Εξέλιξη Βάσεων Δεδομένων και Συντήρηση Εξαρτώμενων Εφαρμογών μέσω Επανεγγραφής Ερωτήσεων

Πέτρος Μανούσης

University of Ioannina, Greece

Problem definition

- Changes on a database schema may cause inconsistency in applications that use that database, is there a way to regulate that?
- If there is such a way of accepting or rejecting a change, could we satisfy it by rewriting the database schema?
- If there are conflicts between the applications on acceptance or rejection of a change, is there a possibility of satisfying both?

Our approach

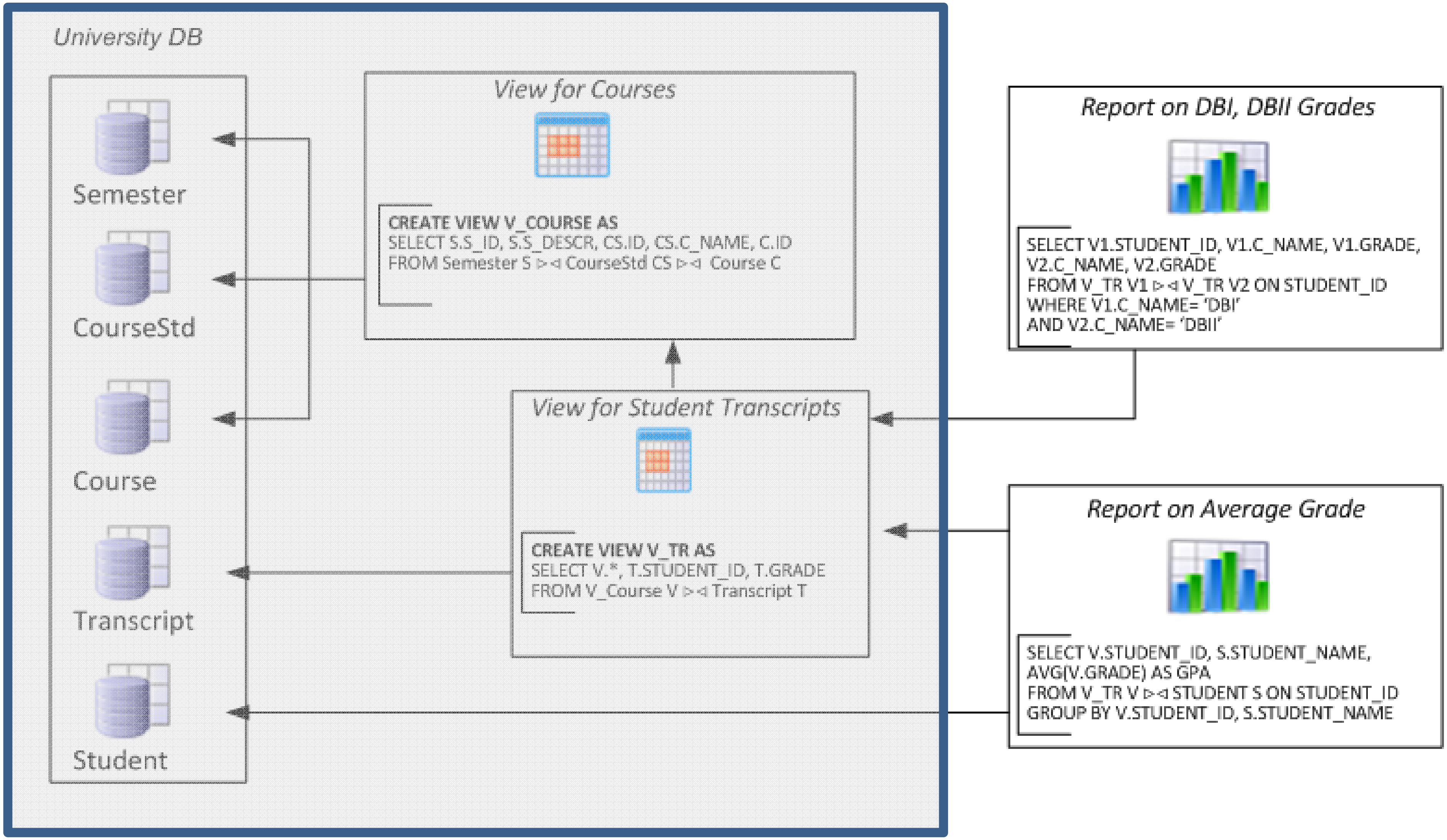
We are going to present you a method that contains 3 steps:

1. Status Determination
2. Path Check
3. Rewrite

Background
Message propagation
Path check
Rewriting
Experiments and Results

Background

Database and queries



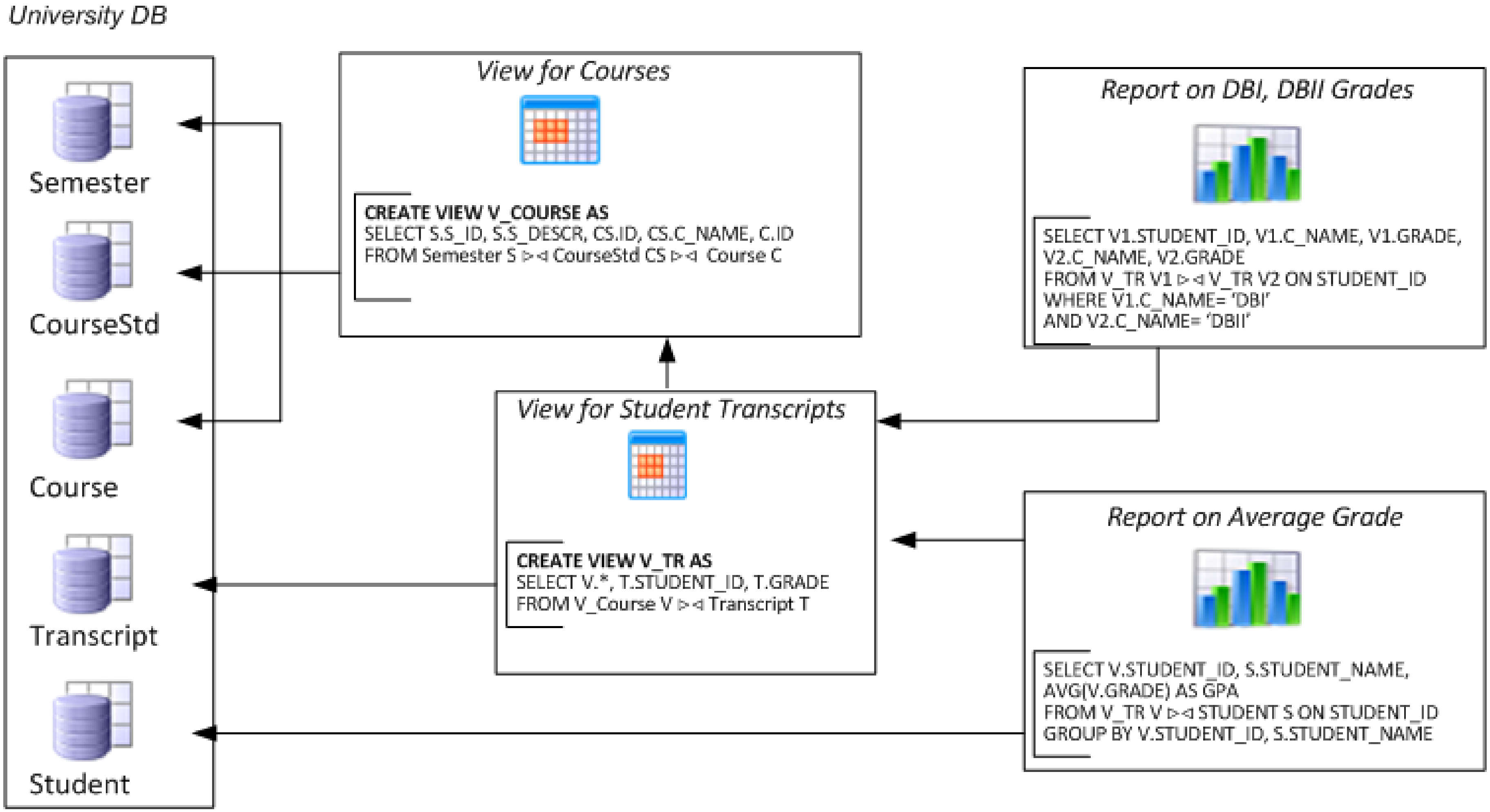
```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⊳⊲ CourseStd CS ⊳⊲  Course C
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⊳⊲ Transcript T
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⊳⊲ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= ‘DBI’
AND V2.C_NAME= ‘DBII’
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⊳⊲ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Data-centric Ecosystem



University DB

Semester

CourseStd

Course

Transcript

Student

View for Courses

```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ▷◁ CourseStd CS ▷◁  Course C
```

View for Student Transcripts

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ▷◁ Transcript T
```

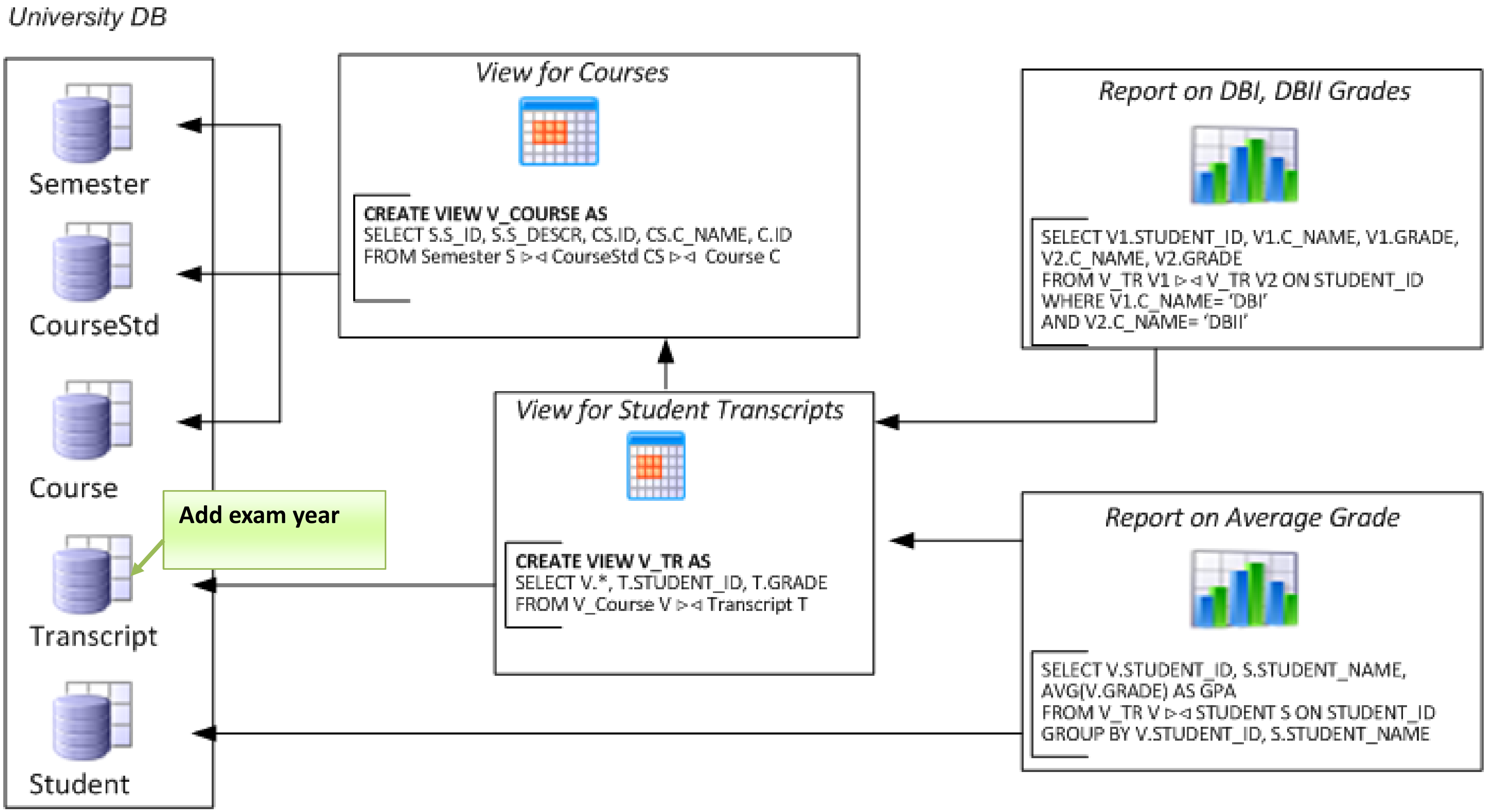
Report on DBI, DBII Grades

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ▷◁ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

Report on Average Grade

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ▷◁ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Evolving data-centric ecosystem



