Topics in Database Systems: Modern Database Systems

CS848 Spring 2022

David Toman

DATABASE IMPLEMENTATION

(UPDATES)

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

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

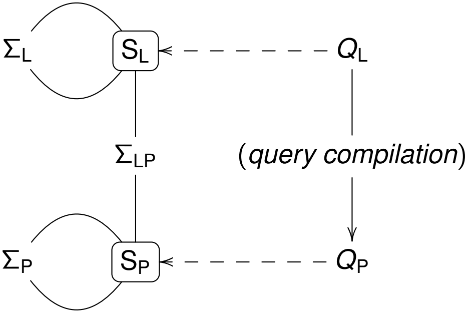
Plan

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

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



Σ_{L}

S_{L}

Q_{L}

Σ_{LP}

(*query compilation*)

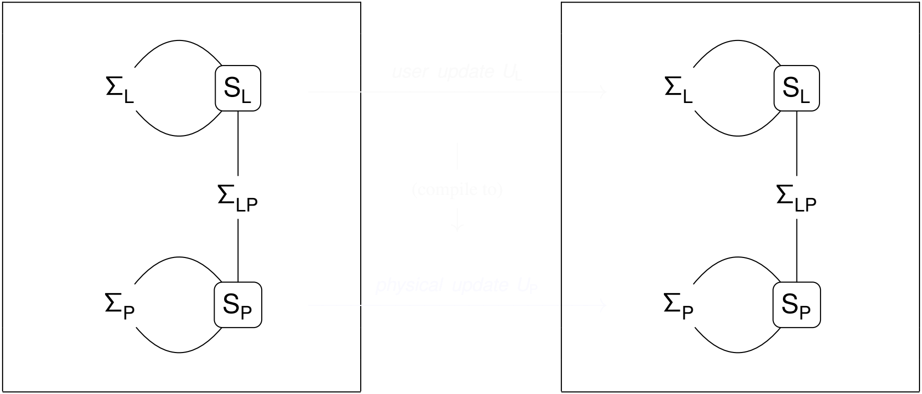
