Query Processing for Non-traditional Applications

CS848 Spring 2013

Cheriton School of CS

Updating Data

Plan

1. What are updates (how to understand dynamic aspects of instances)?

Plan

1. What are updates (how to understand dynamic aspects of instances)?
2. How do we understand updates *in our framework*?

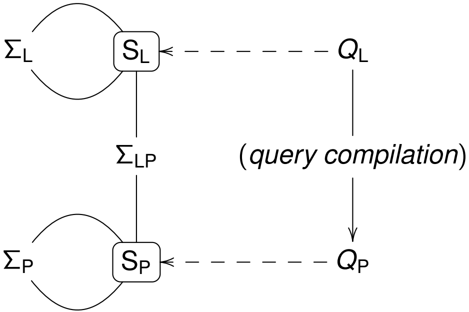
Plan

1. What are updates (how to understand dynamic aspects of instances)?
2. How do we understand updates *in our framework*?
   - updates and logical relations
   - updates and constraints
   - updates and access paths

Plan

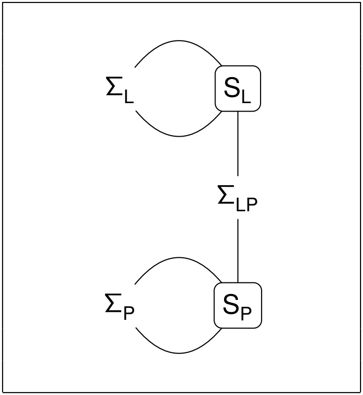
1. What are updates (how to understand dynamic aspects of instances)?
2. How do we understand updates *in our framework*?
    - updates and logical relations
    - updates and constraints
    - updates and access paths
3. Difficulties on the way
    - sequencing updates
    - value invention