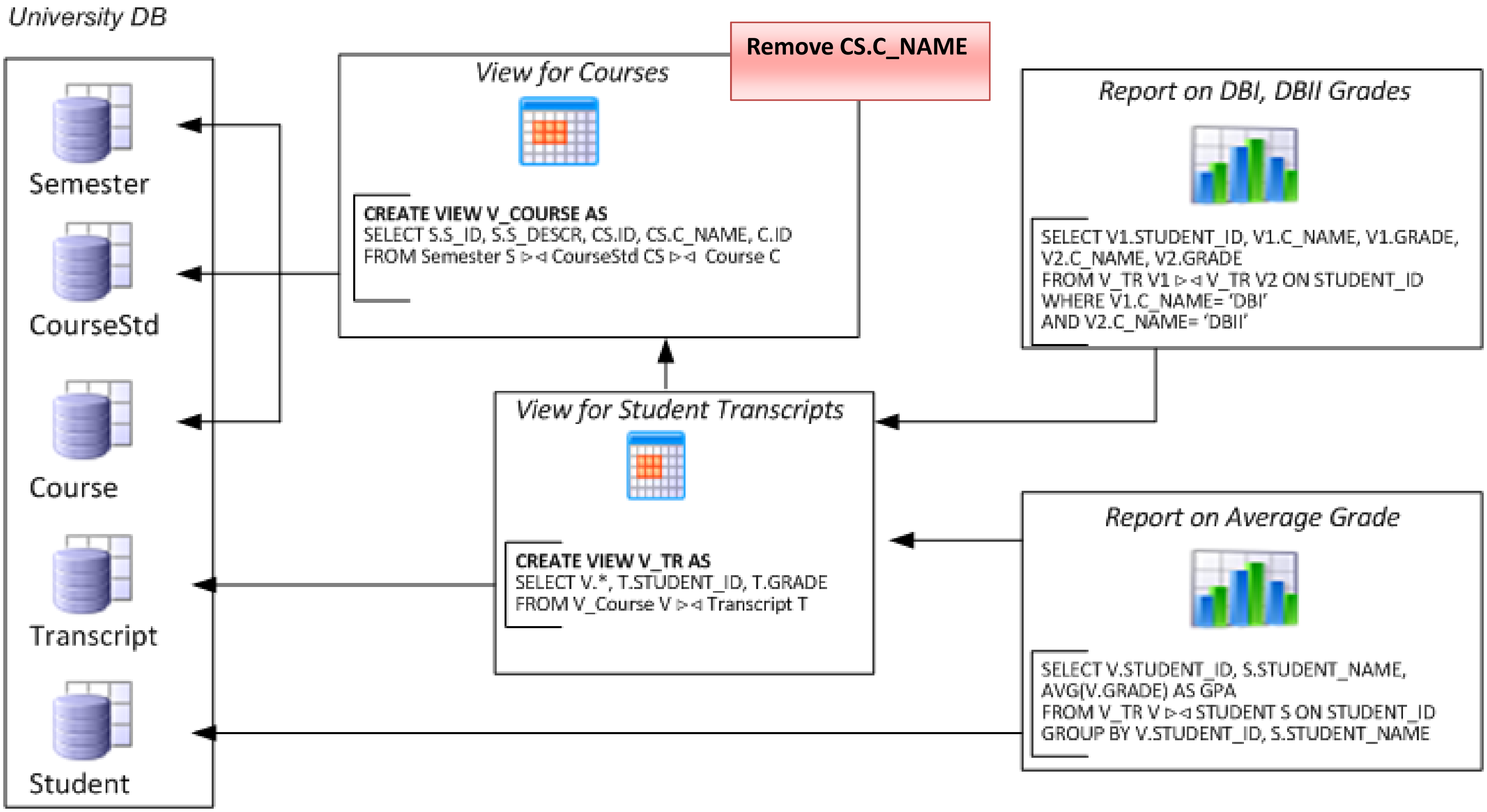
```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⊳⊲ CourseStd CS ⊳⊲  Course C
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⊳⊲ Transcript T
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⊳⊲ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⊳⊲ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Evolving data-centric ecosystem



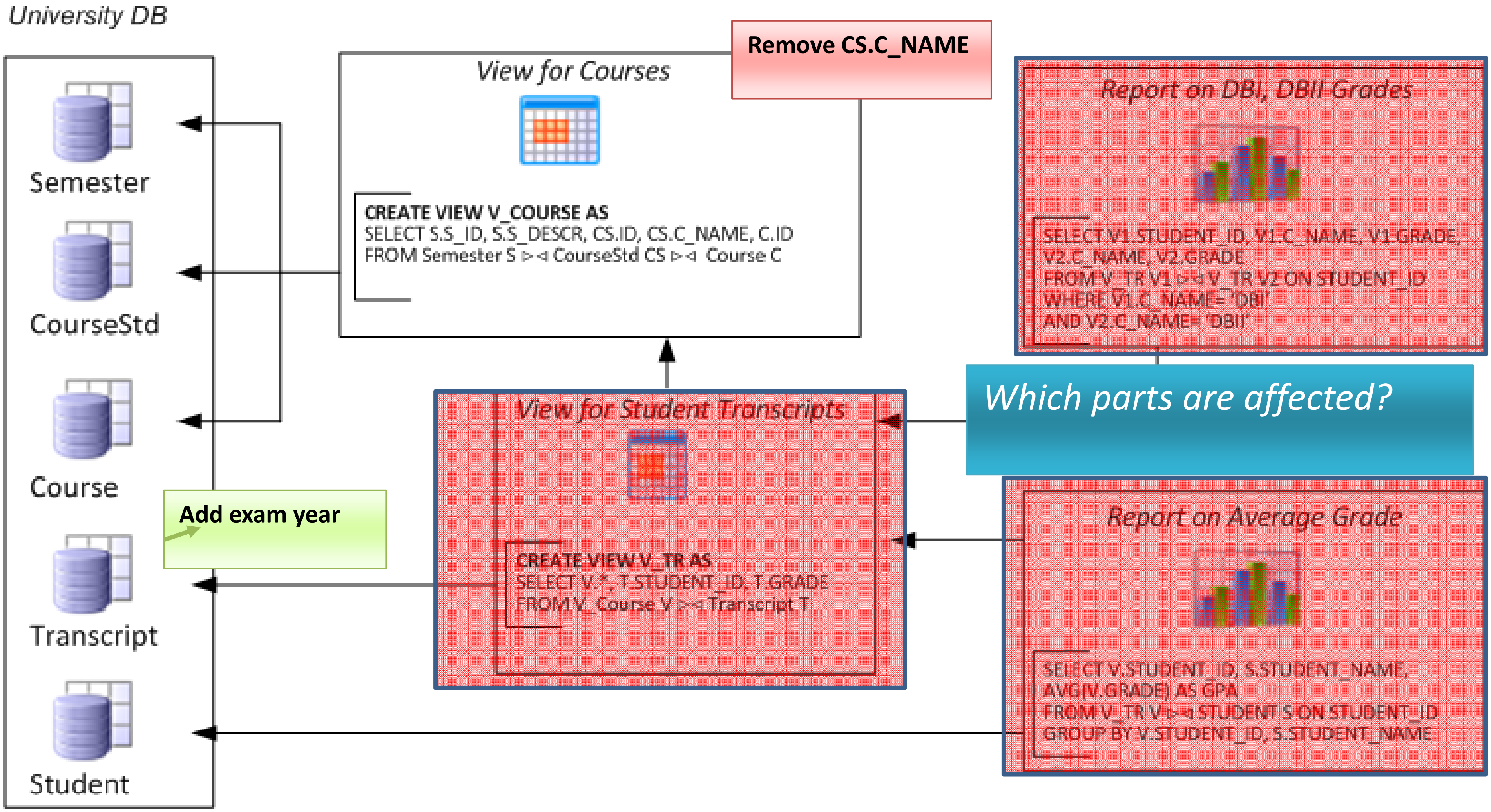
```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⋈ CourseStd CS ⋈  Course C
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⋈ Transcript T
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⋈ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⋈ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Evolving data-centric ecosystem



```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⊳⊲ CourseStd CS ⊳⊲  Course C
```

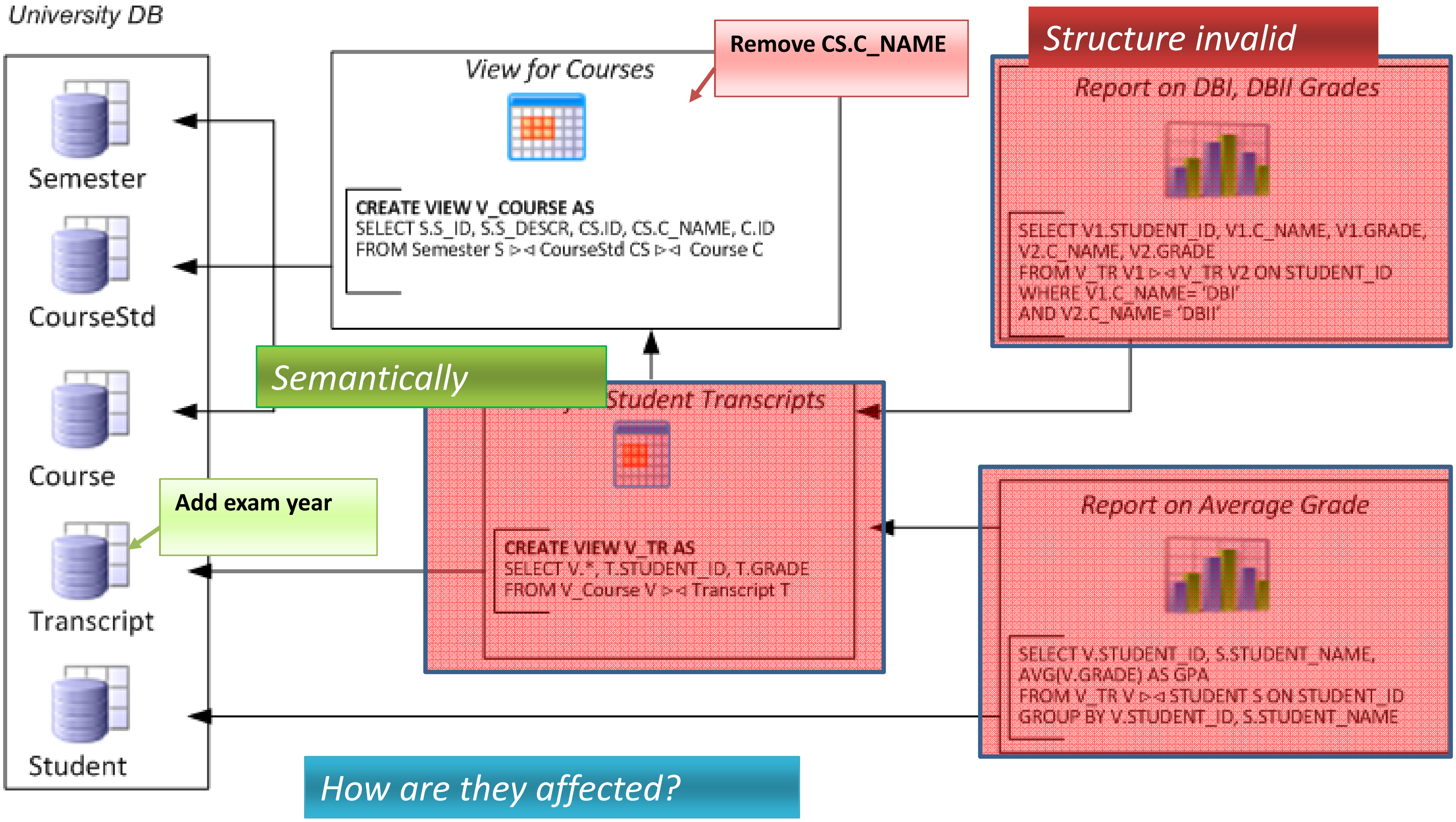
```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⊳⊲ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⊳⊲ Transcript T
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⊳⊲ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Which parts are affected?

Evolving data-centric ecosystem



```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ▷◁ CourseStd CS ▷◁  Course C
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ▷◁ Transcript T
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ▷◁ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ▷◁ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Remove CS.C_NAME