Σ_{P}

S_{P}

Q_{P}

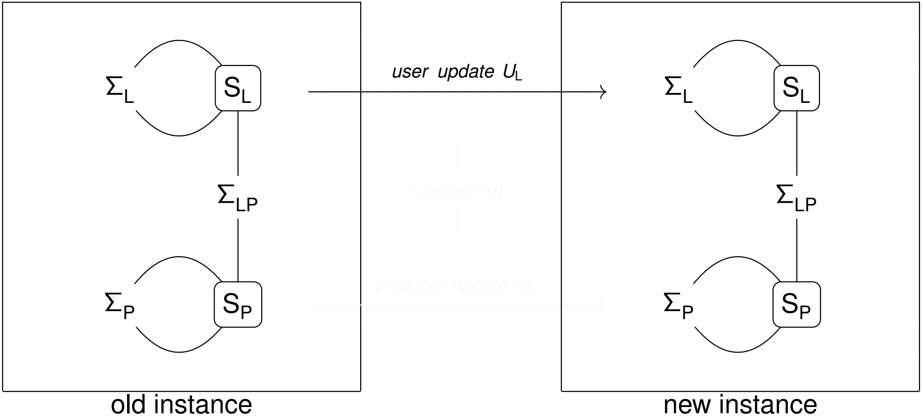
UPDATES IN NUTSHELL

Physical Design and Updates: Overview

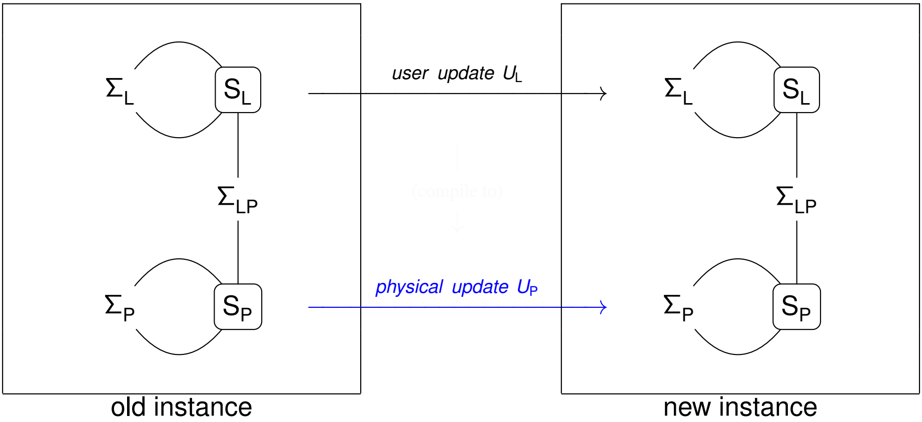


old instance new instance

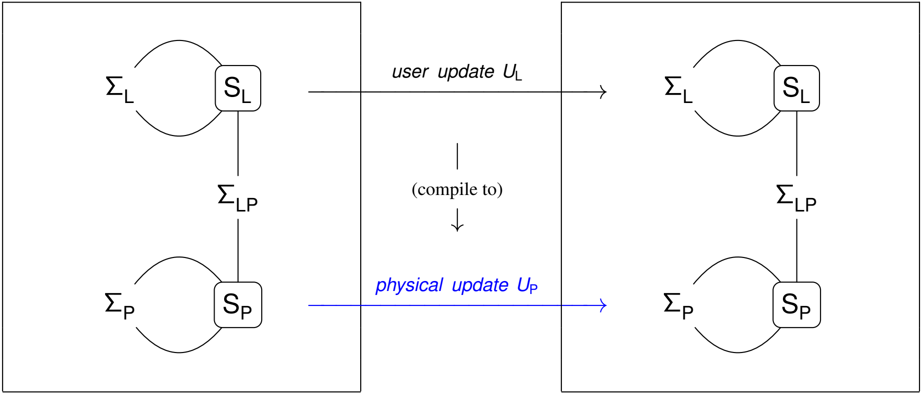
Physical Design and Updates: Overview



Physical Design and Updates: Overview

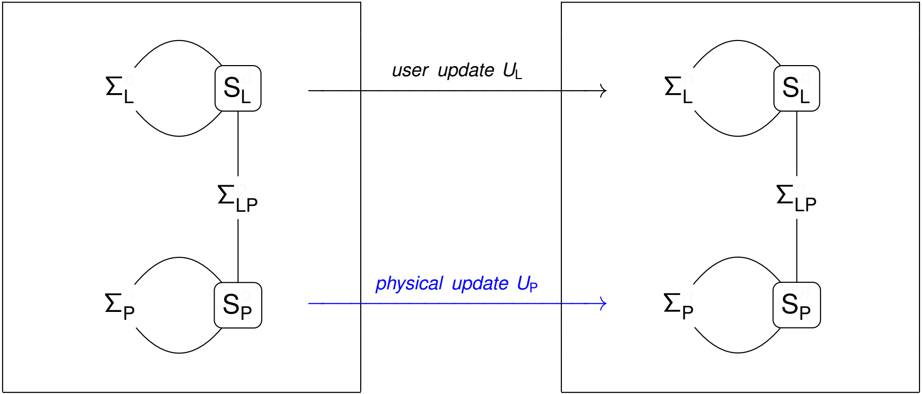


Physical Design and Updates: Overview



old instance new instance

Update Schema



Σ_L^o S_L^o

Σ_{LP}^o

Σ_P^o S_P^o

user update U_L