Physical Design and Query Compilation: Overview

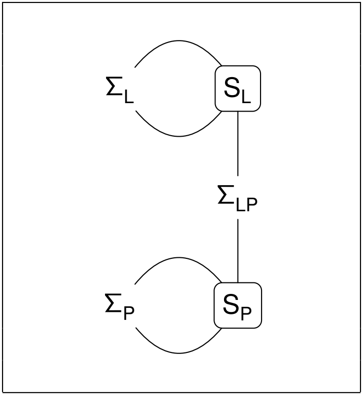


UPDATES IN NUTSHELL

Physical Design and Updates: Overview

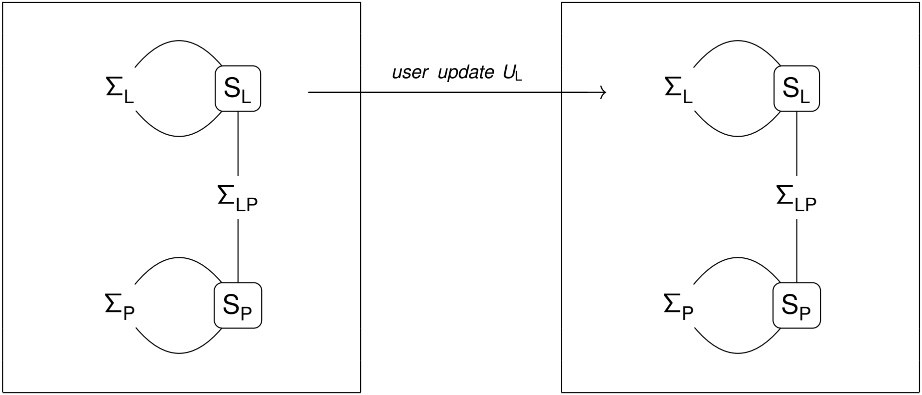


old instance



new instance

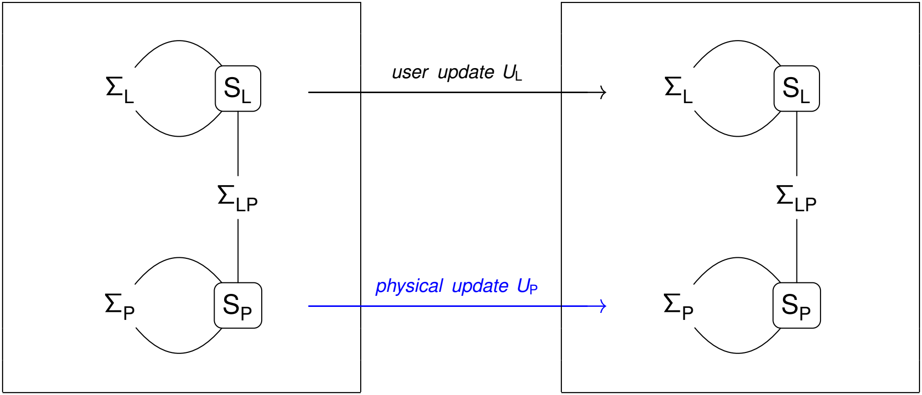
Physical Design and Updates: Overview



old instance

new instance

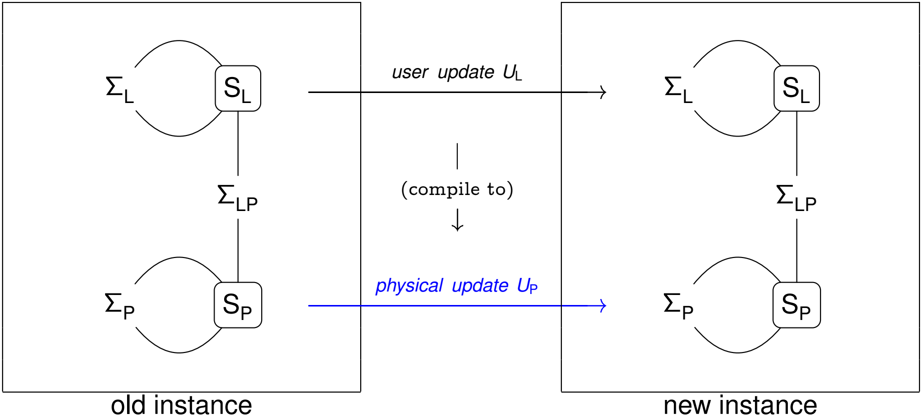
Physical Design and Updates: Overview



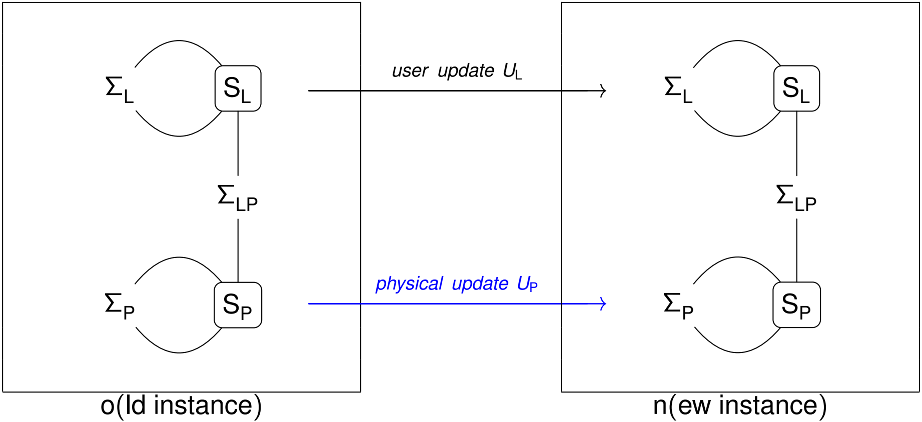
old instance

new instance

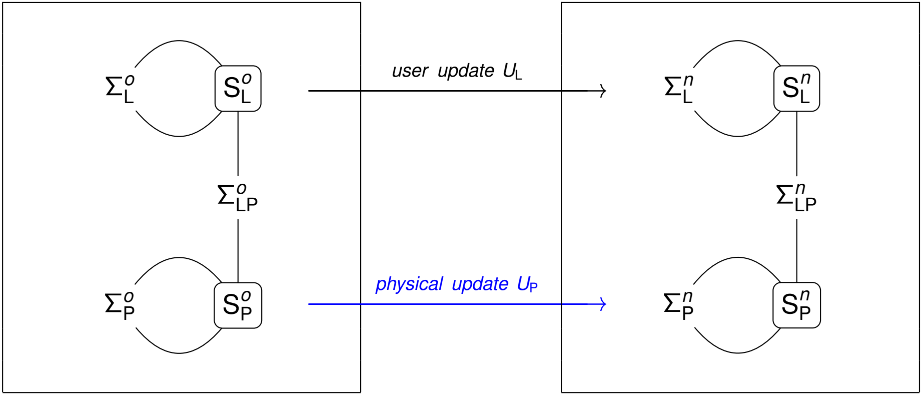
Physical Design and Updates: Overview