Structure invalid

Semantically

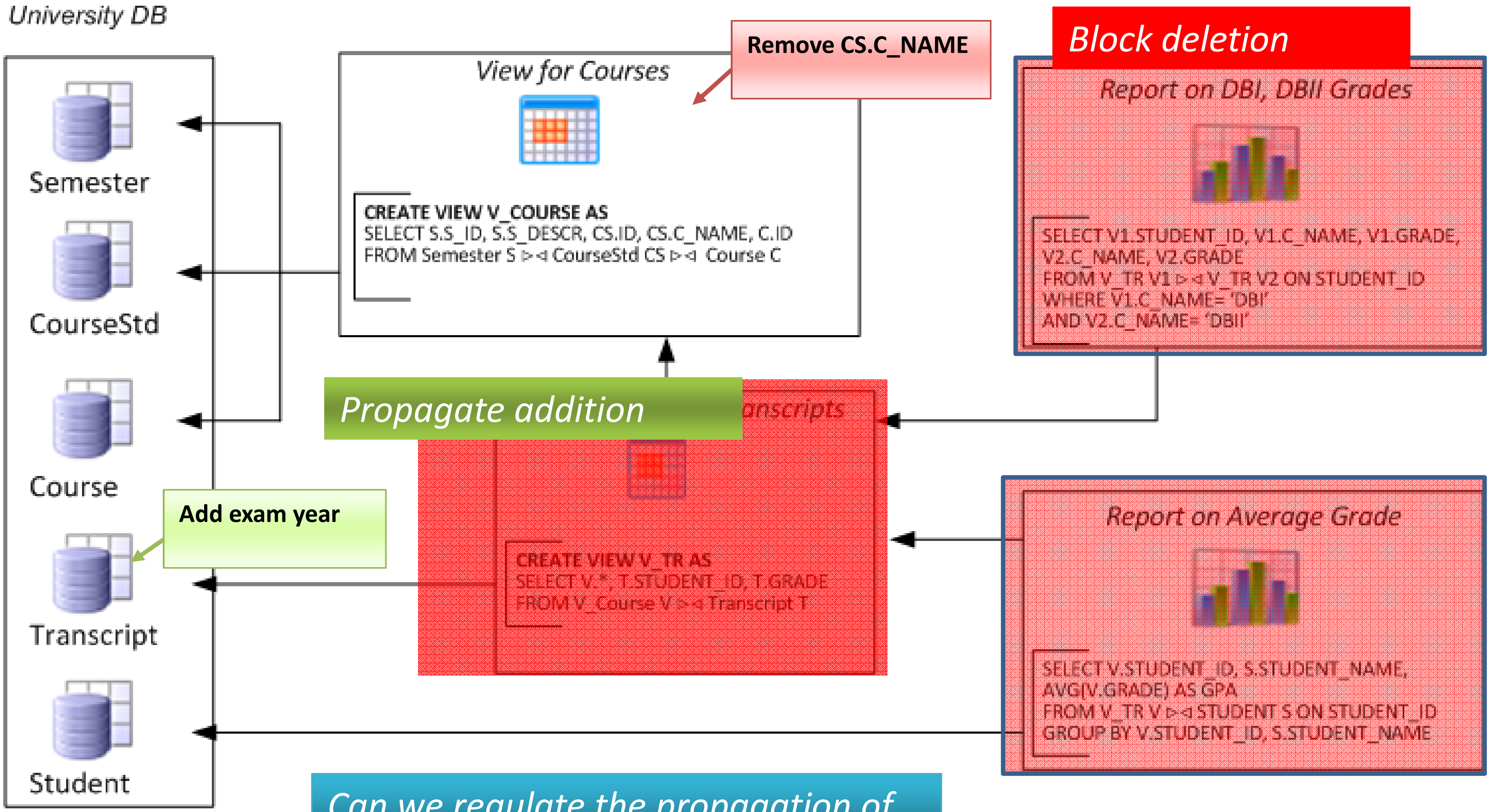
Add exam year

How are they affected?

Problem definition

- **Changes on a database schema may cause inconsistency in applications that use that database, is there a way to regulate that?**
- *If there is such a way of accepting or rejecting a change, could we satisfy it by rewriting the database schema?*
- *If there are conflicts between the applications on acceptance or rejection of a change, is there a possibility of satisfying both?*

Policy-driven evolution



Remove CS.C_NAME

Block deletion

Propagate addition

Add exam year

```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⊳⊲ CourseStd CS ⊳⊲  Course C
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⊳⊲ Transcript T
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⊳⊲ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⊳⊲ STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

Can we regulate the propagation of the events?

Background

Message propagation

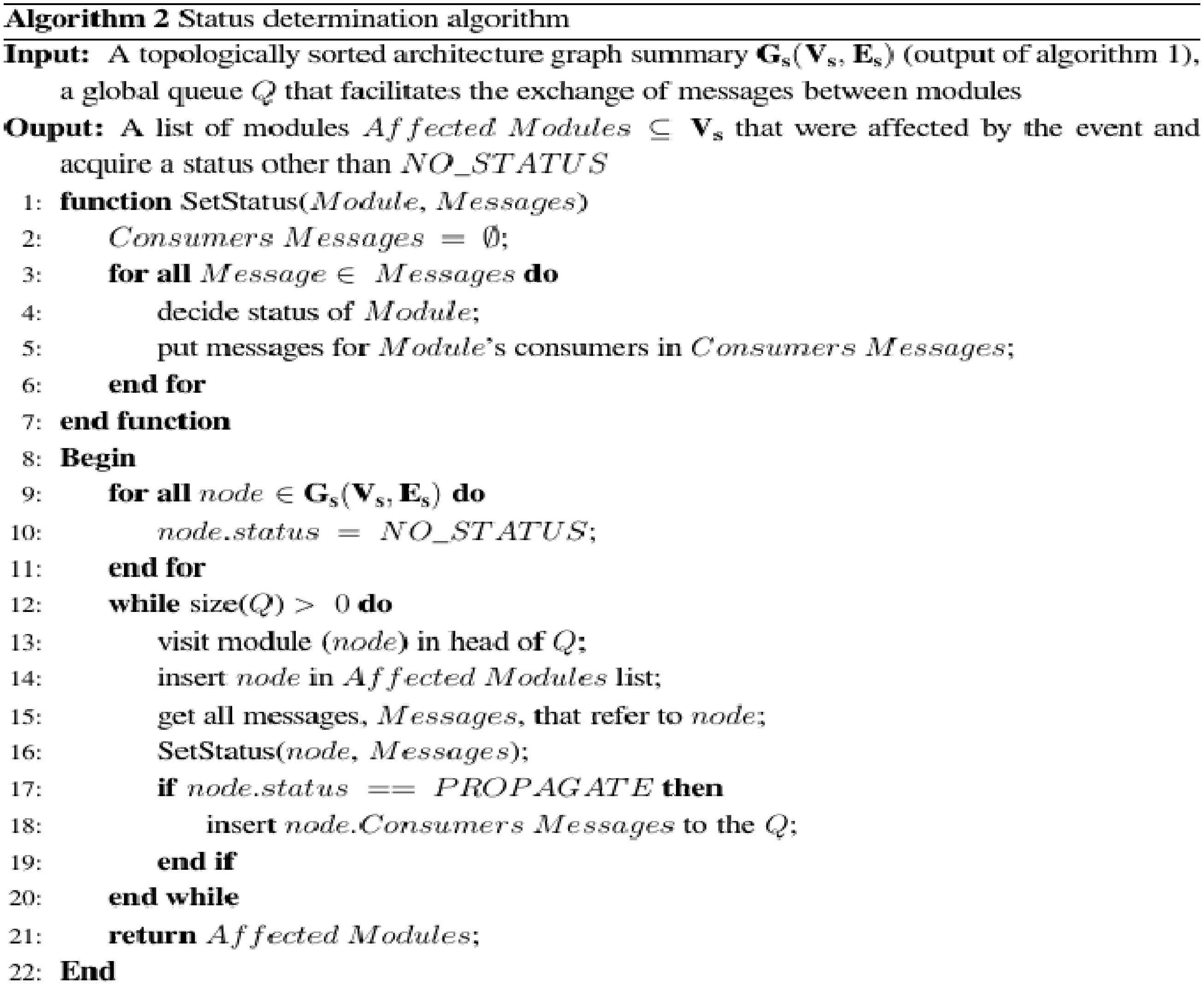
Path check

Rewriting

Experiments and Results

Message propagation

Message propagation algorithm



Algorithm 2 Status determination algorithm

Input: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ (output of algorithm 1), a global queue Q that facilitates the exchange of messages between modules

Ouput: A list of modules $Affected\ Modules \subseteq \mathbf{V_s}$ that were affected by the event and acquire a status other than NO_STATUS

```
function SetStatus($Module$, $Messages$)
    $Consumers\ Messages = \emptyset$;
    for all $Message \in Messages$ do
        decide status of $Module$;
        put messages for $Module$'s consumers in $Consumers\ Messages$;
    end for
end function
Begin
    for all $node \in \mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ do
        $node.status = NO\_STATUS$;
    end for
    while size($Q$) $>$ 0 do
        visit module ($node$) in head of $Q$;
        insert $node$ in $Affected\ Modules$ list;
        get all messages, $Messages$, that refer to $node$;
        SetStatus($node$, $Messages$);
        if $node.status == PROPAGATE$ then
            insert $node.Consumers\ Messages$ to the $Q$;
        end if
    end while
    return $Affected\ Modules$;
End
```

Our approach

•Model data-centric ecosystems with **Architecture Graphs**

•Mechanism for propagating evolution events on the graph, based on

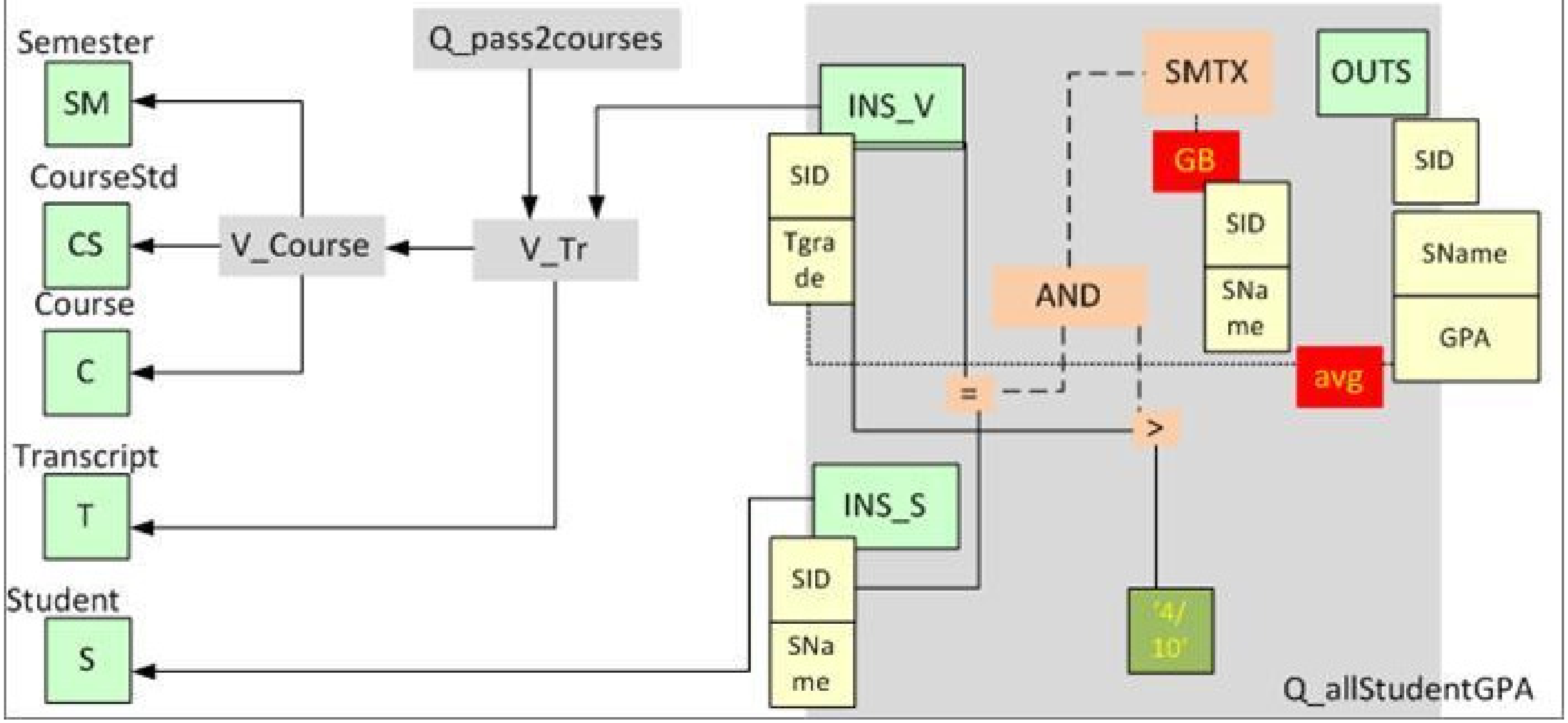
- Graph structure & semantics
- Types of evolution events
- Policies that regulate the message flooding

•Guarantee the termination of the events propagation

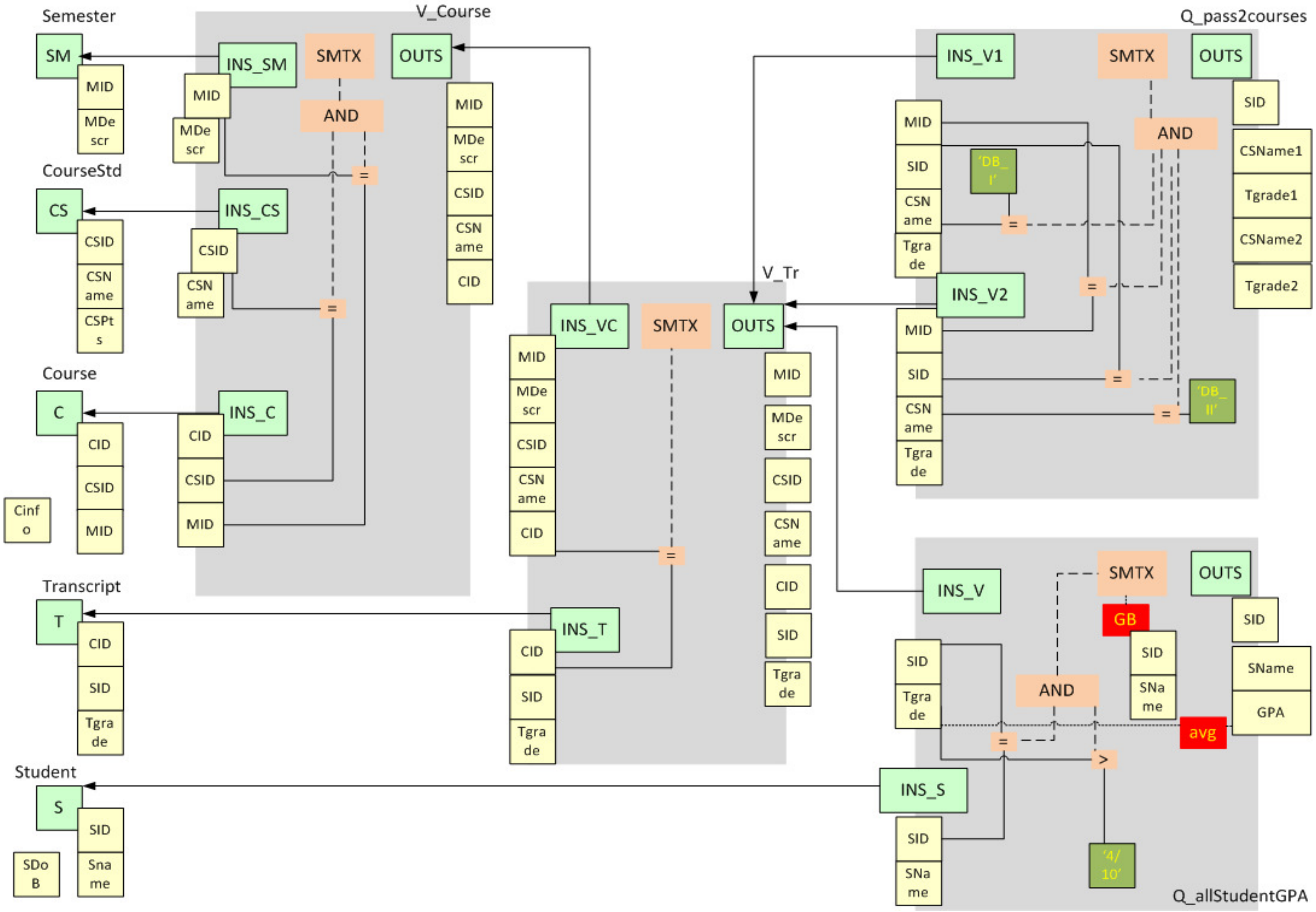
Architecture Graph

