A Logic-Based Approach to Cloud Computing

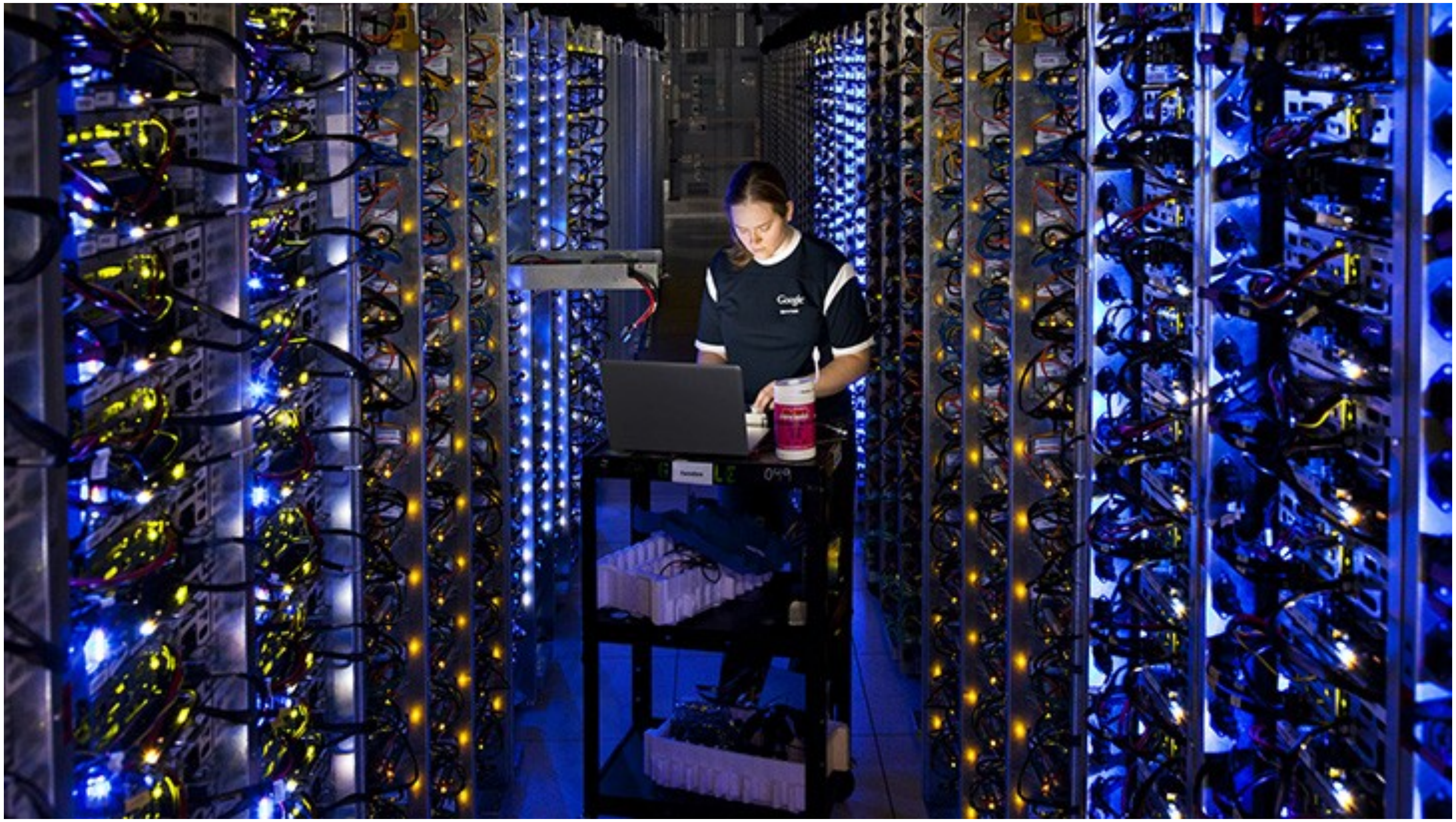
Jan Van den Bussche
(Hasselt University, Belgium)

joint work with Tom Ameloot and Frank Neven

Origins of Cloud Computing

Large websites (Amazon, Google, Facebook)