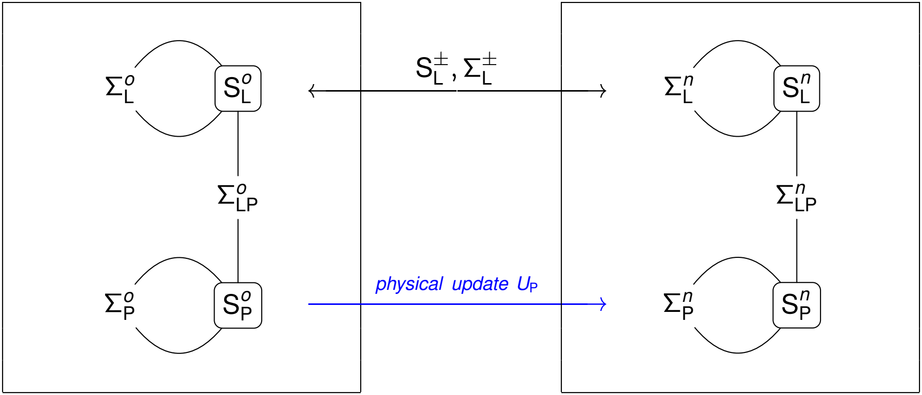
Update Schema



Update Schema



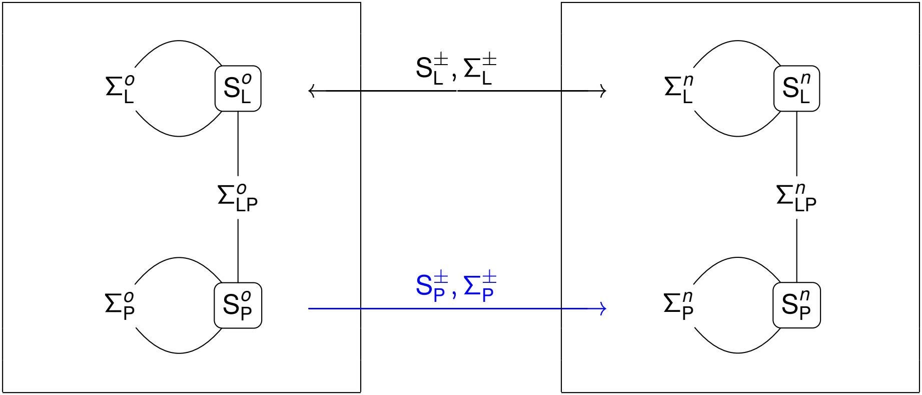
Update Schema



Σ^o_L
S^o_L
Σ^o_LP
Σ^o_P
S^o_P
$\mathsf{S}^\pm_\mathsf{L}, \Sigma^\pm_\mathsf{L}$
physical update U_P
Σ^n_L
S^n_L
Σ^n_LP
Σ^n_P
S^n_P

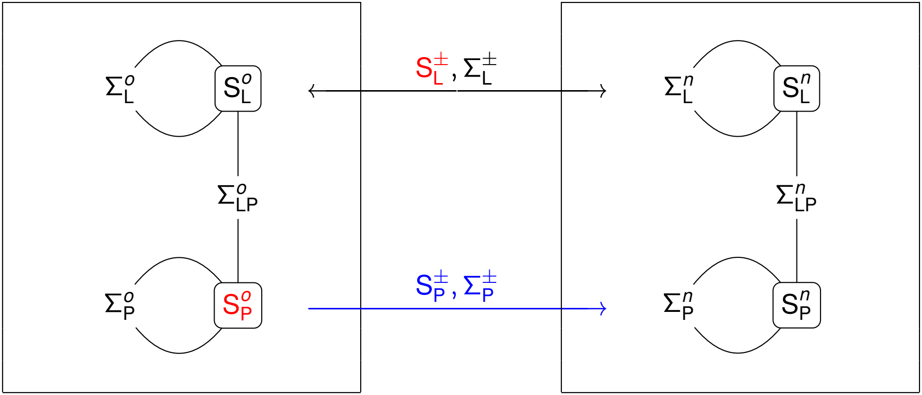
$$\mathsf{S}^\pm_\mathsf{L} = \{P^+, P^- \mid P \in \mathsf{S}_\mathsf{L}\},$$
$$\Sigma^\pm_\mathsf{L} = \{\forall\bar{x}.(P^o(\bar{x}) \vee P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \vee P^-(\bar{x})) \mid P \in \mathsf{S}_\mathsf{L}\}$$

