Sector & Sphere

Exploring Data Parallelism and Locality in Wide Area Networks

Yunhong Gu
Univ. of Illinois at Chicago

Robert Grossman
Univ. of Illinois at Chicago and Open Data Group

Overview

- Cloud Computing and Sector/Sphere Stack
- Sphere Computing Paradigm
- System Design and Implementation
- Applications and Performance Evaluation

Cloud Computing & Sector/Sphere

- Hardware
  - Inexpensive computer cluster
  - High speed wide area networks
- Software
  - Storage cloud: Distributed, big data set
  - Compute cloud: simplified distributed data processing

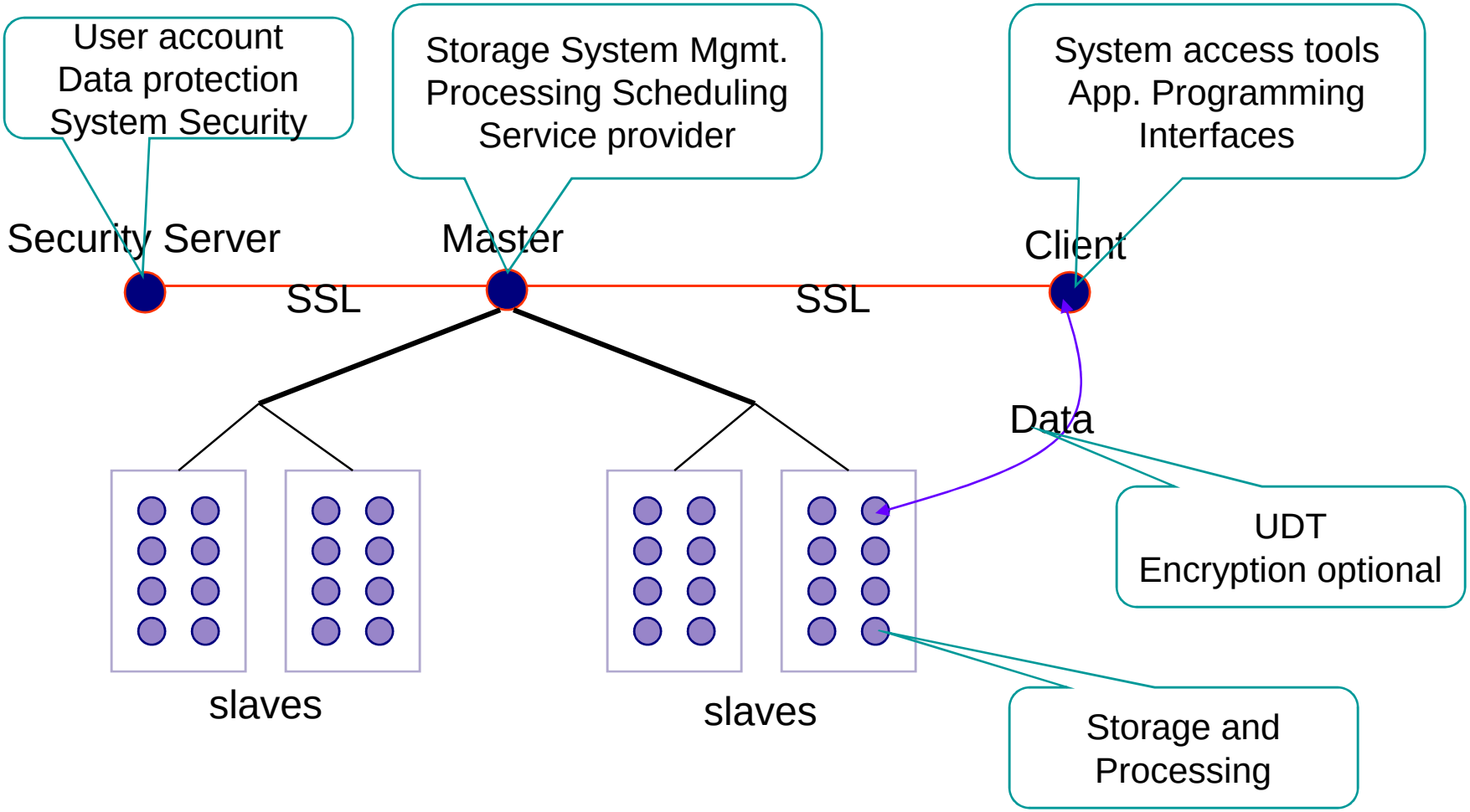