physical update U_P

Σ_L^n S_L^n

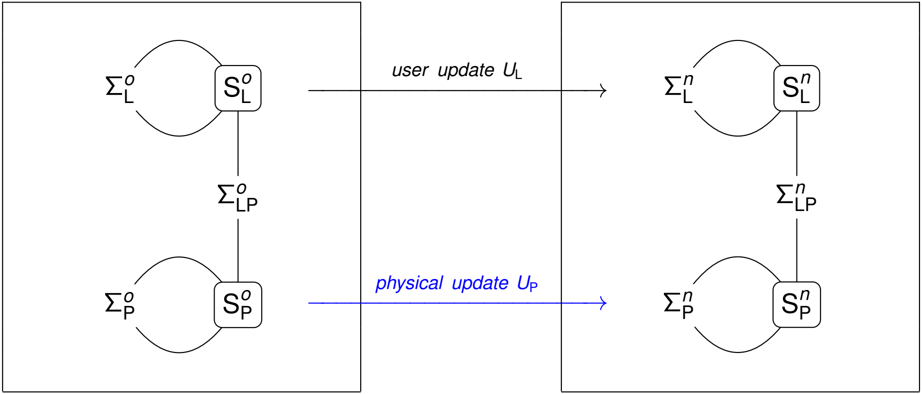
Σ_{LP}^n

Σ_P^n S_P^n

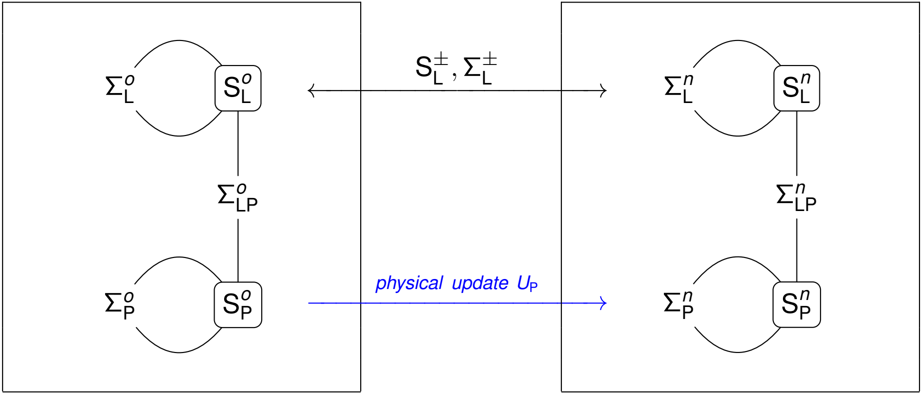
o(ld instance)

n(ew instance)

Update Schema

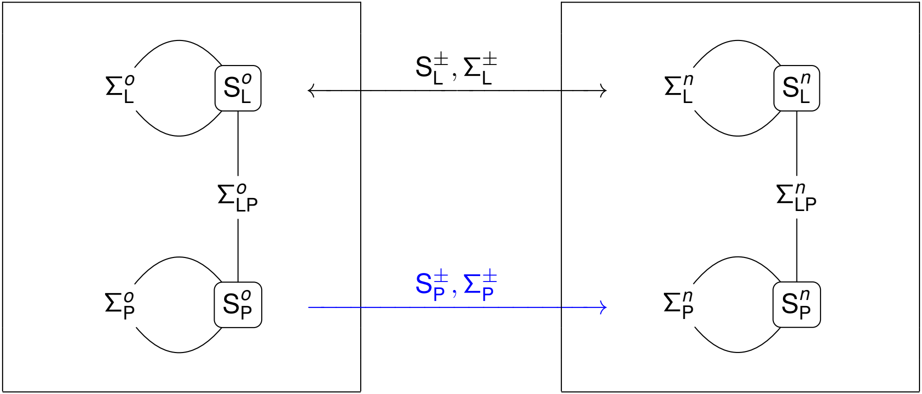


Update Schema



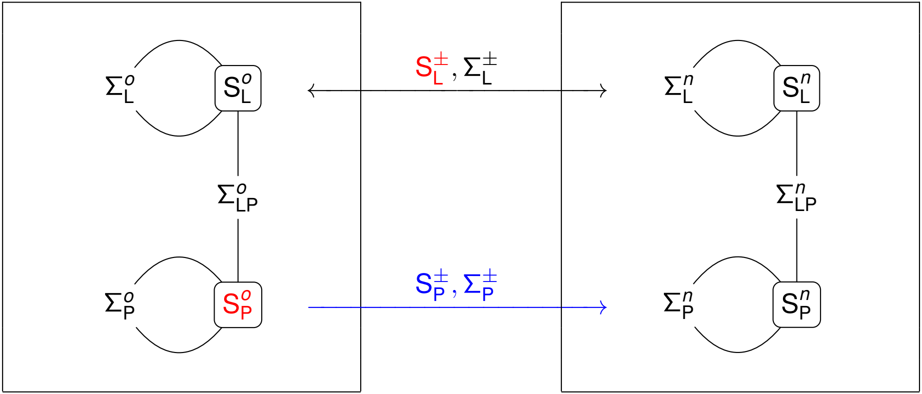
$$S_L^\pm = \{P^+, P^- \mid P \in S_L\},$$
$$\Sigma_L^\pm = \{\forall \bar{x}.(P^o(\bar{x}) \vee P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \vee P^-(\bar{x})) \mid P \in S_L\}$$

