Topics in Database Systems: Main/in-memory and Embedded DBMS
CS848 Spring 2018

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

\[ \Sigma_L \rightarrow S_L \rightarrow Q_L \downarrow \Sigma_{LP} \rightarrow \text{(query compilation)} \rightarrow Q_P \rightarrow S_P \rightarrow \Sigma_P \]
Updates in Nutshell
Physical Design and Updates: Overview

old instance

new instance
Physical Design and Updates: Overview

Update in a Nutshell
Physical Design and Updates: Overview

old instance

new instance

user update $U_L$

(compile to)

physical update $U_P$
Physical Design and Updates: Overview

old instance

\[ \Sigma_L \quad S_L \quad \Sigma_{LP} \quad S_P \]

user update \( U_L \)

(compile to)

physical update \( U_P \)

new instance

\[ \Sigma_L \quad S_L \quad \Sigma_{LP} \quad S_P \]
Update Schema

\[ \Sigma_L \xrightarrow{\text{user update } U_L} \Sigma_L \]

\[ \Sigma_P \xrightarrow{\text{physical update } U_P} \Sigma_P \]

old instance \( \to \) new instance
Update Schema

$\Sigma^o_L \rightarrow S^o_L \rightarrow \Sigma^o_{LP} \rightarrow S^0_P \rightarrow \Sigma^o_P$

user update $U_L$

$\Sigma_L \rightarrow S^n_L \rightarrow \Sigma^n_{LP} \rightarrow S^n_P \rightarrow \Sigma^n_P$

physical update $U_P$
Update in a Nutshell

\[ \Sigma_L^o \rightarrow S_L^o \rightarrow \Sigma_{LP}^o \rightarrow S_P^o \rightarrow \Sigma_P^o \]

\[ \Sigma_L^n \leftarrow S_L^n \leftarrow \Sigma_{LP}^n \leftarrow S_P^n \leftarrow \Sigma_P^n \]

physical update \( U_P \)

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

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

\[ \Sigma_{L}^{o} \rightarrow S_{L}^{o} \]

\[ \Sigma_{L}^{o} \rightarrow \Sigma_{L}^{o} \]

\[ \Sigma_{p}^{o} \rightarrow S_{p}^{o} \]

\[ S_{L}^{\pm}, \Sigma_{L}^{\pm} \rightarrow \Sigma_{L}^{n} \]

\[ S_{L}^{\pm}, \Sigma_{L}^{\pm} \rightarrow \Sigma_{L}^{n} \]

\[ S_{p}^{\pm}, \Sigma_{p}^{\pm} \rightarrow \Sigma_{p}^{n} \]

\[ S_{p}^{\pm}, \Sigma_{p}^{\pm} \rightarrow \Sigma_{p}^{n} \]
Update Schema

\[ \Sigma_L^o, S_L^o, \Sigma_{LP}^o \rightarrow S_L^{\pm}, \Sigma_L^{\pm}, S_P^{\pm} := Q(S_P^o, S_L^{\pm}), \Sigma_P^{\pm} \rightarrow \Sigma_P^n, S_P^n, \Sigma_{LP}^n \]
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 $P \in S_A$ with 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) $P$. 

$U_L$ ← \[\Sigma \left(\text{update compilation}\right)\] $U_P$
[Update in a Nutshell]

- $U_L$ 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) $P$. 

David Toman (University of Waterloo)
• $U_L$ 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) $P$. 

 tíes. 

David Toman  (University of Waterloo)  Update in a Nutshell
**Physical Design and Update Compilation**

- \( U_L \) is a *user query* \( P^+ (\overline{x}) \) (\( P^- (\overline{x}) \)) for \( P \in S_A \);
- \( U_P \) is a *plan* for the user query \( P^+ (\overline{x}) \) (\( P^- (\overline{x}) \)) 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 $\text{Employee}(id, name, salary)$

- 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:
\[ S_L = \{ \text{Employee/3} \}, \Sigma_L = \{ \text{“id is a key”} \} \]

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

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

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

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

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

Update Queries:

\[ \text{empfile}^+(x, y, z) \]
\[ \text{empfile}^-(x, y, z) \]

...similar for emp-name
Example

Setup: standard relational design for \texttt{Employee(id, name, salary)}
- A \textit{base file} \texttt{empfile} of \texttt{emp} records (organized by \texttt{id})
- An \texttt{emp-name} index on \texttt{employee} names (links \texttt{name} to \texttt{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:
\[ S_P = \{ \text{Employee}^+/3, \text{Employee}^-/3, \text{empfile}^o/3, \text{empfile}^o/3, \ldots \}\]
\[ \Sigma_{LP} = \{ \forall x, y, z. (\text{empfile}^o(x, y, z) \lor \text{empfile}^+(x, y, z)) \leftrightarrow (\text{empfile}^n(x, y, z) \lor \text{empfile}^-(x, y, z)), \ldots \}\]
\[ \Sigma_P = \{ \forall x, y, z. \text{Employee}^+(x, y, z) \land \text{Employee}^-(x, y, z) \rightarrow \bot, \ldots \}\]

