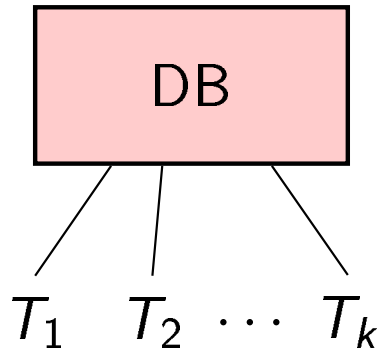
A Model of Independence and Overlap for Transactions on Database Schemata

Stephen J. Hegner
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

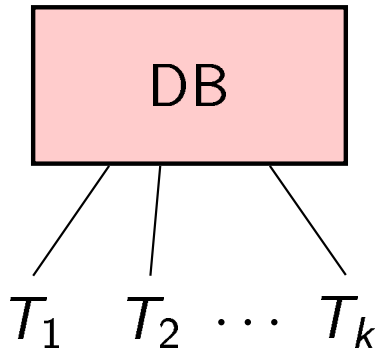
Serializability for Concurrency Control

- It is very common that transactions share access to a database.



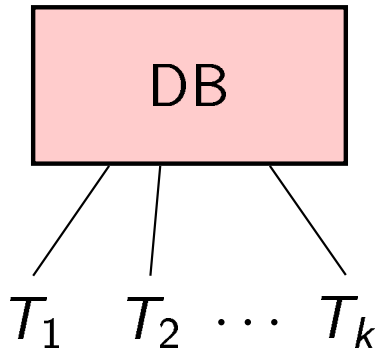
Serializability for Concurrency Control

- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.



Serializability for Concurrency Control

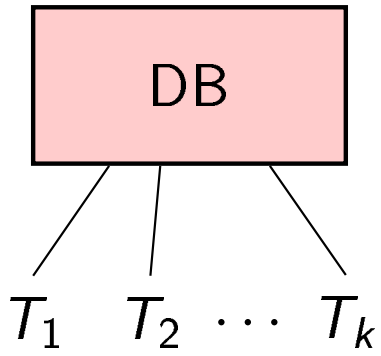
- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.



- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

Serializability for Concurrency Control

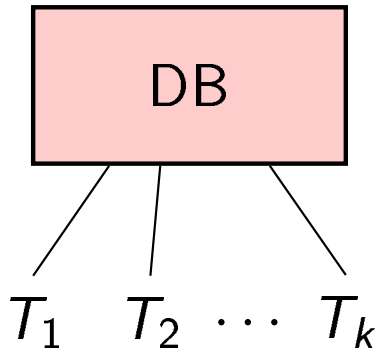
- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.



- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

Not OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, X\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, X\rangle$

Serializability for Concurrency Control

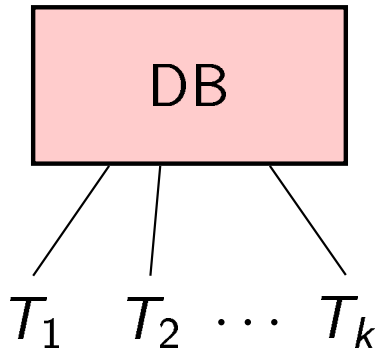


- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.
- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

Not OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, X\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, X\rangle$

- Operations on distinct data objects are never modelled as conflicting.

Serializability for Concurrency Control



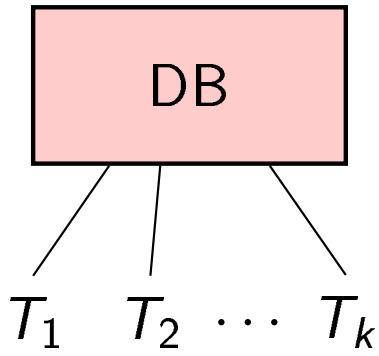
- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.
- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

Not OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, X\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, X\rangle$

- Operations on distinct data objects are never modelled as conflicting.

OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, Y\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, Y\rangle$

Serializability for Concurrency Control



- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.
- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

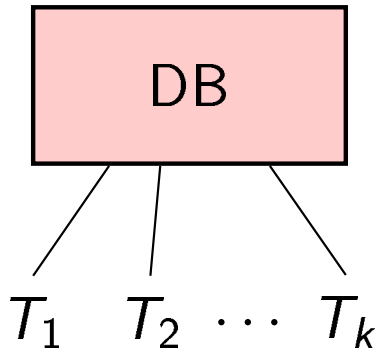
Not OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, X\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, X\rangle$

- Operations on distinct data objects are never modelled as conflicting.

OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, Y\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, Y\rangle$

Questions: Is this model adequate?

Serializability for Concurrency Control



- It is very common that transactions share access to a database.
- The classical solution to concurrency control is *serializability* of the schedule of operations.
- Roughly, serializability requires that the read and write operation interleave as in some serial schedule.

Not OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, X\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, X\rangle$

- Operations on distinct data objects are never modelled as conflicting.

OK: $\text{Read}\langle T_i, X\rangle$ $\text{Read}\langle T_j, Y\rangle$ $\text{Write}\langle T_i, X\rangle$ $\text{Write}\langle T_j, Y\rangle$

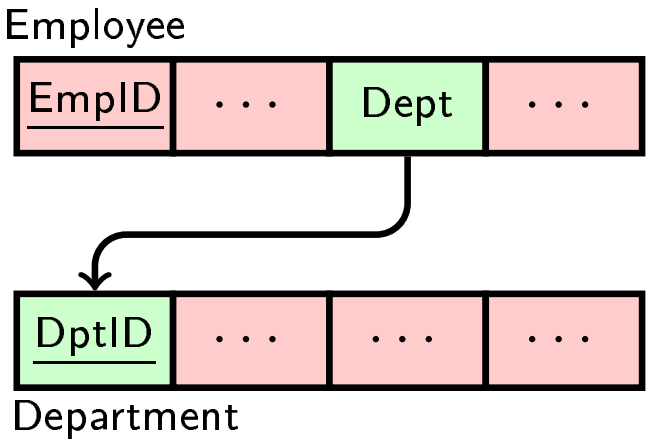
Questions: Is this model adequate?
Can operations on distinct data objects be in conflict?

Interdependent Data Objects

- In the presence of integrity constraints, non-overlapping data objects may be interdependent.

Interdependent Data Objects

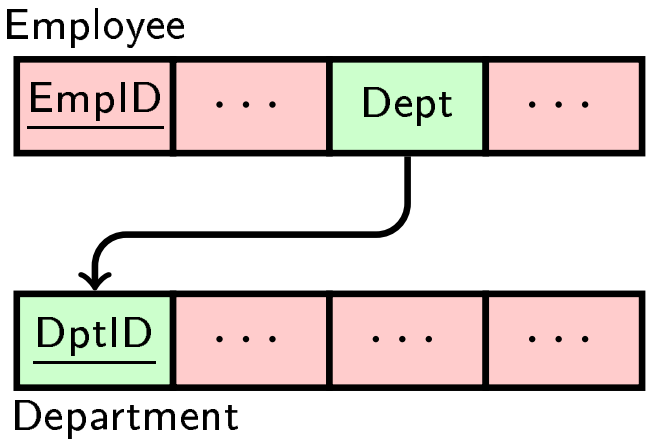
- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.



Interdependent Data Objects

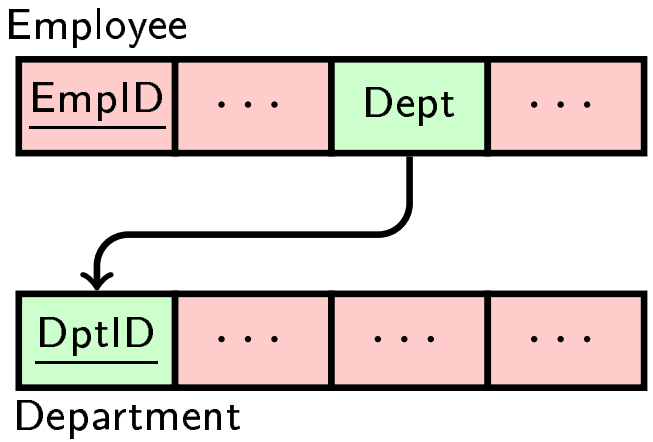
- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.

T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].



Interdependent Data Objects

- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.

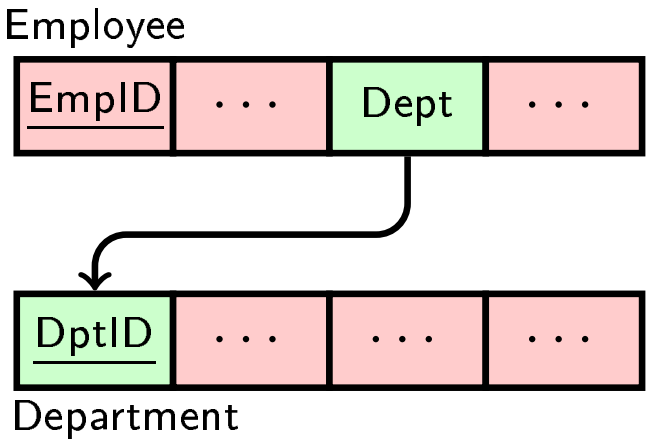


T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].

T_2: Assign Alice to the *Research* department [modifies Employee only].

Interdependent Data Objects

- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.



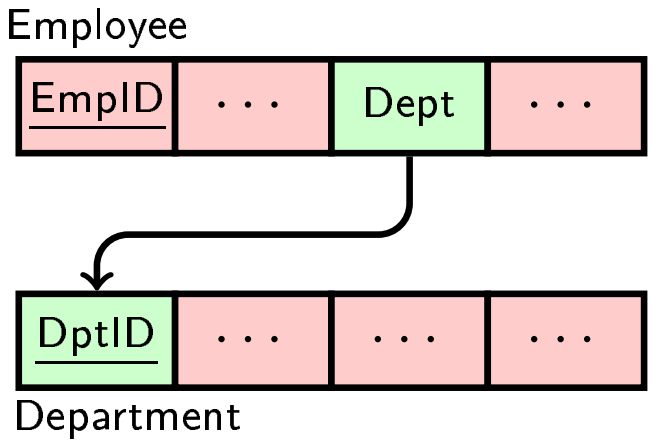
T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].

T_2: Assign Alice to the *Research* department [modifies Employee only].

- Each of T_1 and T_2 may be run by itself with no violation of integrity constraints.

Interdependent Data Objects

- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.