Update Schema

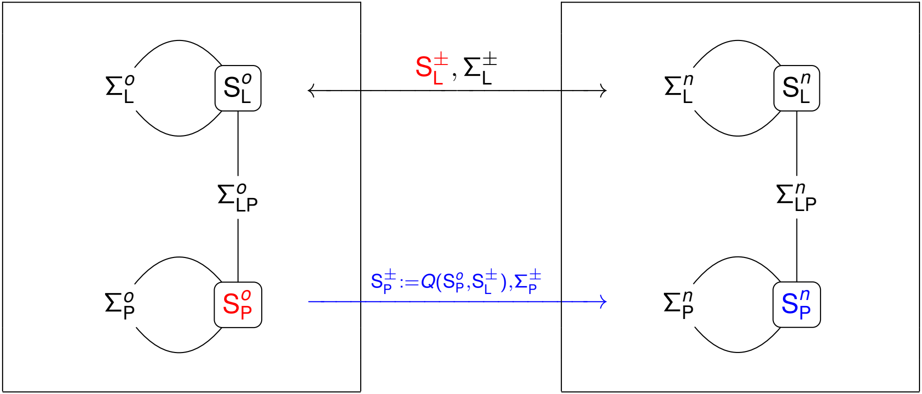


$$S_L^\pm = \{P^+, P^- \mid P \in S_A\},$$
$$\Sigma_L^\pm = \{\forall\bar{x}.(P^o(\bar{x}) \vee P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \vee P^-(\bar{x})) \mid P \in S_A\}$$

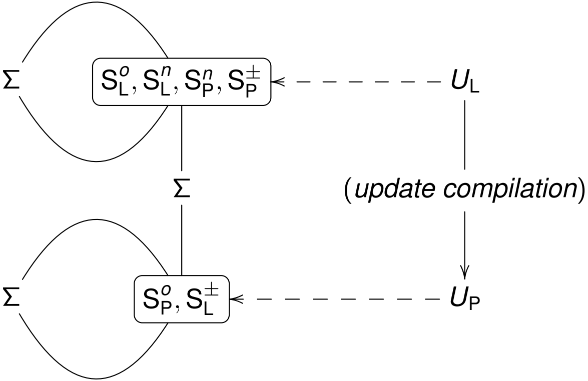
Update Schema



Update Schema



Physical Design and Update Compilation



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

$S^o_P, S^\pm_L$

U_L

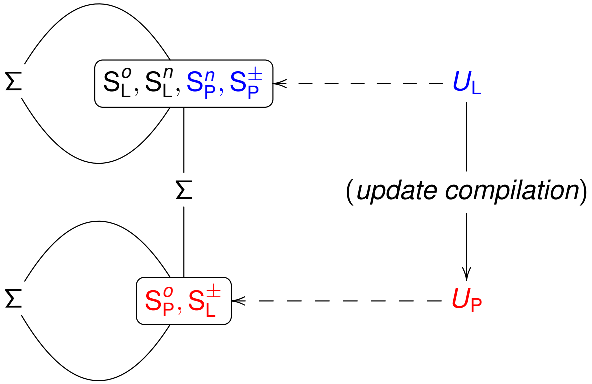
U_P

Σ

(*update compilation*)

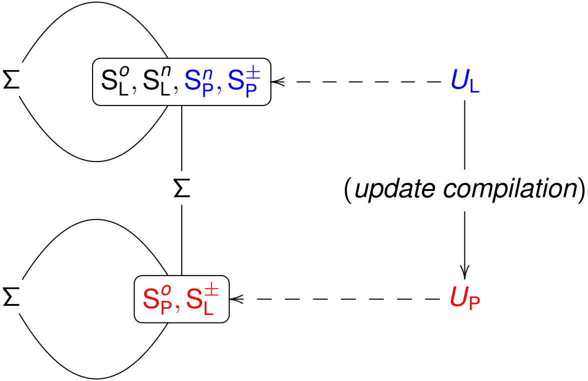
- U_L is a *user query* $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in S_A$;
- U_P is a *plan* for the user query $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in S_A$
  - ⇒ w.r.t. the access paths $S_A \cup S^\pm_L$, and
  - ⇒ aux code that inserts (deletes) the result of the plan into (from) P.