```
SELECT  V.STUDENT_ID, S.STUDENT_NAME, AVG(V.TGRADE) AS GPA
FROM V_TR V ▷◁ STUDENT S ON STUDENT_ID
WHERE V.TGRADE > 4 / 10
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

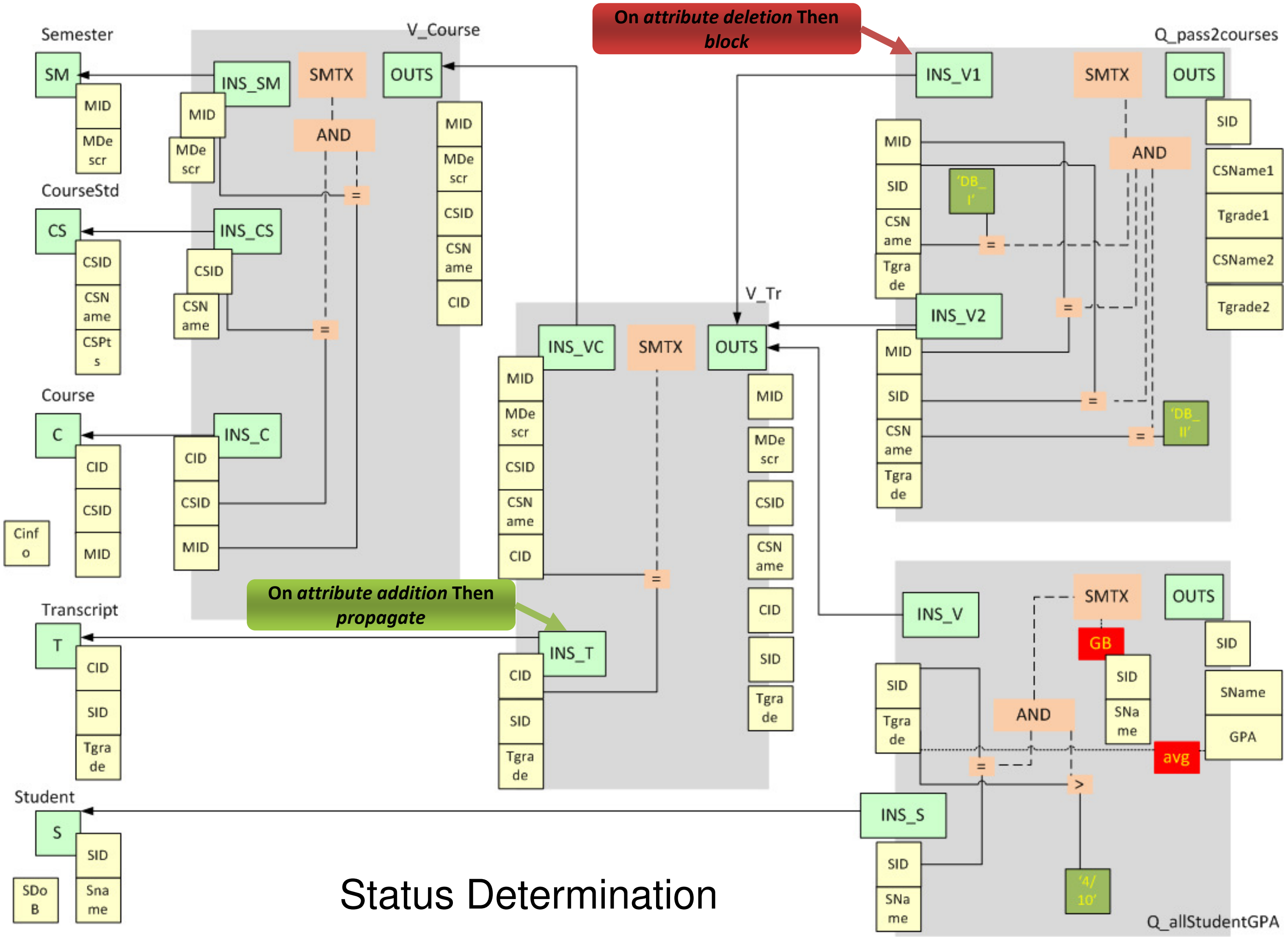
Modules and Module Encapsulation



University E/S Architecture Graph



Annotation with Policies


On *attribute deletion* Then *block*

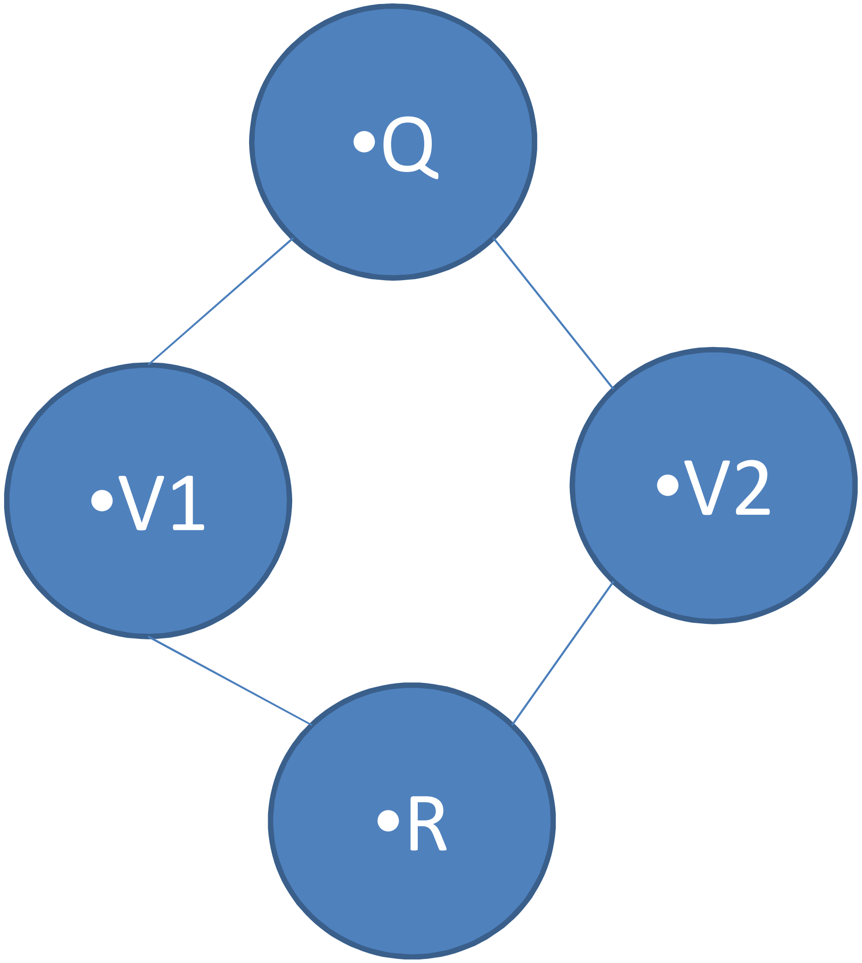
On *attribute addition* Then *propagate*

Status Determination

Implementation problems

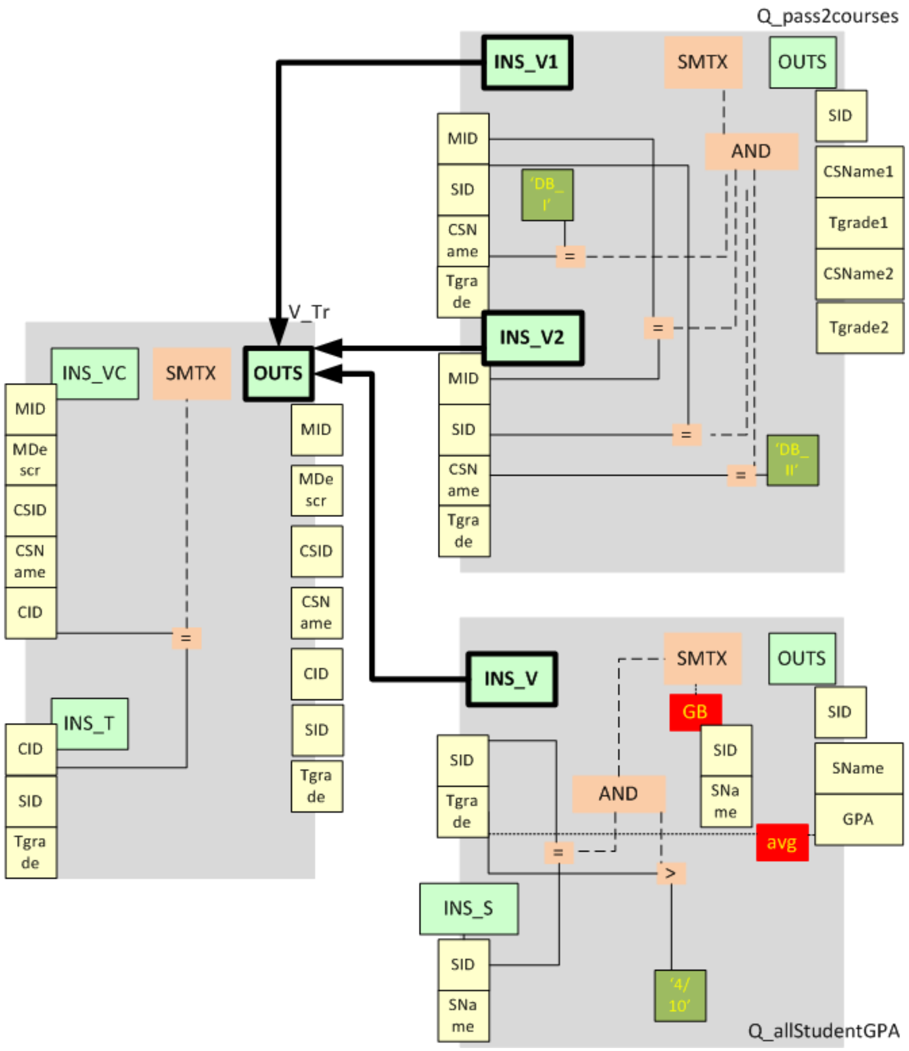
•How do we **guarantee** that when a change occurs at the source nodes of the AG, this is **correctly** propagated to the end nodes of the graph?