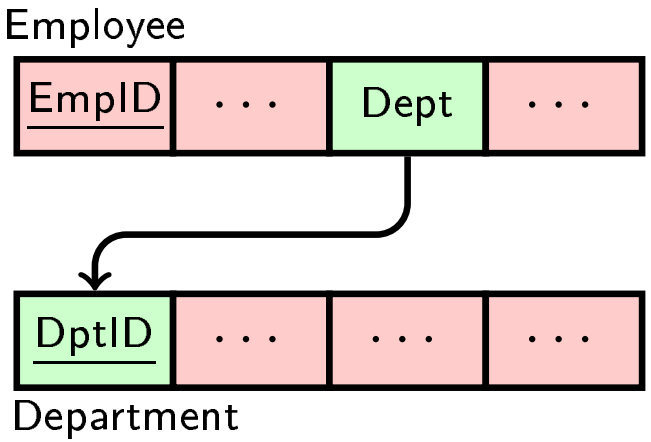
T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].

T_2: Assign Alice to the *Research* department [modifies Employee only].

- Each of T_1 and T_2 may be run by itself with no violation of integrity constraints.
- T_1 and T_2 operate on distinct data objects yet cannot both be run, even serially.

Interdependent Data Objects

- In the presence of integrity constraints, non-overlapping data objects may be interdependent.
- An example is defined by a foreign key constraint.



T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].

T_2: Assign Alice to the *Research* department [modifies Employee only].

- Each of T_1 and T_2 may be run by itself with no violation of integrity constraints.
- T_1 and T_2 operate on distinct data objects yet cannot both be run, even serially.
- Their overlap is only of a very limited read-only nature.

Solutions to the Management of Interdependent Updates

Classical solution:

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

- Abort and re-run should only be used as a last resort.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

- Abort and re-run should only be used as a last resort.
- Relative to computer speed, human decision making and interactive input take a very long time.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

- Abort and re-run should only be used as a last resort.
- Relative to computer speed, human decision making and interactive input take a very long time.
  $\Rightarrow$ Justify increased preprocessing to minimize conflict in concurrency.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

- Abort and re-run should only be used as a last resort.
- Relative to computer speed, human decision making and interactive input take a very long time.
  - $\Rightarrow$ Justify increased preprocessing to minimize conflict in concurrency.
  - $\Rightarrow$ Claiming/locking of data objects as fine-grained as possible.

Solutions to the Management of Interdependent Updates

Classical solution:

- The integrity of the overall update is checked at each commit.
- Inconsistency of potential commits resolved by aborting one or more transactions.
- A locking protocol is employed to ensure that all access is authorized.

Observations about situations with human interaction:

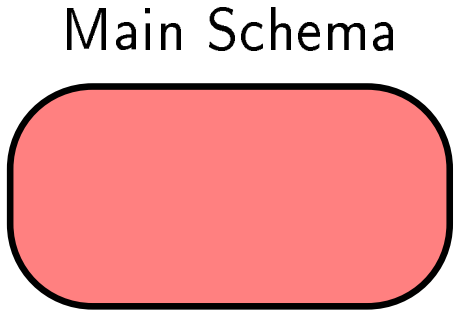
- Abort and re-run should only be used as a last resort.
- Relative to computer speed, human decision making and interactive input take a very long time.
  - $\Rightarrow$ Justify increased preprocessing to minimize conflict in concurrency.
  - $\Rightarrow$ Claiming/locking of data objects as fine-grained as possible.

Focus of this research:

- A fine-grained model of interdependence for data objects.

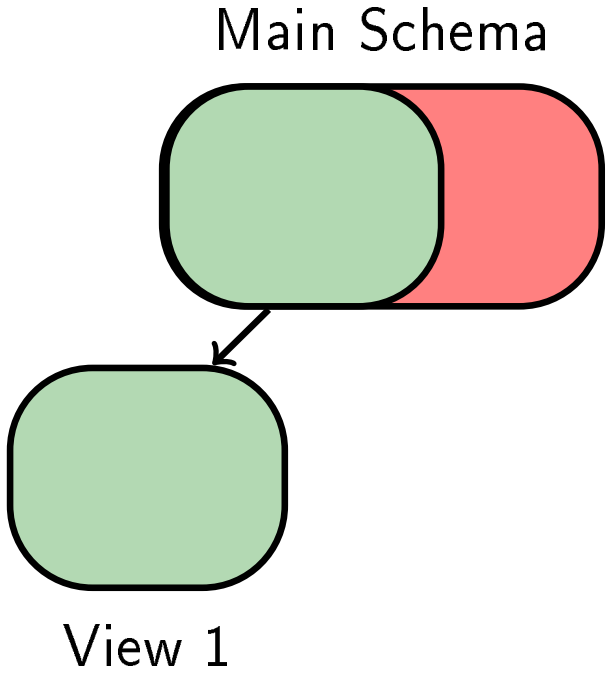
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.



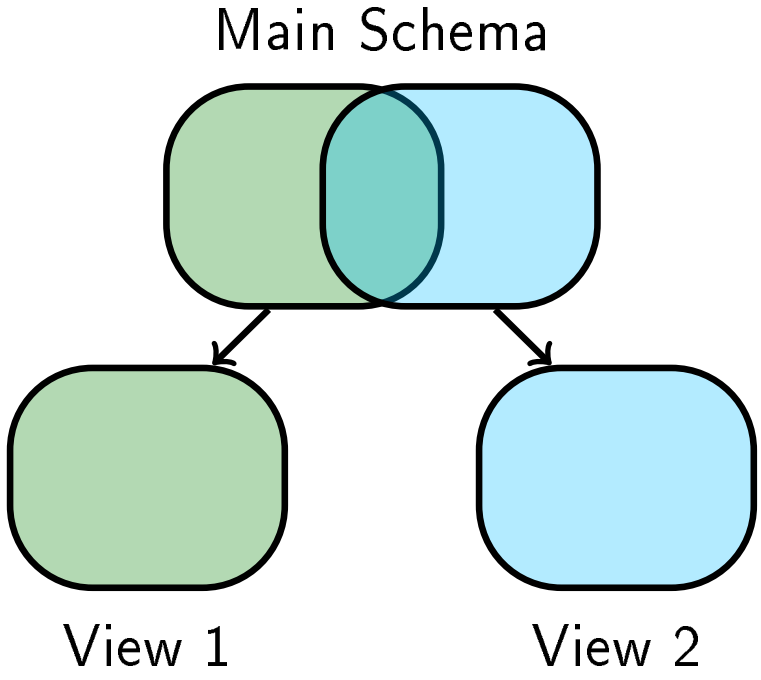
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated



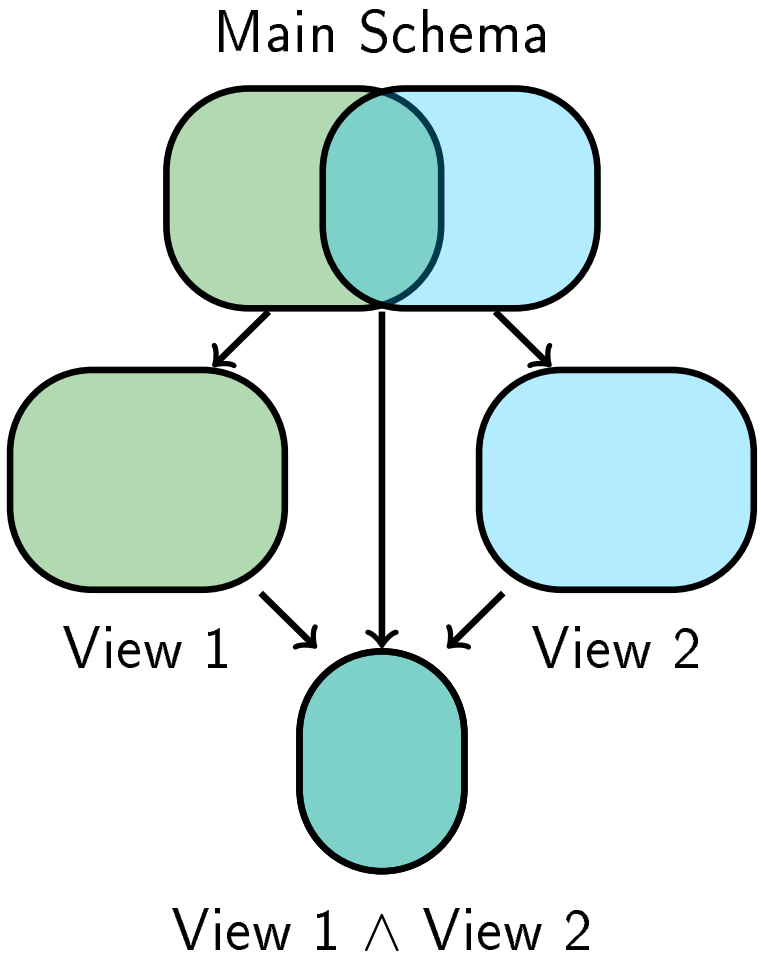
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.



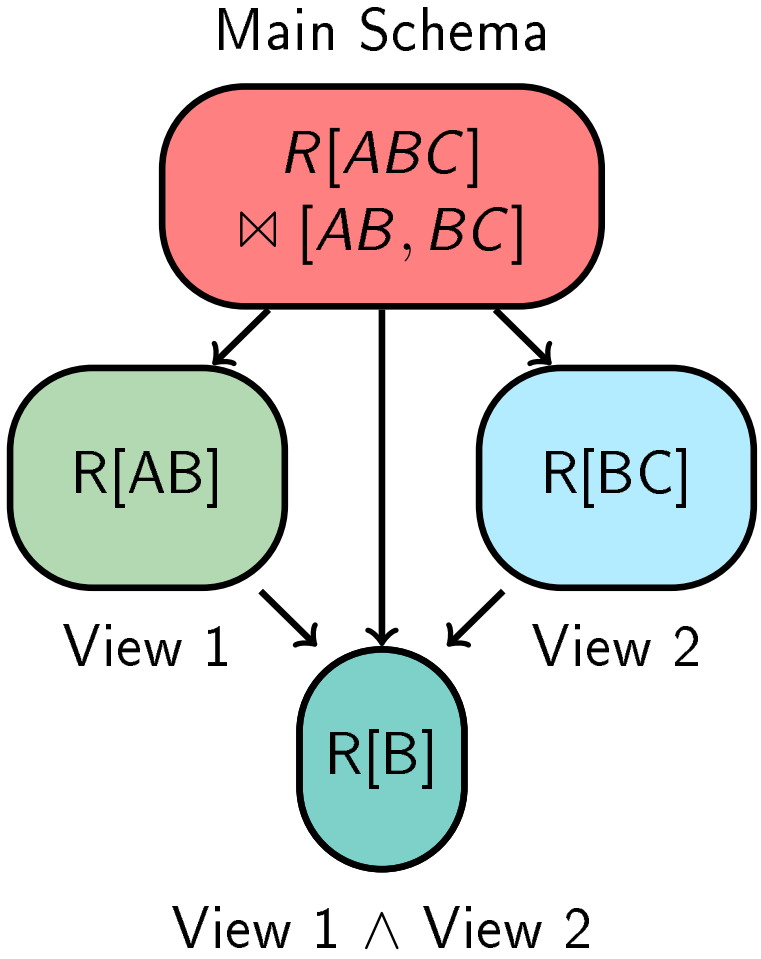
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.