Update Schema

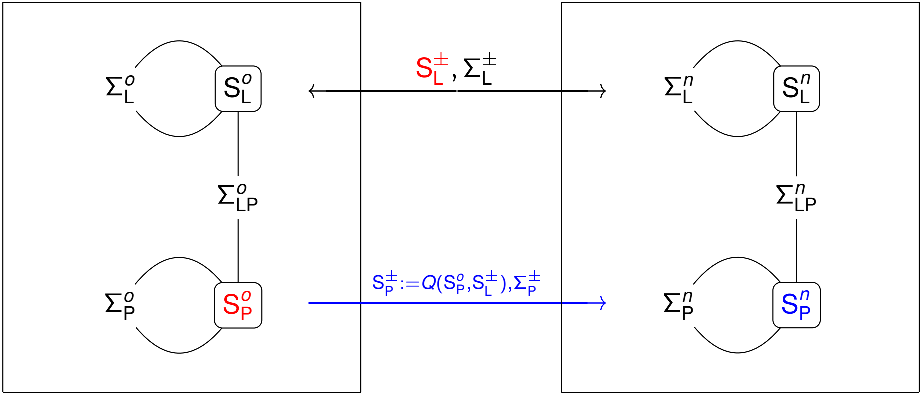


$$S_L^\pm = \{P^+, P^- \mid P \in S_A\},$$
$$\Sigma_L^\pm = \{\forall \bar{x}.(P^o(\bar{x}) \vee P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \vee P^-(\bar{x})) \mid P \in S_A\}$$

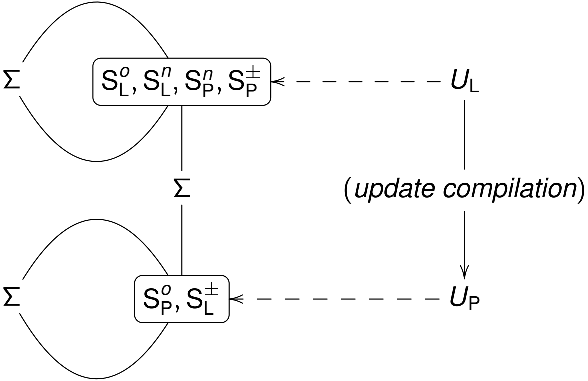
Update Schema



Update Schema



Physical Design and Update Compilation



Σ

$S_L^o, S_L^n, S_P^n, S_P^\pm$

U_L

Σ

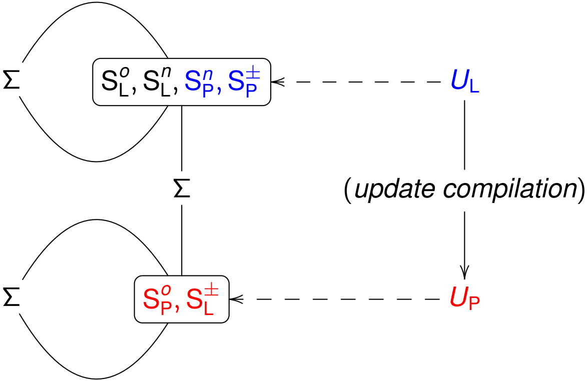
(*update compilation*)

Σ

$S_P^o, S_L^\pm$

U_P

Physical Design and Update Compilation



Σ

$S^o_L, S^n_L, S^n_P, S^\pm_P$

U_L

Σ

(*update compilation*)

Σ

$S^o_P, S^\pm_L$

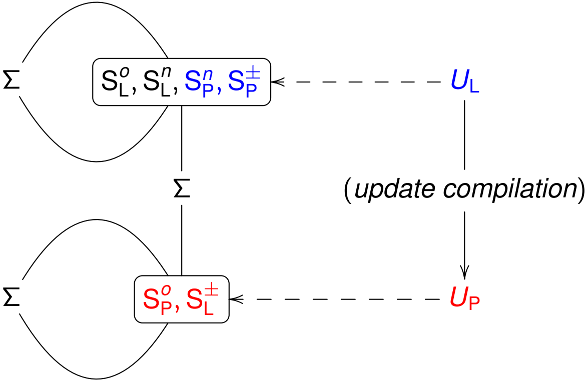
U_P

Physical Design and Update Compilation

$$\begin{tikzcd}
\boxed{S_L^o, S_L^n, S_P^n, S_P^\pm} \arrow[loop left, "\Sigma"] \arrow[d, dash, "\Sigma"] & & U_L \arrow[ll, dashed] \arrow[d, "(\textit{update compilation})"] \\
\boxed{S_P^o, S_L^\pm} \arrow[loop left, "\Sigma"] & & U_P \arrow[ll, dashed]
\end{tikzcd}$$

