Scalable Virtual Data Structures

Sushil Jajodia, George Mason University
Witold Litwin, Université Paris Dauphine
Thomas Schwarz, SJ, Universidad Católica del Uruguay

SDDS

- Scalable Distributed Data Structures
  - Developed in the 90s
    - LH* (Litwin, Neimat, Schneider), RP* (Litwin, Neimat, Schneider), Search Trees (di Pasquale & Nardelli, Kröll & Widmayer), DDH (Devine), …
  - Key-based access to large data sets in time O(1)
    - Key-value pairs
    - Scan operation

Di Pasquale, Adriano, and Enrico Nardelli. "Scalable Distributed Data Structures: A Survey." In *WDAS*, pp. 87-111. 2000.

SDDS

- SDDS
  - No central components (on typical access path)
  - Store records in buckets
  - Split buckets to accommodate growth
  - With high / saciable availability versions

SDDS

- LH* : SDDS based on linear hashing
  - Records stored in buckets
  - Originally *m* buckets
  - Buckets split in fixed order
    - 0,1,…,*m*-1;0,1,…,*m*-1, *m*, … 2*m*-1;0, 1, …
  - Global file state: level *L*, split pointer *s*

SDDS

- Addressing:
  - Based on extensible hash functions
    - Example (with *M* initial number of buckets):

$$h_i(c) = c \pmod{M} \cdot 2^i$$

  - Bucket address calculated from key using level *i* and split pointer *s*

$$a := h_i(c)$$

$$\text{if } a < s \text{ then } a := h_{i+1}(c)$$

SDDS

- Clients do not necessarily know the file state, they know an image of the file state
- Same is true for buckets
- Key-based query:
  - Clients use their file state image to find the bucket where record is
  - They can be wrong
  - A bucket uses its image of the file state to see whether a request from a client is directed to a record that it has. If not, then it *forwards*

SDDS

- All requests reach the correct bucket with at most two additional forwards
- Clients never make the same mistake twice:
  - If a request is forwarded, the correct bucket sends an image-adjustment message so that the client has the correct file state
- Active clients commit few addressing mistakes
  - If they commit one, in general only one forward

SDDS

- LH* allows scans:
  - Client sends request to all buckets it knows together with its image
  - Buckets can determine whether they need to forward a scan request to other buckets

SDDS

- Take home:
  - Data structure is autonomous from clients
    - Adjusts to growth (and shrinking)
    - Manages failure tolerance
    - Hides complexity from clients

What do we want

- Organize calculation in the cloud in the same way
  - Data structure that distributes brute force work over as many nodes as needed
  - Is autonomous
  - Is scalable

- Paradigm is the *scan* operation in an SDDS

What do we want

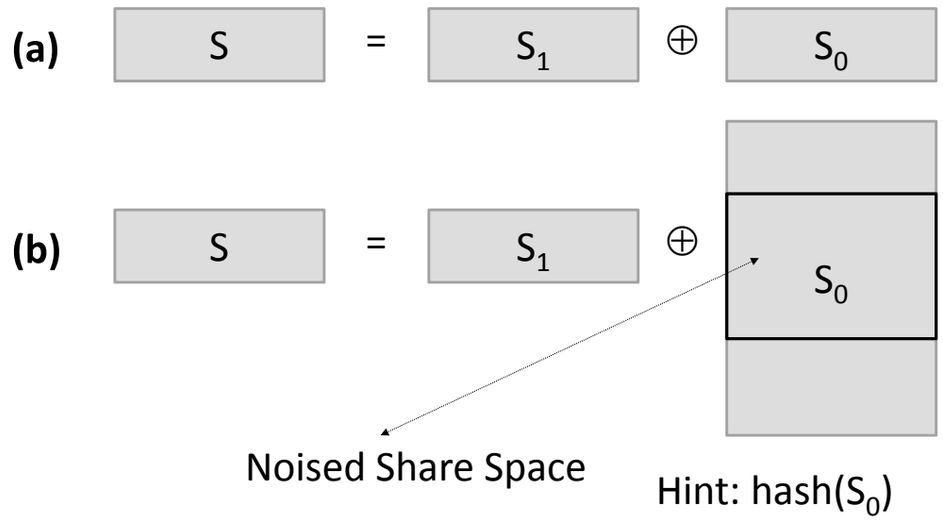
- Cloud resources
  - are fungible and easily obtained
  - suffer a high rate of failure
  - provides various levels of service
  - are cheap in comparison with programming effort
  - push limits of brute force calculation by 2 or 3 orders of decadic magnitude

