A Logical Approach to Data Provenance

James Cheney

Logic & Semantics Seminar

University of Cambridge

November 17, 2006

based on joint work with P. Buneman, A. Chapman, S. Vansummeren

What is provenance?

- Informally:
  - History: creation, modifications; times, places, identities
  - Input data that explains “why” data in output
  - Input data that explains “where” output data “came from”
- Formally:
  - Figuring this out is purpose of this talk.
  - Caveat: Haven’t run most of this by colleages (or anyone) yet...

Examples

- Time/date/owner stamps in file system; system logs
- Line number info in parser/compiler; “diff/patch”
- Intermediate data structures for incremental computation
- Annotations/citations in scientific databases
- Problem: Automated support lacking
- Problem: Few guarantees, especially when data is mobile
- Danger: Solutions that give us a warm, fuzzy feeling but no solid foundation

Provenance Challenges

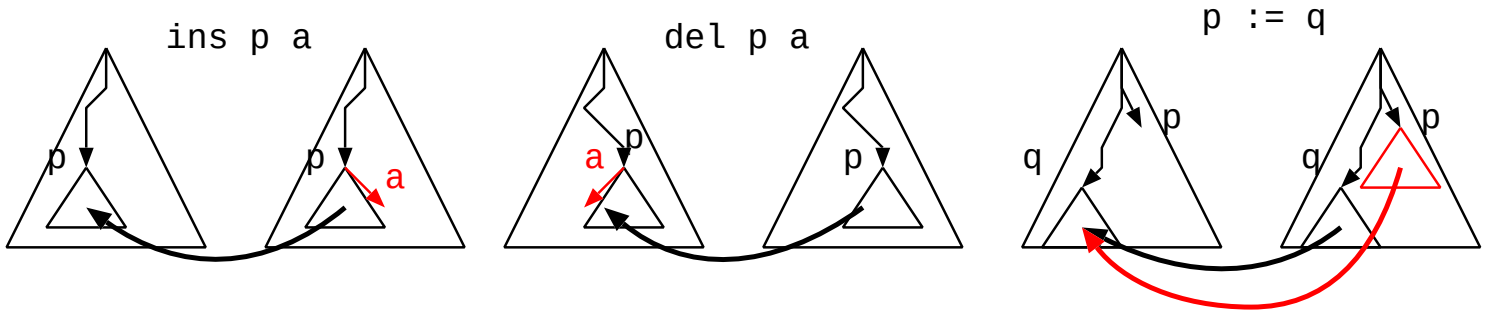
- There are (at least) two significant challenges in tracking and managing provenance
  1. *Policy*: that is, what should we be doing, and how can we argue that what we do is sufficient/correct?
  2. *Mechanism*: that is, how to build systems that efficiently capture (and exploit) some kind of provenance information?
- Most existing work focuses on (2), but without a good answer for (1), it's not clear how to judge whether such approaches are solving the "right" problem.
- Goal: Framework for comparing
  - *expressiveness* of tracking approaches
  - *complexity* of exact provenance problems.

Provenance/lineage in databases

- Lineage tracing (Woodruff, Stonebraker 1997)
  - Required external “inverse” functions
- Lineage for DB views (Cui, Wiener, Widom 2000,2003)
  - Defined lineage for arbitrary relational calculus queries
  - Doesn’t work well for negation, aggregation
- “Why-” and “where-” provenance (Buneman, Khanna, Tan 2001)
  - Distinguished “source” of copied data from “witness” (similar to lineage)
  - Where-provenance sensitive to query syntax.
  - Later related to view annotation propagation/deletion problem (PODS 2002)

Provenance in the presence of updates

- Copy-paste provenance (Buneman, Chapman, Cheney, Vansummeren 2006, 2007)
  - View database(s) as tree-structured namespace
  - Model data updates as atomic insert, delete, copy operations
  - Define & store provenance links showing “where data came from”



Complaints/questions

- Syntactic definitions without semantic foundation (esp. where-provenance, copy-paste)
- Focus on implementing “something reasonable” efficiently, not on what makes a solution reasonable.
- What makes some interesting-seeming data “provenance”?
- How can we generalize provenance to full query languages (or other computational settings?)