- We notify exactly the nodes that should be notified
- The status of a node is determined independently of how messages arrive at the node
- Without infinite looping



Propagation mechanism

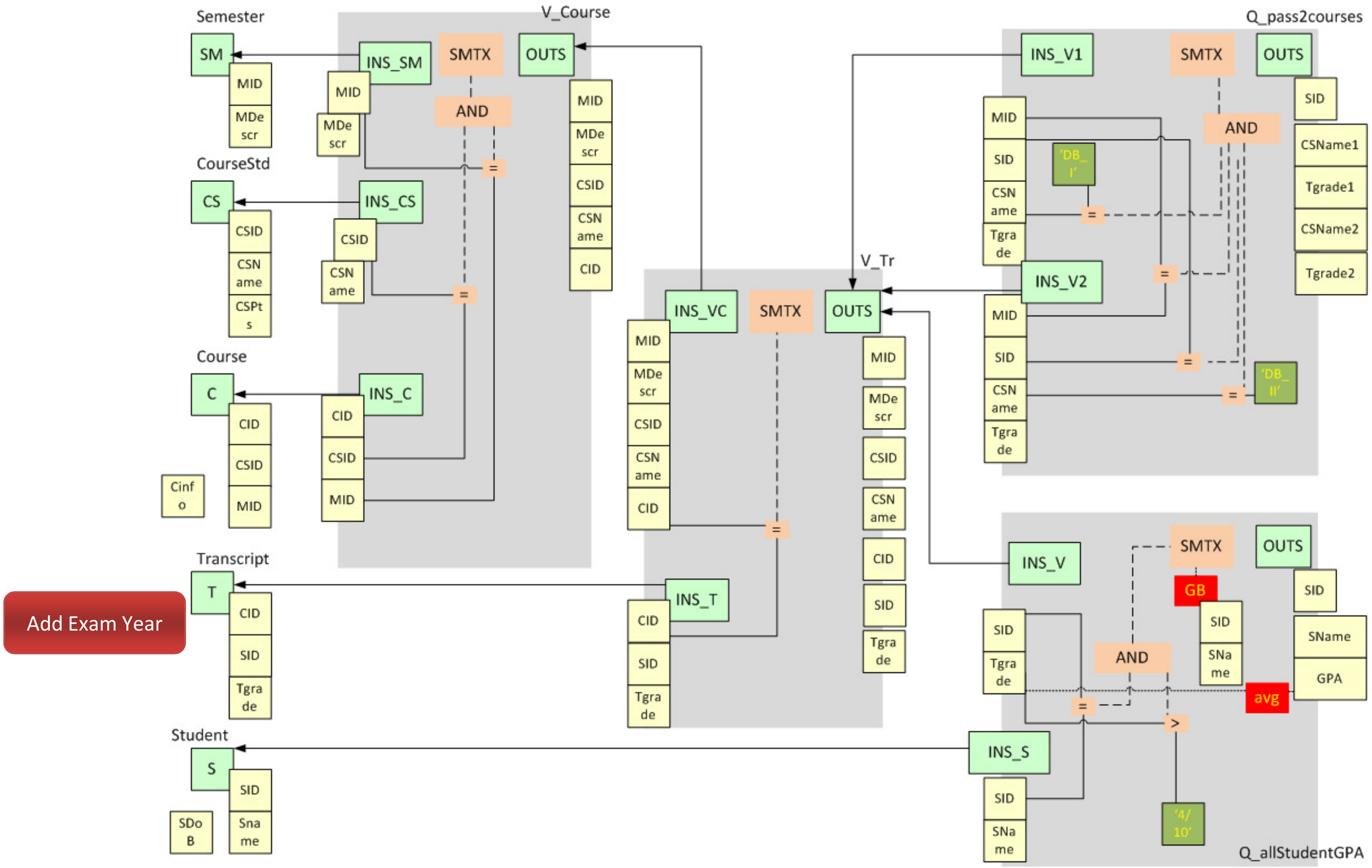
- Modules communicate with each other via a single means: the schema of a provider module notifies the input schema of a consumer module when this is necessary
- Propagation
    - At the module level
    - Intra-module level



At the module level

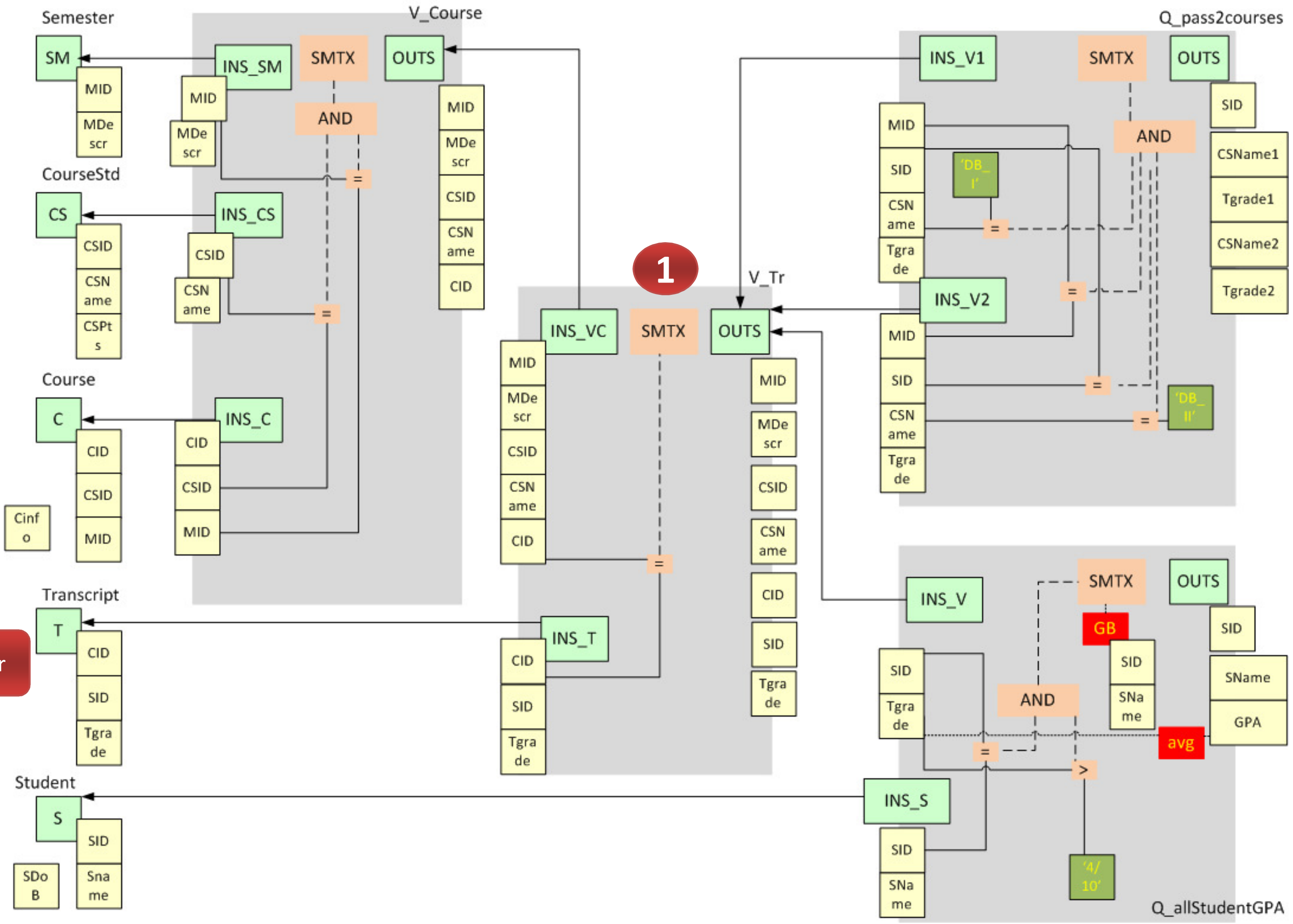
1.Topologically sort the graph
2.Visit affected modules with its topological order and process its incoming messages for it.
3.Process locally the incoming messages

Module Level Propagation



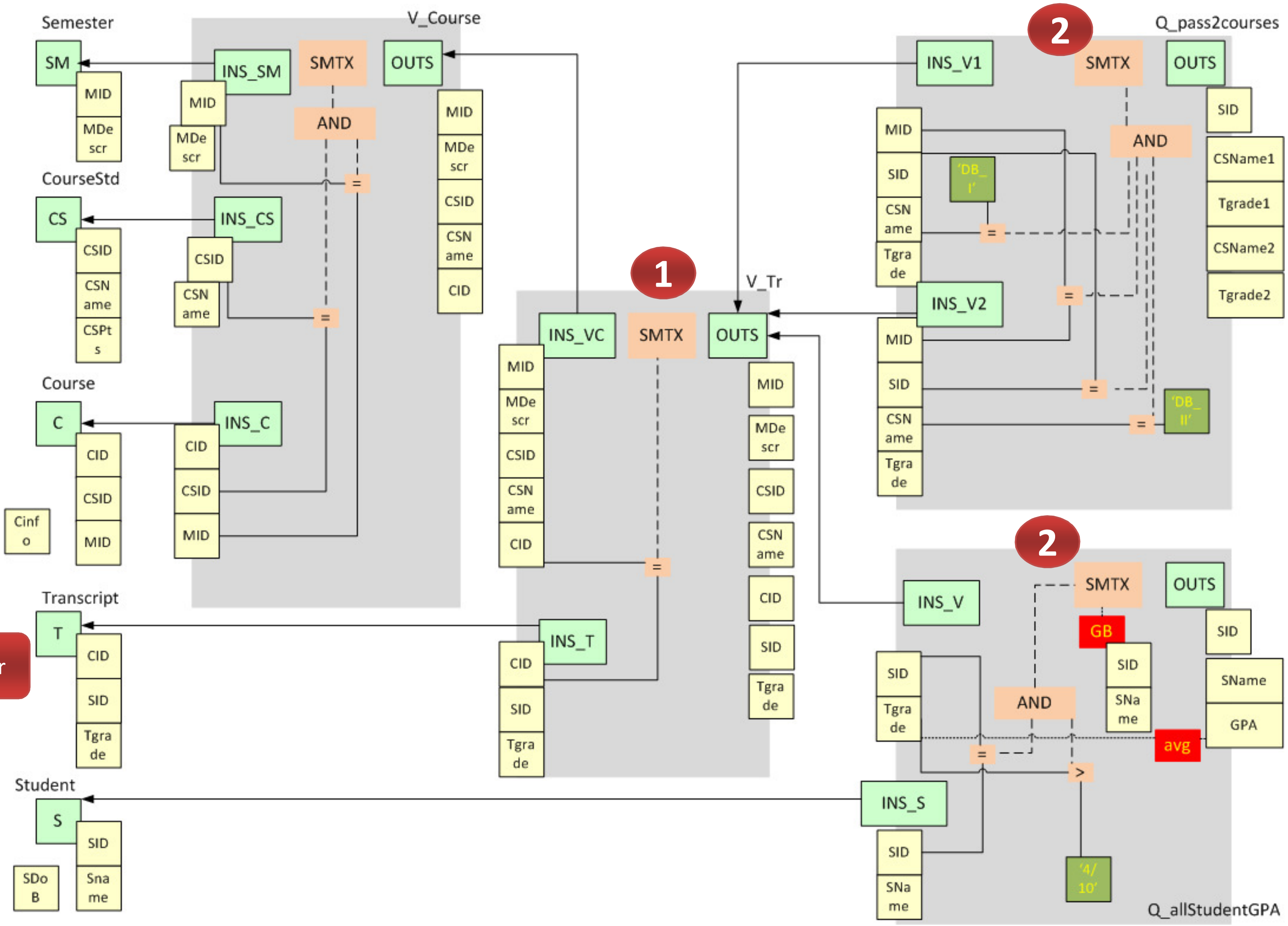
Add Exam Year

Module Level Propagation



Add Exam Year

Module Level Propagation



Add Exam Year

Message initiation

The Message is processed in one of the following schemata:

- Output schema and its attributes if the user wants to change the output of a module (add / delete / rename attribute).
- Semantics schema if the user wants to change the semantics tree of the module.