SVDS

- Scalable Virtual Data Structures
- Extend SDDS principles to brute-force computing in the cloud
  - A scan operation where records are virtual

Secret Sharing with Noised Share

- Back-up scheme for key *S*
- Key is broken into two shares
- Escrow agency stores one share and the hint
- Escrow agency can recover by using a cloud to invert the hint
- Size of the noised share space and speed of hashing algorithm controls the complexity of the operation
- Costs of share recovery too high for escrow service to precompute and low enough to obtain share in an emergency



Jajodia, Sushil, Witold Litwin, and Thomas Schwarz. "Key Recovery Using Noised Secret Sharing with Discounts over Large Clouds." In *Social Computing (SocialCom), 2013 International Conference on*, pp. 700-707. IEEE, 2013.

Jajodia, Sushil, and Witold Litwin. "Recoverable encryption through a noised secret over a large cloud." In *Transactions on Large-Scale Data-and Knowledge-Centered Systems IX*, pp. 42-64. Springer Berlin Heidelberg, 2013.

SVDS

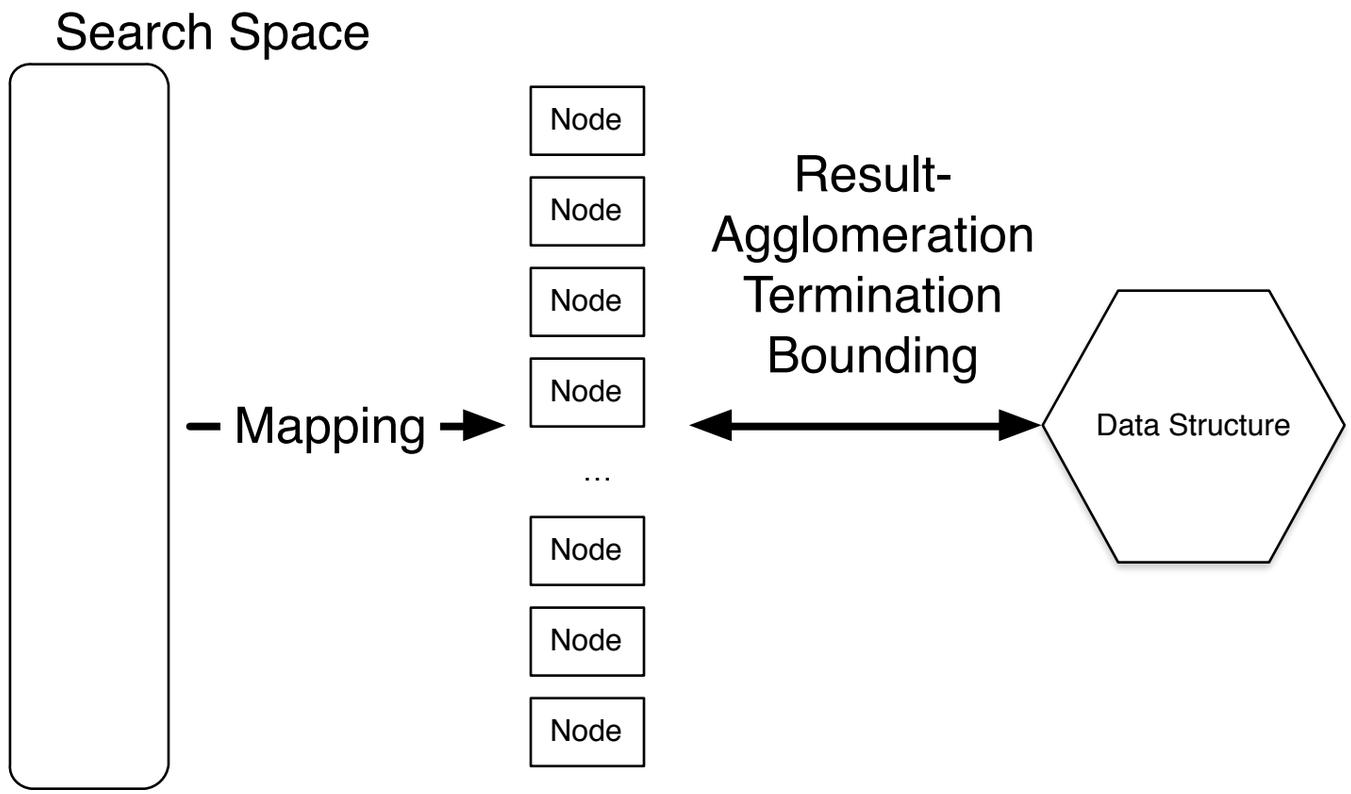
- Solving generic optimization problems
  - using brute-force
  - in a cloud environment