New principles

- Provenance is metadata describing properties of the query
- It should tell us both
  - (Value correctness) How the actual output value was actually computed. (“What happened?”)
  - (Dependency correctness) How the output value depends on the input. (“What would happen if ...?”)
- It should be defined in terms of the semantics, not the syntax, of the operation
  - although we can ultimately only *compute* (or *approximate*) the provenance using syntactic representation.

Defining dependency

- Notions of *dependency* are important in many settings
  - In database theory, functional, inclusion, multivalued dependencies are used in database design.
  - In incremental computation, dependency graphs help identify what parts of computation to rebuild when inputs change.
  - In program analysis/automated debugging, dependences among program variables, functions, etc. play central role.
  - Dependences also arise naturally in probability theory, differential calculus
- The notion of dependency we want is similar to, but not equal to, any of the above.

Framework

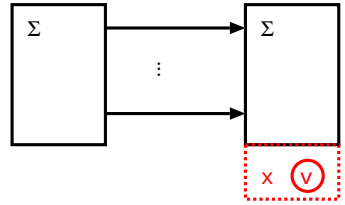
- Consider simple case: data stored in key-value maps

(Labels)	x, y	$\in$	Lab
(Values)	v	$\in$	Val
(Signatures)	Σ	$\subseteq$	Lab
(States)	σ	$:$	$\Sigma \to Val$
(State sets)	$[\![\Sigma]\!]$	$=$	$\{\sigma : \Sigma \to Val\}$
(Functions)	F	$\in$	$[\![\Sigma]\!] \to [\![\Sigma']\!]$

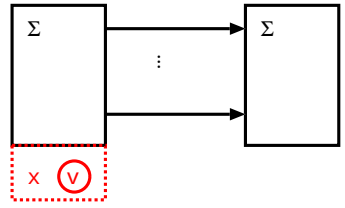
- We'll often cheat and write
  - $\sigma : \Sigma$ for $\sigma \in [\![\Sigma]\!]$
  - $F : \Sigma \to \Sigma'$ for $F \in [\![\Sigma]\!] \to [\![\Sigma']\!]$.
  - $\Sigma = x, y, \ldots$ instead of $\{x\} \uplus \{y\} \uplus \ldots$.

Examples

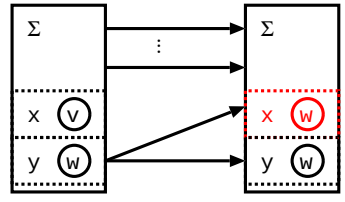
- Insertion: ins $x = v : \Sigma \rightarrow \Sigma, x$



- Deletion: del $x : \Sigma, x \rightarrow \Sigma$



- Copying: $x := y : \Sigma, x, y \rightarrow \Sigma, x, y$



Dependency

Definition 1. *Given* $F : \Sigma \to \Sigma'$*, we say that* $x' \in \Sigma'$ depends on $x \in \Sigma$ *if*

$$\exists \sigma : \Sigma, v \in Val.F(\sigma)(x') \neq F(\sigma[x := v])(x')$$

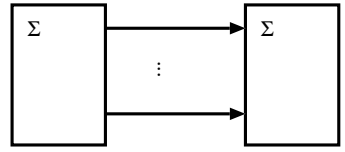
- That is, x' depends on x in F if changing x in the input can change x' in the output.

Definition 2. *Given* $F : \Sigma \to \Sigma'$*, we say that* $x' \in \Sigma'$ *is* constant *if for some* v*,*

$$\forall \sigma : \Sigma.F(\sigma)(x') = v$$

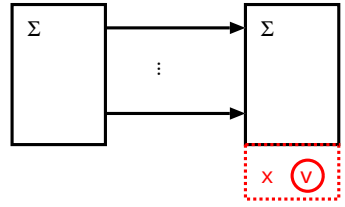
Examples

- $F = id_\Sigma$: every $x \in \Sigma$ depends on itself (only).



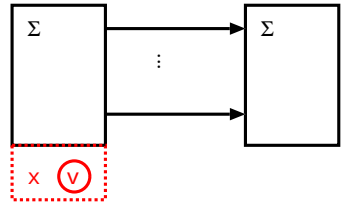
Examples

- $F = id_\Sigma$: every $x \in \Sigma$ depends on itself (only).
- $F =$ ins $x = v : \Sigma \to \Sigma, x$: every $y \in \Sigma$ depends on itself (only), x depends on nothing and is constant.