Compute Cloud: Sphere

Storage Cloud: Sector

Routing: C/S or P2P

High Speed Transport Protocol UDT

Sector: Distributed Storage System



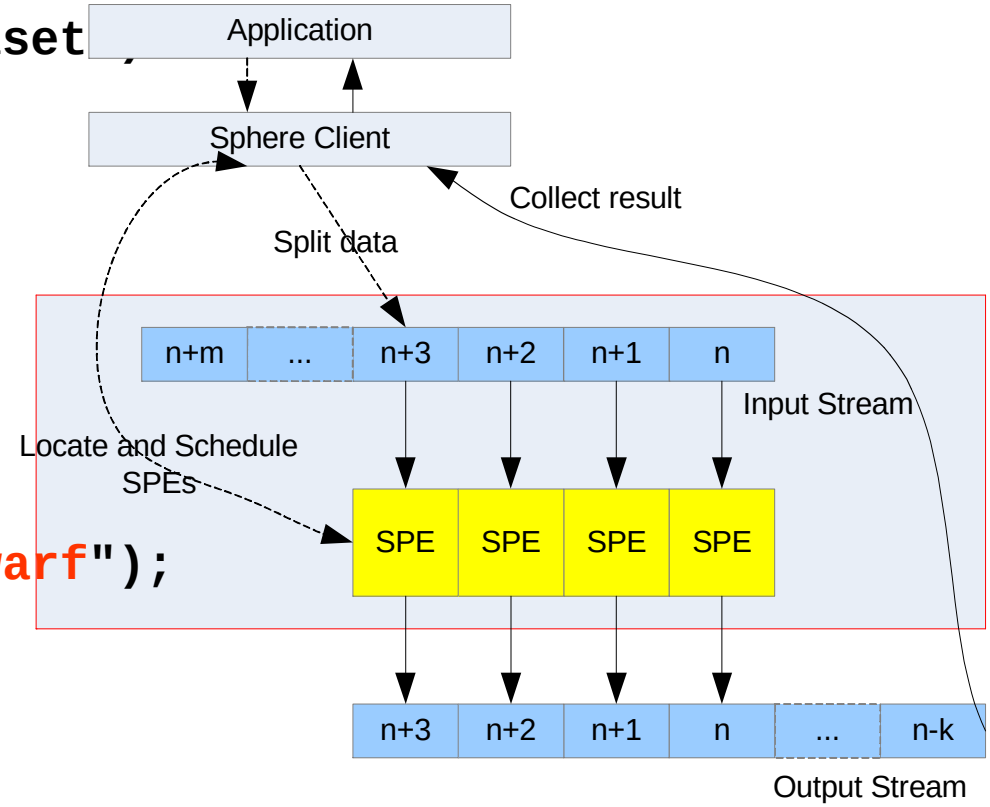
Sphere: Simply Data Processing

- Data is processed at where it resides (locality)
- Same user defined functions (UDF) can be applied on all elements (records, blocks, or files)
- Processing output can be written to Sector files, on the same node or other nodes
- Generalize Map/Reduce

Sphere: Simplify Data Processing

```
for each file F in (SDSS dataset)
  for each image I in F
     findBrownDwarf(I, …);
```