Physical Design and Update Compilation



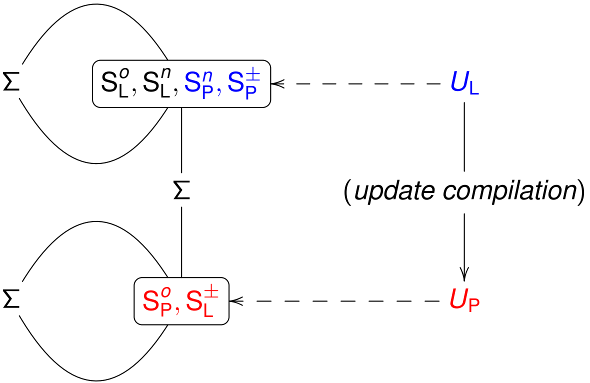
- U_L is a *user query* $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in S_A$;
- U_P is a *plan* for the user query $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in S_A$
  - ⇒ w.r.t. the access paths $S_A \cup S_L^\pm$, and
  - ⇒ aux code that inserts (deletes) the result of the plan into (from) P.

Physical Design and Update Compilation



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

Physical Design and Update Compilation



Σ

$\mathsf{S}_\mathsf{L}^o, \mathsf{S}_\mathsf{L}^n, \mathsf{S}_\mathsf{P}^n, \mathsf{S}_\mathsf{P}^\pm$

U_L

Σ

(*update compilation*)

Σ

$\mathsf{S}_\mathsf{P}^o, \mathsf{S}_\mathsf{L}^\pm$

U_P

- U_L is a *user query* $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in \mathsf{S}_\mathsf{A}$;
- U_P is a *plan* for the user query $P^+(\bar{x})$ ($P^-(\bar{x})$) for $P \in \mathsf{S}_\mathsf{A}$
  $\Rightarrow$ w.r.t. the access paths $\mathsf{S}_\mathsf{A} \cup \mathsf{S}_\mathsf{L}^\pm$, 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:

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

Physical Schema:

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

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

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:

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

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)

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

Physical Update Schema:

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

Update Queries:

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

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

…similar for `emp-name`

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)

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

Physical Update Schema:

$$\begin{aligned}
\mathsf{S}_{\mathsf{P}} &= \{\texttt{Employee}^{+}/3, \texttt{Employee}^{-}/3, \texttt{empfile}^{o}/3, \texttt{empfile}^{o}/3, \ldots\} \\
\Sigma_{\mathsf{LP}} &= \{\forall x, y, z.(\texttt{empfile}^{o}(x, y, z) \vee \texttt{empfile}^{+}(x, y, z)) \\
&\qquad\qquad \leftrightarrow (\texttt{empfile}^{n}(x, y, z) \vee \texttt{empfile}^{-}(x, y, z)), \ldots\} \\
\Sigma_{\mathsf{P}} &= \{\forall x, y, z.\texttt{Employee}^{+}(x, y, z) \wedge \texttt{Employee}^{-}(x, y, z) \rightarrow \bot, \ldots\}
\end{aligned}$$

Update Queries:

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

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

…similar for `emp-name`

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)

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

Physical Update Schema:

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

Update Queries:

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

…similar for `emp-name`

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. …

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. …

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

Answer

Define *update schema* for V and $\mathsf{S_L}$ (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 $P \in \mathsf{S_L}$.

$\Rightarrow$ when the answer is positive, we construct a corresponding *update* queries.

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?

$\Rightarrow$ *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 $P \in \mathrm{S_L}$.

$\Rightarrow$ when the answer is positive, we construct a corresponding *update* queries.

ADVANCED ISSUES
IN UPDATE COMPILATION

Progressive Updates

Update Queries:

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

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.

Progressive Updates

Update Queries:

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

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

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., $P^- \subseteq P^o$);
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.

Progressive Updates

Update Queries:

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

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)

⇒ 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

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

- A *base file* $\texttt{empfile}(r, x, y, z)$ of `emp` records *with* `RId`*s "r"*
- An $\texttt{emp-name}(y, r)$ 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: $\forall x, y, z.\texttt{Employee}(x, y, z) \leftrightarrow (\exists r.\texttt{empfile}(r, x, y, z))$

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)

$\Rightarrow$ update query for $\texttt{emp-name}^{+}$ must execute *after* $\texttt{empfile}^{+}$.

Value Invention

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

- A *base file* `empfile`(r, x, y, z) of `emp` records *with* `RIds` *“r”*
- An `emp-name`(y, r) index on employee `names` (links `name` to `RIds`)

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

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

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

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)?

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.

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.

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.

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.

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.

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.

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.

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.