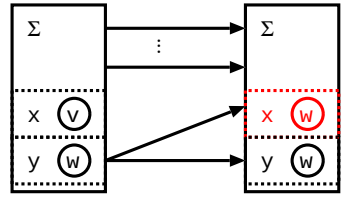
Examples

- $F = id_\Sigma$: every $x \in \Sigma$ depends on itself (only).
- $F =$ ins $x = v : \Sigma \to \Sigma, x$: every $y \in \Sigma$ depends on itself (only), x depends on nothing and is constant.
- $F =$ del $x : \Sigma, x \to \Sigma$: every $y \in \Sigma$ depends on itself (only), x doesn't exist in output.



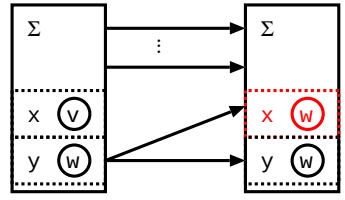
Examples

- $F = id_\Sigma$: every $x \in \Sigma$ depends on itself (only).
- $F =$ ins $x = v : \Sigma \to \Sigma, x$: every $y \in \Sigma$ depends on itself (only), x depends on nothing and is constant.
- $F =$ del $x : \Sigma, x \to \Sigma$: every $y \in \Sigma$ depends on itself (only), x doesn't exist in output.
- $F = x := y : \Sigma, x, y \to \Sigma, x, y$: every $z \in \Sigma, y$ depends on itself; x depends on y.



Examples

- $F = id_\Sigma$: every $x \in \Sigma$ depends on itself (only).
- $F =$ ins $x = v : \Sigma \to \Sigma, x$: every $y \in \Sigma$ depends on itself (only), x depends on nothing.
- $F =$ del $x : \Sigma, x \to \Sigma$: every $y \in \Sigma$ depends on itself (only), x doesn't exist in output.
- $F = x := y : \Sigma, x, y \to \Sigma, x, y$: every $z \in \Sigma, y$ depends on itself; x depends on y.



- Hey! Those arrows in the "informal" diagrams seem to correspond to dependences!
- This is not a coincidence.

Composition

- What about the composition of two functions?
- If x' depends on x in $F : \Sigma \to \Sigma'$, and x'' depends on x' in $G : \Sigma' \to \Sigma''$, does x'' depend on x in $G \circ F$?
- In general: No.
- Example:

$$[\![x := \neg y]\!] \models x \leftarrow y, y \leftarrow y$$
$$[\![z := x \vee y]\!] \models z \leftarrow x, z \leftarrow y$$
$$[\![x := \neg y; z := x \vee y]\!] \circ G \not\models z \leftarrow y$$

- Without compositionality, we can't do much.
- Insert-delete-copy dependences turn out to be compositional.

New notation

- Let's introduce more compact/suggestive notation.
- Think of F as a *model*
- Think of assertions such as "x' depends on x" as *formulas* $x' \leftarrow x$
- Write "x' depends on x in F" as $F \models x' \leftarrow x$
- Write "x' constant in F" as $F \models const(x')$
- Can also allow FO connectives $\wedge, \vee, \supset$, and quantifiers (over Σ, Σ'); semantics standard.
- Let IDC be the set of functions constructible by composing insert, delete, and copy operations.
- Write $S \models \phi$ for $\forall F \in S.F \models \phi$.

Identical dependences

- For insert-delete-copy operations, dependences are of a special form.

Definition 3. *Given* $F : \Sigma \to \Sigma'$*, we say that* $x' \in \Sigma'$ *is* identically dependent on $x \in \Sigma$ *(*$F \models x' \Leftarrow x$*) if* $F(\sigma)(x') = \sigma(x)$ *for every* $\sigma : \Sigma$*.*

Lemma 4 (Identity dependences unique)**.**
$\models x'_1 \Leftarrow x \wedge x'_2 \Leftarrow x \supset x_1 = x_2$*.*

Lemma 5 (Identity dependences compose)**.** *If* $F \models x' \Leftarrow x$ *and* $G \models x'' \Leftarrow x'$ *then* $G \circ F \models x'' \Leftarrow x$*.*