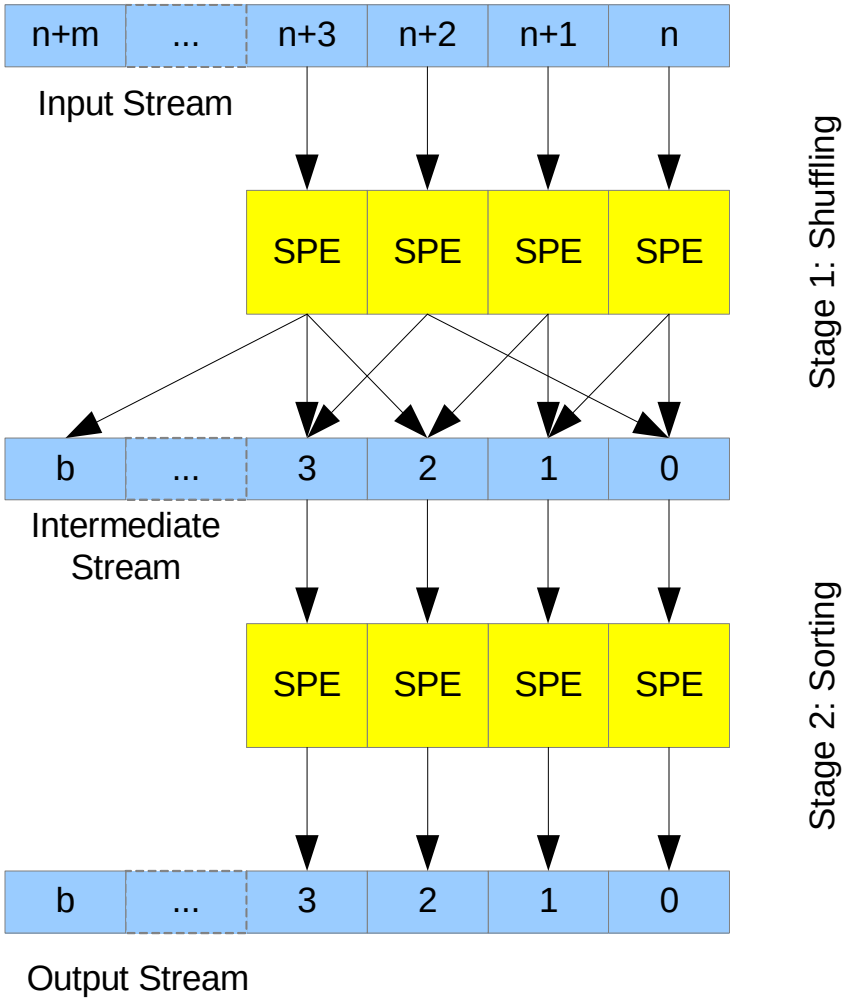
```
SphereStream sdss;
sdss.init("sdss files");
SphereProcess myproc;
myproc->run(sdss,"findBrownDwarf");
myproc->read(result);
```




```
findBrownDwarf(char* image, int isize, char* result, int rsize);
```