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Hand-crafted Solution

```
while ¬end-of(emppages) do
    read emppages to $p$;
    while ¬end-of(emprecords($p$)) do
        read emprecords($p$) to $r$;
        if $r \rightarrow$ salary $< 100k$ then
            $r \rightarrow$ salary $*= 1.1$;
            write $r$ to emprecords($p$);
    write $p$ to emppages;
```

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Hand-crafted Solution

```
while ¬end-of(emppages) do
    read emppages to $p$;
    while ¬end-of(emprecords($p$)) do
        read emprecords($p$) to $r$;
        if $r \to$ salary $< 100k$ then
            $r \to$ salary $* = 1.1$;
            write $r$ to emprecords($p$);
    write $p$ to emppages;      // only if p is "dirty"?
```

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Current Situation

- Our (current) solution–behaves as if *pages* were just pointers:

⇒ $\texttt{emprecords}^-$ becomes “all old records”
$\texttt{emprecords}^+$ becomes “all changed records”

⇒ we completely *miss* the need to write `emppages`...

Project Idea

How do we deal with *temporarily replicated data* and *intermediate results*?

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Current Situation

- Our (current) solution–behaves as if *pages* were just pointers:

⇒ $\texttt{emprecords}^-$ becomes “all old records”
$\texttt{emprecords}^+$ becomes “all changed records”

⇒ we completely *miss* the need to write `emppages`...

Project Idea

How do we deal with *temporarily replicated data* and *intermediate results*?

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Current Situation

- Our (current) solution–behaves as if *pages* were just pointers:

⇒ $\texttt{emprecords}^-$ becomes “all old records”
$\texttt{emprecords}^+$ becomes “all changed records”

⇒ we completely *miss* the need to write `emppages`...

Project Idea

How do we deal with *temporarily replicated data* and *intermediate results*?

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Ideas for Solution(s)

- extensions with *updates-in-place*
- modified operators (NLJ) so that it *writes data back*

  $\Rightarrow \mathrm{NLJ}(\mathtt{emppages}(p), \mathrm{NLJ}(\mathtt{emprecords}(p,r), \mathrm{modify}\ r))$

- or more schema design??

  ⇒ separate "access paths" for reading/writing

  ⇒ sequencing, e.g., via `union`, etc.

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Ideas for Solution(s)

- extensions with *updates-in-place*
- modified operators (NLJ) so that it *writes data back*

  $\Rightarrow \mathrm{NLJ}(\mathtt{emppages}(p), \mathrm{NLJ}(\mathtt{emprecords}(p, r), \mathrm{modify}\ r))$
- or more schema design??

  ⇒ separate “access paths” for reading/writing

  ⇒ sequencing, e.g., via `union`, etc.

Updates and 2-level Store (open problem)

Example

- Design: employees stored in `emppages/1/0` and `emprecords/2/1`;
- Update: *every employee (making <100k) gets 10% salary increase*

Ideas for Solution(s)

- extensions with *updates-in-place*
- modified operators (NLJ) so that it *writes data back*

  $\Rightarrow \mathrm{NLJ}(\texttt{emppages}(p), \mathrm{NLJ}(\texttt{emprecords}(p, r), \mathrm{modify}\ r))$
- or more schema design??

  ⇒ separate “access paths” for reading/writing

  ⇒ sequencing, e.g., via `union`, etc.

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

Project Idea

Detecting and rectifying the Halloween problem

⇒ what is the *correct* semantics anyway? (this alone is a project topic)

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

(Naive) Hand-crafted Solution

```
while ¬end-of(emplist) do
    read emplist to $r$;
    if $r \rightarrow$ salary $< 100k$ then
        $r \rightarrow$ salary $+= 10k$;
    write $r$ to emprecords($p$);
```

Project Idea

Detecting and rectifying the Halloween problem

⇒ what is the *correct* semantics anyway? (this alone is a project topic)

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

(Naive) Hand-crafted Solution

```
while ¬end-of(emplist) do
    read emplist to $r$;
    if $r \to$ salary $< 100k$ then
        $r \to$ salary $+= 10k$;
    write $r$ to emprecords($p$);
```

Does this work?? NO!!!

⇒ consider `emplist` = [(Fred, $10k$), (Wilma, $15k$)] to start with;

⇒ result `emplist` = [(Fred, $100k$), (Wilma, $105k$)] at the end…

Project Idea

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

(Naive) Hand-crafted Solution

```
while ¬end-of(emplist) do
    read emplist to $r$;
    if $r \to$ salary $< 100k$ then
        $r \to$ salary $+= 10k$;
    write $r$ to emprecords($p$);    // insert into ordered list
```

Does this work?? NO!!!