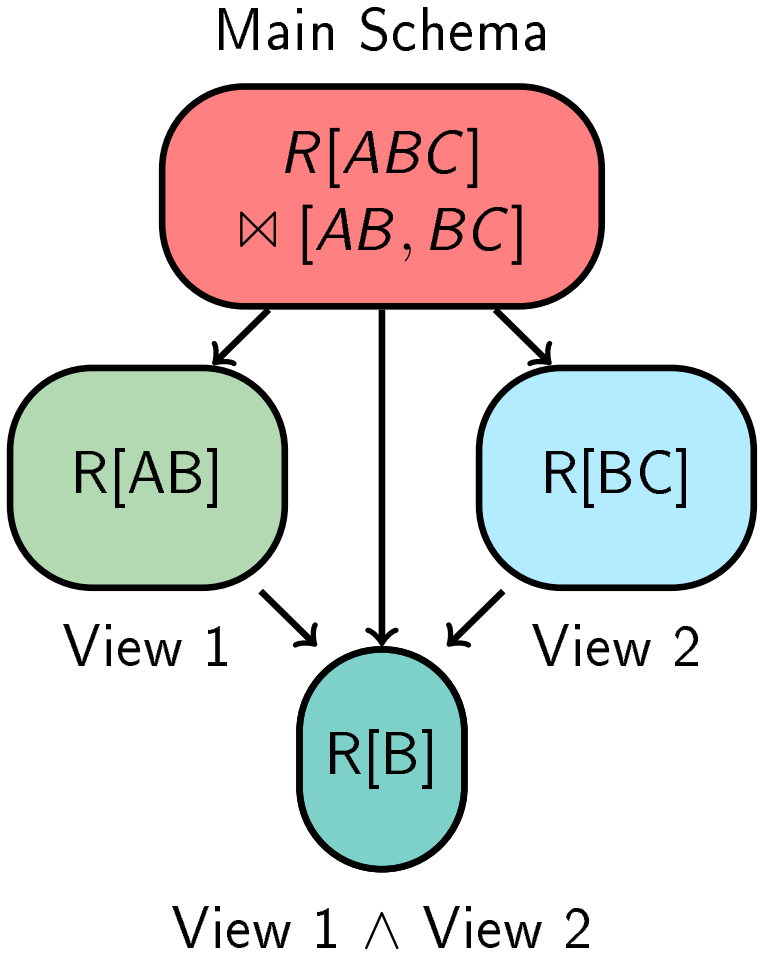
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.



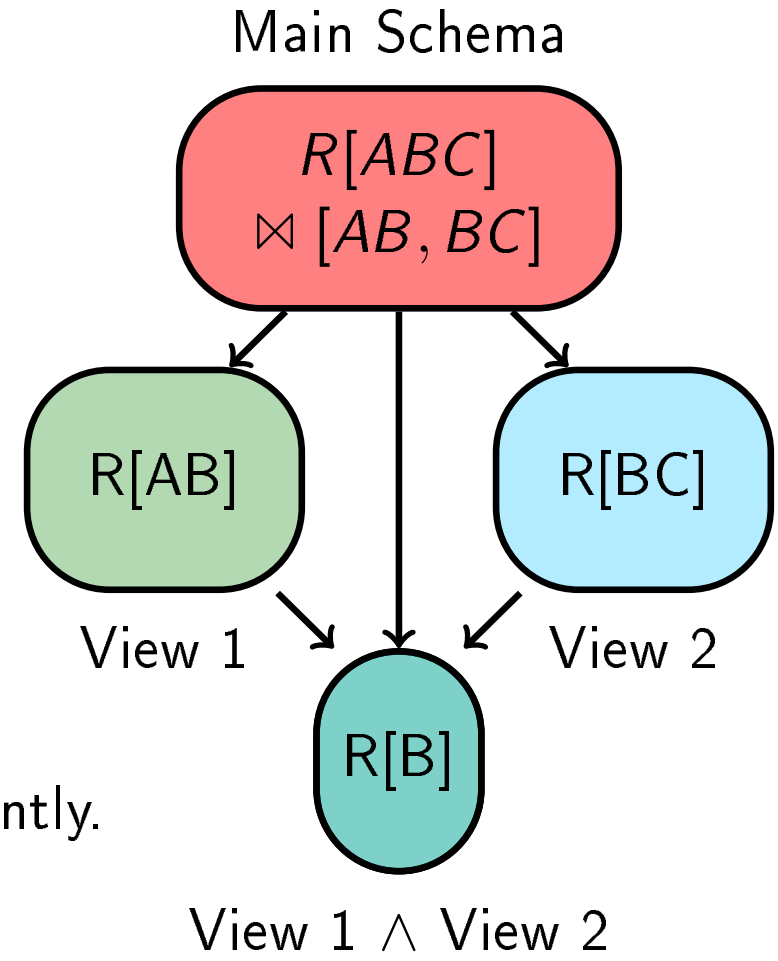
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.
- The solution is actually symmetric.



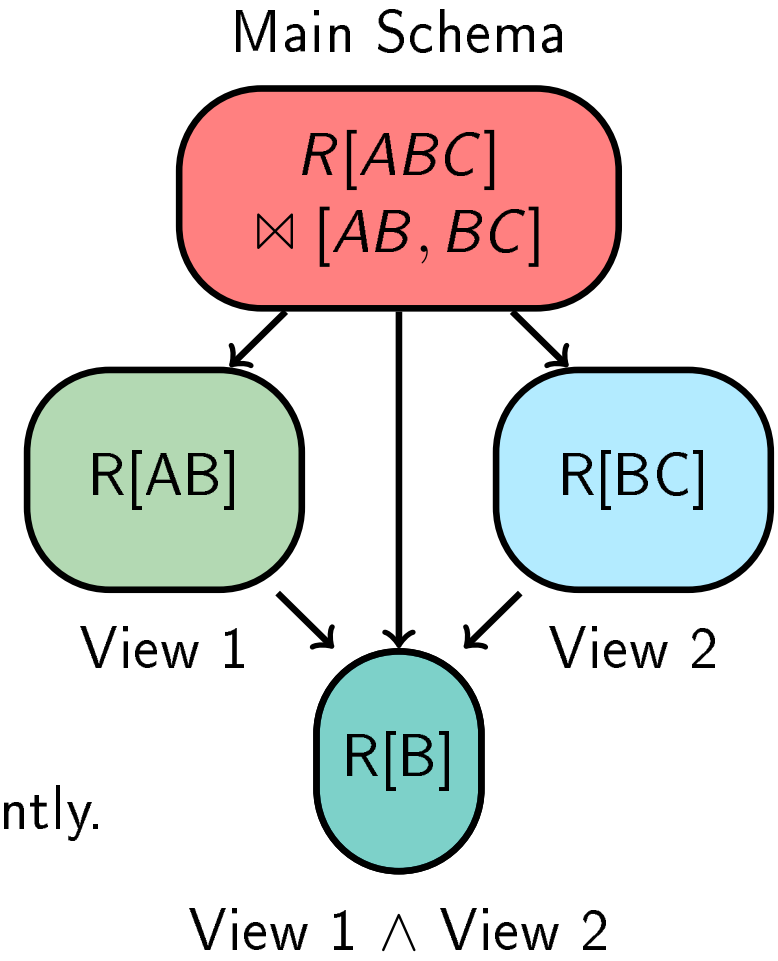
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.
- The solution is actually symmetric.
- The two views may be updated independently.



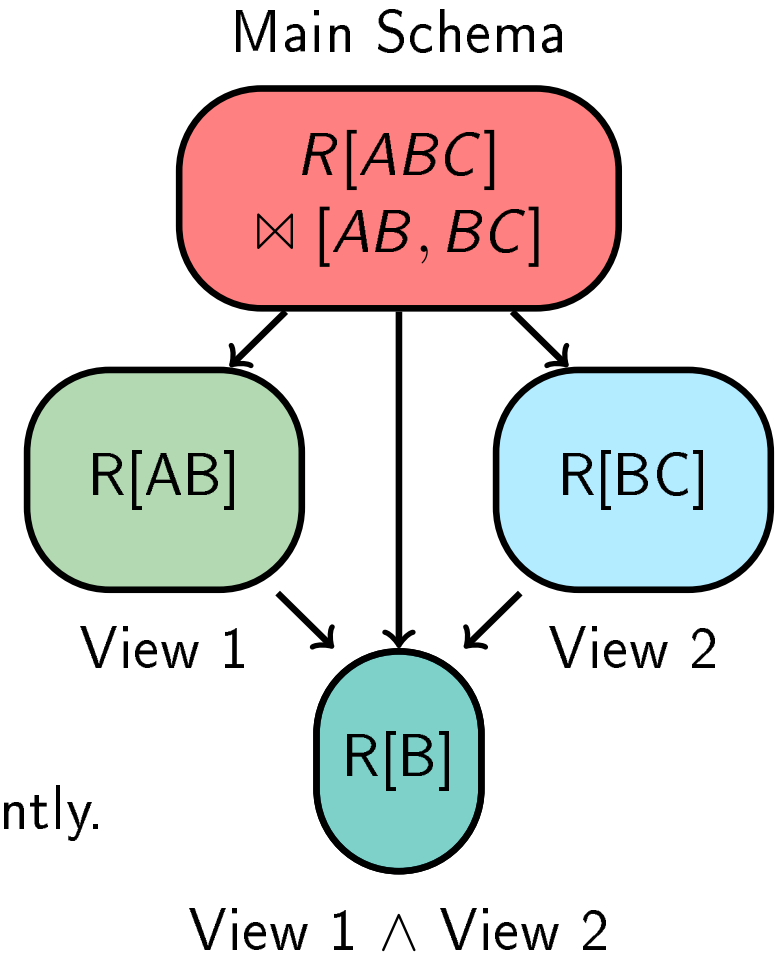
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.
- The solution is actually symmetric.
- The two views may be updated independently.
- Think of View 1 and View 2 defining data objects.