- large underlying database
- large number of simultaneous users
- use of computer clusters



The Cloud and Big Data

Make datacenters available for

- storing user data (DropBox, Google Drive)
- programmers (Amazon Elastic Compute Cloud)

Big Data: use computer clusters for **data**

- previously mainly for compute-intensive tasks
- now for data-intensive tasks

Our goal in this talk

Develop a theoretical model of data clouds

Using the tools of database queries (Ph.Kolaitis talk part 1)

Formalize properties of cloud systems (eventual consistency, coordination-freeness)

Model practical cloud programming languages (Bloom)

Investigate Hellerstein's conjectures (CALM, CRON)

Obtain more fundamental understanding of data clouds

What's in a cloud?

Set of **compute nodes**

Each node can:

- manage a local database
- receive messages from other nodes
- send messages to other nodes

Nodes work concurrently

Nodes continually cycle 3 phases: receive–update–send

(All nodes run the same program)

Example 1: Find all the red items

Each node has a piece of a dataset of items

Each node outputs its own red items

No communication is necessary

Recall Example 2: Library search (Google query)

One master-node and many slave-nodes

Each slave has a piece of the library: table of item–keyword pairs

Clustered: all items satisfying a certain keyword are on the same slave

Master has an index: mapping from keywords to slave ids

Master receives a keyword query

Consults index and forwards request to responsible slave

Slaves output matching items

Example 3: AND of two keywords (join)

If slaves have a copy of the index

Slave can process first keyword

Send matching items for confirmation te slave responsible for second keyword

Example 4: Keyword count (Map-Reduce)

Master gets a large list of keywords

Master still has an index, but slaves have no data yet

Master distributes entries to the slaves according to index

Slaves keep counts for each distinct received keyword

Problem: when does a slave know he has received everything?

Asynchronous communication

Messages may arrive out of order!

A simple “end of stream” message will not work

Coordination is needed for keyword counting

- Master keeps counts: # of messages sent to each slave
- Slaves periodically report # of messages received so far
- Master can ask for final results when counts match

More complicated communication than mere exchange of data

Relational transducer [Abiteboul,Vianu,Yesha 2000]

Formal model of a compute node in a data cloud

A machine that maintains a state and produces outputs in response to inputs

Choose three disjoint database schemas $\mathcal{S}_{\text{state}}$, $\mathcal{S}_{\text{in}}$, $\mathcal{S}_{\text{out}}$

States are instances of $\mathcal{S}_{\text{state}}$

Inputs and outputs are instances of $\mathcal{S}_{\text{in}}$ and $\mathcal{S}_{\text{out}}$

Behavior is given by a query $Q : \mathcal{S}_{\text{state}} \cup \mathcal{S}_{\text{in}} \to \mathcal{S}_{\text{state}} \cup \mathcal{S}_{\text{out}}$

Example of a relational transducer

Maintains a library of item–keyword pairs

Responds to insert/delete/keyword requests

$$\begin{aligned} \mathcal{S}_{\text{state}} &= \{\textit{Lib}/2\} \\ \mathcal{S}_{\text{in}} &= \{\textit{Ins}/2, \textit{Del}/2, \textit{Key}/1\} \\ \mathcal{S}_{\text{out}} &= \{R/2\} \end{aligned}$$

$$\begin{aligned} Q_{\textit{Lib}} &= \{(i,k) \mid (\textit{Lib}(i,k) \vee \textit{Ins}(i,k)) \wedge \neg \textit{Del}(i,k)\} \\ Q_R &= \{(i) \mid \exists k(\textit{Lib}(i,k) \wedge \textit{Key}(k))\} \end{aligned}$$

Example of a run

Lib		R	Ins		Del		Key
Prague	capital						
Prague	Czech Rep.		Telc	Moravia	Telc	boring	
Telc	Czech Rep.		Telc	EATCS			
Telc	boring						
Prague	capital						
Prague	Czech Rep.						
Telc	Czech Rep.						Czech Rep.
Telc	Moravia						
Telc	EATCS						
Prague	capital						
Prague	Czech Rep.	Prague					EATCS
Telc	Czech Rep.	Telc	Telc	fun	Telc	EATCS	fun
Telc	EATCS						
Telc	Moravia						
Prague	capital						
Prague	Czech Rep.						
Telc	Czech Rep.	Telc					
Telc	Moravia						
Telc	fun						