Sphere: Data Movement

- Slave -> Slave Local
- Slave -> Slaves (Shuffle)
- Slave -> Client



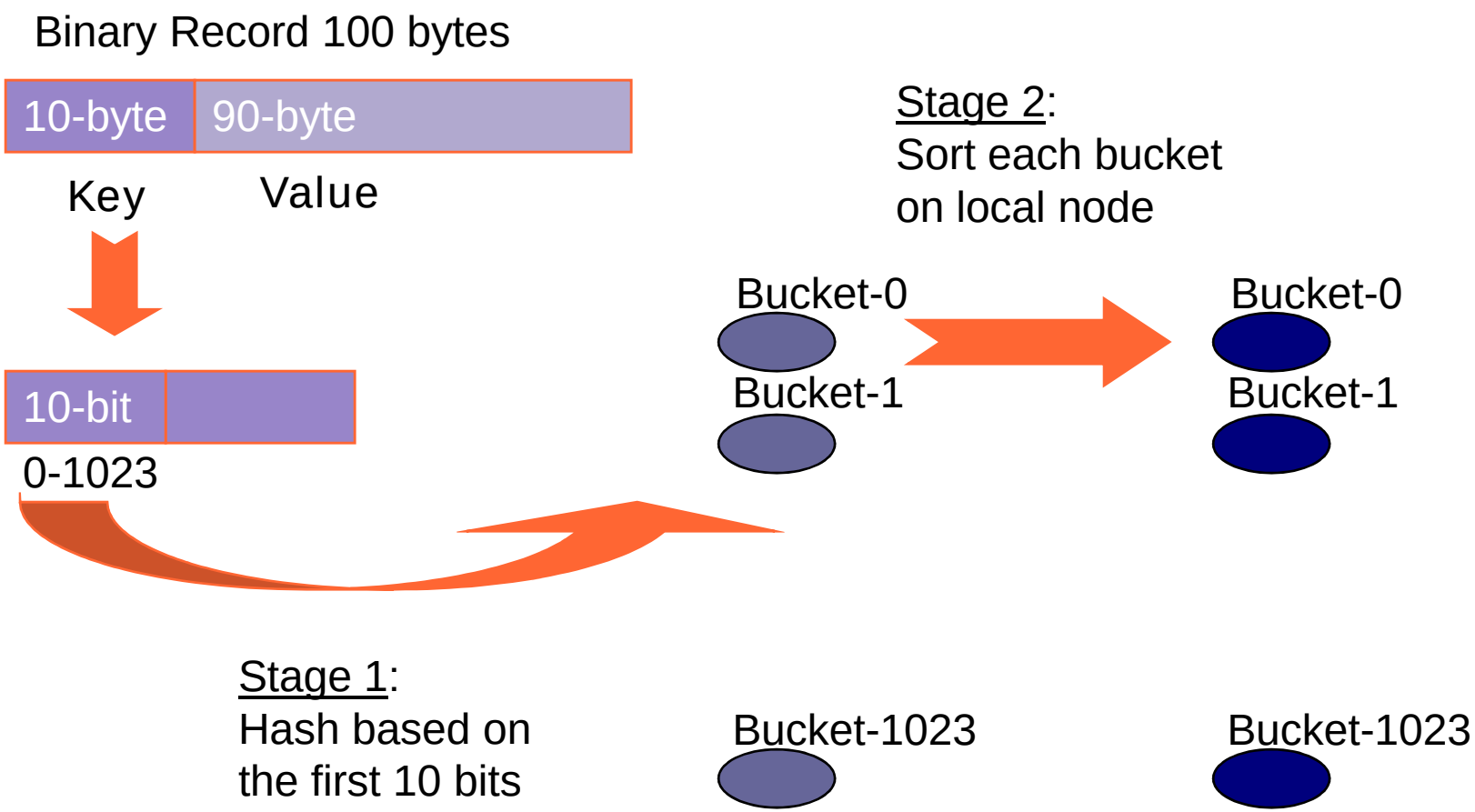
Load Balance & Fault Tolerance

- The number of data segments is much more than the number of SPEs. When an SPE completes a data segment, a new segment will be assigned to the SPE.
- If one SPE fails, the data segment assigned to it will be re-assigned to another SPE and be processed again.

Example: Sorting a TeraByte

- Data is split into small files, scattered on all slaves
- Stage 1: On each slave, an SPE scans local files, sends each record to a bucket file on a remote node according to the key, so that all buckets are sorted.
- Stage 2: On each destination node, an SPE sort all data inside each bucket.

TeraSort


Binary Record 100 bytes
10-byte
90-byte
Key
Value
10-bit
0-1023
Stage 2:
Sort each bucket
on local node
Bucket-0
Bucket-1
Bucket-1023
Bucket-0
Bucket-1
Bucket-1023
Stage 1:
Hash based on
the first 10 bits

Implementation

- Sector/Sphere is written C++ and released as open source software
  - http://sector.sf.net
  - Current release version 1.5
- TeraSort Example Application
  - 50 lines of C++ code, in addition to local sorting code
  - Prepare input files, output destinations, and call user defined functions (hash and sort)

Open Cloud Testbed

- 4 Racks in Baltimore (JHU), Chicago (StarLight and UIC), and San Diego (Calit2)
- 10Gb/s inter-site connection on CiscoWave
- 1Gb/s inter-rack connection
- Two dual-core AMD CPU, 12GB ram, 1TB single disk

