Cooperative Update Exchange in the Youtopia System

Łucja Kot, Christoph Koch

Cornell University

August 25, 2009

Youtopia

A system for collaborative sharing of relational data

- useful whenever a community needs to maintain a database

Many applications:

- scientific data sharing
- vertical social networking sites
- grass-roots collaboration involving data
- ....

Collaborative data sharing

Currently carried out through custom-built solutions

Youtopia = "Wiki software for relational data"

- users create and maintain a database together
- minimal understanding of databases required
- extensible to allow for organic growth in data and schema

Building a collaborative DBMS

Many challenges; needs to:

- be completely decentralized
- support dirty and incomplete data
- handle disagreement among users
- ...

Related work: Orchestra, Dataspaces, Fusion Tables, Cimple / DBLife, peer-to-peer data integration systems...

Update exchange

In this talk, introduce the logical data model for Youtopia

- an update exchange model (cf. Orchestra)

At logical level, a Youtopia repository consists of relations connected by mappings

- mappings are tuple-generating dependencies (tgds) - for now
- mappings propagate data changes between relations using a chase
  - different from the way mappings are used in data integration
- users can create new relations and mappings at any time
- no centralized restrictions or components (global schema, mapping acyclicity)

Chase example

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto

R
(NYS Tour Reviews)

company	attraction	review
XYZ	Geneva Winery	Great!
x_1	Niagara Falls	Nice

Chase example

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto

R
(NYS Tour Reviews)

company	attraction	review
XYZ	Geneva Winery	Great!
x_1	Niagara Falls	Nice

$$\sigma_1 : \mathrm{A}(l, n) \wedge \mathrm{T}(n, c, c') \rightarrow \exists r\ \mathrm{R}(c, n, r)$$

Chase example

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto
Niagara Falls	ABC	Buffalo

R
(NYS Tour Reviews)

company	attraction	review
XYZ	Geneva Winery	Great!
x_1	Niagara Falls	Nice

$$\sigma_1 : \mathrm{A}(l, n) \wedge \mathrm{T}(n, c, c') \rightarrow \exists r\ \mathrm{R}(c, n, r)$$

Chase example

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto
Niagara Falls	ABC	Buffalo

R
(NYS Tour Reviews)

company	attraction	review
XYZ	Geneva Winery	Great!
x_1	Niagara Falls	Nice
ABC	Niagara Falls	x_2

$$\sigma_1 : \mathrm{A}(l, n) \wedge \mathrm{T}(n, c, c') \rightarrow \exists r\ \mathrm{R}(c, n, r)$$

What about deletions?

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto

R
(NYS Tour Reviews)

company	attraction	review
~~XYZ~~	~~Geneva Winery~~	~~Great!~~
x_1	Niagara Falls	Nice

$$\sigma_1 : \mathrm{A}(l, n) \wedge \mathrm{T}(n, c, c') \rightarrow \exists r\ \mathrm{R}(c, n, r)$$

Cascading deletions - the backward chase

A
(NYS Attractions)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T
(Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto

R
(NYS Tour Reviews)

company	attraction	review
x_1	Niagara Falls	Nice

$$\sigma_1 : \mathtt{A}(l, n) \wedge \mathtt{T}(n, c, c') \rightarrow \exists r\ \mathtt{R}(c, n, r)$$

Handling deletions

Other ways to handle deletions:

- regenerate - user deletion was "misguided"
- ignore - user is requesting exception to mapping

We show how to handle the "hardest" case.

- If ignoring, nothing to do. If regenerating, run forward chase.
- Obvious practical design decisions - security, etc.

What makes the Youtopia chase different?

The algorithm allows for user participation

User steps in at points of ambiguity

- Sometimes, tuple generated by chase might already be in the database
- Or there may be more than one valid path for the chase
- User can specify that manually

Side effect - cycles allowed in mappings

Another chase example

C
(City)

city
Ithaca
Syracuse

S
(Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$

$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

Another chase example

C
(City)

city
Ithaca
Syracuse

S
(Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca
JFK	NYC	Ithaca

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$

$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

Another chase example

C
(City)

city
Ithaca
Syracuse
NYC

S
(Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca
JFK	NYC	Ithaca

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$

$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

Another chase example

C
(City)

city
Ithaca
Syracuse
NYC

S
(Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca
JFK	NYC	Ithaca
x_1	x_2	NYC

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$
$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

Another chase example

C
(City)

city
Ithaca
Syracuse
NYC
x_2

S
(Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca
JFK	NYC	Ithaca
x_1	x_2	NYC

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$

$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

The problem

Cyclical mappings generate infinite cascade of insertions

- reason for ubiquitous acyclicity restriction and variants

Observation: the problem arises from a lack of information on the part of the algorithm

- in reality, x_2 and `NYC` may be the same, but the system cannot assume that

Our solution: ask human to supply information

- done through simple and intuitive frontier operations

Frontier operations in action

C
(City)

city
Ithaca
Syracuse
NYC
x_2

S
(Suggested
Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca
JFK	NYC	Ithaca
x_1	x_2	NYC

$$\sigma_2 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$

$$\sigma_3 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$

Frontier operations in action

User is shown the frontier tuple and asked: is it new?