- U_L is a *user query* $P^+(\bar{x})$ $(P^-(\bar{x}))$ for $P \in S_A$;

Physical Design and Update Compilation



Σ

$S_L^o, S_L^n, S_P^n, S_P^\pm$

U_L

Σ

$(\textit{update compilation})$

Σ

$S_P^o, S_L^\pm$

U_P

- U_L is a *user query* $P^+(\bar{x})$ $(P^-(\bar{x}))$ for $P \in S_A$;
- U_P is a *plan* for the user query $P^+(\bar{x})$ $(P^-(\bar{x}))$ for $P \in S_A$
  $\Rightarrow$ w.r.t. the access paths $S_A \cup S_L^\pm$, and
  $\Rightarrow$ aux code that inserts (deletes) the result of the plan into (from) P.

Example

Setup: standard relational design for `Employee(id,name,salary)`

- A *base file* `empfile` of `emp` records (organized by `id`)
- An `emp-name` index on employee `name`s (links `name` to `id`)

Example

Setup: standard relational design for `Employee(id,name,salary)`

- A *base file* `empfile` of `emp` records (organized by `id`)
- An `emp-name` index on employee `name`s (links `name` to `id`)

Logical Schema:

$$\mathsf{S_L} = \{\texttt{Employee}/3\}, \Sigma_\mathsf{L} = \{\text{“}\texttt{id}\text{ is a key”}\}$$

Physical Schema:

$$\mathsf{S_P} = \mathsf{S_A} = \{\texttt{empfile}/3/0, \texttt{emp-id}/3/1, \texttt{emp-name}/2/1\}$$

$$\Sigma_\mathsf{LP} = \left\{ \begin{array}{l} \forall x,y,z.\texttt{Employee}(x,y,z) \leftrightarrow \texttt{empfile}(x,y,z) \\ \forall x,y,z.\texttt{Employee}(x,y,z) \leftrightarrow \texttt{emp-id}(x,y,z) \\ \forall x,y,z.\texttt{Employee}(x,y,z) \leftrightarrow \texttt{emp-name}(y,x) \end{array} \right\}$$

Logical Update Schema: (just the signature)

$$\mathsf{S_L} = \{\texttt{empfile}^+/3, \texttt{empfile}^-/3, \texttt{emp-name}^+/2, \texttt{emp-name}^-/2\}$$

Physical Update Schema:

$$\mathsf{S_P} = \{\texttt{Employee}^+/3, \texttt{Employee}^-/3, \texttt{empfile}^o/3, \texttt{empfile}^o/3, \ldots\}$$

Example

Setup: standard relational design for `Employee(id,name,salary)`

- A *base file* `empfile` of `emp` records (organized by `id`)
- An `emp-name` index on employee `name`s (links `name` to `id`)

Logical Update Schema: (just the signature)

$$\mathsf{S_L} = \{\mathtt{empfile}^{+}/3, \mathtt{empfile}^{-}/3, \mathtt{emp\text{-}name}^{+}/2, \mathtt{emp\text{-}name}^{-}/2\}$$

Physical Update Schema:

$$\begin{aligned} \mathsf{S_P} &= \{\mathtt{Employee}^{+}/3, \mathtt{Employee}^{-}/3, \mathtt{empfile}^{o}/3, \mathtt{empfile}^{o}/3, \ldots\} \\ \Sigma_{\mathsf{LP}} &= \{\forall x, y, z.(\mathtt{empfile}^{o}(x, y, z) \lor \mathtt{empfile}^{+}(x, y, z)) \\ &\qquad\qquad \leftrightarrow (\mathtt{empfile}^{n}(x, y, z) \lor \mathtt{empfile}^{-}(x, y, z)), \ldots\} \\ \Sigma_{\mathsf{P}} &= \{\forall x, y, z.\mathtt{Employee}^{+}(x, y, z) \land \mathtt{Employee}^{-}(x, y, z) \to \bot, \ldots\} \end{aligned}$$

Update Queries:

$$\mathtt{empfile}^{+}(x, y, z)$$

$$\mathtt{empfile}^{-}(x, y, z)$$

Example

Setup: standard relational design for `Employee(id,name,salary)`

- A *base file* `empfile` of `emp` records (organized by `id`)
- An `emp-name` index on employee `name`s (links `name` to `id`)