- Need data structure that
  - is scalable
  - distributes load efficiently
  - manages nodes autonomously
  - provides failure tolerance

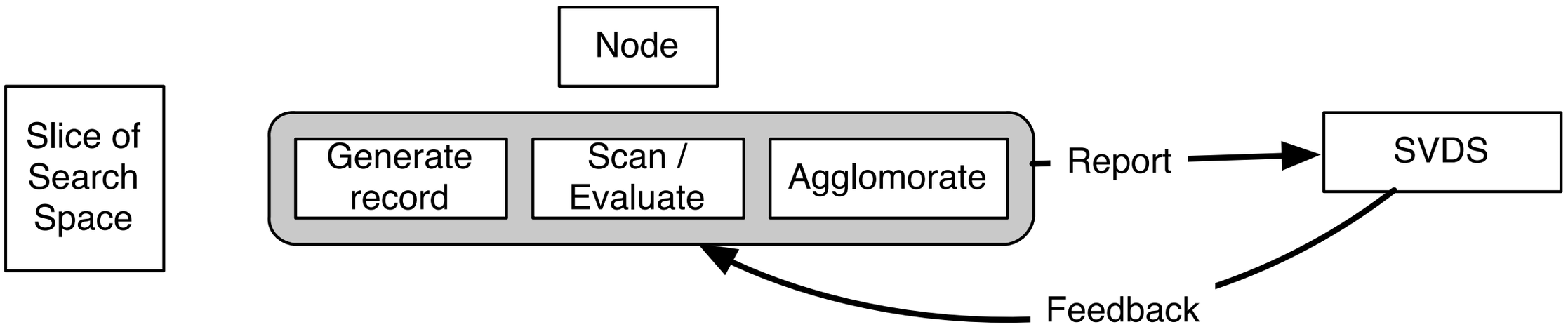
SVDS

- Examples
  - Inverting hashes
  - The classical 0-1 knapsack problem: maximize a linear function subject to a linear constraints
  - The traveling salesman problem: minimize the sum of edge values of a roundtrip through all the nodes of a graph
  - Integer linear programming with general constraint and objective functions
  - 3SAT

