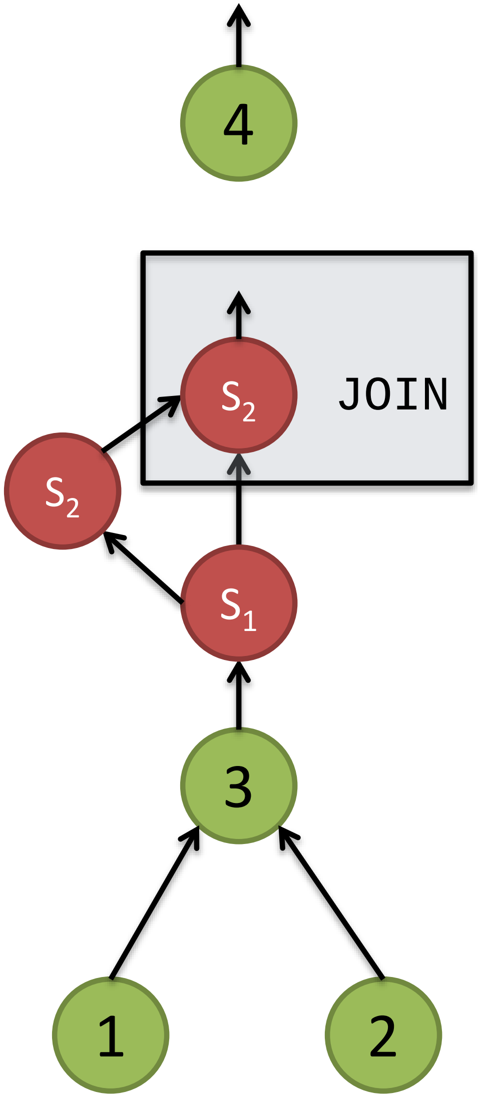



| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

# Stratified samples

```
SELECT AVG(data)
FROM tbl
GROUP BY city
```

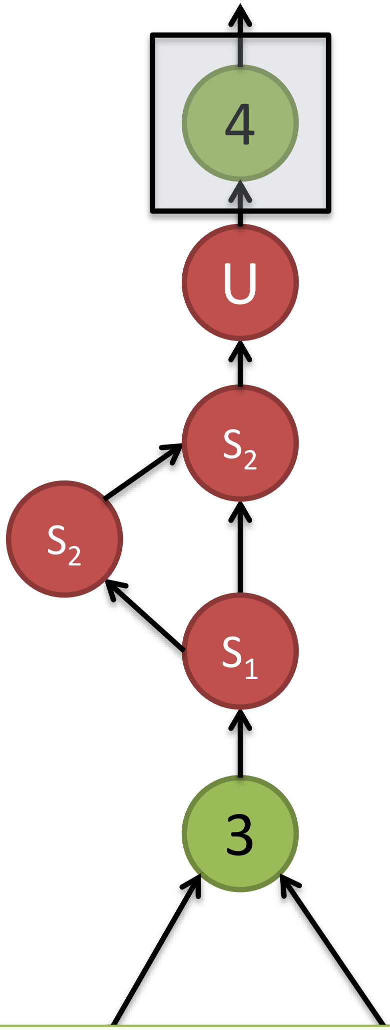
| City | Count | Rate |
|---|---|---|
| NYC | **7** | **2/7** |
| Berkeley | **5** | **2/5** |




| ID | City | Data |
|---|---|---|
| 1 | NYC | 0.78 |
| 2 | NYC | 0.13 |
| 3 | Berkeley | 0.25 |
| 4 | NYC | 0.19 |
| 5 | NYC | 0.11 |
| 6 | Berkeley | 0.09 |
| 7 | NYC | 0.18 |
| 8 | NYC | 0.15 |
| 9 | Berkeley | 0.13 |
| 10 | Berkeley | 0.49 |
| 11 | NYC | 0.19 |
| 12 | Berkeley | 0.10 |

# Stratified samples

SELECT AVG(data)
FROM tbl
GROUP BY city




| ID | City | Data | Rate |
|---|---|---|---|
| 2 | NYC | 0.13 | **2/7** |
| 8 | NYC | 0.25 | **2/7** |
| 6 | Berkeley | 0.09 | **2/5** |
| 12 | Berkeley | 0.49 | **2/5** |

Does not change query semantics

Are rare values represented in the sample now?

