Generating Big Data

Tilmann Rabl – msrg.org / U of T
Motivation

- What makes a benchmark successful?

  - Central factor
    - Easy to use
    - Good / complete tool chain

  - Good examples:
    - TPC-H, YCSB
Benchmarking Tool Chain

- **Data generator**
- **Workload generator**
- **Driver**
- **Metric computation**
Ways to Generate Data

- Application specific
  - Implementation overhead
  - Limited adaptability
  - Fast outdated
- Client simulation
  - Graph based
  - Very accurate (complex dependencies)
  - **Slow**
  - Limited repeatability
- Statistical distributions
  - Based on probability
  - Fast
  - Repeatable
  - **Simple data**
What Do We Need?

- Big data
  - Terabytes, Petabytes, Exabytes
And
- Complex data
  - User history, click streams
  - Useful information
  - In different stages (end-to-end)
Data Example
(Quickly made up last night)

- Users have names and email addresses and user data
- Messages connect two users
  - Both have to exist
- Social graph
  - Includes all connections of two users (directed or undirected)
Hierarchical seeding strategy

- Schema → Table → Column → Update → Row → Generator
- Uses deterministic seeds
- Guarantees that n-th random number determines n-th value
- Even for large schemas all seeds can be cached

Repeatable, **parallel**, deterministic generation
Parallel Data Generation Framework (PDGF) is

- **Generic**
  - Can generate any schema

- **Configurable**
  - XML configuration files for schema and output format

- **Extensible**
  - Plug-in mechanism for
    - Distributions
    - Specialized data generation formats

- **Efficient**
  - Utilizes all system resources to a maximum degree (if desired)

- **Scalable**
  - Parallel generation for modern multi-core SMPs and clustered systems
To generate data the user defines:

- Schema XML file
  - Defines relational schema
- Generation XML file
  - Defines output format (CSV, XML, merging tables)
- Configures PDGF for the schema
- Corresponds to ER of logical DBMS, i.e. tables, columns
- Defines content of columns (Field value generators)
- Defines table and column references
- Defines update properties
Generation XML File

- Defines the output
  - Scheduling
  - Data format
  - Sorting
  - File name and location

- Post processing
  - Filtering of values
  - Merging of tables
  - Splitting of tables
  - Templates (e.g. XML)
Implemented Data Generators

- **SetQuery** (The Set Query Benchmark by Patrick E. O'Neil)
  - Single table, 21 columns
  - 250 lines in schema and generation XML files
  - TPC TC’10

- **TPC-H** (Data Warehouse Benchmark by TPC)
  - 8 tables, 61 columns
  - 500 lines in schema and generation XML files
  - TPC TC’11

- **TPC-DI** (ETL Benchmark by TPC)
  - 20 tables, more than 200 columns
  - 6000 lines in schema and generation XML files
  - In progress

- More to come…
Evaluation

- TPC-ETL excerpt
  - Trade table
  - Historic data
  - 2 change data captures
  - SF $1,000,000 = 18 \text{ GB}$

- Test system
  - SMP server, 4 x X5670 Intel Xeon CPUs (2.93 GHz, 12 MB cache, 6 cores), 140 GB RAM, 24 cores total
  - All writes to /dev/null
Scaling the System Size

- 18 GB produced data
- Almost linear speed up for 8 threads
- Decreasing speed for more threads than cores

Legend:
Solid Line – Generation Time
Dotted Line – Throughput
Scaling the Problem Size

- 24 core system, 32 threads
- Generation of 18 – 72 gigabytes
- Constant throughput
- Linear generation time
Summary

- Requirements of big data generation
  - Large data, large systems, complex data

- Parallel Data Generation Framework
  - Fast, parallel, generic data generation
  - Support for complex inter value dependencies
  - Support for different data stages

- Current work
  - TPC-ETL, SSB
  - Query workload
  - Your benchmark?
Thank You!

- More info and download soon at
  - [www.paralleldatageneration.org](http://www.paralleldatageneration.org)
Backup Slides
SPARC T3-4, 4 x T3 CPUs (1.65 GHz, 6 MB cache 16 cores), 8 hardware threads per core, 512 GB RAM, 512 virtual processors

1.8 GB produced data

Linear speed up for 32 threads

Decreasing speed for more threads than cores
Data Generation in PDGF

- Data generation is done in so called Field Value Generators
- Field Value Generators are functions
  - Domain: random values
  - Co-domain: data domain
  - Built-in Field Value Generators can be extended with plugins
  - Based on pseudo random number generators
  - Deterministic data generation

- Sample built-in Field Value Generators
  - Dictionary: Random number modulo DictionaryRowCount
  - Number: Random number modulo (range + offset)
Architecture PDGF

- Controller ➔ Initialization
- Meta Scheduler ➔ Inter node scheduling
- Scheduler ➔ Inter thread scheduling
- Worker ➔ Blockwise data generation
- Update Black Box ➔ Co-ordination of data updates
- Seeding System ➔ Random sequence adaption
- Generators ➔ Value generation
- Output system ➔ Data formating
Random Number Generation

- Pseudo random numbers (xorshift)
  - Fast
  - Repeatable

Parallel random number generation
- Fast random numbers
- Random hash
- \( \text{rng}(n) = \text{prng}(\text{seed}+n) \)

```c
void skip(long step){
    seed += step;
}

long next() {
    ++seed;
    long x = seed;
    x = x ^ (x >>> 15);       //XOR1
    x = x ^ (x << 35);        //XOR1
    x = x ^ (x >>> 4);        //XOR1
    x = 4768777513237032739L * x; //MWC
    x = x ^ (x << 17);        //XOR2
    x = x ^ (x >>> 31);       //XOR2
    x = x ^ (x << 8);         //XOR2
    return x;
}
```
Bijective Permutation

- Pseudo random numbers are not sufficient to generate all types of complex data

- Bijective permutation
  - Allows sampling without replacement
    - Choosing a unique key from a set of keys
  - Necessary for
    - Random unique values
    - Random subsets (e.g. account managers)
  - Static size

```
perm(x){
    y = (x * b + c) mod p
    return y;
}
```

```
invperm(y){
x=((y - c) * b_inv) mod p
    return x;
}
```
Growing Permutation

- Growing permutation with offsets
  - Abstract time (generation / update ID)
  - Bijective permutation per generation

- In each generation
  - Adding of values
  - Removing of values
  - Changing of values

- Growing, shrinking or static number of values