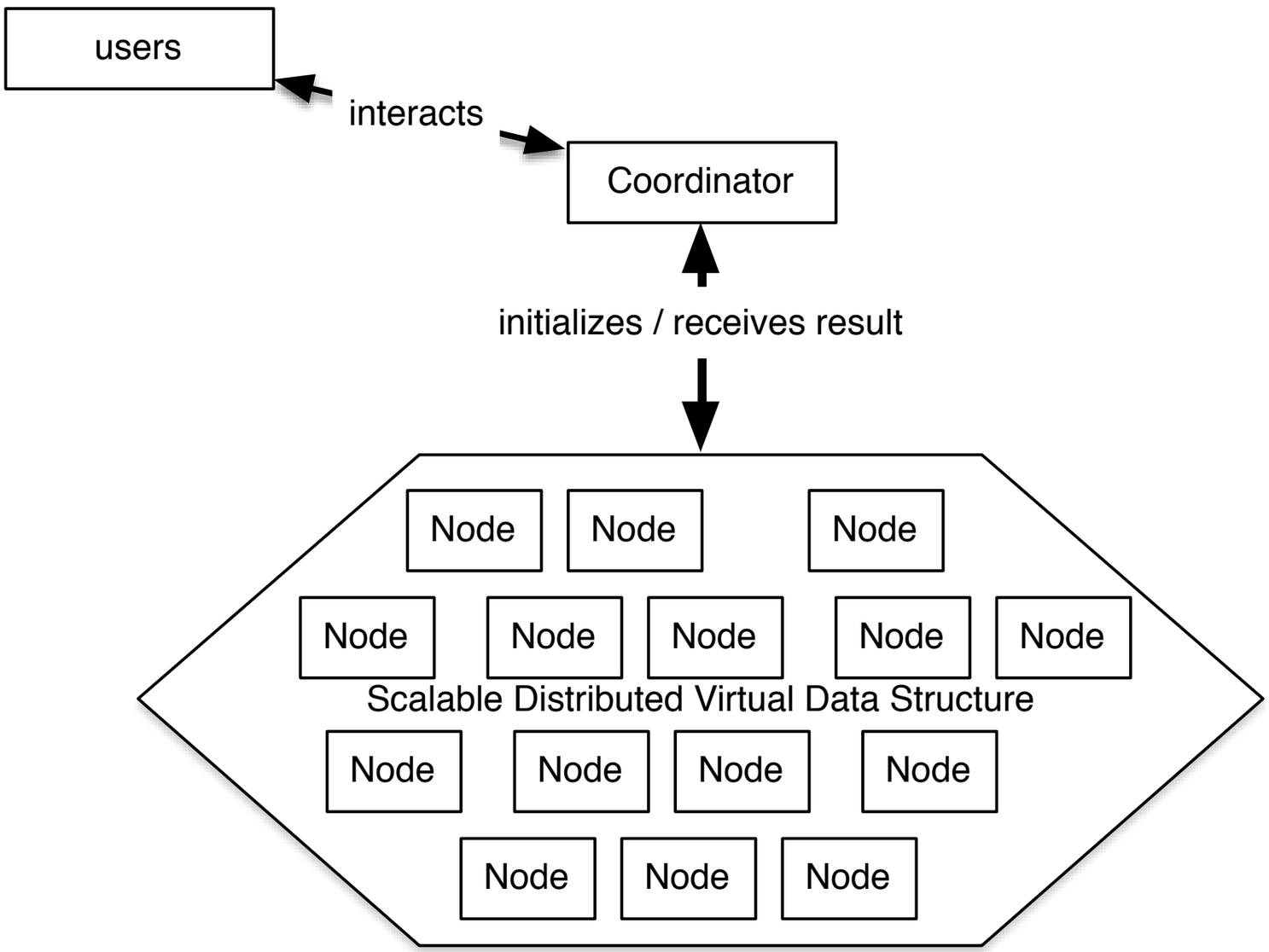
SVDS


Search Space
Mapping
Node
Node
Node
Node
...
Node
Node
Node
Result-
Agglomeration
Termination
Bounding
Data Structure

SVDS


Node
Slice of Search Space
Generate record
Scan / Evaluate
Agglomorate
Report
SVDS
Feedback