Two options, two frontier operations

- "no, it's a duplicate" - user unifies with an existing tuple
- "it's a new tuple" - user expands the frontier tuple

Chase may now be able to proceed further deterministically

- an update is the sequence of all operations that are the consequences of a single initial user insertion, deletion or null-replacement

Cycles OK because a point of ambiguity will be reached sooner or later on every chase path (forward chase)

Analogous situation for deletions

$$\sigma_1 : \mathtt{A}(l, n) \wedge \mathtt{T}(n, c, c') \rightarrow \exists r\ \mathtt{R}(c, n, r)$$

A deletion from R could cascade to either A or T

- again, choice left to the user
- system marks all potential deletions as deletion candidates
- user chooses which ones actually get deleted

The Youtopia chase in practice

Frontier operations powerful but slow

- can't lock down system while awaiting a frontier operation
- need to allow queries and other updates to proceed

If multiple chases are running, undesirable interleavings and interference may occur!

- We show how to prevent such interference

Interference example

R

A	B
a	b
c	d

S

B	C
b	f
d	g

T

A	C	D
a	f	i
c	g	j

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w\ \mathtt{T}(x, z, w)$$

Interference example

Update 1

R

A	B
a	b
c	d

S

B	C
b	f
d	g

T

A	C	D
~~a~~	~~f~~	~~i~~
c	g	j

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w \ \mathtt{T}(x, z, w)$$

Interference example

Update 1

R

A	B
a	b
c	d

S

B	C
b	f
d	g

T

A	C	D
c	g	j

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w \ \mathtt{T}(x, z, w)$$

Interference example

Update 2

R

A	B
a	b
c	d
k	b

S

B	C
b	f
d	g

T

A	C	D
c	g	j

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w \ \mathtt{T}(x, z, w)$$

Interference example

Update 2

R

A	B
a	b
c	d
k	b

S

B	C
b	f
d	g

T

A	C	D
c	g	j
k	f	x_1

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w \ \mathtt{T}(x, z, w)$$

Interference example

Update 1

R

A	B
a	b
c	d
k	b

S

B	C
~~b~~	~~f~~
d	g

T

A	C	D
c	g	j
k	f	x_1

$$\mathtt{R}(x, y) \wedge \mathtt{S}(y, z) \rightarrow \exists w \ \mathtt{T}(x, z, w)$$

Avoiding interference

Usually, such interference is undesirable

- should be disallowed or at least flagged for the user

Our contributions:

- define serializability
  - based on some ordering on the updates (eg. timestamps)
- present algorithm framework to enforce it
  - our framework can also be used to detect and flag violations

Defining serializability

All details in paper

Model the chase in a way that abstracts away human intervention

- basic unit of granularity: chase step
- each chase step associated with a set of writes and a set of reads

A sequence of chase steps is a schedule

- what does it mean for a (potentially incomplete) schedule to be serializable?
- intuition: "we haven't messed up anything yet"

Two definitions of serializability

Final-state serializability: in all possible futures in which updates execute serially to termination, resulting DB is identical to that produced in serial execution

Conflict-serializability: a definition based on read/write conflicts between updates

- if update *i* writes a data item, the write
  - should not be visible to reads by higher-priority updates
  - should not retroactively change the result of a read by a lower-priority update (unsafe reads)
- reads defined intensionally (read queries)

Enforcing conflict-serializability

Half the problem solved by use of versioning

- update i cannot see versions created by updates with lower priority

To prevent unsafe reads, two options:

1. prevent read if may be unsafe
2. allow read, abort reader if retroactively determined unsafe due to write

We take the second approach

- aborts will occur; require human attention in redo
- need to minimize number of aborts

Reducing the number of aborts

Reduce conflicts through scheduling

- Workload-specific; future work

Reduce cascading aborts: if update number i aborts, anyone who has read its writes must abort too

- can just abort all updates with lower priority than i (NAÏVE)
- or we can be more clever in computing read dependencies

Computing read dependencies

Dependencies can be computed in multiple ways

- trade precision (and number aborts) for running time
- `COARSE` computes dependencies from names of relations written to
- `PRECISE` computes dependencies from actual tuples written

See paper for experiments showing the relative performance of these algorithms

Current work

Currently building a Youtopia prototype

- Hopefully demo coming soon

Thank you!