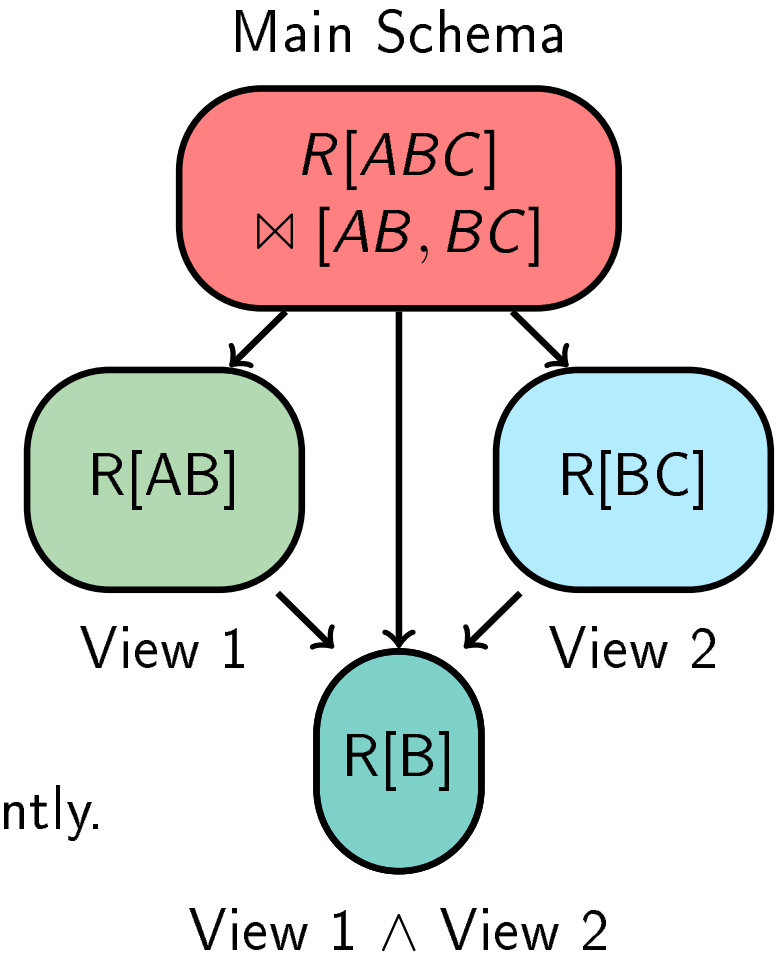
Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.
- The solution is actually symmetric.
- The two views may be updated independently.
- Think of View 1 and View 2 defining data objects.
- They may be updated independently, in any order, provided that View 1 $\wedge$ View 2 is held constant.



Characterization of Independence for Two Data Objects

- The *constant complement strategy* is a classical solution to the *view-update-translation problem*.
- View 1 to be updated is matched with a *meet complement* View 2.
- Meet View 1 $\wedge$ View 2 held constant.
- A classical example is dependency-preserving decomposition via a JD.
- The solution is actually symmetric.
- The two views may be updated independently.
- Think of View 1 and View 2 defining data objects.



- They may be updated independently, in any order, provided that View 1 $\wedge$ View 2 is held constant.
- This forms the basic idea for independent data objects.

An Overview of the Algebra of Data Objects

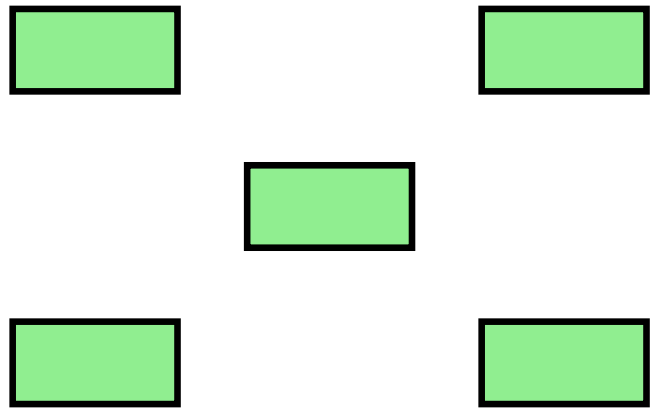
- The idea of independence may be extended to many components.

An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.

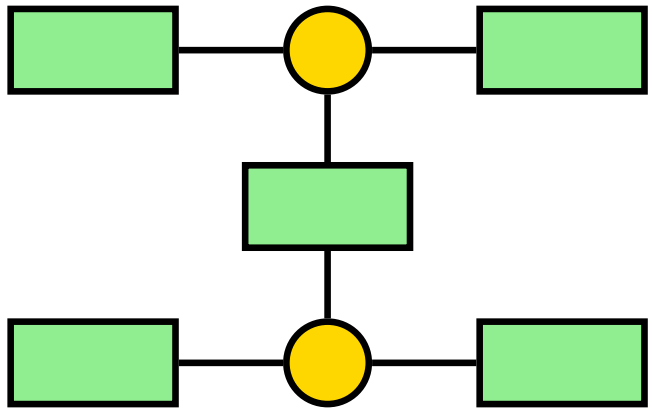
An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.



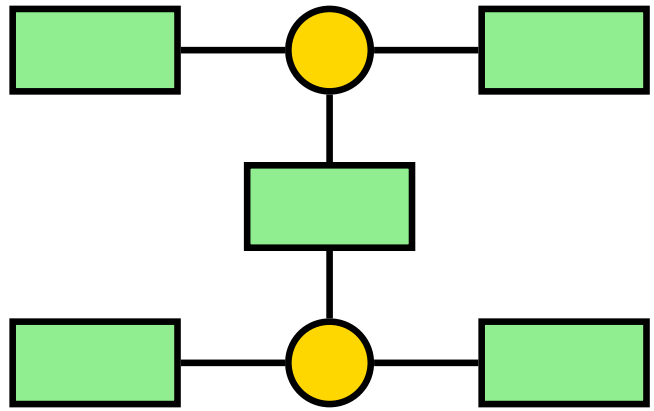
An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.
- Each data object has zero or more read-only sub-views called ports.



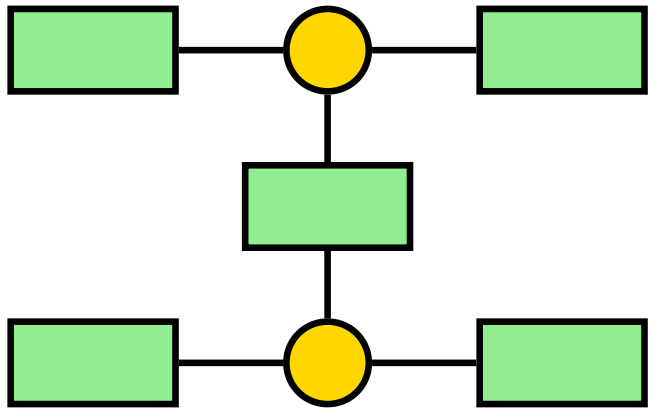
An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.
- Each data object has zero or more read-only sub-views called ports.
- Data objects overlap by sharing ports.



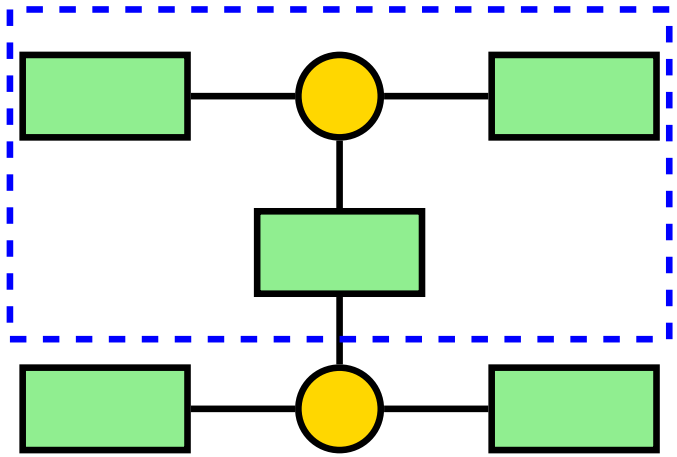
An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.
- Each data object has zero or more read-only sub-views called ports.
- Data objects overlap by sharing ports.
- Updates to a data object must keep the ports constant.



An Overview of the Algebra of Data Objects

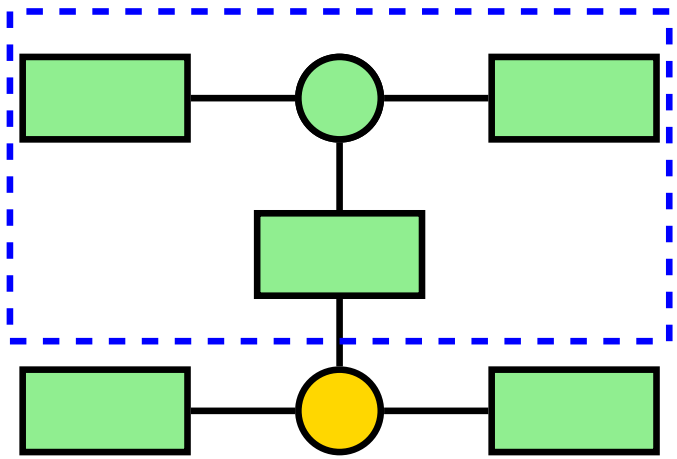
- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.
- Each data object has zero or more read-only sub-views called ports.
- Data objects overlap by sharing ports.



- Updates to a data object must keep the ports constant.
- Data objects may be combined to form larger objects.

An Overview of the Algebra of Data Objects

- The idea of independence may be extended to many components.
- The idea is based upon *database schema components* and has roots in classical *pairwise definability*.
- Each data object is a view of the main schema.
- Each data object has zero or more read-only sub-views called ports.
- Data objects overlap by sharing ports.



- Updates to a data object must keep the ports constant.
- Data objects may be combined to form larger objects.
- To obtain a write claim on a port, all basic components which share that port must be combined into a larger complex object.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Planes of decomposition: There are two important planes along which data objects are constructed.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Planes of decomposition: There are two important planes along which data objects are constructed.

Vertical decomposition: Classical DB decomposition is based upon *projection* π.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Planes of decomposition: There are two important planes along which data objects are constructed.

Vertical decomposition: Classical DB decomposition is based upon *projection* π.

Horizontal decomposition: Transactions often need to claim parts of the DB based upon attribute *selection* σ.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Planes of decomposition: There are two important planes along which data objects are constructed.

Vertical decomposition: Classical DB decomposition is based upon *projection* π.

Horizontal decomposition: Transactions often need to claim parts of the DB based upon attribute *selection* σ.

- The basic data objects represent parts of the DB obtained by operations along both of these planes.

Interdependent Data Objects in a Common Context

- It is important to show that these ideas may be concretized to common DBMS contexts.

Model: The relational model is still by far the most widely used in DBMS.

Dependencies: Two forms of constraints dominate in practice:

FDs: Functional dependencies, in particular key dependencies.

FKDs: Foreign key dependencies, special case of *inclusion dependencies*.

Planes of decomposition: There are two important planes along which data objects are constructed.

Vertical decomposition: Classical DB decomposition is based upon *projection* π.

Horizontal decomposition: Transactions often need to claim parts of the DB based upon attribute *selection* σ.

- The basic data objects represent parts of the DB obtained by operations along both of these planes.
- The remainder of the talk will sketch how these goals are realized.

Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

Data Objects Defined by Vertical Decomposition

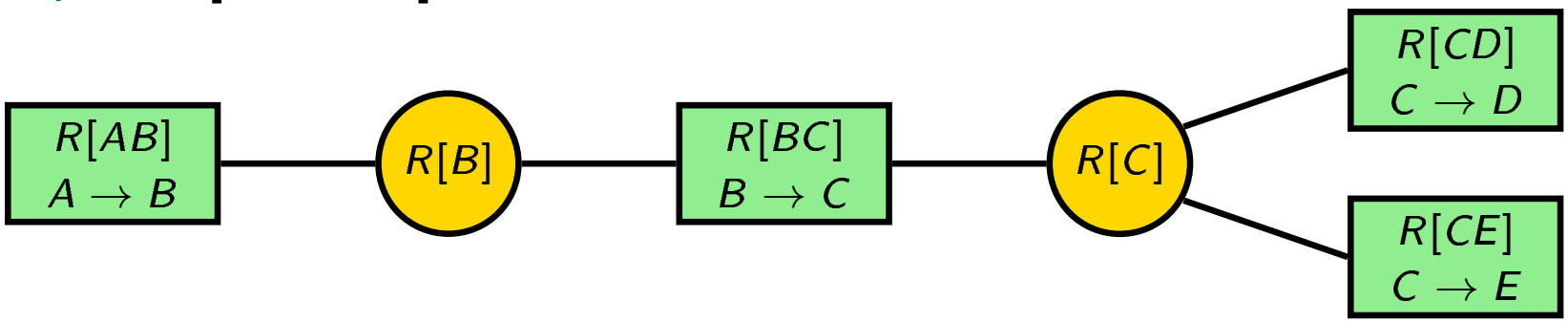
- The vertical plane of components is defined by standard pairwise decomposition.

Example: $R[ABCDE]$; $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow DE$.

Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

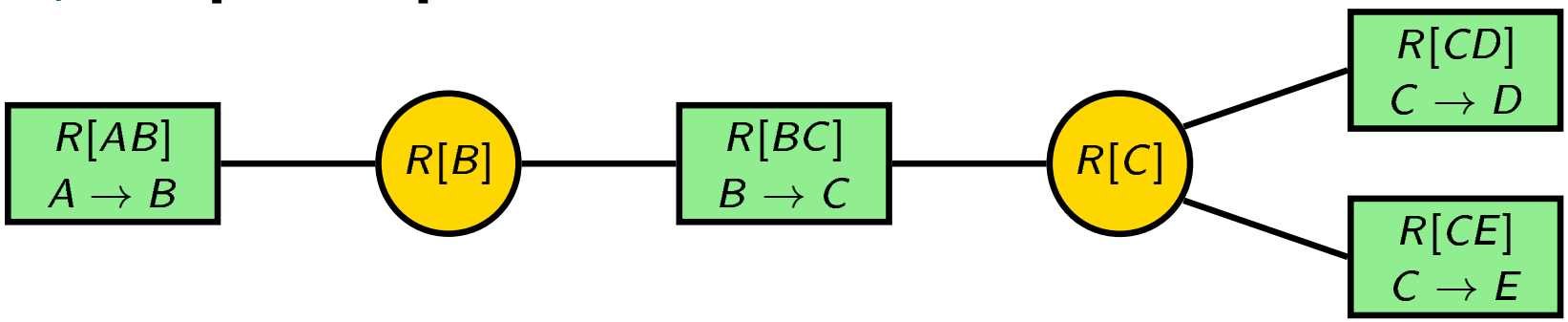
Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.



Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.

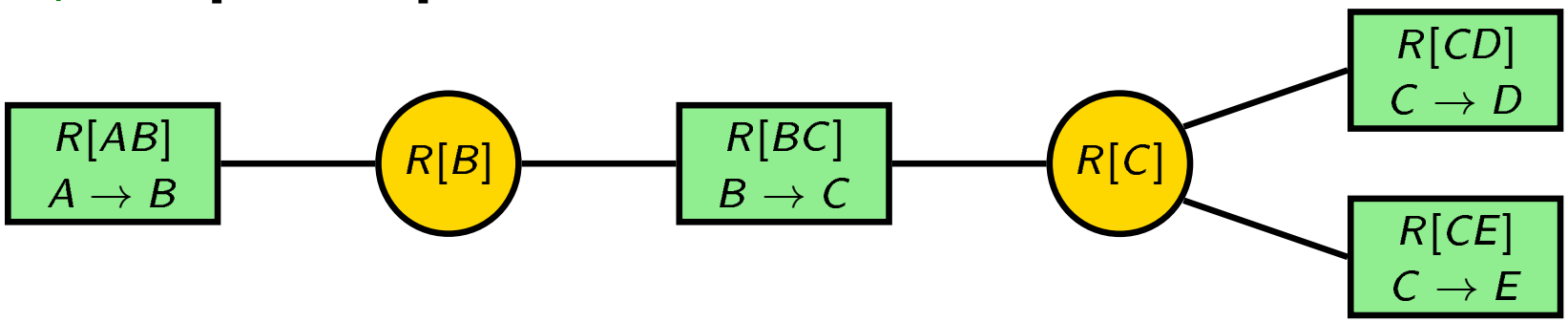


Additional requirement for data objects: Each component may be governed by at most one (key) FD.

Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.



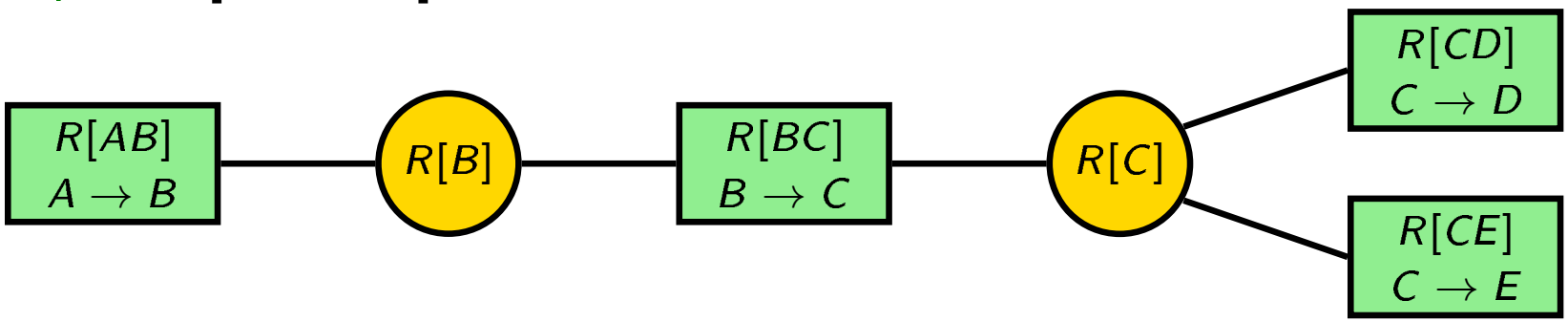
Additional requirement for data objects: Each component may be governed by at most one (key) FD.

- This is accomplished via "redundant" decomposition.

Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.



Additional requirement for data objects: Each component may be governed by at most one (key) FD.

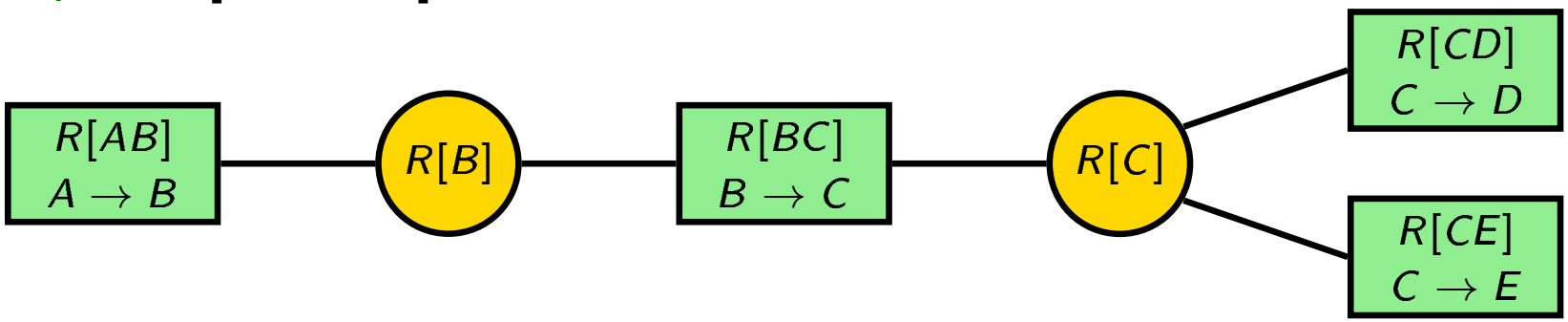
- This is accomplished via "redundant" decomposition.

Example: $R[ABC]$, $AB \to C$, $C \to B$.

Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

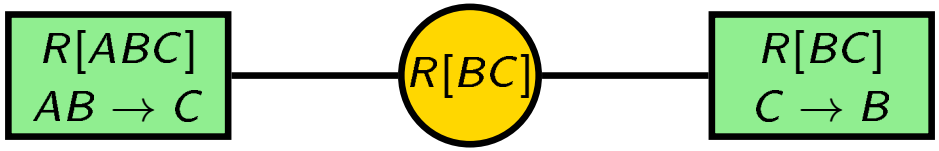
Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.



Additional requirement for data objects: Each component may be governed by at most one (key) FD.

- This is accomplished via "redundant" decomposition.

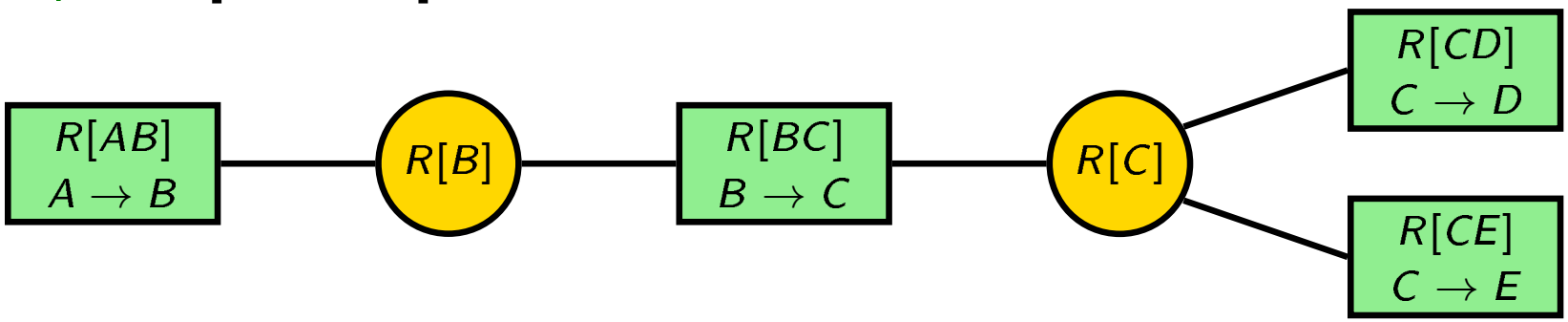
Example: $R[ABC]$, $AB \to C$, $C \to B$.