Theorem 6. *If* $IDC \models (x' \leftarrow x) \iff (x' \Leftarrow x)$*.*

Complexity

- For IDC functions $u : \Sigma_1 \to \Sigma_2$, we can decide whether $[\![u]\!] \models x' \Leftarrow x$ in $PTIME$
- Idea: Evaluate u symbolically
  - Consider symbolic states $\sigma^* : \Sigma' \to \Sigma \cup \{\bot\}$
  - Start with $\mathsf{id}^*_\Sigma = [x_1 := x_1, \ldots, x_n := x_n]$, if $\Sigma = \{x_1, \ldots, x_n\}$
  - Given $u : \Sigma' \to \Sigma''$, evaluate $P[\![u]\!] : (\Sigma' \to \Sigma) \to (\Sigma'' \to \Sigma)$ as follows:

$$
\begin{aligned}
P[\![\mathsf{ins}\ x := v]\!](\sigma^*) &= \sigma^*[x := \bot] \\
P[\![\mathsf{del}\ x]\!](\sigma^*) &= \sigma^* - x \\
P[\![x := y]\!](\sigma^*) &= \sigma^*[x := y] \\
P[\![u; u']\!](\sigma^*) &= P[\![u']\!](P[\![u]\!](\sigma^*))
\end{aligned}
$$

Correctness of provenance tracking

- Example:

$$P[\![\mathsf{ins}\ w = 1; x := z; \mathsf{del}\ z]\!](\mathsf{id}^*_{x,y,z}) = [w := \bot, x := z, y := y]$$

- Claim: Symbolic evaluation of

  Theorem 7. *If* $u \in IDC$, *then*

  1. $[\![u]\!] \models x \Leftarrow y$ *iff* $P[\![u]\!](\mathsf{id}^*_\Sigma)(x) = y$ *and*
  2. $[\![u]\!] \models const(x)$ *iff* $P[\![u]\!](\mathsf{id}^*_\Sigma)(x) = \bot$.

- That is, the provenance-tracking analysis captures exactly the true dependence information.

So far, so good... so what?

- I claim that we now fully understand dependences/provenance for IDC updates.
- This is kind of unexciting since IDC is an utterly trivial model of computation.
- Way more limited than previously studied query languages or other treatments of provenance (e.g. trees, relations)
- So let's see how far we can push this approach.
- Things get more interesting if we consider
  - Functions on domain ($\wedge, \vee, +, *$)
  - Conditionals (if then else)
  - Structured data (trees, sets/relations)

Built-in functions on domain

- Generalization #1: Suppose we allow primitive functions over domain Val.
- Example: $Val = (\mathbb{B}, 0, 1, \wedge, \vee, \neg)$. Call this $IDC + \mathbb{B}$.
- Example: $Val = (\mathbb{Z}, 0, 1, +, -, \cdot, zero?)$. Call this $IDC + \mathbb{Z}$
- Many other possible constraint domains ($\mathbb{R}, \mathbb{R}_{lin}$, etc.)
- Effect on complexity?

Built-ins: complexity

- In general, complexity depends on domain.
- For finite domains, generally remains decidable.

Theorem 8. *For any $F \in IDC + \mathbb{B}$, deciding whether $\leftarrow$ is NP-complete; deciding $\Leftarrow$ or $const$ is co-NP-complete.*

Proof. Upper bounds straightforward. For lower bounds, let $P(\vec{y})$ be a Boolean formula.

- $[\![x := P(\vec{y}) \wedge x]\!] \models x \leftarrow x$ iff P satisfiable
- $[\![x := P(\vec{y})]\!] \models const(x)$ iff P valid
- $[\![x := P(\vec{y}) \wedge x]\!] \models x \Leftarrow x$ iff P valid

□

Built-ins: complexity

- For infinite domains, can easily be undecidable:

  Theorem 9. *There exists $F \in IDC + \mathbb{Z}$ such that deciding whether $F \models x \leftarrow x$ is undecidable.*

- Proof idea: Let $P(\vec{y}) = 0$ be a Diophantine equation over $\vec{y}$. Let F be

  $$F = zero?(P(\vec{y})) \cdot x$$

  where $zero?(0) = 1, zero?(n) = 0$ otherwise

- If P has a solution, then x depends on x; otherwise not.