Update Queries:
\[ \text{empfile}^+(x, y, z) \stackrel{\text{compiles}}{\rightarrow} \text{Employee}^+(x, y, z) \land \neg \text{empfile}^o(x, y, z) \]
\[ \text{empfile}^-(x, y, z) \stackrel{\text{compiles}}{\rightarrow} \text{Employee}^-(x, y, z) \land \text{empfile}^o(x, y, z) \]

...similar for \texttt{emp-name}
Example

Setup: standard relational design for \texttt{Employee}(id, name, salary)

- A \textit{base file} \texttt{empfile} of \texttt{emp} records (organized by \texttt{id})
- An \texttt{emp-name} index on \texttt{employee} names (links \texttt{name} to \texttt{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:

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

\[
\Sigma_{LP} = \{\forall x, y, z. (\texttt{empfile}^o(x, y, z) \lor \texttt{empfile}^+(x, y, z))
\]

\[
\quad\quad\leftrightarrow (\texttt{empfile}^n(x, y, z) \lor \texttt{empfile}^-(x, y, z)), \ldots\}
\]

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

Update Queries:

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

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

\[
\ldots\text{similar for \texttt{emp-name}}\]
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)
Update in a Nutshell
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 $S_L$, 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?

\[ \Rightarrow \text{insertable, deletable, and updatable views} \]

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 \( P \in S_L \).

\[ \Rightarrow \text{when the answer is positive, we construct a corresponding update queries.} \]
ADVANCED ISSUES IN UPDATE COMPILATION
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 empfile^0!

Possible Solutions:

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

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

3. evolving physical schema one AP at a time
   \( \Rightarrow \) sequence of update schemas with a subset of \( S_\Lambda \) “updated”,
   \( \Rightarrow \) 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:

- **Simultaneous relational assignment:**
  - compute all deltas and store results in temporary storage,
  - only then apply all deltas to \( S_A \);

- **Using independent deltas:**
  - add constraints to avoid the problem (e.g., \( P^- \subseteq P^0 \));

- **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:

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

This doesn’t quite work:

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

Possible Solutions:

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

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

3. evolving physical schema one AP at a time
   - \( \Rightarrow \) sequence of update schemas with a subset of \( S_A \) “updated”,
   - \( \Rightarrow \) 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 compute all deltas and store results in temporary storage,
   - \Rightarrow \textit{only then} apply all deltas to \( S_A \);

2. using independent deltas:
   - \Rightarrow 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”,
   - \Rightarrow 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 we no longer have \text{empfile}^0!

Possible Solutions:

1. 
   simultaneous \textit{relational} assignment:
   - compute all deltas and store results in temporary storage,
   - \textit{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 \texttt{Employee(id, name, salary)}

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

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

\textbf{IDEA (Constant Complement [Bancilhon and Spyridos])}

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 (\texttt{AP+} doesn’t have the attribute)

⇒ 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 \( \text{empfile}(r, x, y, z) \) of emp records with RIds “r”
- An emp-name\((y, r)\) index on employee names (links name to RIds)

\[ \Rightarrow \text{no update query, e.g., for empfile}^+(r, x, y, z): \text{no “source” of RIds!} \]

(due to: \( \forall x, y, z. \text{Employee}(x, y, z) \Leftrightarrow (\exists r. \text{empfile}(r, x, y, z)) \))

IDEA (Constant Complement [Bancilhon and Spyranos])

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)

\[ \Rightarrow \text{update query for emp-name}^+ \text{must execute after empfile}^+. \]
Value Invention

Setup: advanced relational design for $\text{Employee}(id, name, salary)$

- A base file $\text{empfile}(r, x, y, z)$ of emp records with RIds “r”
- An $\text{emp-name}(y, r)$ index on employee names (links name to RIds)

$\Rightarrow$ no update query, e.g., for $\text{empfile}^+(r, x, y, z)$: no “source” of RIds!

IDEA (Constant Complement [Bancilhon and Spyritos])

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 $\text{emp-name}^+$ must execute after $\text{empfile}^+$. 
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 Spyritos])

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 and Spyroatos])

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 \textit{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

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

\[ \Rightarrow \text{impossible to insert the 1st employee and 1st department!} \]

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 \texttt{emp-id} AP (yields address of \texttt{emp});
2. insert department record (the above value used for the manager field; yields address of \texttt{dept});
3. insert the same employee into \texttt{emp-dept} AP using the \texttt{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.
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 \textit{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.
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*?
  - \(\Rightarrow\) guaranteed by the user (e.g., extra user queries to make sure)
  - \(\Rightarrow\) system-generated checks—HARD!
Additional Issues

• How to know when an \textit{constant complement} is needed?
• How to determine the \textit{ordering} of the individual AP updates?
• How to identify when \textit{reification} is needed?
• How to determine if the user update preserves \textit{consistency}?
  \[\Rightarrow\text{guaranteed by the user (e.g., extra user queries to make sure)}\]
  \[\Rightarrow\text{system-generated checks—HARD!}\]
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!
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!