Logical Update Schema: (just the signature)

$$\mathrm{S_L} = \{\mathtt{empfile}^{+}/3, \mathtt{empfile}^{-}/3, \mathtt{emp{-}name}^{+}/2, \mathtt{emp{-}name}^{-}/2\}$$

Physical Update Schema:

$$\begin{aligned}
\mathrm{S_P} &= \{\mathtt{Employee}^{+}/3, \mathtt{Employee}^{-}/3, \mathtt{empfile}^{o}/3, \mathtt{empfile}^{o}/3, \ldots\}\\
\Sigma_{\mathrm{LP}} &= \{\forall x, y, z.(\mathtt{empfile}^{o}(x, y, z) \lor \mathtt{empfile}^{+}(x, y, z))\\
&\qquad\qquad \leftrightarrow (\mathtt{empfile}^{n}(x, y, z) \lor \mathtt{empfile}^{-}(x, y, z)), \ldots\}\\
\Sigma_{\mathrm{P}} &= \{\forall x, y, z.\mathtt{Employee}^{+}(x, y, z) \land \mathtt{Employee}^{-}(x, y, z) \to \bot, \ldots\}
\end{aligned}$$

Update Queries:

$$\begin{aligned}
\mathtt{empfile}^{+}(x, y, z) &\xrightarrow{\textit{compiles}} \mathtt{Employee}^{+}(x, y, z) \land \neg\mathtt{empfile}^{o}(x, y, z)\\
\mathtt{empfile}^{-}(x, y, z) &\xrightarrow{\textit{compiles}} \mathtt{Employee}^{-}(x, y, z) \land \mathtt{empfile}^{o}(x, y, z)
\end{aligned}$$

Example

Setup: standard relational design for `Employee(id,name,salary)`

- A *base file* `empfile` of `emp` records (organized by `id`)
- An `emp-name` index on employee `name`s (links `name` to `id`)

Logical Update Schema: (just the signature)

$$S_L = \{\texttt{empfile}^+/3, \texttt{empfile}^-/3, \texttt{emp-name}^+/2, \texttt{emp-name}^-/2\}$$

Physical Update Schema:

$$S_P = \{\texttt{Employee}^+/3, \texttt{Employee}^-/3, \texttt{empfile}^o/3, \texttt{empfile}^o/3, \ldots\}$$

$$\Sigma_{LP} = \{\forall x, y, z.(\texttt{empfile}^o(x, y, z) \lor \texttt{empfile}^+(x, y, z)) \leftrightarrow (\texttt{empfile}^n(x, y, z) \lor \texttt{empfile}^-(x, y, z)), \ldots\}$$

$$\Sigma_P = \{\forall x, y, z.\texttt{Employee}^+(x, y, z) \land \texttt{Employee}^-(x, y, z) \to \bot, \ldots\}$$

Update Queries:

$$\texttt{empfile}^+(x, y, z) \xrightarrow{\textit{compiles}} \texttt{Employee}^+(x, y, z) \land \neg\texttt{empfile}^o(x, y, z)$$

$$\texttt{empfile}^-(x, y, z) \xrightarrow{\textit{compiles}} \texttt{Employee}^-(x, y, z) \land \texttt{empfile}^o(x, y, z)$$

. . . similar for `emp-name`, no-op for `emp-id` (why?)

Transaction Types

Transactions

A user update (expressed as diffs on *logical* symbols) that transforms an consistent instance to another consistent instance.

Transaction Types

Transactions

A user update (expressed as diffs on *logical* symbols) that transforms an consistent instance to another consistent instance.

Additional information about transaction behaviour?

1. transaction only adds tuples to a certain relation,
2. transaction only modifies certain relations,
3. . . .

Transaction Types

Transactions

A user update (expressed as diffs on *logical* symbols) that transforms an consistent instance to another consistent instance.

Additional information about transaction behaviour?

1. transaction only adds tuples to a certain relation,
2. transaction only modifies certain relations,
3. ...

Additional information ⇒ additional constraints:

1. $P^- = \emptyset$ for the “insert-only” relation P,
2. $P^+ = P^- = \emptyset$ for unmodified relations.
3. ...

The View Update Problem

Classical View Update Problem

Given a relational view

$$\forall \bar{x}. V(\bar{x}) \leftrightarrow Q(\bar{x})$$

with Q expressed over $\mathsf{S_L}$, is it possible to update the content of V by appropriately modifying the interpretation of the $\mathsf{S_L}$ symbols?