⇒ consider `emplist` $= [(\text{Fred}, 10k), (\text{Wilma}, 15k)]$ to start with;

⇒ result `emplist` $= [(\text{Fred}, 100k), (\text{Wilma}, 105k)]$ at the end…

Project Idea

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

(Naive) Hand-crafted Solution

```
while $\neg$end-of(emplist) do
    read emplist to $r$;
    if $r \rightarrow$ salary $< 100k$ then
        $r \rightarrow$ salary $+ = 10k$;
    write $r$ to emprecords($p$);    // insert into ordered list
```

Does this work?? NO!!!

$\Rightarrow$ consider `emplist` $= [(\text{Fred}, 10k), (\text{Wilma}, 15k)]$ to start with;

$\Rightarrow$ result `emplist` $= [(\text{Fred}, 100k), (\text{Wilma}, 105k)]$ at the end…

Project Idea

The Halloween Problem (open problem)

Example

- Design: employees stored in a list `emplist/2/0` *ordered by* `salary`
- Update: *every employee (making <100k) gets 10k salary increase*

(Naive) Hand-crafted Solution

```
while ¬end-of(emplist) do
    read emplist to $r$;
    if $r \rightarrow$ salary $< 100k$ then
        $r \rightarrow$ salary $+= 10k$;
    write $r$ to emprecords($p$);    // insert into ordered list
```

Project Idea

Detecting and rectifying the Halloween problem

⇒ what is the *correct* semantics anyway? (this alone is a project topic)

Concurrency and Durability (open problem)

Isolation: what if others access the data too??

⇒ schematic description of CC rather that *lock manager et al.*
e.g., the RCU style approach (used by the Linux kernel)

⇒ deadlock-free solutions (why?)

⇒ compile-time (just what one really needs)

Durability: what if a permanent record is needed??

⇒ additional *physical design* for LOGs (or for COW?)

⇒ how to deal with (lazy) replication? (see 2-level store)

⇒ transactions, rollbacks, and recovery?

Project Ideas

Each of the above (alone) can be a project

⇒ even just analyzing the problems without a clear/favourite solution!

Concurrency and Durability (open problem)

Isolation: what if others access the data too??

⇒ schematic description of CC rather that *lock manager et al.*
e.g., the RCU style approach (used by the Linux kernel)

⇒ deadlock-free solutions (why?)

⇒ compile-time (just what one really needs)

Durability: what if a permanent record is needed??

⇒ additional *physical design* for LOGs (or for COW?)

⇒ how to deal with (lazy) replication? (see 2-level store)

⇒ transactions, rollbacks, and recovery?

Project Ideas

Each of the above (alone) can be a project

⇒ even just analyzing the problems without a clear/favourite solution!

Concurrency and Durability (open problem)

Isolation: what if others access the data too??

⇒ schematic description of CC rather that *lock manager et al.*
e.g., the RCU style approach (used by the Linux kernel)

⇒ deadlock-free solutions (why?)

⇒ compile-time (just what one really needs)

Durability: what if a permanent record is needed??

⇒ additional *physical design* for LOGs (or for COW?)

⇒ how to deal with (lazy) replication? (see 2-level store)

⇒ transactions, rollbacks, and recovery?

Project Ideas

Each of the above (alone) can be a project

⇒ even just analyzing the problems without a clear/favourite solution!

More Issues

- How do we know *what APs* to update? (so we don't miss `emppages`!)
- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
  ⇒ what about interleaving??
- How to identify cycles and 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!

More Issues

- How do we know *what APs* to update? (so we don't miss `emppages`!)
- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
  ⇒ what about interleaving??
- How to identify cycles and 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!

More Issues

- How do we know *what APs* to update? (so we don't miss `emppages`!)
- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
  ⇒ what about interleaving??
- How to identify cycles and 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!

More Issues

- How do we know *what APs* to update? (so we don't miss `emppages`!)
- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
  ⇒ what about interleaving??
- How to identify cycles and 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!

More Issues

- How do we know *what APs* to update? (so we don't miss `emppages`!)
- How to know when an *constant complement* is needed?
- How to determine the *ordering* of the individual AP updates?
  ⇒ what about interleaving??
- How to identify cycles and 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!