# Sampling for rare QCS values

- When a QCS value is very rare, all records end up in the sample

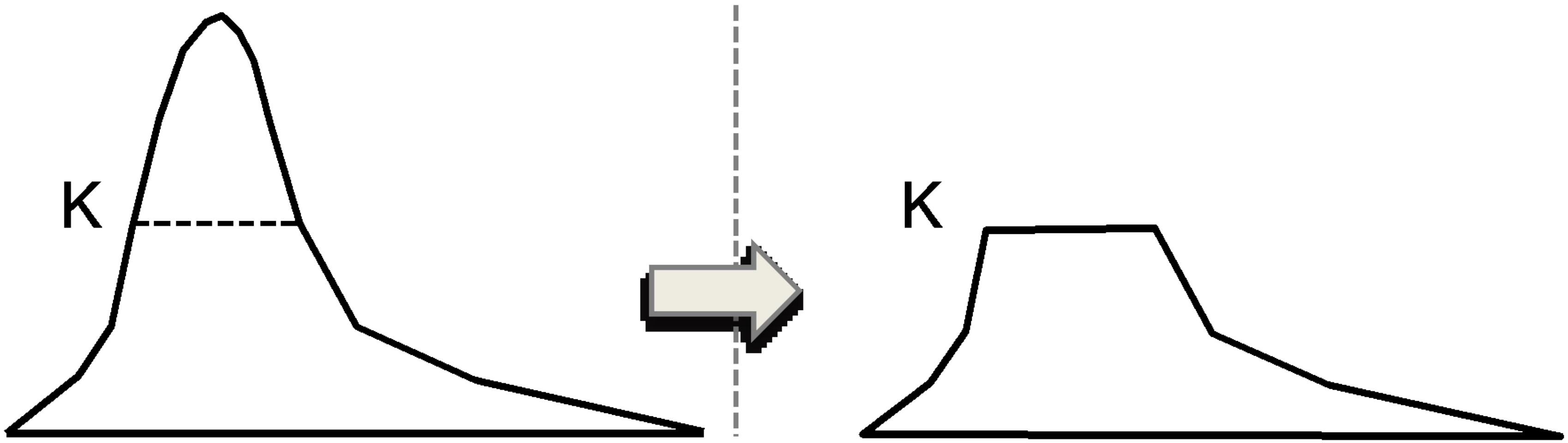



**Figure 4.** Example of a stratified sample associated with a set of columns, $\phi$.

# What is BlinkDB?

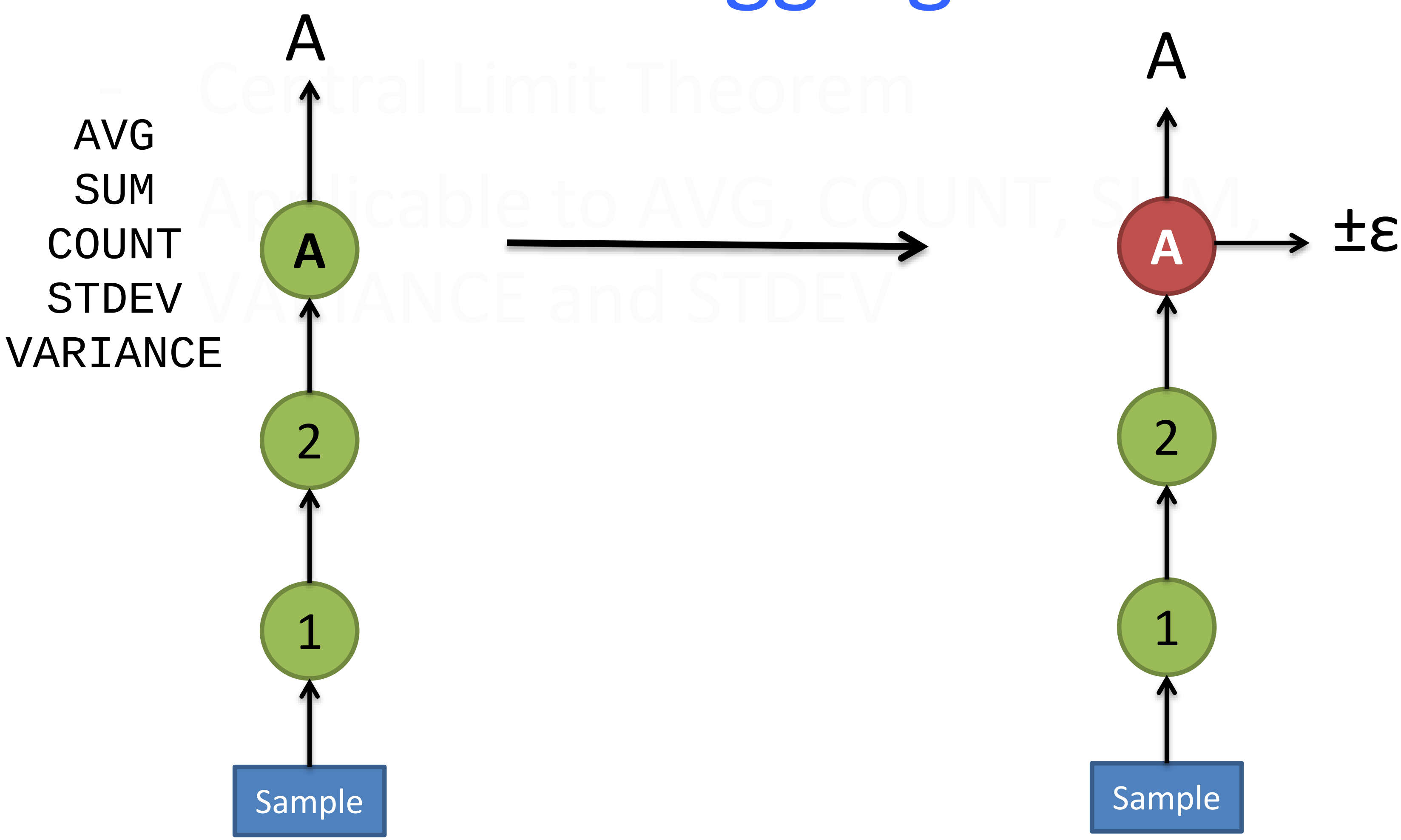
- A framework that …
  - creates and maintains a variety of uniform and stratified samples from underlying data
  - returns fast, approximate answers with error bars by executing queries on samples of data
  - verifies the correctness of the error bars that it returns at runtime

# Error Estimation

- Closed Form Aggregate Functions
  - Central Limit Theorem: When adding independent random variables, their sum tends towards a normal distribution, even if the original variables themselves are not normally distributed
  - Applicable to `AVG`, `COUNT`, `SUM`, `VARIANCE` and `STDEV`

# Error estimation

- Closed Form Aggregate Functions
  - Central Limit Theorem
  - Applicable to AVG, COUNT, SUM, VARIANCE and STDEV



Optional reading!