$\Rightarrow$ *insertable*, *deletable*, and *updatable* views

The View Update Problem

Classical View Update Problem

Given a relational view

$$\forall \bar{x}. V(\bar{x}) \leftrightarrow Q(\bar{x})$$

with Q expressed over $\mathrm{S_L}$, is it possible to update the content of V by appropriately modifying the interpretation of the $\mathrm{S_L}$ symbols?

⇒ *insertable*, *deletable*, and *updatable* views

Answer

Define *update schema* for V and $\mathrm{S_L}$ (where every symbol is also an access path). Then V is

- *insertable* if P^n is definable w.r.t. the update design with $V^- = \emptyset$,
- *deletable* if P^n is definable w.r.t. the update design with $V^+ = \emptyset$, and
- *updatable* if P^n and V^- are definable w.r.t. the update design

for all $P \in \mathrm{S_L}$.

⇒ when the answer is positive, we construct a corresponding *update* queries.

ADVANCED ISSUES IN UPDATE COMPILATION

Progressive Updates

Update Queries:

$$\mathtt{empfile}^{+}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{+}(x,y,z) \land \neg\mathtt{empfile}^{o}(x,y,z)$$

$$\mathtt{empfile}^{-}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{-}(x,y,z) \land \mathtt{empfile}^{o}(x,y,z)$$

Progressive Updates

Update Queries:

$$\texttt{empfile}^{+}(x,y,z) \xrightarrow{\textit{compiles}} \texttt{Employee}^{+}(x,y,z) \land \neg\texttt{empfile}^{o}(x,y,z)$$

$$\texttt{empfile}^{-}(x,y,z) \xrightarrow{\textit{compiles}} \texttt{Employee}^{-}(x,y,z) \land \texttt{empfile}^{o}(x,y,z)$$

This doesn't quite work:

after *executing* the 1st update query we no longer have $\texttt{empfile}^{o}$!

Progressive Updates

Update Queries:

$$\mathtt{empfile}^{+}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{+}(x,y,z) \land \neg\mathtt{empfile}^{o}(x,y,z)$$

$$\mathtt{empfile}^{-}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{-}(x,y,z) \land \mathtt{empfile}^{o}(x,y,z)$$

This doesn't quite work:

after *executing* the 1st update query we no longer have $\mathtt{empfile}^{o}$!

Possible Solutions:

1. simultaneous *relational* assignment:

   ⇒ compute all deltas and store results in temporary storage,

   ⇒ *only then* apply all deltas to S_A;

Progressive Updates

Update Queries:

$$\mathtt{empfile}^{+}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{+}(x,y,z) \land \neg\mathtt{empfile}^{o}(x,y,z)$$

$$\mathtt{empfile}^{-}(x,y,z) \xrightarrow{\textit{compiles}} \mathtt{Employee}^{-}(x,y,z) \land \mathtt{empfile}^{o}(x,y,z)$$

This doesn't quite work:

after *executing* the 1st update query we no longer have $\mathtt{empfile}^{o}$!

Possible Solutions:

1. simultaneous *relational* assignment:

   ⇒ compute all deltas and store results in temporary storage,

   ⇒ *only then* apply all deltas to S_{A};

2. using independent deltas:

   ⇒ add constraints to avoid the problem (e.g., $P^{-} \subseteq P^{o}$);

Progressive Updates

Update Queries:

$$\texttt{empfile}^{+}(x,y,z) \xrightarrow{\textit{compiles}} \texttt{Employee}^{+}(x,y,z) \land \neg\texttt{empfile}^{o}(x,y,z)$$
$$\texttt{empfile}^{-}(x,y,z) \xrightarrow{\textit{compiles}} \texttt{Employee}^{-}(x,y,z) \land \texttt{empfile}^{o}(x,y,z)$$

This doesn't quite work:

after *executing* the 1st update query we no longer have $\texttt{empfile}^{o}$!

Possible Solutions:

1. simultaneous *relational* assignment:

   ⇒ compute all deltas and store results in temporary storage,
   ⇒ *only then* apply all deltas to S_A;

2. using independent deltas:

   ⇒ add constraints to avoid the problem (e.g., $P^{-} \subseteq P^{o}$);

3. evolving physical schema one AP at a time

   ⇒ sequence of update schemas with a subset of S_A "updated",
   ⇒ subsequent updates compiled w.r.t. partially updated schema.

Value Invention

Setup: advanced relational design for `Employee(id,name,salary)`

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RId`*s "r"*
- An `emp-name`(y, r) index on employee `name`s (links `name` to `RId`s)

Value Invention

Setup: advanced relational design for `Employee(id,name,salary)`

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RId`*s "r"*
- An `emp-name`(y, r) index on employee `name`s (links `name` to `RId`s)

$\Rightarrow$ no update query, e.g., for $\texttt{empfile}^{+}(r, x, y, z)$: no "source" of `RId`s!