Finally, Messages are produced within the module for its consumers, containing the necessary parameters for its consumers.

Within each module

A Message arrives at a module, through the propagation mechanism, these steps describe module's way of handling:

1) Input schema and its attributes if applicable, are probed.

2) If the parameter of the Message has any kind of connection with the semantics tree, then the Semantics schema is probed.

3) Likewise if the parameter of the Message has any kind of connection with the output schema, then the Output schema and its attributes (if applicable) is probed.

Finally, Messages are produced within the module for its consumers.s

Theoretical Guarantees

- At the inter-module level
  - *Theorem 1* (*termination*). The message propagation at the inter-module level terminates.
  - *Theorem 2 (unique status)*. Each module in the graph will assume a unique status once the message propagation terminates.
  - *Theorem 3 (correctness)*. Messages are correctly propagated to the modules of the graph
- At the intra-module level
  - *Theorem 4 (termination and correctness)*. The message propagation at the intramodule level terminates and each node assumes a status.

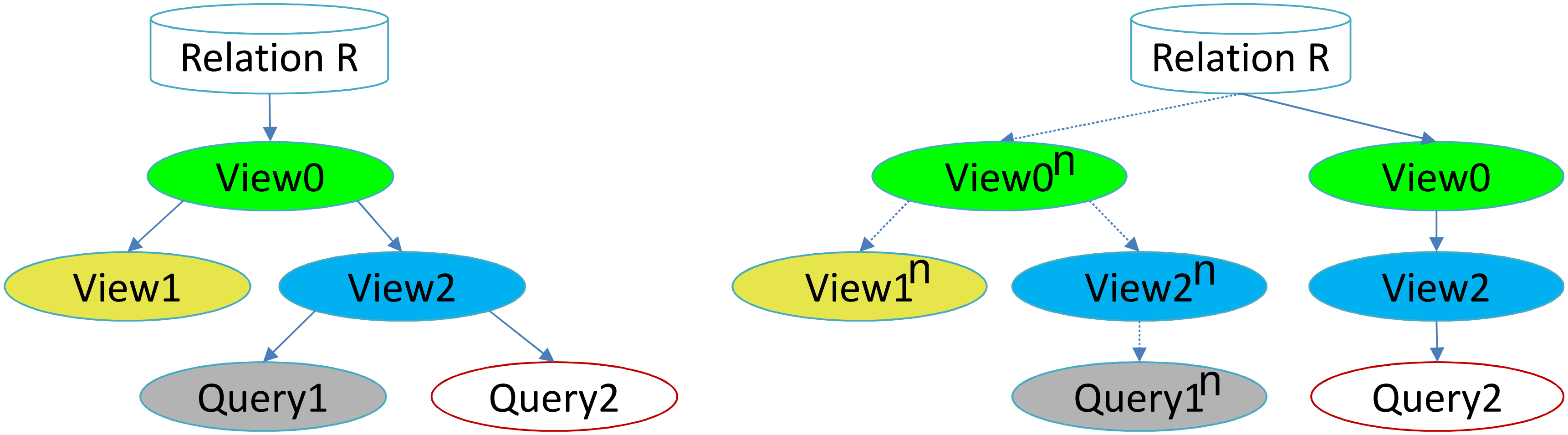
Background
Message propagation
Path check
Rewriting
Experiments and Results

Path check

Problem definition

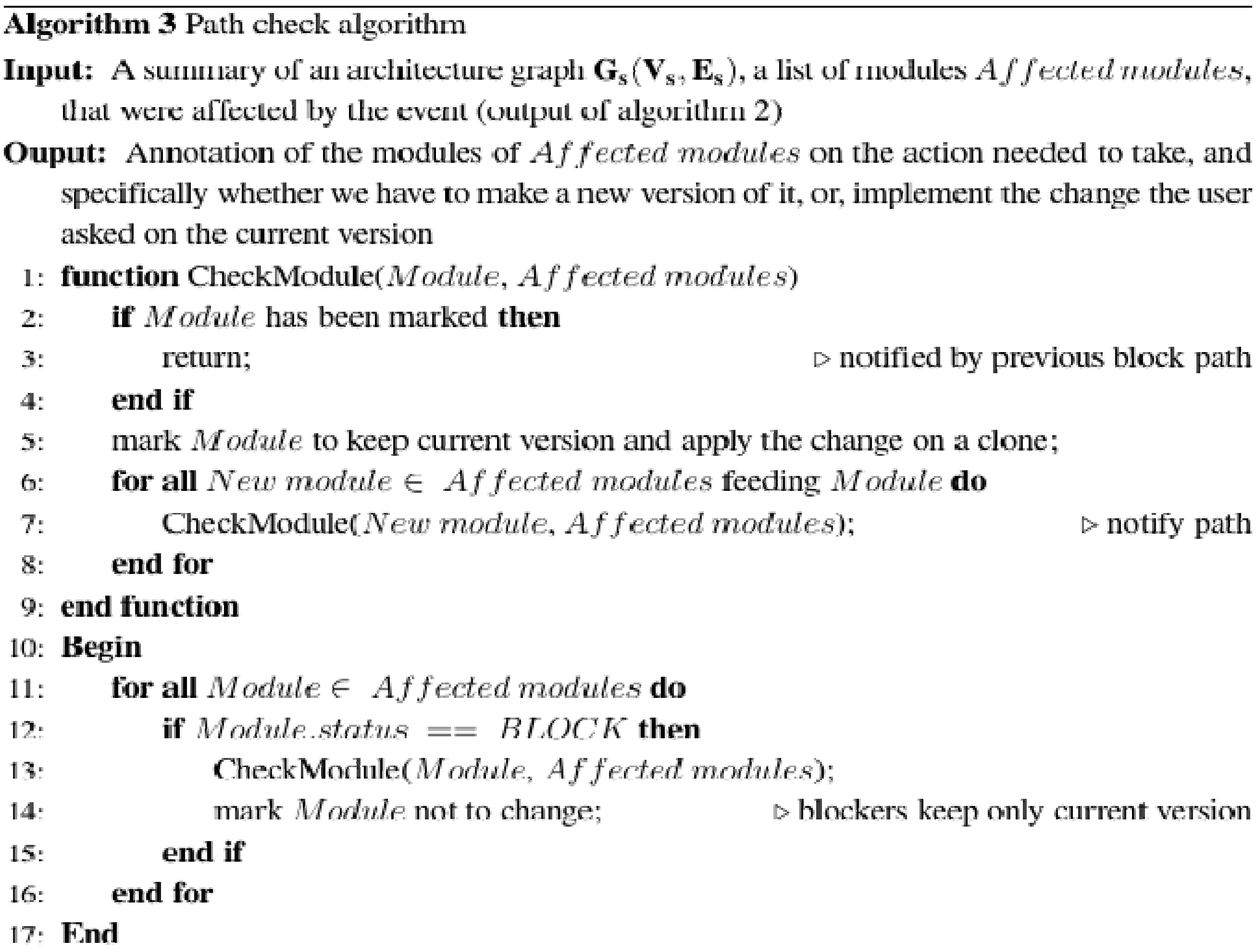
- *Changes on a database schema may cause inconsistency in applications that use that database, is there a way to regulate that?*
- *If there is such a way of accepting or rejecting a change, could we satisfy it by rewriting the database schema?*
- **If there are conflicts between the applications on acceptance or rejection of a change, is there a possibility of satisfying both?**

Rewriting



- View0 initiates a change.
- Query2 rejects the change.
- Query1 accepts the change.

Path Check algorithm



Algorithm 3 Path check algorithm

Input: A summary of an architecture graph $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a list of modules $Affected\ modules$, that were affected by the event (output of algorithm 2)

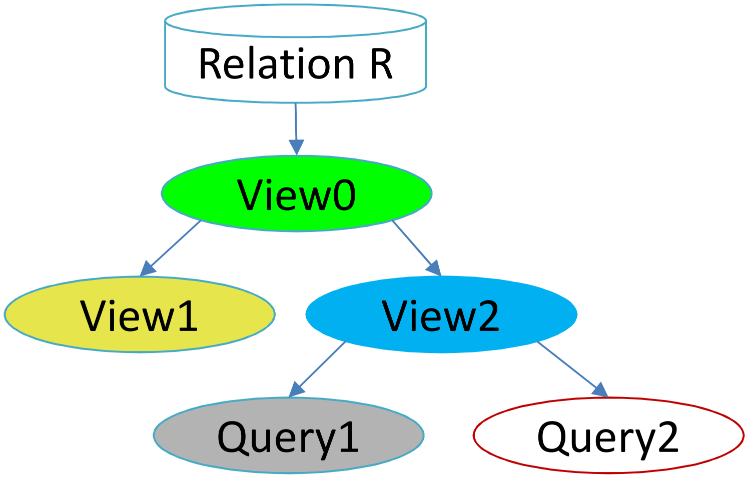
Ouput: Annotation of the modules of $Affected\ modules$ on the action needed to take, and specifically whether we have to make a new version of it, or, implement the change the user asked on the current version

```
function CheckModule($Module$, $Affected\ modules$)
    if $Module$ has been marked then
        return;                                          ▷ notified by previous block path
    end if
    mark $Module$ to keep current version and apply the change on a clone;
    for all $New\ module \in Affected\ modules$ feeding $Module$ do
        CheckModule($New\ module$, $Affected\ modules$);                ▷ notify path
    end for
end function
Begin
    for all $Module \in Affected\ modules$ do
        if $Module.status == BLOCK$ then
            CheckModule($Module$, $Affected\ modules$);
            mark $Module$ not to change;              ▷ blockers keep only current version
        end if
    end for
End
```

Path Check

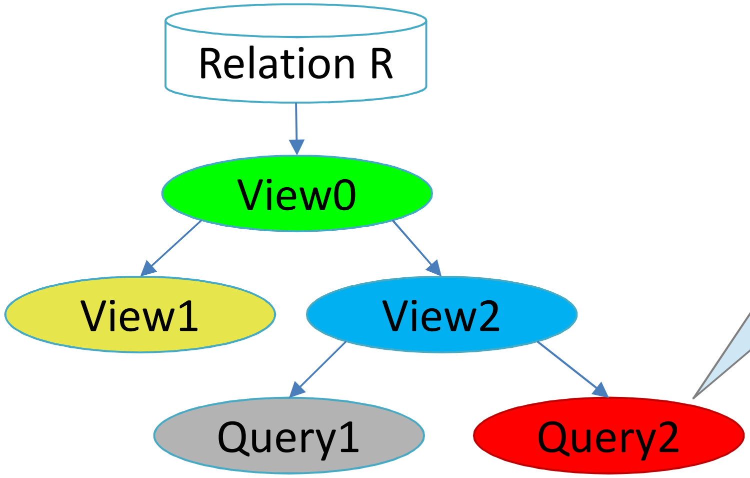
- If there exists any Block Module we travel in reverse the Architecture Graph from blocker node to initiator of change
- In each step we inform the Module to keep current version and produce a new one adapting to the change
- We inform the blocker node that it should not change at all.