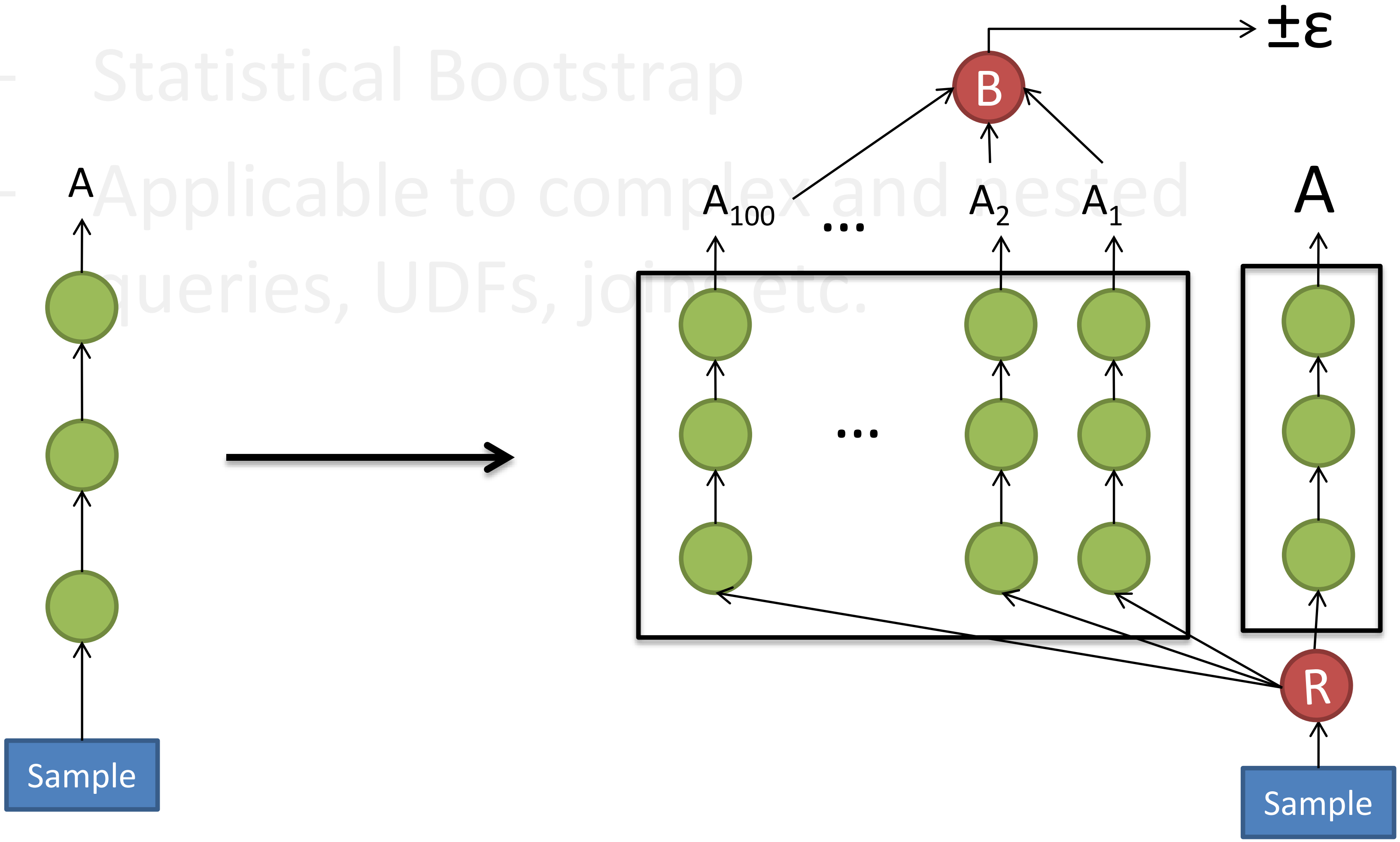
# Error estimation

- Generalized Aggregate Functions
  - Statistical Bootstrap
  - Applicable to complex and nested queries, UDFs, joins etc.
- Main idea
  - Execute the query on x sub-samples
  - Variance of answers: indicator of accuracy

Optional reading!

# Error estimation

- Generalized Aggregate Functions
  - Statistical Bootstrap
  - Applicable to complex and nested queries, UDFs, joins etc.