SVDS

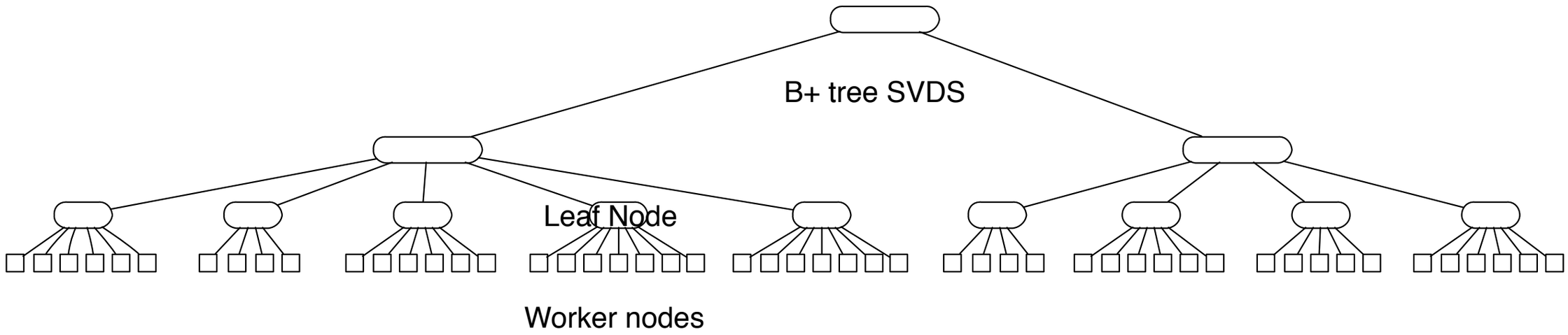

users
interacts
Coordinator
initializes / receives result
Node
Node
Node
Node
Node
Node
Node
Node
Scalable Distributed Virtual Data Structure
Node
Node
Node
Node
Node
Node
Node

SVDS

- Structures
  - LH* Assignment
    - Initial phase
      - Coordinator estimates conservatively number of nodes M
      - Each assigned node checks its capacity
      - Splits if its load is larger than its capacity
    - Agglomeration phase
      - Nodes report to their parents
      - Initial nodes report to coordinator

SVDS

- Structures based on range partitioning
  - Organize nodes in a B+-tree.
  - Leaf nodes organizes group of worker nodes



SVDS

- Structures based on range partitioning
  - Initial assignment:
    - Coordinator assigns leaf node leaders and generates communication structure of interior leaves.
    - Leaf nodes have between *k* and 2*k*-1 nodes
    - Leaf node leader assigns load according to capacity
    - If capacity is not sufficient, try to shift load to left or right neighbor (rotation)
    - If necessary, requests additional worker nodes
    - If number of worker nodes is larger than 2*k*, split

SVDS

- Structures based on range partitioning
  - Failure Tolerance
    - Every worker node reports partial results of slices to all other nodes in the same group
    - Leader detects failure based on outstanding reports
    - Leader failure is detected in the same way and leads to election of a new leader
    - Failed node is replaced
    - Already reported results do not need to be regenerated
- Provides tolerance against *k*-1 failures

SVDS

- Changes in load
  - If load lowers:
    - redistribute load internally to free nodes
    - try rotates
    - place freed nodes into a global pool as substitutes for failed nodes
  - If load increases
    - redistribute load internally
    - try rotates
    - grow leaf node by additional worker node

SVDS

- Structures based on range partitioning
  - Final agglomeration uses internal tree structure
    - Logarithmic delay -> scalable only within reason
  - Use tree structure for partial agglomeration to provide bounds

SVDS

- Programming:
  - User needs to provide:
    - Record creation code
    - Scan / evaluation code
    - Agglomeration code
      - Usually trivial
- Scan code can make use of globally already seen best results

SVDS

- Initial load distribution
  - Leaf nodes of 8
  - Node capacity between 50% and 150% normal distributed
  - ~10000 nodes in batch
  - load distribution consecutive
  - capacity estimate between 1 and 1/2 of actual expected capacity
- Appears impressive

8	1	10000	3.378%	±0.004%
8	1.1	10000	1.380%	±0.001%
8	1.2	10000	0.361%	±0.0005%
8	1.3	10000	0.038%	$\pm 5 \cdot 10^{-5}\%$
8	1.4	10000	0.006%	$\pm 8 \cdot 10^{-6}\%$
8	1.5	10000	0.019%	$\pm 2 \cdot 10^{-5}\%$
8	1.6	10000	0.215%	±0.0003%
8	1.7	10000	0.941%	±0.001%
8	1.8	10000	9.182%	±0.012%
8	1.9	10000	33.691%	±0.043%
8	2.0	10000	53.549%	±0.069%

SVDS

- Group survival
  - 30 minutes work time
  - Time between failures 120 minutes
  - Replacement in 5 minutes
  - Groups of 4
  - >99% survival rate

SVDS

- Practical question
  - Assume simple interaction
    - User designates agent who obtains new computing resources
    - Pays a start-up fee and otherwise pays per time-slice

Research Question

- Do we need a feedback operation?
  - Example:
    - 0-1 integer programming problem or 0-1 knapsack problem
    - Scan code can exclude large parts of the search space if we already know a good solution

Conclusions

- Outlined a tentative paradigm for self-organizing brute force calculations in the cloud
- Paradigm is SDDS
- Goal is simplicity of MapReduce