Data Objects Defined by Vertical Decomposition

- The vertical plane of components is defined by standard pairwise decomposition.

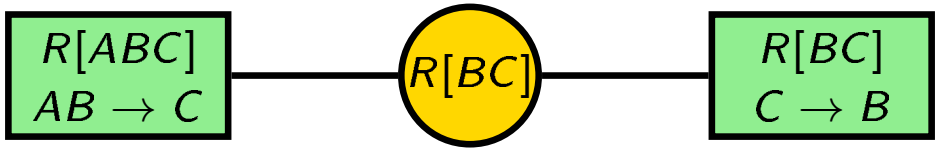
Example: $R[ABCDE]$; $A \to B$, $B \to C$, $C \to DE$.



Additional requirement for data objects: Each component may be governed by at most one (key) FD.

- This is accomplished via "redundant" decomposition.

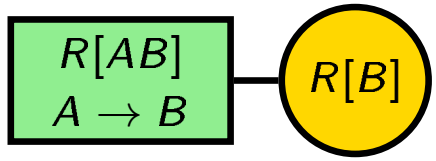
Example: $R[ABC]$, $AB \to C$, $C \to B$.



- These are object definitions, not materialized views!!

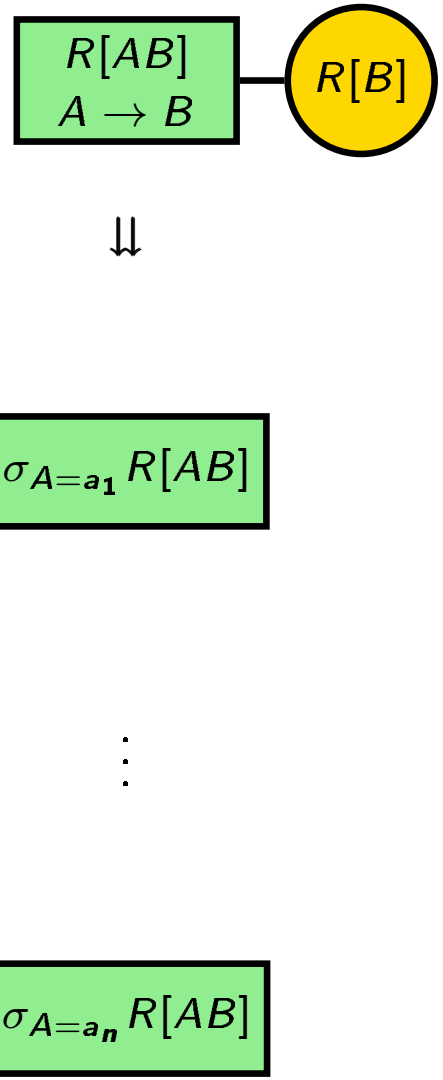
Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.



Data Objects Defined by Horizontal Decomposition

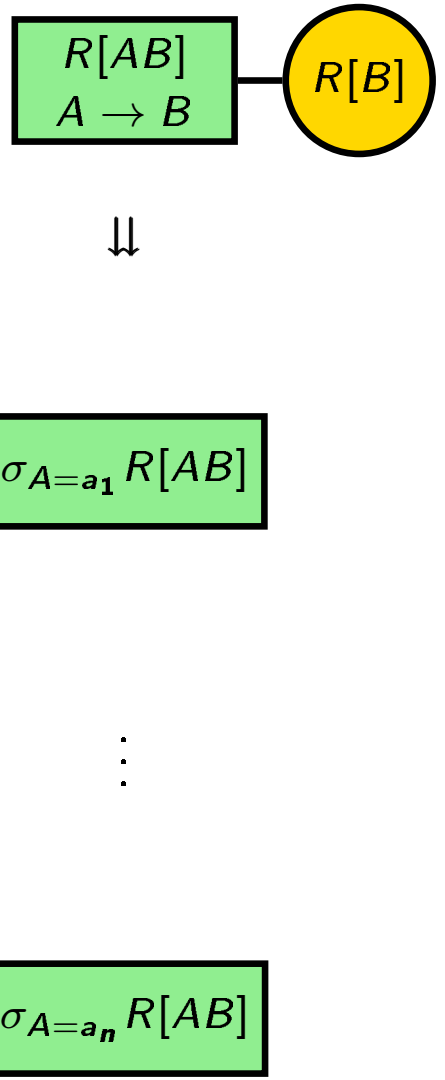
- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

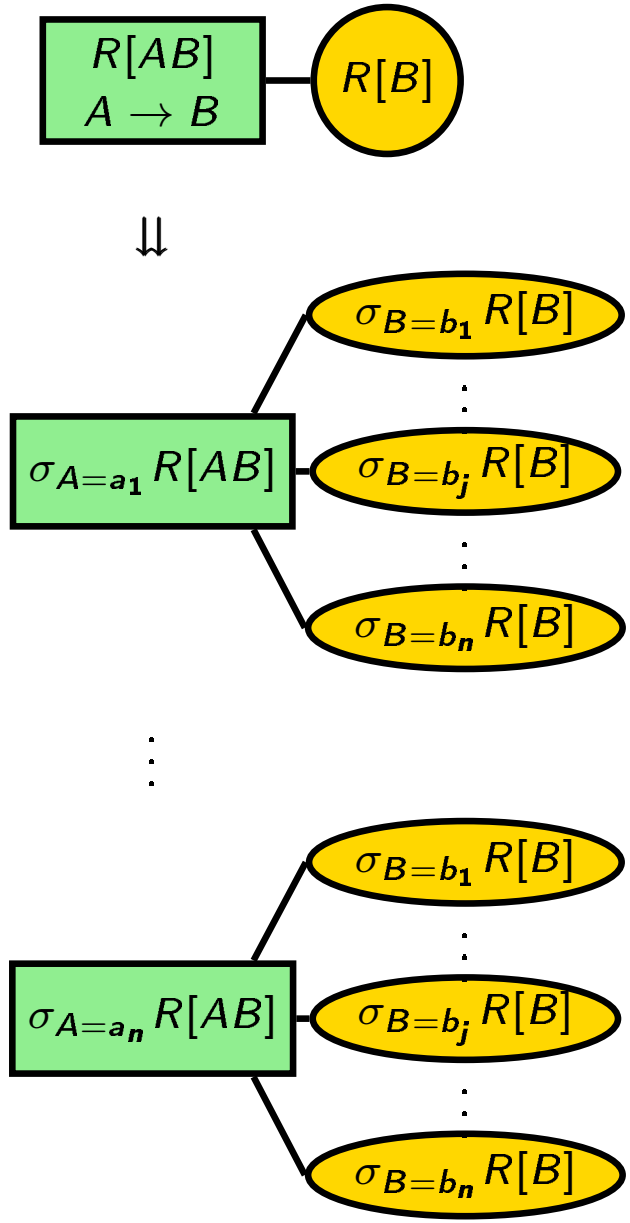
  Assumption: All domains are finite.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

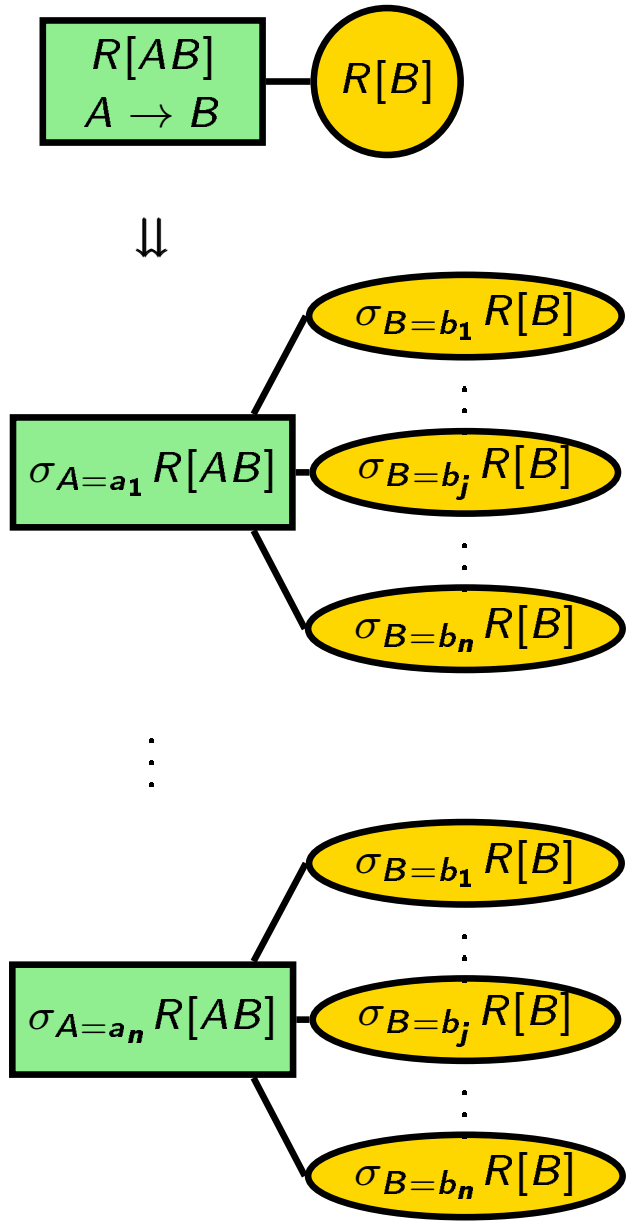
  Assumption: All domains are finite.
- The port attributes are similarly divided.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

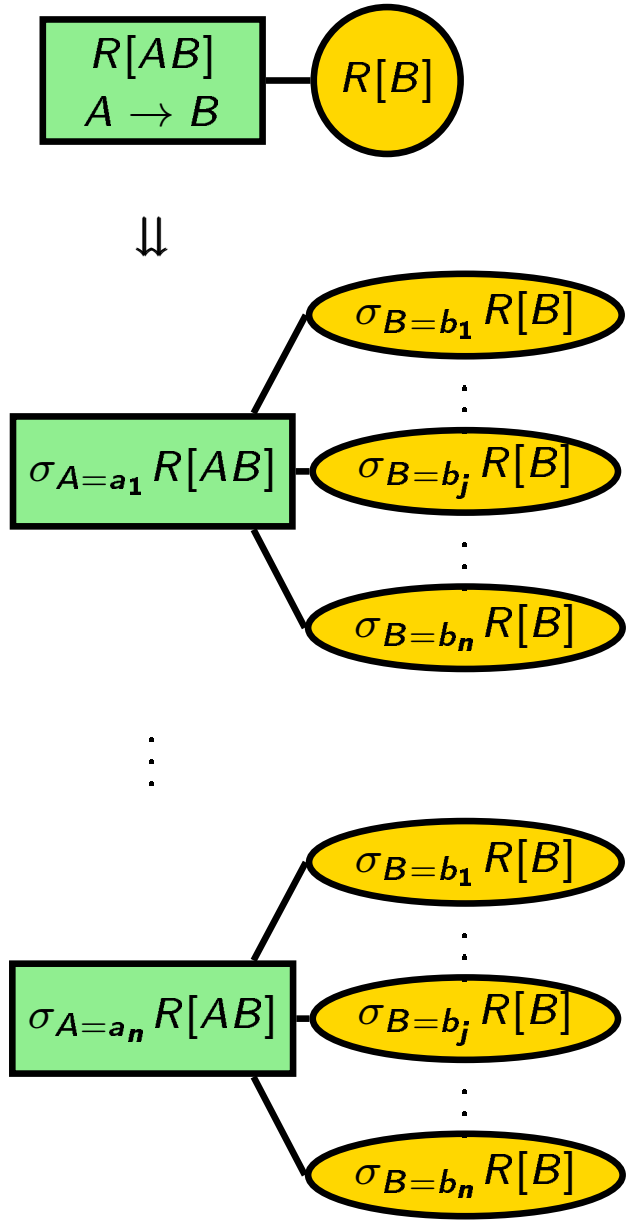
  Assumption: All domains are finite.
- The port attributes are similarly divided.
- Only selection on the key attribute(s) is used.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