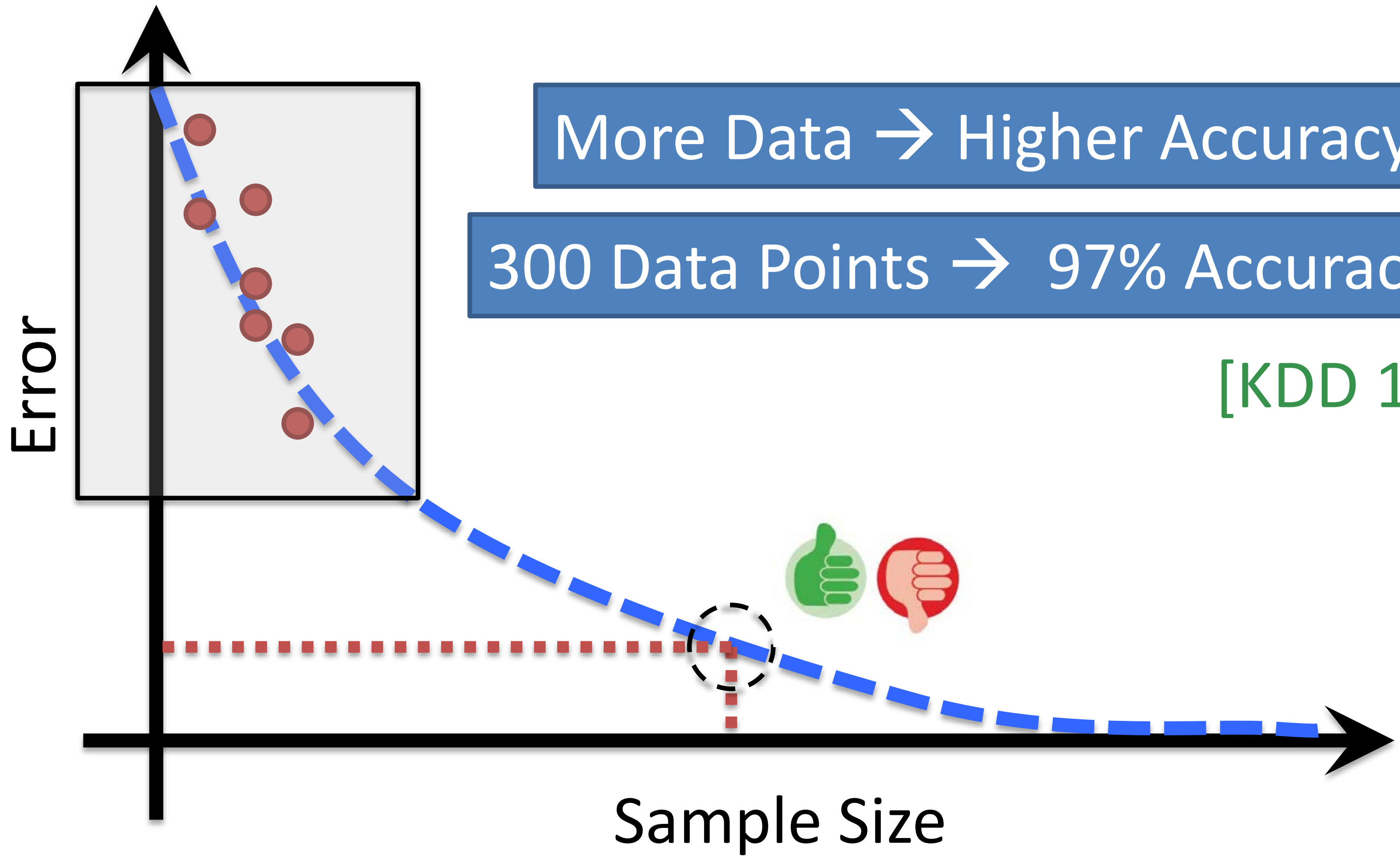
Optional reading!

# What is BlinkDB?

- A framework that …
  - creates and maintains a variety of random and stratified samples from underlying data
  - returns fast, approximate answers with error bars by executing queries on samples of data
  - verifies the correctness of the error bars that it returns at runtime

Optional reading!

# Kleiner's Diagnostics




More Data → Higher Accuracy

300 Data Points → 97% Accuracy

[KDD 13]

# What is BlinkDB?

- A framework that …

- creates and maintains a variety of random and stratified samples from underlying data
- returns fast, approximate answers with error bars by executing queries on samples of data
- verifies the correctness of the error bars that it returns at runtime

# Readings

- BlinkDB
  - Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica: BlinkDB: queries with bounded errors and bounded response times on very large data. EuroSys 2013: 29-42 https://sameeragarwal.github.io/blinkdb_eurosys13.pdf
  - Paper 2 (optional, will not be examined): https://sameeragarwal.github.io/mod282-agarwal.pdf
- Streams
  - Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica: Discretized streams: fault-tolerant streaming computation at scale. SOSP 2013: 423-438 http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf
  - Introduction & time models
    - **Section 4** from http://users.monash.edu/~mgaber/Muthu-Survey.pdf
    - **Section II.A** from http://dimacs.rutgers.edu/~graham/pubs/papers/fwddecay.pdf for decay models
    - **Section 3** from http://dl.acm.org/citation.cfm?id=1060753 covers sliding windows, jumping windows, landmark windows
- Technical/How-to/API/dev…
  - http://spark.apache.org/sql/ http://spark.apache.org/streaming/ http://blinkdb.org