Built-ins: How to fix

- What can we do in practice?
- Option #1: Seek decidable/tractable conservative approximation to true set of dependences
- Simple approach: each output may depend on all inputs read
- Examples:

$$\begin{aligned} x := x^3 + y + 1 &\implies MayDep(x) = \{x, y\} \\ z := \neg y; x := y \vee z &\implies MayDep(x) = \{y\} \\ x := 1 &\implies MayDep(x) = \{\} \end{aligned}$$

- Note that analysis result can depend on syntax, but "true" dependences do not.

Built-ins: How to fix

- What can we do in practice?
- Option #2: Generalize dependence formulas to include *expressions*
- Say that x' identically depends on expression e ($F \models x \Leftarrow e$ if $F(\sigma)(x') = [\![e]\!]\sigma$ for every σ.
- Examples:

$$\begin{aligned} [\![x := y^3 + y + 1]\!] &\models x \Leftarrow y^3 + y + 1 \\ [\![z := \neg y; x := y \vee z]\!] &\models x \Leftarrow y \vee \neg y \\ [\![x := 1]\!] &\models x \Leftarrow 1 \end{aligned}$$

- This is exact & compositional, but hard to decide implications/equivalences among dependences.

Conditionals

- Generalization #1: Conditionals (if then else).

$$\begin{aligned} u &::= \cdots \mid \text{if } c \text{ then } u_1 \text{ else } u_2 \\ c &::= x = v \mid x = y \end{aligned}$$

- Call this language $IDC + \text{if}$.
- Non-identity dependences become possible.
- e.g. in

$$F = \text{if } x = 1 \text{ then } y = 1 \text{ else } z = w$$

y, z depend on x, and z also depends on w, but y, z are not always copies of x (or w).

Conditionals: complexity

- Perhaps unsurprisingly, as soon as we have conditionals, testing for data dependence becomes (co)NP-hard.

  Theorem 10. *For $F \in IDC+$* if*, deciding $\leftarrow$ is NP-hard; deciding $\Leftarrow$ and $const$ is co-NP-hard.*

- Proof sketch: Encode $3SAT$ instance P using conditionals; set things up so that identity dependence holds iff P is valid, or data dependence exists iff P is satisfiable.

  if P then $x := x$ else $x := 0$

  if P then $x := 0$ else $x := x$

Conditionals: how to fix?

- In order to regain a measure of compositionality, we consider *conditional dependences*.
- Idea: Limit the states we consider according to conditions we encounter.
- For example, recall

$$F = \text{if } x = 1 \text{ then } y = 1 \text{ else } z = w$$

- Over states σ satisfying $x = 1$, $F \models const(y) \wedge z \Leftarrow z$.
- Also, over states σ satisfying $x \neq 1$, $F \models y \Leftarrow y \wedge z \Leftarrow w$.

Conditional dependences

- We generalize models so that F can be a *partial* function. States $\sigma : \Sigma$ range only over F's domain.
- We introduce new dependence formulas $c \Vdash \phi$, where ϕ is a dependence logic formula and c is a formula over input signature Σ.

Definition 11. *We say that F satisfies ϕ under condition c (written $F, S \models\Vdash \phi$ if $F \circ \delta_c \models \phi$, where*

$$\delta_c(\sigma) = \begin{cases} \sigma & (c \models \sigma) \\ \bot & (c \not\models \sigma) \end{cases}$$

- That is, $c \Vdash \phi$ holds in a model if ϕ holds with respect to possible input states satisfying c.

Tracking conditional dependences

- For a $IDC+$ if update, consider conditional, annotated states $c \Vdash \sigma^*$
- Track ordinary operations on σ^* as usual (ignoring c).
- Track if-then-else statements as follows:

$$[\![\text{if } c' \text{ then } u_1 \text{ else } u_2]\!](c \Vdash \sigma^*) = \begin{cases} [\![u_1]\!](c \wedge c' \Vdash \sigma^*) & \sigma^* \models c' \\ [\![u_2]\!](c \wedge \neg c' \Vdash \sigma^*) & \sigma^* \not\models c' \end{cases}$$

- (Note: I'm cheating here; c' needs to be specialized using provenance information in σ^*)
- Generates valid conditional dependency information; may not tell us anything about F's behavior off c

