Topics in Database Systems: Modern Database Systems
CS848 Spring 2021

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 \textit{in our framework}? 
   - updates and logical relations
   - updates and constraints
   - updates and access paths

3. Difficulties on the way
   - sequencing updates
   - value invention
计划

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

$\Sigma_L$ $S_L$ $\rightarrow$ $Q_L$

$\Sigma_{LP}$

$\Sigma_P$ $S_P$ $\rightarrow$ $Q_P$

(query compilation)
Updates in Nutshell
Physical Design and Updates: Overview

Update in a Nutshell

old instance

new instance

\[ \Sigma_L \rightarrow \Sigma_{LP} \]

\[ S_L \rightarrow \Sigma_{LP} \]

\[ \Sigma_P \rightarrow S_P \]

\[ \text{user update } U_L \]

\[ \text{physical update } U_P \]
Physical Design and Updates: Overview

old instance

new instance

\[ \Sigma_L \rightarrow S_L \]

\[ \Sigma_{LP} \]

\[ \Sigma_P \rightarrow S_P \]

user update \( U_L \)

(compile to)

physical update \( U_P \)
Physical Design and Updates: Overview

old instance

new instance

user update $U_L$

physical update $U_P$

David Toman (University of Waterloo)
Physical Design and Updates: Overview

user update $U_L$

(compile to)

physical update $U_P$

old instance

new instance
Update Schema

\[ \Sigma^0 \]

\[ S_L \]

\[ \Sigma_{LP} \]

\[ \Sigma^0 \]

\[ S_L \]

\[ \Sigma_{LP} \]

\[ S_P \]

\[ \Sigma^0 \]

\[ S_P \]

\[ \Sigma_{LP} \]

\[ \Sigma^0 \]

\[ S_P \]

\[ \Sigma_{LP} \]

\[ \Sigma^0 \]

\[ S_P \]

\[ \Sigma_{LP} \]

user update \( U_L \)

physical update \( U_P \)

old instance

new instance

David Toman (University of Waterloo)

Update in a Nutshell
Update in a Nutshell
Update Schema

\[ \Sigma^o_L \rightarrow S^o_L \rightarrow \Sigma^o_{LP} \rightarrow S^o_P \rightarrow \Sigma^o_P \]

\[ \Sigma^i_L, \Sigma^i \rightarrow \Sigma^i_L^\pm, \Sigma^i^\pm \rightarrow \Sigma^i_L \rightarrow S^i_L \rightarrow \Sigma^i_{LP} \rightarrow S^i_P \rightarrow \Sigma^i_P \]

\[ S_L^\pm = \{ P^+, P^- \mid P \in S_L \}, \]
\[ \Sigma_L^\pm = \{ \forall \bar{x}. (P^o(\bar{x}) \lor P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \lor P^-(\bar{x})) \mid P \in S_L \} \]
Update Schema

\[ \Sigma_L^o \xrightarrow{S_L, \Sigma_L^\pm} S_L^o \]

\[ \Sigma_L^o \quad \Sigma_L^o \]

\[ \Sigma_L^p \xrightarrow{S_L^\pm, \Sigma_L^\pm} \]

\[ \Sigma_L^p \quad \Sigma_L^p \]

\[ \Sigma_P^o \xrightarrow{S_P, \Sigma_P^\pm} S_P^o \]

\[ \Sigma_P^o \quad \Sigma_P^o \]

\[ \Sigma_P^p \xrightarrow{S_P^\pm, \Sigma_P^\pm} \]

\[ \Sigma_P^p \quad \Sigma_P^p \]

\[ S_L^\pm = \{ P^+, P^- \mid P \in S_A \}, \]

\[ \Sigma_L^\pm = \{ \forall \bar{x}. (P^o(\bar{x}) \lor P^+(\bar{x})) \leftrightarrow (P^n(\bar{x}) \lor P^-(\bar{x})) \mid P \in S_A \} \]
Update Schema

\[ \Sigma^o_L S^o_L \]

\[ \Sigma^o_{LP} S^o_P \]

\[ \Sigma^{n}_L S^n_L \]

\[ \Sigma^{n}_{LP} S^n_P \]

\[ S^{\pm}_L, \Sigma^{\pm}_L \]

\[ S^{\pm}_P, \Sigma^{\pm}_P \]
Update Schema

\[ \Sigma^0_L, S^0_L \rightarrow \Sigma^\pm_L, \Sigma^\pm_L \rightarrow \Sigma^n_L, S^n_L \]

\[ \Sigma^0_P, S^0_P \rightarrow S^\pm_P := Q(S^0_P, S^\pm_L), \Sigma^\pm_P \rightarrow \Sigma^n_P, S^n_P \]
• $U_L$ 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 $P \in S_A$
  ⇒ 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

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

Logical Schema:
\[ S_L = \{ \text{Employee}/3 \}, \Sigma_L = \{ "id \text{ is a key}" \} \]