Performance Results: TeraSort

Run time: seconds

	Data Size	Sphere	Hadoop (3 replicas)	Hadoop (1 replica)
UIC	300GB	1265	2889	2252
UIC + StarLight	600GB	1361	2896	2617
UIC + StarLight + Calit2	900GB	1430	4341	3069
UIC + StarLight + Calit2 + JHU	1.2TB	1526	6675	3702

Performance Results: TeraSort

- Sorting 1.2TB on 120 nodes
- Hash vs. Local Sort: 60% : 40%
- Hash
  - Per rack: 680GB in/out; Per node: 10GB in/out
  - CPU: 50% MEM: 900MB
- Local Sort
  - No network IO
  - CPU: xx MEM: XX
- Hadoop: CPU 150% MEM 2GB

For More Information

- Sector/Sphere code & docs: http://sector.sf.net
- Open Cloud Consortium: http://www.opencloudconsortium.org
- NCDM: http://www.ncdm.uic.edu
- @SC08: Booth 321

CreditStone

Text Record

Trans ID|Time|Merchant ID|Fraud|Amount
01491200300|2007-09-27|2451330|0|66.49

Transform

Text Record

Merchant ID	Time	Fraud

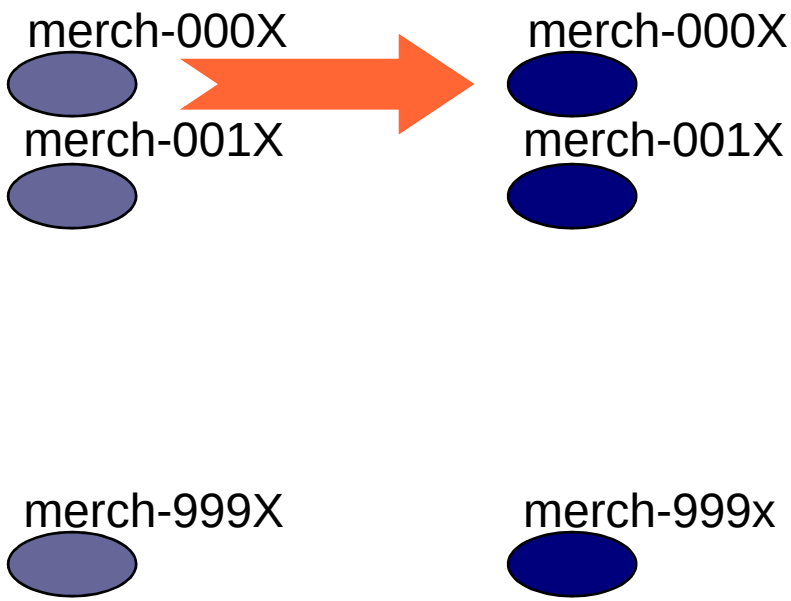
Key Value

3-byte	

000-999

Stage 1:
Process each record and hash into buckets according to merchant ID

Stage 2:
Compute fraudulent rate for each merchant



Performance Results: CreditStone

Racks	JHU	JHU, SL	JHU, SL, Calit2	JHU, SL, Calit2, UIC
Number of Nodes	30	59	89	117
Size of Dataset (GB)	840	1652	2492	3276
Size of Dataset (rows)	15B	29.5B	44.5B	58.5B
Hadoop (min)	179	180	191	189
Sector with Index (min)	46	47	64	71
Sector w/o Index (min)	36	37	53	55

* Courtesy of Jonathan Seidman of Open Data Group.

Inverted Index

HTML page_1

word_x word_y word_y word_z

word_x	1
word_y	1
word_z	1

1st letter	

Stage 1:
Process each HTML file and hash (word, file_id) pair to buckets

Stage 2:
Sort each bucket on local node, merge same word

Bucket-A

Bucket-B

Bucket-Z

word_z	1
word_z	5
word_z	10

Bucket-A

Bucket-B

Bucket-Z

word_z	1, 5, 10