Trees

- Generalization #3: Consider states with hierarchical structure (trees) rather than just flat structure.
- Values are now either base values or finite partial tree-valued maps.

$$Tree = \mu X.Val \uplus (Lab \rightharpoonup_{\mathsf{fin}} X)$$

- Consider insert-delete-copy operations on trees:

$$\begin{array}{lcl} p & ::= & \epsilon \mid p.x \\ u & ::= & \mathsf{ins}\ x = v\ \mathsf{into}\ p \mid \mathsf{del}\ x\ \mathsf{from}\ p \mid p := q \end{array}$$

- Problem: operations can now fail (if path missing).
- Consider partial functions $F : Tree \rightharpoonup Tree$

Trees: problems

- Because of subtree copying, data can grow exponentially with update size

$$(\text{ins } a = q, b = q \text{ into } p; q := p; \text{del } a, b \text{ from } p)^n$$

- So, “obvious” algorithms for computing dependences exponential
- However, no obvious (co-)NP-hardness reduction...
- If we leave out subtree copying, then trees are no more complicated (or interesting) than flat maps.

Trees: problems

- Previous definition of identity dependence is “global”.

$$F \models p \Leftarrow q$$

  means that entire subtree p is a copy of q.

- Easy to violate using nested copying:

$$a := c/d; \mathsf{ins}\ b = v\ \mathsf{into}\ a; a/b := c/e$$

  Here, the result tree a depends on the input, but is not a copy of a part of the input.

- But a is a *composite* of copies.
- That is, each part of a is “locally” identified with part of the input.

Local dependences

- Two values T and U are similar modulo $C \subset Lab$ ($T \sim_C U$) if either
  1. they are equal data values, or
  2. they are both trees, and $dom(T) - C = dom(U) - C$.
- Define *local dependences* as follows:

  Definition 12. *Given a function $F : Tree \rightharpoonup Tree$, path q is* locally dependent *on path p (written $F \models q \hookleftarrow p$) if there exists a finite $C \subseteq Lab$ such that, whenever $T \in dom(F), p \in T$, we have $q \in F(T)$, and $F(T).q \sim_C T.p$.*
- Idea: $T.p$ and $F(T).q$ are the same "almost everywhere" (i.e., except for a finite number of children explicitly modified by F).

Trees: good news!

- Example: Insertion. $[\![\text{ins } x = v \text{ into } p]\!] \models p \hookleftarrow p$ since $C = \{x\}$ works.
- Example: Deletion. $[\![\text{del } x \text{ from } p]\!] \models p \hookleftarrow p$ since $C = \{x\}$ works.
- Local dependency is compositional.

Theorem 13. *If* $F, G : Tree \rightharpoonup Tree$ *and* $F \models q \hookleftarrow p$ *and* $G \models r \hookleftarrow q$ *then* $G \circ F \models r \hookleftarrow p$.

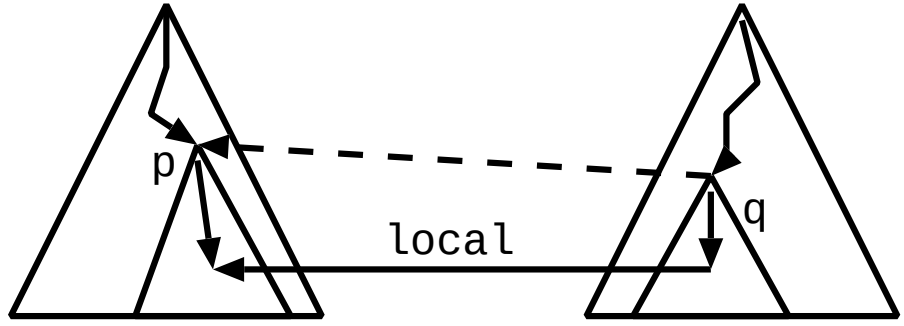
Proof. Definition chasing; if $G(F(T)).r \sim_D F(T).q$ and $F(T).q \sim_C T.p$ then $G(F(T)).r \sim_{C \cup D} T.p$. □

Trees: good news!