Physical Schema:
\[ S_P = S_A = \{ \text{empfile}/3/0, \text{emp-name}/2/1 \} \]
\[ \Sigma_{LP} = \{ \forall x, y, z. \text{Employee}(x, y, z) \leftrightarrow \text{empfile}(x, y, z) \]
\[ \forall x, y, z. \text{Employee}(x, y, z) \leftrightarrow \text{emp-name}(y, x) \} \]

Logical Update Schema: (just the signature)
\[ S_L = \{ \text{empfile}^+/3, \text{empfile}^-/3, \text{emp-name}^+/2, \text{emp-name}^-/2 \} \]

Physical Update Schema:
\[ S_P = \{ \text{Employee}^+/3, \text{Employee}^-/3, \text{empfile}^o/3, \text{empfile}^o/3, \ldots \} \]
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 names (links name to id)

Logical Update Schema: (just the signature)

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

Physical Update Schema:

\[ S_P = \{\text{Employee}^+/3, \text{Employee}^-/3, \text{empfile}^0/3, \text{empfile}^0/3, \ldots\} \]

\[ \Sigma_{LP} = \{\forall x, y, z. (\text{empfile}^0(x, y, z) \lor \text{empfile}^+(x, y, z)) \leftrightarrow (\text{empfile}^n(x, y, z) \lor \text{empfile}^-(x, y, z)), \ldots\} \]

\[ \Sigma_P = \{\forall x, y, z. \text{Employee}^+(x, y, z) \land \text{Employee}^-(x, y, z) \rightarrow \perp, \ldots\} \]

Update Queries:

\[ \text{empfile}^+(x, y, z) \text{ compiles } \rightarrow \text{Employee}^+(x, y, z) \land \neg \text{empfile}^0(x, y, z) \]

\[ \text{empfile}^-(x, y, z) \text{ compiles } \rightarrow \text{Employee}^-(x, y, z) \land \text{empfile}^0(x, y, z) \]

...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 names (links name to id)

Logical Update Schema: (just the signature)

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

Physical Update Schema:

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

$$\Sigma_{LP} = \{\forall x, y, z. (\text{empfile}^o(x, y, z) \lor \text{empfile}^+(x, y, z))$$

$$\quad \quad \leftrightarrow (\text{empfile}^n(x, y, z) \lor \text{empfile}^-(x, y, z)), \ldots\}$$

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

Update Queries:

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

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

...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 names (links name to id)

Logical Update Schema: (just the signature)

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

Physical Update Schema:

\[ S_P = \{\text{Employee}^+/3, \text{Employee}^-/3, \text{empfile}^0/3, \text{empfile}^o/3, \ldots\} \]

\[ \Sigma_{LP} = \{\forall x, y, z. (\text{empfile}^0(x, y, z) \lor \text{empfile}^+ (x, y, z)) \leftrightarrow (\text{empfile}^n(x, y, z) \lor \text{empfile}^- (x, y, z)), \ldots\} \]

\[ \Sigma_P = \{\forall x, y, z. \text{Employee}^+(x, y, z) \land \text{Employee}^-(x, y, z) \rightarrow \perp, \ldots\} \]

Update Queries:

\[ \text{empfile}^+(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^+(x, y, z) \land \neg \text{empfile}^0(x, y, z) \]

\[ \text{empfile}^-(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^-(x, y, z) \land \text{empfile}^0(x, y, z) \]

... 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. ...

David Toman (University of Waterloo)
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. ...

David Toman (University of Waterloo)
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 $S_L$, is it possible to update the content of $V$ by appropriately modifying the interpretation of the $S_L$ symbols?

⇒ insertable, deletable, and updatable views

Answer

Define update schema for $V$ and $S_L$ (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 $P \in S_L$.

⇒ 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 $S_L$, is it possible to update the content of $V$ by appropriately modifying the interpretation of the $S_L$ symbols?

$\Rightarrow$ insertable, deletable, and updatable views

Answer

Define update schema for $V$ and $S_L$ (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 $P \in S_L$.

$\Rightarrow$ when the answer is positive, we construct a corresponding update queries.
ADVANCED ISSUES IN UPDATE COMPILATION
Progressive Updates

Update Queries:

\[
\begin{align*}
\text{empfile}^+(x, y, z) &\xrightarrow{\text{compiles}} \text{Employee}^+(x, y, z) \land \neg \text{empfile}^o(x, y, z) \\
\text{empfile}^-(x, y, z) &\xrightarrow{\text{compiles}} \text{Employee}^-(x, y, z) \land \text{empfile}^o(x, y, z)
\end{align*}
\]

This doesn't quite work:

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

Possible Solutions:

1. simultaneous relational assignment:
   \[
   \rightarrow \text{compute all deltas and store results in temporary storage,} \\
   \rightarrow \text{only then apply all deltas to } S_A;
   \]

2. using independent deltas:
   \[
   \rightarrow \text{add constraints to avoid the problem (e.g., } P^- \subseteq P^o); \]

3. evolving physical schema one AP at a time
   \[
   \rightarrow \text{sequence of update schemas with a subset of } S_A \text{ “updated”,} \\
   \rightarrow \text{subsequent updates compiled w.r.t. partially updated schema.} \]
Progressive Updates

Update Queries:

\[
\text{empfile}^+(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^+(x, y, z) \land \neg \text{empfile}^0(x, y, z)
\]

\[
\text{empfile}^-(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^-(x, y, z) \land \text{empfile}^0(x, y, z)
\]

This doesn’t quite work:

after \textit{executing} the 1st update query \textit{we no longer have} \text{empfile}^0!