(due to: $\forall x, y, z.\texttt{Employee}(x, y, z) \leftrightarrow (\exists r.\texttt{empfile}(r, x, y, z))$)

Value Invention

Setup: advanced relational design for `Employee(id,name,salary)`

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RId`*s "r"*
- An `emp-name`(y, r) index on employee `name`s (links `name` to `RId`s)

  ⇒ no update query, e.g., for `empfile`$^+(r, x, y, z)$: no "source" of `RId`s!

IDEA (Constant Complement [Bancilhon and Spyratos])

An *oracle access path* that provides the required value
given the values of remaining attributes as parameters.

Value Invention

Setup: advanced relational design for `Employee(id,name,salary)`

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RId`*s "r"*
- An `emp-name`(y, r) index on employee `name`s (links `name` to `RId`s)

⇒ no update query, e.g., for `empfile`$^+$(r, x, y, z): no "source" of `RId`s!

IDEA (Constant Complement [Bancilhon and Spyratos])

An *oracle access path* that provides the required value given the values of remaining attributes as parameters.

In practice: a record allocation mechanism (e.g., `malloc`+code that initializes fields of the allocated record)

Value Invention

Setup: advanced relational design for `Employee(id,name,salary)`

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RId`*s "r"*
- An `emp-name`(y, r) index on employee `name`s (links `name` to `RId`s)

⇒ no update query, e.g., for `empfile`$^+$(r, x, y, z): no "source" of `RId`s!

IDEA (Constant Complement [Bancilhon and Spyratos])

An *oracle access path* that provides the required value given the values of remaining attributes as parameters.

In practice: a record allocation mechanism (e.g., `malloc`+code that initializes fields of the allocated record)

- a separate access path (may need to "remember" all allocated records!)
- a part of the record insertion code (AP^+ doesn't have the attribute)
  ⇒ update query for `emp-name`$^+$ must execute *after* `empfile`$^+$.

Value Invention and Schematic Cycles

Can we *always* schedule the updates of record `ID`s before using these as values (e.g., in an index)?

Value Invention and Schematic Cycles

Can we *always* schedule the updates of record `ID`s before using these as values (e.g., in an index)?

NO: recall our Employee-Works-Department physical schema in which

- `emp` records have a pointer to a `dept` record (for the `Works` relationship),
- `dept` records have a pointer to an `emp` record (to the “manager”).

Value Invention and Schematic Cycles

Can we *always* schedule the updates of record `ID`s before using these as values (e.g., in an index)?

NO: recall our Employee-Works-Department physical schema in which

- `emp` records have a pointer to a `dept` record (for the `Works` relationship),
- `dept` records have a pointer to an `emp` record (to the "manager").

⇒ impossible to insert the 1st employee and 1st department!

Value Invention and Schematic Cycles

Can we *always* schedule the updates of record `ID`s before using these as values (e.g., in an index)?

NO: recall our Employee-Works-Department physical schema in which

- `emp` records have a pointer to a `dept` record (for the `Works` relationship),
- `dept` records have a pointer to an `emp` record (to the “manager”).

⇒ impossible to insert the 1st employee and 1st department!

IDEA: reify (one of) the AP (we have done that already in our example) and then interleave updates to the reified relations.

Value Invention and Schematic Cycles

Can we *always* schedule the updates of record `IDs` before using these as values (e.g., in an index)?

NO: recall our Employee-Works-Department physical schema in which

- `emp` records have a pointer to a `dept` record (for the `Works` relationship),
- `dept` records have a pointer to an `emp` record (to the “manager”).

⇒ impossible to insert the 1st employee and 1st department!

IDEA: reify (one of) the AP (we have done that already in our example) and then interleave updates to the reified relations.

1. insert an employee’s `Id` into `emp-id` AP (yields address of `emp`);
2. insert department record (the above value used for the manager field; yields address of `dept`);
3. insert the same employee into `emp-dept` AP using the `dept` address.

Additional Issues

- How to know when an *constant complement* is needed?

Additional Issues

- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?

Additional Issues

- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
- How to identify when *reification* is needed?

Additional Issues

- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
- How to identify when *reification* is needed?
- How to determine if the user update preserves *consistency*?

  ⇒ guaranteed by the user (e.g., extra user queries to make sure)

  ⇒ system-generated checks—HARD!