- All dependences in $IDC(Tree)$ operations are generated by local dependences. “I depend on you if my descendant locally depends on your descendant”

  Conjecture 14 (Babysitting)**.** *Let $F \in IDC(Tree)$ be given. Then $F \models q \leftarrow p$ if and only if there exist $q' \geq q, p' \geq p$ such that $F \models q' \hookleftarrow p'$.*

- Pretty sure this is true, but not sure how to prove it yet...

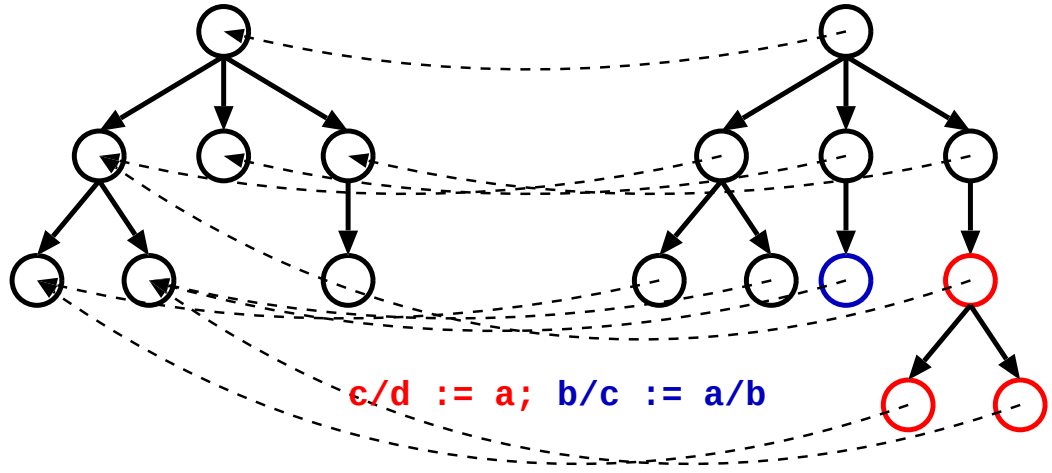


Trees: good news!

- Assuming conjecture, can compute full dependence information in $PTIME$ from local dependences.
- Local dependence info worst-case exponential, but this seems pathological
  - Could fix this by allowing “moving” (cut-paste) subtrees but not “duplicating” (copy-paste)
- Also, identity dependences can be computed from local dependences; skip details

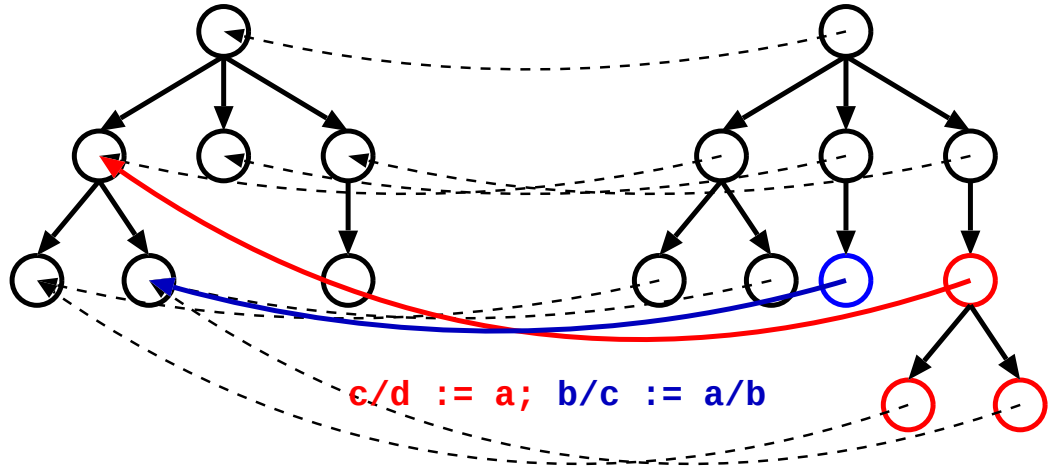
Trees: computing provenance

- For pure $IDC(Tree)$ operations, can easily compute local dependences “online” (looking at data)