Relational transducer as a node in a data cloud

Each node has a unique id

State contains

- relations Id and All, initialized with own id and all ids
- local state relations
- initial input relations (piece of distributed database)
- relations for producing final output

Modeling of messages

Messages are facts of the form

$$R(\mathsf{dest\_id}, a_1, \ldots, a_k)$$

Set of incoming messages is input for the transducer

Transducer's output is set of outgoing messages

Example 2: Library search (Google query)

One master-node and many slave-nodes

Each slave has a piece of the library: table of item–keyword pairs

Master has an index: mapping from keywords to slave ids

Master receives a keyword query

Consults index and forwards request to responsible slave

Slaves output matching items

Master receives requests as $\mathit{Req}(r, k)$-facts

Messages from master to slave are $F(s, r, k, m)$-facts

Replies from slaves to master are $R(m, r, i)$-facts

Query for F:

$$\{(s, r, k, m) \mid \mathit{Req}(r, k) \wedge \mathit{Index}(k, s) \wedge \mathit{Id}(m)\}$$

Query for R:

$$\{(m, r, i) \mid \exists s \exists k (F(s, r, k, m) \wedge \mathit{Lib}(i, k))\}$$

In Datalog notation:

$$F(s, r, k, m) \leftarrow \mathit{Req}(r, k), \mathit{Index}(k, s), \mathit{Id}(m).$$
$$R(m, r, i) \leftarrow F(s, r, k, m), \mathit{Lib}(i, k).$$

Transducer network

A set of node ids and a transducer Q running on the nodes

While the network runs,

- each node has its local state
- a set of messages is in transit

This information constitutes the network's **configuration**

Network transition

One network transition:

1. pick a node v

2. some of the messages to v are delivered (possibly none: "heartbeat")

3. v makes a transition (according to Q)

4. v's outgoing messages are added to the network

In this way the network steps from configuration to configuration

Highly nondeterministic!

Eventual consistency

Facebook: Alice is in Prague, Bob in New York, Cynthia in Tokyo

A: “Just passed my violin exam!”

B: “Congrats!” — C: “Mine is tomorrow!”

Eventually everybody will see both reactions (in no particular order)

Amazon.com: A and B both view a book, only 1 copy left, A orders it

Until this info reaches B’s server, B can also order the book

Consistent transducer networks

Consider transducer network initialized with a distributed database

During a run, observe output facts being produced by the nodes

Active domain is finite, so a quiescence point will be reached after which no new output facts will be produced

Definition: The network is called **consistent** if, for every initial configuration, all fair runs yield the same set of output facts

Example 1: not consistent

Node 1 sends out two messages A and B

If Node 2 receives A first, output C

$A(v) \leftarrow \mathsf{All}(v).$
$B(v) \leftarrow \mathsf{All}(v).$
$MB() \leftarrow B(v).$
$C() \leftarrow A(v), \neg MB().$

Example 2: not consistent

Node 1 sends out two messages A and B

If Node 2 receives A and B simultaneously, output C

$A(v) \leftarrow \mathit{All}(v).$
$B(v) \leftarrow \mathit{All}(v).$
$C() \leftarrow A(v), B(v).$

Cloud programs and distributed queries

A transducer that is consistent on every set of nodes is called a **cloud program**

A cloud program computes a well-defined **distributed query:** a mapping from distributed databases to databases

- input: initial contents of state relations
- output: set of all output facts

Theorem: Consider a distributed query $\mathcal{Q}$.