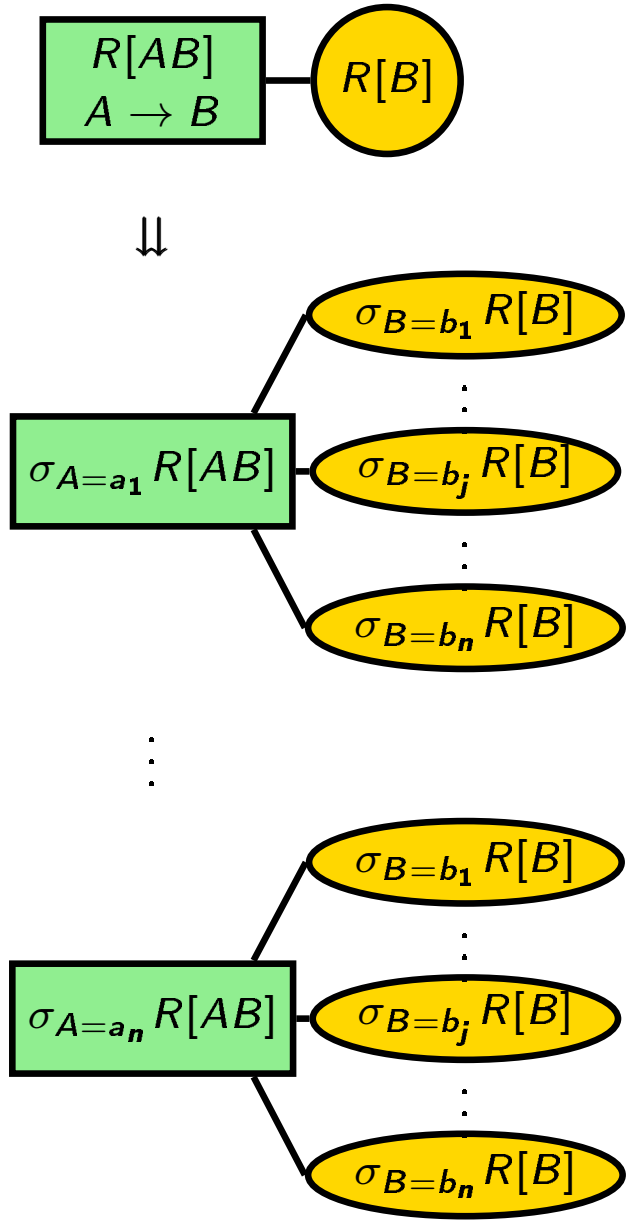
  Assumption: All domains are finite.
- The port attributes are similarly divided.
- Only selection on the key attribute(s) is used.
  - By construction, only one (key) FD is enforced in each object.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

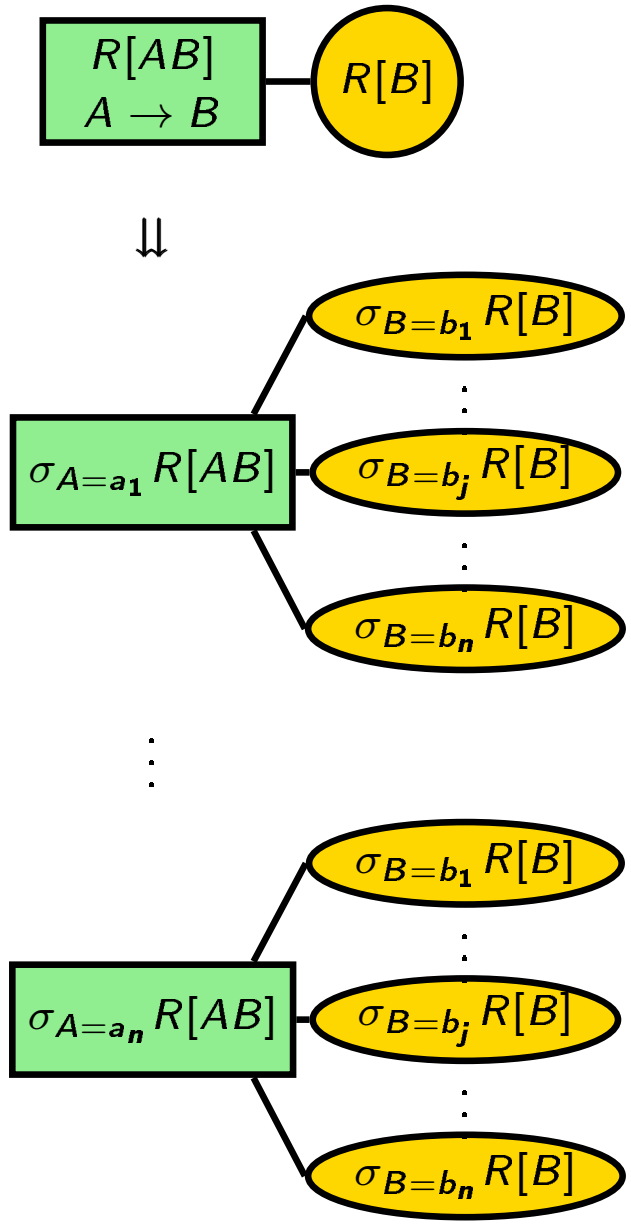
  Assumption: All domains are finite.
- The port attributes are similarly divided.
- Only selection on the key attribute(s) is used.
  - By construction, only one (key) FD is enforced in each object.
- A workable definition of more general select objects is difficult.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

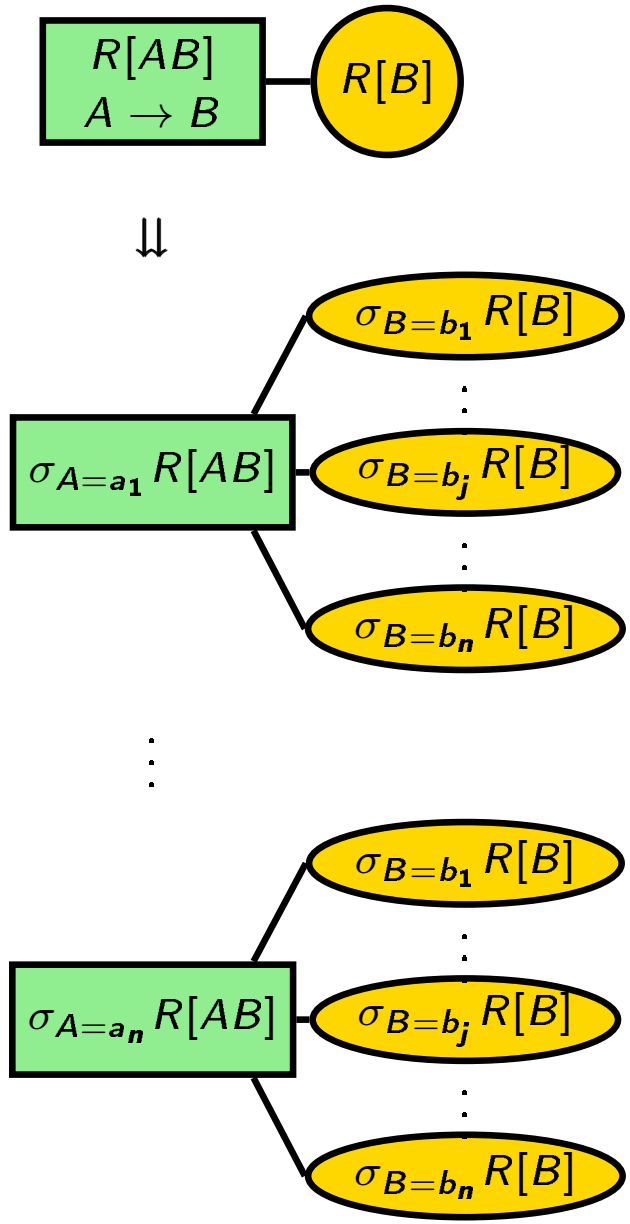
  Assumption: All domains are finite.
- The port attributes are similarly divided.
- Only selection on the key attribute(s) is used.
  - By construction, only one (key) FD is enforced in each object.
- A workable definition of more general select objects is difficult.
  - This is not a shortcoming of this particular approach.



Data Objects Defined by Horizontal Decomposition

- Data objects defined by vertically (via projection) are not adequate by themselves.
- Each "vertical" projection is further divided into "horizontal" selection slices, one for each value of its key attribute(s).

  Assumption: All domains are finite.
- The port attributes are similarly divided.
- Only selection on the key attribute(s) is used.
  - By construction, only one (key) FD is enforced in each object.
- A workable definition of more general select objects is difficult.
  - This is not a shortcoming of this particular approach.
  - Identifying the scope of updates with the key not specified are difficult by nature.



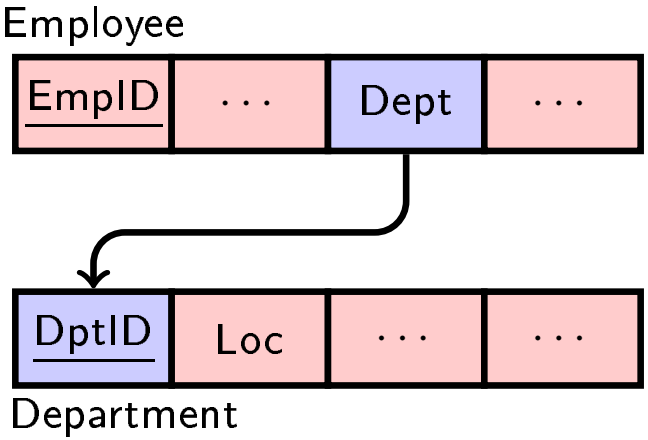
Managing Foreign-Key Dependencies

- Foreign-key dependencies require special attention.