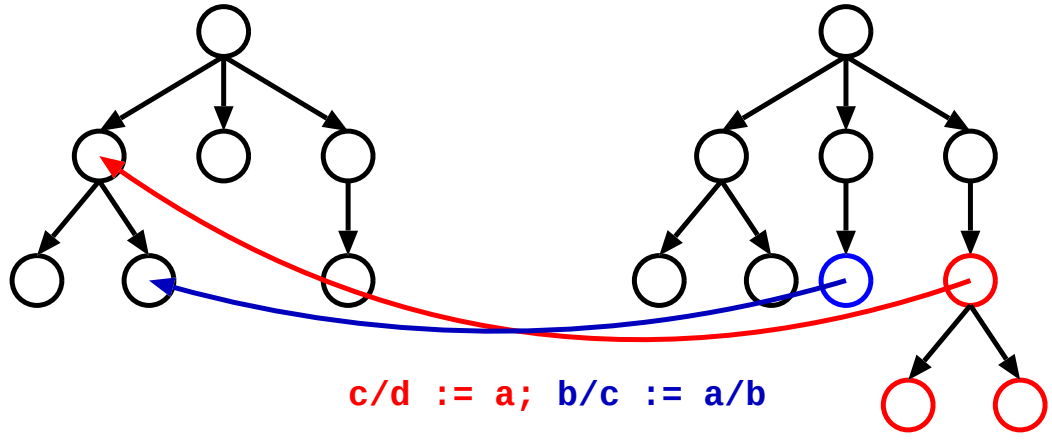
Trees: computing provenance

- For pure $IDC(Tree)$ operations, can easily compute local dependences “online” (looking at data)
- This is expensive (lots of redundant edges); much better to compute “interesting” edges “offline”



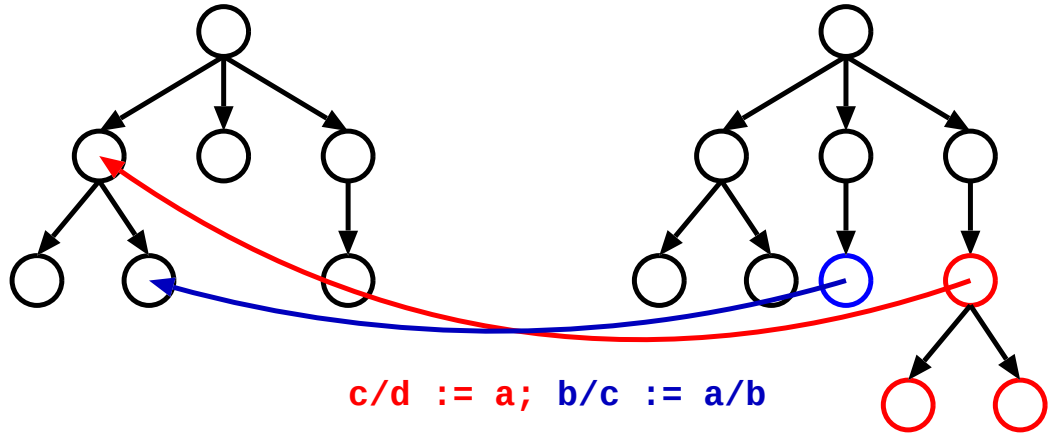
Trees: computing provenance

- For pure $IDC(Tree)$ operations, can easily compute local dependences “online” (looking at data)
- This is expensive (lots of redundant edges); much better to compute “interesting” edges “offline”
- This is exactly what we implemented in [Buneman, Chapman, Cheney 2006]



Trees: computing provenance

- For pure $IDC(Tree)$ operations, can easily compute local dependences "online" (looking at data)
- This is expensive (lots of redundant edges); much better to compute "interesting" edges "offline"
- This is exactly what we implemented in [Buneman, Chapman, Cheney 2006]
  - Warm, fuzzy feeling can be justified with some math!



Conclusions

- Provenance tracking:
  - important problem
  - many competing solutions
  - few design principles
  - few dimensions for formal comparison
- This talk:
  - “complete” formal foundation for dirt simple computational model
  - considered “baby step” extensions in several directions
  - bad news: conditionals, built-in functions lead to high complexity
  - good news: copying, trees well-behaved.

Conclusions

- Despite my early doubts, logic and semantics crucial to understanding and elucidating approaches to data provenance problem.
- Crucial ingredient: concept of *data dependence*
- Recent work: developed a dynamic provenance tracking technique for general (nested) relational queries
- Future work: combine features considered so far, “scale up” to real query languages, other models of computation...