1. $\mathcal{Q}$ is computable by a cloud program that uses only FO-queries (first-order logic, relational calculus)
2. $\underline{\text{if and only if}}$ $\mathcal{Q}$ is expressible in FO + while

Example of FO + while

Transitive closure T of relation E

```
$T := E$;
**while** $\exists x, y, z(T(x, y) \wedge T(y, z) \wedge \neg T(x, z))$
**do**
   $T := \{(x, z) \mid T(x, z) \vee \exists y(T(x, y) \wedge T(y, z))\}$
**end**
```

From FO + while to cloud program

Two steps:

1. Gather the entire distributed database in a single node
2. Simulate FO + while on a relational transducer

For step 1 the nodes send their local data around

Difficulty is to know that we have received everything

Need to coordinate (recall word-counting example)

Programming coordination: Example

Each node has a piece of a distributed database. Use relation A for the pieces.

They must together determine that all local pieces are empty

Each node sends around its id if its piece is empty. Use message relation M.

Keep record of received ids in state S and compare with All relation

Use output relation *Yes*

$$\begin{aligned} Q_M &= \{(j,i) \mid \mathit{All}(j) \land \mathit{Id}(i) \land \neg\exists x\, A(x)\} \\ Q_S &= \{(i) \mid \exists j\, M(j,i)\} \\ Q_{\mathit{Yes}} &= \{() \mid \forall i(\mathit{All}(i) \Rightarrow S(i))\} \end{aligned}$$

No coordination needed for monotone queries

<u>Definition</u>: A query $\mathcal{Q}$ is **monotone** if

$$\mathcal{I} \subseteq \mathcal{J} \quad \Rightarrow \quad \mathcal{Q}(\mathcal{I}) \subseteq \mathcal{Q}(\mathcal{J})$$

E.g. “All A’s are false” is not monotone

<u>Definition</u>: A transducer is **oblivious** if it does not use the relation All except for sending messages

Theorem: Consider a <u>monotone</u> distributed query $\mathcal{Q}$.

1. $\mathcal{Q}$ is computable by an <u>oblivious</u> cloud program that uses only FO-queries (first-order logic, relational calculus)

2. if and only if $\mathcal{Q}$ is expressible in FO + while

The CALM conjecture

When is coordination not needed? (When is parallelism most efficient?)

Hellerstein 2010 conjectured (CALM) that this is exactly when the distributed query to be computed is not monotone

To formalize this conjecture we need a formal definition of “coordination-freeness”

We propose such a definition and prove the conjecture under the assumption of **network independence**.

Network independence

Definition: The **global database** of a distributed database is the union of all its parts

A distributed query $\mathcal{Q}$ is called **network-independent** if, whenever $\mathcal{I}$ and $\mathcal{J}$ have the same global database, then $\mathcal{Q}(\mathcal{I}) = \mathcal{Q}(\mathcal{J})$

Example: "Output the contents of only those nodes that contain less than 3 facts" is **not** network-independent

Coordination-freeness: Intuition

There are two kinds of communication:

1. Data exchange among the nodes

2. Communication for other reasons (coordination)

When there is no coordination, and the data is already suitably partitioned, data exchange is not needed either

- every node can compute its part of the output by itself

Consider **redistributions** of distributed databases: same network, same global database, but distribution is different

Coordination-freeness: Definition

A cloud program $\mathcal{P}$, computing a distributed query $\mathcal{Q}$, is called **coordination-free** if for every distributed database $\mathcal{I}$ there exists a redistribution $\mathcal{I}'$ so that $\mathcal{P}(\mathcal{I}')$ already produces the entire result $\mathcal{Q}(\mathcal{I})$ without communication (only self-messages are allowed)

E.g. Program for "All A's are empty" is not coordination-free

- Only redistribution of $(\emptyset, \emptyset)$ is $(\emptyset, \emptyset)$ itself!
- Program does not produce Yes without communication

Transitive closure by a coordination-free program

Binary relation E distributed over a set of nodes

Nodes incrementally compute transitive closure of their local copy (state relation T)

But also send around their piece of relation E (message relation M)

Nodes add incoming facts to their local copy of E

$M(j,x,y) \leftarrow \mathit{All}(j), E(x,y).$
$E(x,y) \leftarrow M(j,x,y).$
$T(x,y) \leftarrow E(x,y).$
$T(x,z) \leftarrow T(x,y), T(y,z).$

CALM Theorem

The following are equivalent for a network-independent distributed query $\mathcal{Q}$:

1. $\mathcal{Q}$ is monotone

2. $\mathcal{Q}$ is computable by an oblivious cloud program

3. $\mathcal{Q}$ is computable by a coordination-free cloud program

We already know $1 \Rightarrow 2$

Proof: Oblivious implies coordination-free

Given: oblivious cloud program $\mathcal{P}$ computing distributed query $\mathcal{Q}$

Given a distributed database $\mathcal{I}$ on n nodes

Let I_0 be the global database of $\mathcal{I}$ in a single node

Since $\mathcal{Q}$ is network-independent, $\mathcal{P}(I_0)$ produces $\mathcal{Q}(\mathcal{I})$

This does not involve communication!

Redistribute $\mathcal{I}$ as n copies of I_0

$\mathcal{P}$ is oblivious, so the partial run of $\mathcal{P}$ on I_0 is still valid

Hence $\mathcal{Q}(\mathcal{I})$ is produced as desired

Proof: Coordination-free implies monotone

Given: coordination-free cloud program $\mathcal{P}$ computing distributed query $\mathcal{Q}$

Consider $\mathcal{I} \subseteq \mathcal{J}$ and fact $f \in \mathcal{Q}(\mathcal{I})$

Let I and J be global databases of $\mathcal{I}$ and $\mathcal{J}$

Consider two-node network (I, I); since $\mathcal{Q}$ is network-independent, $\mathcal{Q}(I, I) = \mathcal{Q}(\mathcal{I})$

There exists redistribution (I_1, I_2) of (I, I) so that $\mathcal{P}(I_1, I_2)$ produces $Q(\mathcal{I})$ without communication; wlog assume f is produced on node 1

Add $J - I$ to node 2; partial run of node 1 that produces f is still valid

Automated verification of cloud programs

Recall property of **consistency** of a transducer network: all runs produce the same set of output facts

Undecidable for FO-transducers

Decidable for “simple” FO-transducers [Ameloot 2014]

Assumes local piece of data is given in read-only “input” relations

Simple transducer

All queries are $\mathrm{UCQ}^{\neg}$ (unions of conjunctive queries with negation)

No recursive dependencies between message and state relations

No deletions (state relations can only grow)

All queries must be positive in message relations

Queries for sending must not mention state relations (only inputs and other messages)

Queries for state update must be **message-bounded**: existential variables must be bound to messages and can only occur (possibly negated) in input relations and other messages)

Theorem: Simple transducer networks can express precisely the distributed queries expressible (globally) in $UCQ^{\neg}$

Theorem: Consistency for simple transducer networks is decidable (coNEXPTIME-complete)

Note that even on a given finite distributed database as input, the transition system is still **infinite** due to resending of messages

Procedural vs declarative semantics

We have seen operational semantics of transducer network as a transition system

Can also express FO-transducers in Datalog$^\neg$ and give a declarative semantics using stable models

These can be formally proven equivalent

Causality

While messages may arrive out of order, there is still some causal order in the semantics of transducer networks

Facebook Anne: “I passed my violin exam!”

Facebook Bob and Cynthia: “Congratulations”—“Great”

It would be strange if reactions arrived at Anne before she sends the message!

Hellerstein 2010 has asked when we do not need causality?

It is possible to give a non-causal semantics to cloud programs expressed in positive Datalog

Theorem: The causal and the non-causal semantics coincide

Conclusion

Data clouds are a big thing (Big Data)

Logical methods can help understanding them better

Berkeley Bloom language is essentially FO-transducers

Expressive power

Behavior

Verification

Efficiency

References

T.J. Ameloot, F. Neven, J. Van den Bussche: Relational transducers for declarative networking

T.J. Ameloot: Deciding correctness with Fairness for simple transducer networks

T.J. Ameloot, J. Van den Bussche, P. Alvaro, W.R. Marczak, J.M. Hellerstein: A declarative semantics for Dedalus

T.J. Ameloot, J. Van den Bussche: Positive Dedalus programs tolerate non-causality

Follow-up works by Green, Ludaescher, Zinn

Ameloot, Ketsman, Neven PODS 2014 Best Paper Award