Possible Solutions:

1. simultaneous relational assignment:
   \(
   \Rightarrow \text{compute all deltas and store results in temporary storage,}
   \Rightarrow \text{only then apply all deltas to } S_A;
   \)

2. using independent deltas:
   \(
   \Rightarrow \text{add constraints to avoid the problem (e.g., } P^- \subseteq P^0);\)

3. evolving physical schema one AP at a time
   \(
   \Rightarrow \text{sequence of update schemas with a subset of } S_A \text{ “updated”,}
   \Rightarrow \text{subsequent updates compiled w.r.t. partially updated schema.}
   \)
Progressive Updates

Update Queries:

\[
\text{empfile}^+(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^+(x, y, z) \land \neg \text{empfile}^0(x, y, z)
\]

\[
\text{empfile}^-(x, y, z) \xrightarrow{\text{compiles}} \text{Employee}^-(x, y, z) \land \text{empfile}^0(x, y, z)
\]

This doesn’t quite work:

after executing the 1st update query we no longer have \text{empfile}^0!

Possible Solutions:

1. simultaneous \textit{relational} assignment:
   \[\Rightarrow\text{compute all deltas and store results in temporary storage,}\]
   \[\Rightarrow\text{only then apply all deltas to } S_A;\]

2. using independent deltas:
   \[\Rightarrow\text{add constraints to avoid the problem (e.g., } P^- \subseteq P^0);\]

3. evolving physical schema one AP at a time
   \[\Rightarrow\text{sequence of update schemas with a subset of } S_A \text{ “updated”,}\]
   \[\Rightarrow\text{subsequent updates compiled w.r.t. partially updated schema.}\]
Value Invention

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

- A base file empfile(r, x, y, z) of emp records with 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 Spyrou]

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 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!
(due to: ∀x, y, z.Employee(x, y, z) ↔ (∃r.empfile(r, x, y, z))

IDEA (Constant Complement [Bancilhon and Spyra])

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 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 andSpyratos])

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 \texttt{Employee(id, name, salary)}
- A base file \texttt{empfile(r, x, y, z)} of emp records \textit{with} RIDs “r”
- An \texttt{emp-name(y, r)} index on employee names (links name to RIDs)

\implies no update query, e.g., for \texttt{empfile^+(r, x, y, z)}: no “source” of RIDs!

IDEA (Constant Complement [Bancilhon and Spyrtatos])

An \textit{oracle access path} that provides the required value
given the values of remaining attributes as parameters.

In practice: a record allocation mechanism
(e.g., \texttt{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 (\textit{AP}^+ doesn’t have the attribute)

\implies update query for \texttt{emp-name^+} must execute \textit{after} \texttt{empfile^+}. 
Value Invention

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

- A base file `empfile(r, x, y, z)` of employee 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+`. 
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.
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.
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.
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 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.
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

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

David Toman  (University of Waterloo)
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

```plaintext
while ¬end-of(emppages) do
    read emppages to p;
    while ¬end-of(emprecords(p)) do
        read emprecords(p) to r;
        if r → salary < 100k then
            r → 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:
  ⇒ `emprecords−` becomes “all old records”
  `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:
  - `emprecords^-` becomes “all old records”
  - `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:
  - `emprecords^−` becomes “all old records”
  - `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 $\text{emppages}/1/0$ and $\text{emprecords}/2/1$;
- 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*

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

- or more schema design??
  $$\Rightarrow$$ separate “access paths” for reading/writing
  $$\Rightarrow$$ 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*
  \[ \Rightarrow \text{NLJ(emppages}(p)\text{, NLJ(emprecords}(p, r), \text{modify } r)) \]
  - or more schema design??
    \[ \Rightarrow \text{separate “access paths” for reading/writing} \]
    \[ \Rightarrow \text{sequencing, e.g., via union, etc.} \]
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*
  
  ⇒ NLJ(emppages(p), NLJ(emprecords(p, r), 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

```plaintext
while ¬end-of(emplist) do
  read emplist to r;
  if r → salary < 100k then
    r → 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

```haskell
while ¬end-of(emplist) do
  read emplist to r;
  if r → salary < 100k then
    r → 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

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

```plaintext
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
```

Does this work?? **NO!!!**

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

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

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

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

Does this work?? NO!!!

⇒ consider `emplist = [(Fred, 10k), (Wilma, 15k)]` to start with;
⇒ result `emplist = [(Fred, 100k), (Wilma, 105k)]` at the end...
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 → salary < 100k then
    r → 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 than 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 than 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 *emp* pages!)
- 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 *emp* pages!)
- 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 any pages!)
- 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!