Path Check

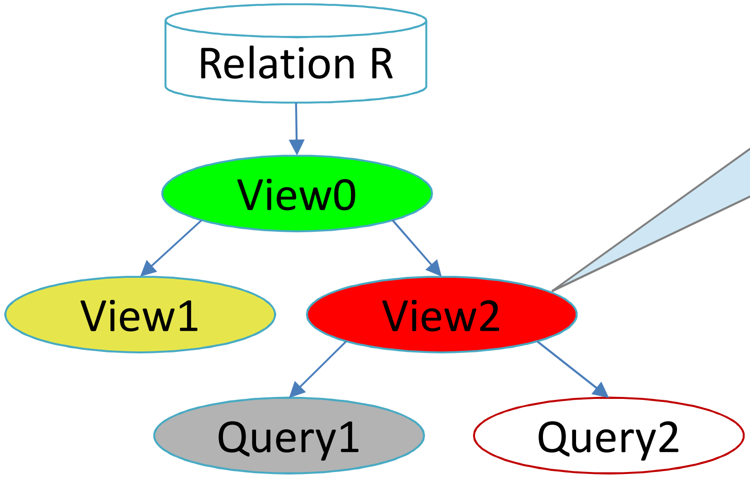


Path Check

Query2 starts Path Check algorithm Searching which of his providers sent him the message and notify him that he does not want to change

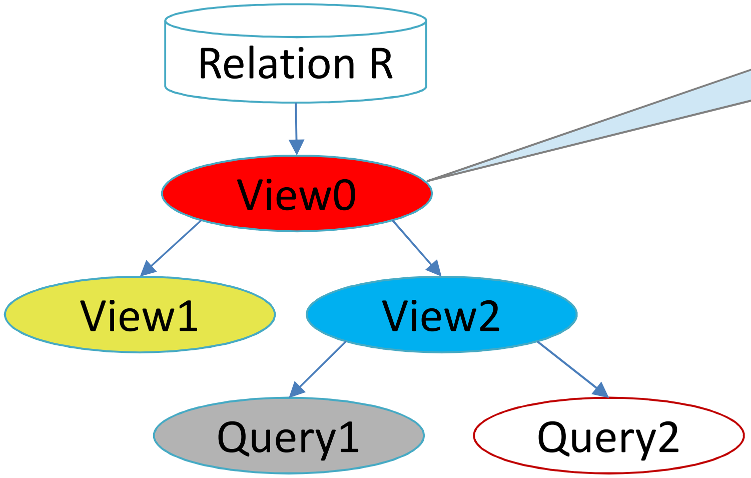


Path Check



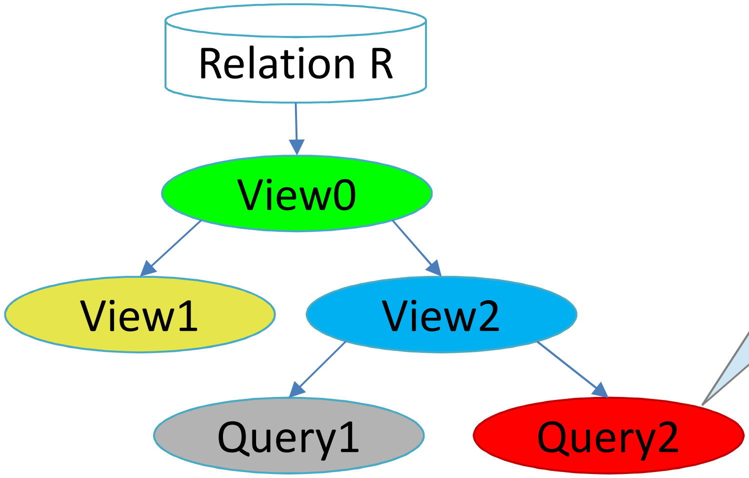
View2 is notified
to keep current version for Query2 and
produce new version for Query1

Path Check



View0 is notified
To keep current version for Query2 and
Produce new version for Query1

Path Check



We make sure that Query2 will not change since it is the blocker

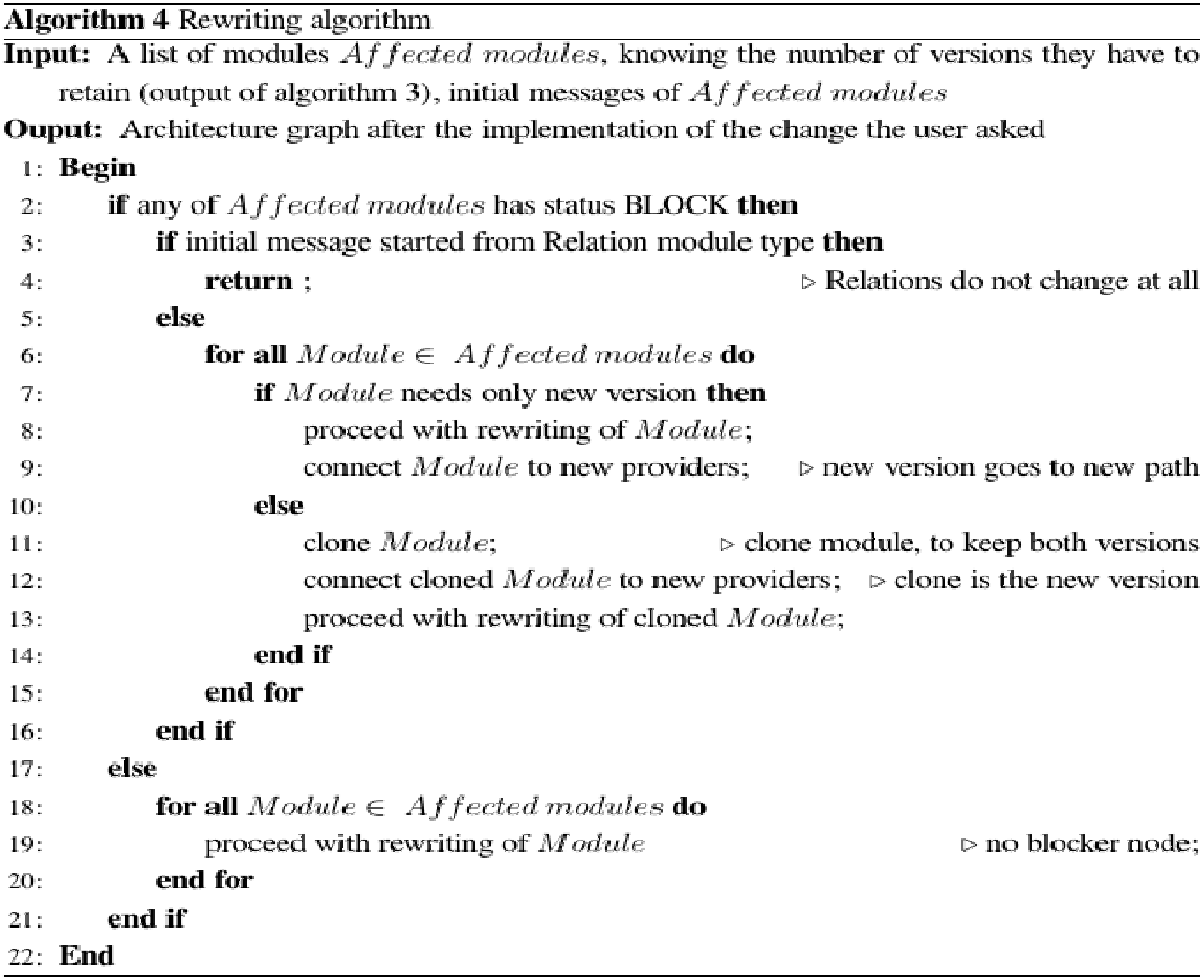
Problem definition

- *Changes on a database schema may cause inconsistency in applications that use that database, is there a way to regulate that?*
- ***If there is such a way of accepting or rejecting a change, could we satisfy it by rewriting the database schema?***
- *If there are conflicts between the applications on acceptance or rejection of a change, is there a possibility of satisfying both?*

Background Information
Message propagation
Path check
Rewriting
Experiments and Results

Rewriting

Rewriting algorithm



Algorithm 4 Rewriting algorithm

Input: A list of modules $Affected\ modules$, knowing the number of versions they have to retain (output of algorithm 3), initial messages of $Affected\ modules$

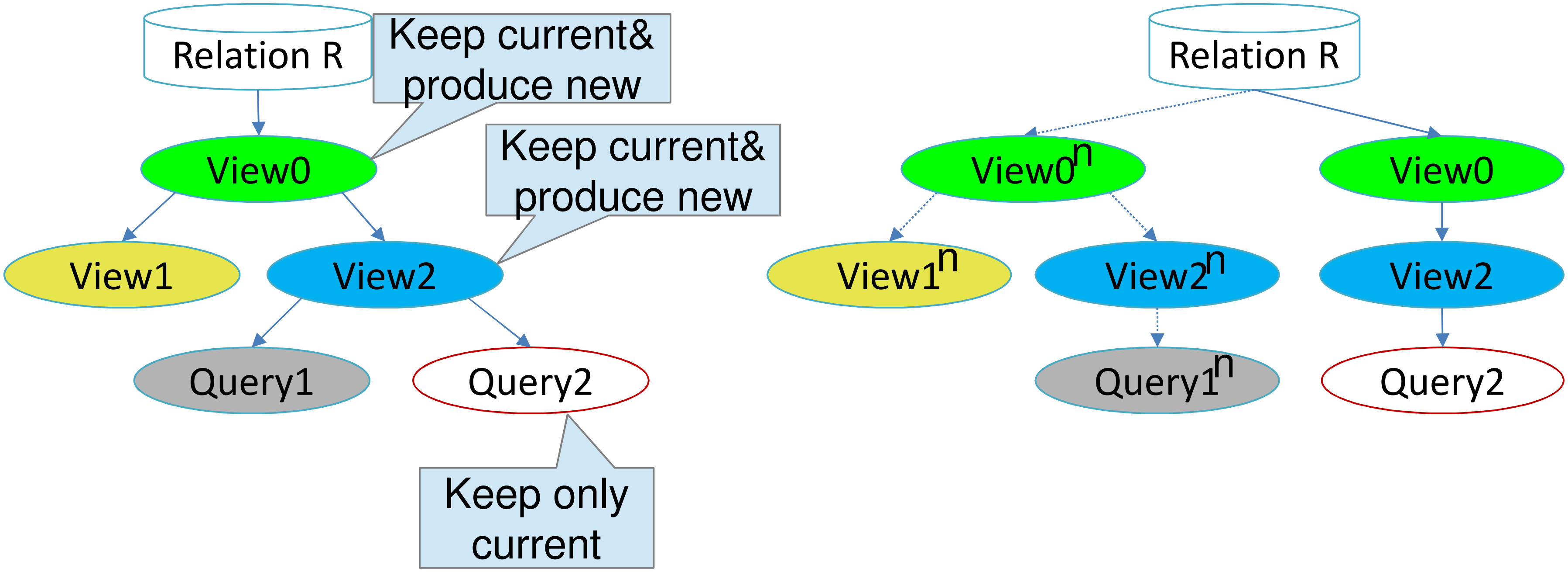
Ouput: Architecture graph after the implementation of the change the user asked

```
Begin
    if any of $Affected\ modules$ has status BLOCK then
        if initial message started from Relation module type then
            return ;                                              ▷ Relations do not change at all
        else
            for all $Module \in Affected\ modules$ do
                if $Module$ needs only new version then
                    proceed with rewriting of $Module$;
                    connect $Module$ to new providers;           ▷ new version goes to new path
                else
                    clone $Module$;                    ▷ clone module, to keep both versions
                    connect cloned $Module$ to new providers;   ▷ clone is the new version
                    proceed with rewriting of cloned $Module$;
                end if
            end for
        end if
    else
        for all $Module \in Affected\ modules$ do
            proceed with rewriting of $Module$                       ▷ no blocker node;
        end for
    end if
End
```

Rewriting

- If there is no Block, we perform the rewriting.
- If there is Block and the initiator of the change is a relation we stop further processing.
- Otherwise:
  - We clone the Modules that are part of a block path and were informed by Path Check and we perform the rewrite on the clones
  - We perform the rewrite on the Module if it is not part of a block path.

Rewriting



Relation R

Keep current& produce new

View0

Keep current& produce new

View1

View2

Query1

Query2

Keep only current

Relation R

View0^n

View0

View1^n

View2^n

View2

Query1^n

Query2

Background Information
Message propagation
Path check
Rewriting
Experiments and Results

Experiments and results

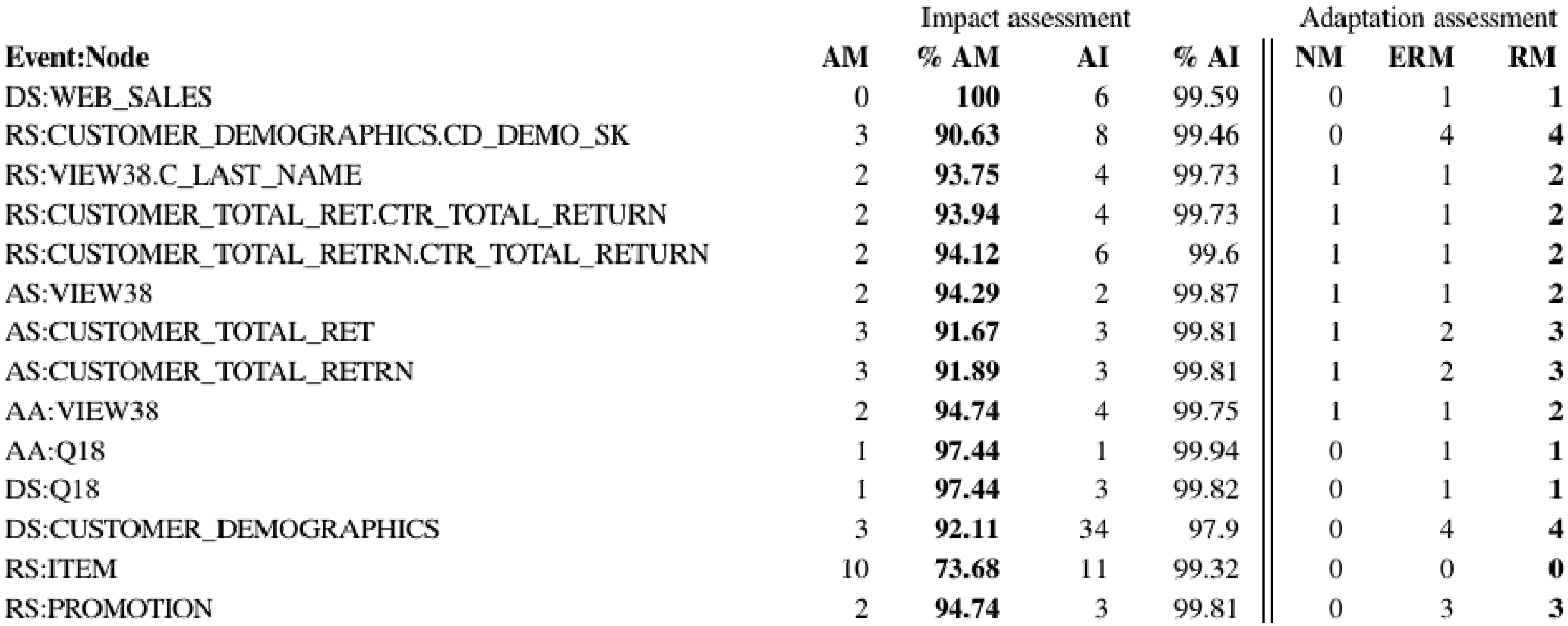
Experimental setup

University database ecosystem (the one of we used in previous slides, consisted of 5 relations, 2 view and 2 queries)
TPC-DS ecosystem (consisted of 15 relations, 5 views and 27 queries) where we used two workloads of events