Managing Foreign-Key Dependencies

- Foreign-key dependencies require special attention.
- Consider the Employee-Department example, with the foreign-key constraint:

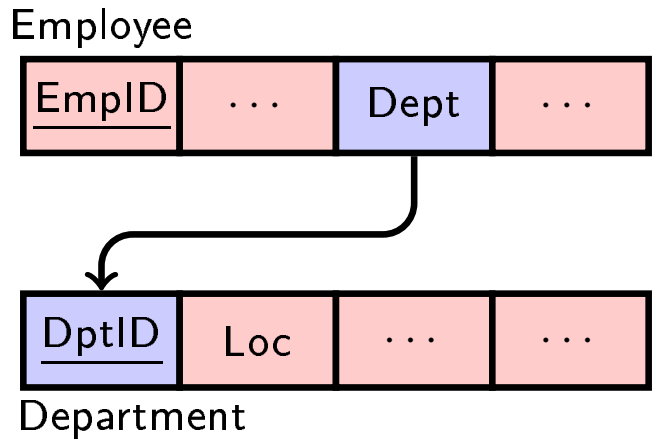
  $\text{Employee}[\text{Dept}] \subseteq \text{Department}[\text{DptID}]$

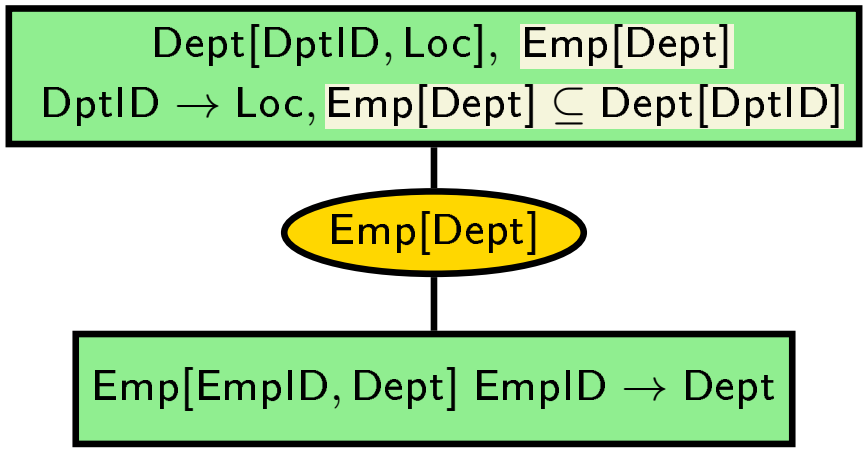


Managing Foreign-Key Dependencies

- Foreign-key dependencies require special attention.
- Consider the Employee-Department example, with the foreign-key constraint:

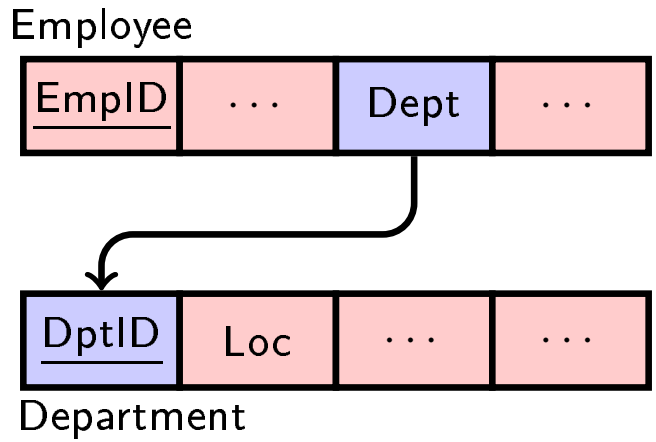
  $\text{Employee}[\text{Dept}] \subseteq \text{Department}[\text{DptID}]$
- To accommodate this FK dependency, the foreign key is included in the data object containing the key and associated attribute of the other relation.

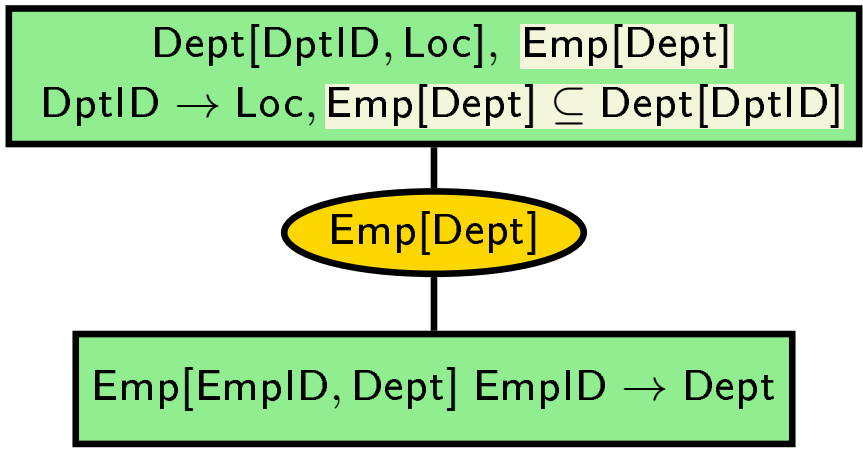




Managing Foreign-Key Dependencies

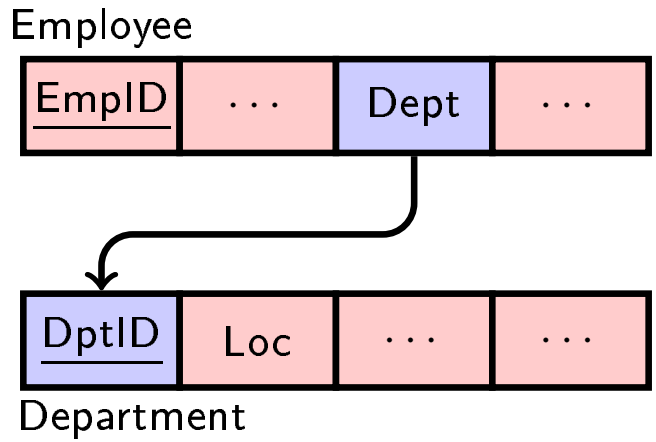
- Foreign-key dependencies require special attention.
- Consider the Employee-Department example, with the foreign-key constraint:
  $\text{Employee[Dept]} \subseteq \text{Department[DptID]}$
- To accommodate this FK dependency, the foreign key is included in the data object containing the key and associated attribute of the other relation.
- Any update to the key of the Department relation also requires a claim/lock on the foreign key in the Employee relation, and conversely.

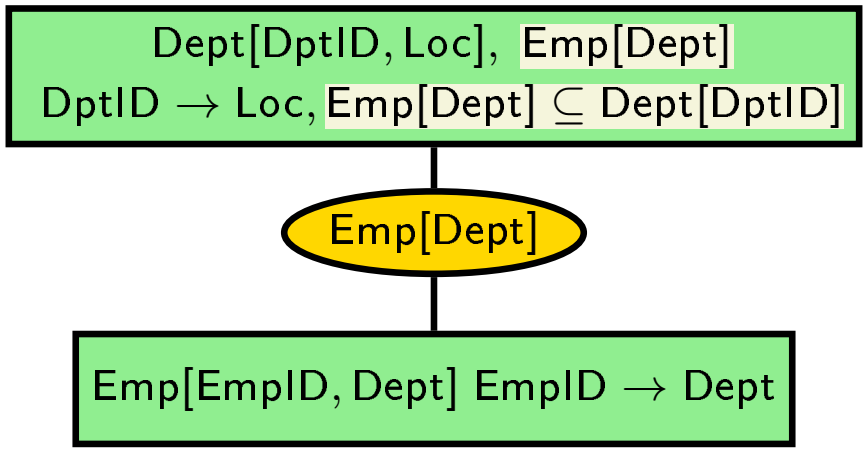




Managing Foreign-Key Dependencies

- Foreign-key dependencies require special attention.
- Consider the Employee-Department example, with the foreign-key constraint:
  $\text{Employee[Dept]} \subseteq \text{Department[DptID]}$
- To accommodate this FK dependency, the foreign key is included in the data object containing the key and associated attribute of the other relation.
- Any update to the key of the Department relation also requires a claim/lock on the foreign key in the Employee relation, and conversely.
- These objects also divide horizontally.





Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

- It is important to note that simply requiring data objects to be comprised of physically disjoint tuples does not guarantee such consistency.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

- It is important to note that simply requiring data objects to be comprised of physically disjoint tuples does not guarantee such consistency.
- Independence is needed to guarantee such consistency.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

- It is important to note that simply requiring data objects to be comprised of physically disjoint tuples does not guarantee such consistency.
- Independence is needed to guarantee such consistency.
- Serializability limits operations on the *same* data object.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

- It is important to note that simply requiring data objects to be comprised of physically disjoint tuples does not guarantee such consistency.
- Independence is needed to guarantee such consistency.
- Serializability limits operations on the *same* data object.
- Indepdendence limits operations on *distinct* data objects.

Independent Transactions

Model: Each transaction T claims a set Claim(T) of data objects.

Independence: Call a set $\mathfrak{T} = \{T_1, T_2, \ldots, T_n\}$ *independent* if for any distinct $T_i, T_j \in \mathfrak{T}$, Claim($T_i$) and Claim($T_j$) have no ports in common.

Observation: If $\mathfrak{T}$ is independent, then its transactions may run with any concurrency whatever

- yielding the same result, guaranteed to be globally consistent provided that the transactions execute locally consistent updates. □

- It is important to note that simply requiring data objects to be comprised of physically disjoint tuples does not guarantee such consistency.
- Independence is needed to guarantee such consistency.
- Serializability limits operations on the *same* data object.
- Indepdendence limits operations on *distinct* data objects.
- Independence is a *complement* to serializability, not an alternative.

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.

Further Directions:

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.

Further Directions:

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.
- The objects may be combined to form larger objects.

Further Directions:

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.
- The objects may be combined to form larger objects.
- Distinct updates which respect this structure are guaranteed to result in a legal database state.

Further Directions:

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.
- The objects may be combined to form larger objects.
- Distinct updates which respect this structure are guaranteed to result in a legal database state.

Further Directions:

- Implementation on top of existing systems.

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.
- The objects may be combined to form larger objects.
- Distinct updates which respect this structure are guaranteed to result in a legal database state.

Further Directions:

- Implementation on top of existing systems.
- Application to the concurrency problems in the context which motivated this research:

Conclusions and Further Directions

Conclusions:

- A model of structured data objects which supports a notion of strong independence has been developed.
- A key feature of the model is that each object has a writable area and a read-only area.
- The objects may be combined to form larger objects.
- Distinct updates which respect this structure are guaranteed to result in a legal database state.

Further Directions:

- Implementation on top of existing systems.
- Application to the concurrency problems in the context which motivated this research:

  Cooperative update: Updates which require the cooperation of many actors/views.