For both ecosystems: propagate all policy and mixture policy (20% blockers)

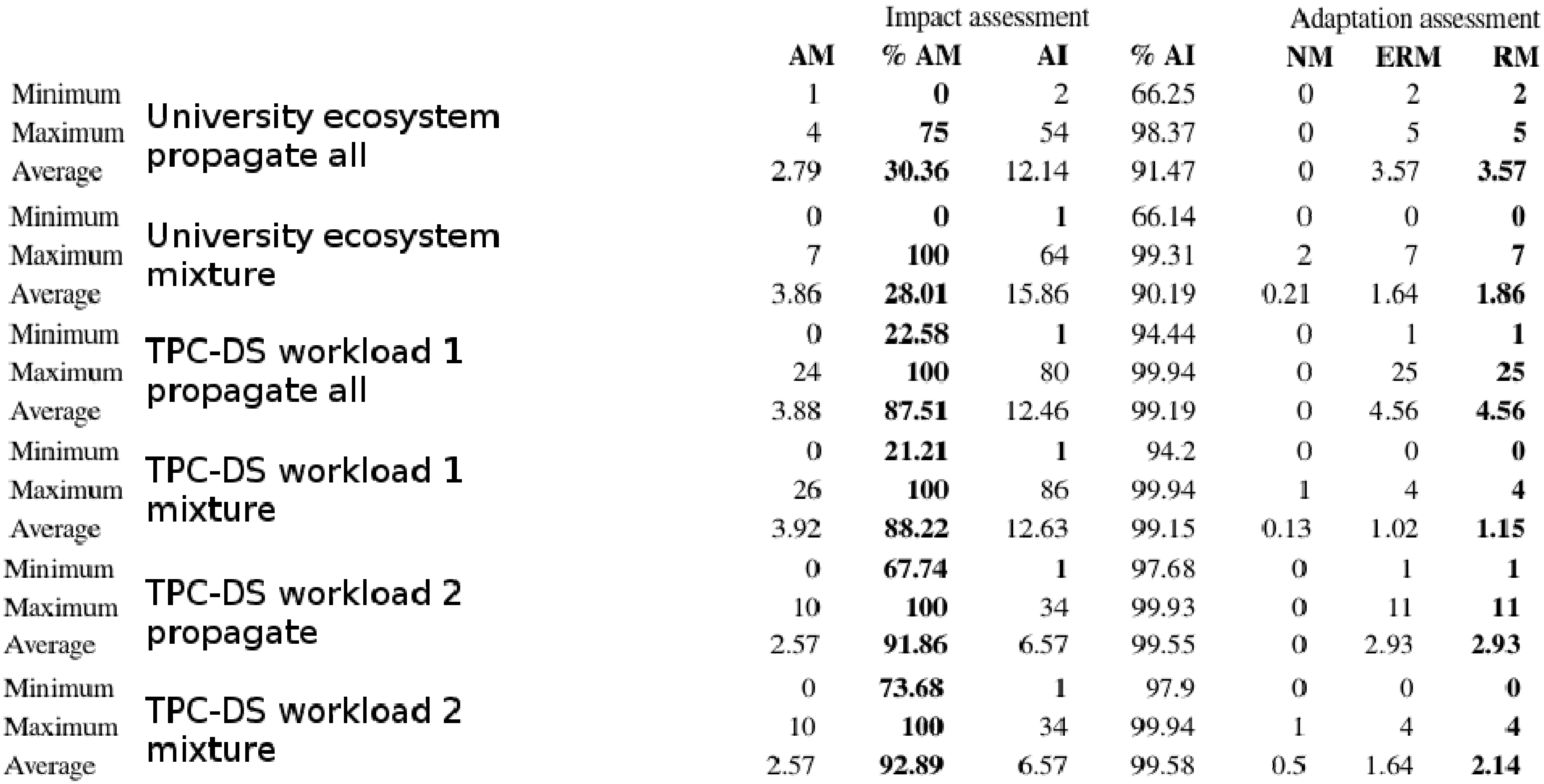
Measurements: effectiveness & cost

Impact & adaptation assessment for TPC-DS



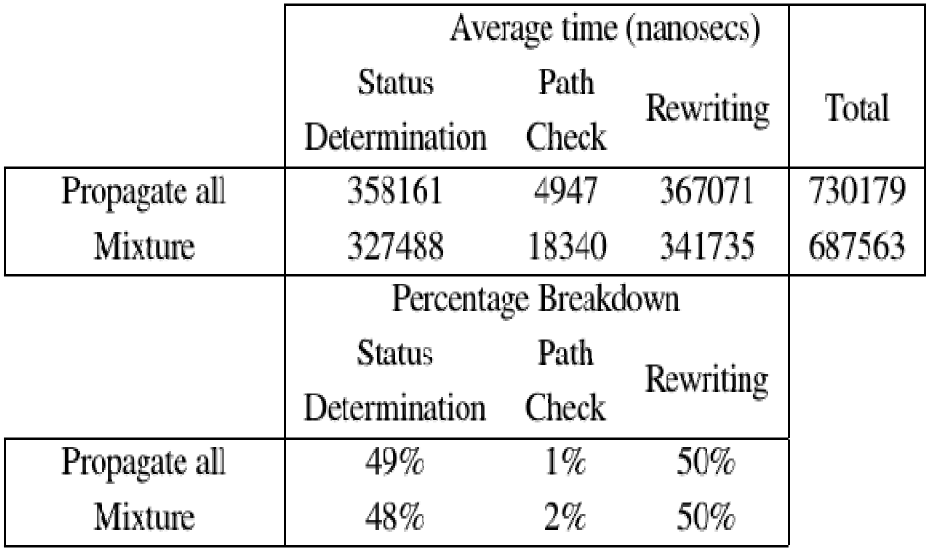
	Impact assessment				Adaptation assessment		
Event:Node	**AM**	**% AM**	**AI**	**% AI**	**NM**	**ERM**	**RM**
DS:WEB_SALES	0	**100**	6	99.59	0	1	**1**
RS:CUSTOMER_DEMOGRAPHICS.CD_DEMO_SK	3	**90.63**	8	99.46	0	4	**4**
RS:VIEW38.C_LAST_NAME	2	**93.75**	4	99.73	1	1	**2**
RS:CUSTOMER_TOTAL_RET.CTR_TOTAL_RETURN	2	**93.94**	4	99.73	1	1	**2**
RS:CUSTOMER_TOTAL_RETRN.CTR_TOTAL_RETURN	2	**94.12**	6	99.6	1	1	**2**
AS:VIEW38	2	**94.29**	2	99.87	1	1	**2**
AS:CUSTOMER_TOTAL_RET	3	**91.67**	3	99.81	1	2	**3**
AS:CUSTOMER_TOTAL_RETRN	3	**91.89**	3	99.81	1	2	**3**
AA:VIEW38	2	**94.74**	4	99.75	1	1	**2**
AA:Q18	1	**97.44**	1	99.94	0	1	**1**
DS:Q18	1	**97.44**	3	99.82	0	1	**1**
DS:CUSTOMER_DEMOGRAPHICS	3	**92.11**	34	97.9	0	4	**4**
RS:ITEM	10	**73.68**	11	99.32	0	0	**0**
RS:PROMOTION	2	**94.74**	3	99.81	0	3	**3**

Impact & adaptation assessment



		Impact assessment				Adaptation assessment		
		AM	% AM	AI	% AI	NM	ERM	RM
Minimum	University ecosystem propagate all	1	**0**	2	66.25	0	2	**2**
Maximum		4	**75**	54	98.37	0	5	**5**
Average		2.79	**30.36**	12.14	91.47	0	3.57	**3.57**
Minimum	University ecosystem mixture	0	**0**	1	66.14	0	0	**0**
Maximum		7	**100**	64	99.31	2	7	**7**
Average		3.86	**28.01**	15.86	90.19	0.21	1.64	**1.86**
Minimum	TPC-DS workload 1 propagate all	0	**22.58**	1	94.44	0	1	**1**
Maximum		24	**100**	80	99.94	0	25	**25**
Average		3.88	**87.51**	12.46	99.19	0	4.56	**4.56**
Minimum	TPC-DS workload 1 mixture	0	**21.21**	1	94.2	0	0	**0**
Maximum		26	**100**	86	99.94	1	4	**4**
Average		3.92	**88.22**	12.63	99.15	0.13	1.02	**1.15**
Minimum	TPC-DS workload 2 propagate	0	**67.74**	1	97.68	0	1	**1**
Maximum		10	**100**	34	99.93	0	11	**11**
Average		2.57	**91.86**	6.57	99.55	0	2.93	**2.93**
Minimum	TPC-DS workload 2 mixture	0	**73.68**	1	97.9	0	0	**0**
Maximum		10	**100**	34	99.94	1	4	**4**
Average		2.57	**92.89**	6.57	99.58	0.5	1.64	**2.14**

Cost analysis

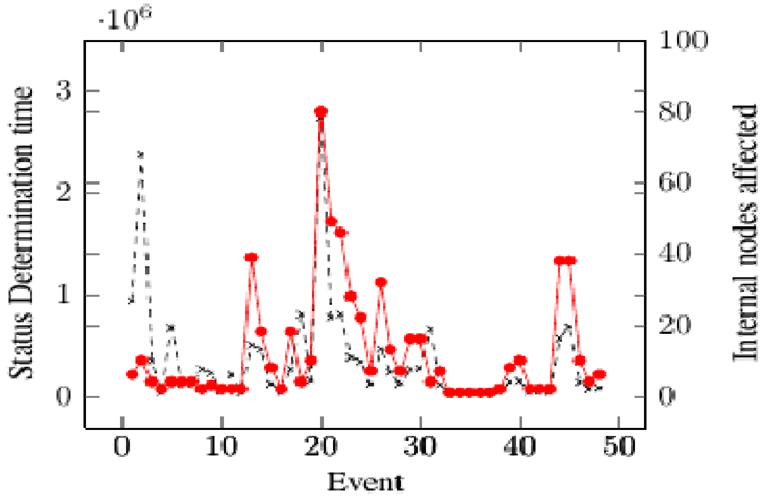


	Average time (nanosecs)			
	Status Determination	Path Check	Rewriting	Total
Propagate all	358161	4947	367071	730179
Mixture	327488	18340	341735	687563

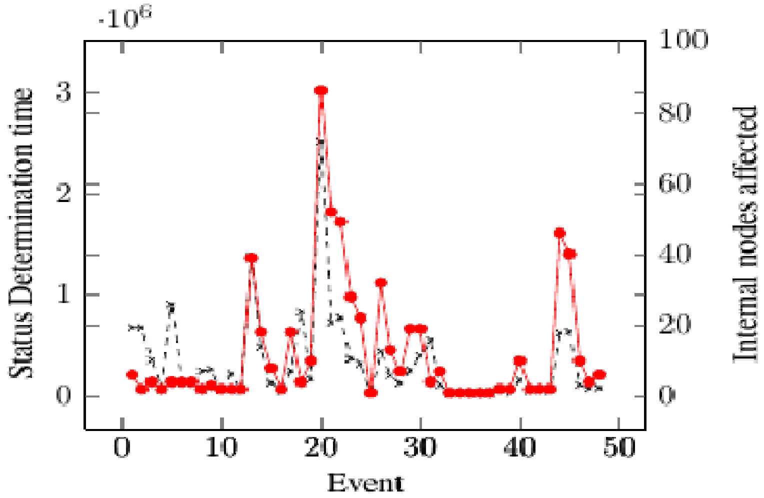
	Percentage Breakdown		
	Status Determination	Path Check	Rewriting
Propagate all	49%	1%	50%
Mixture	48%	2%	50%

- The results of TPC-DS ecosystem in workload 1
- Path check nearly no cost at all, but in 20% blockers doubled its value

Status Determination Cost



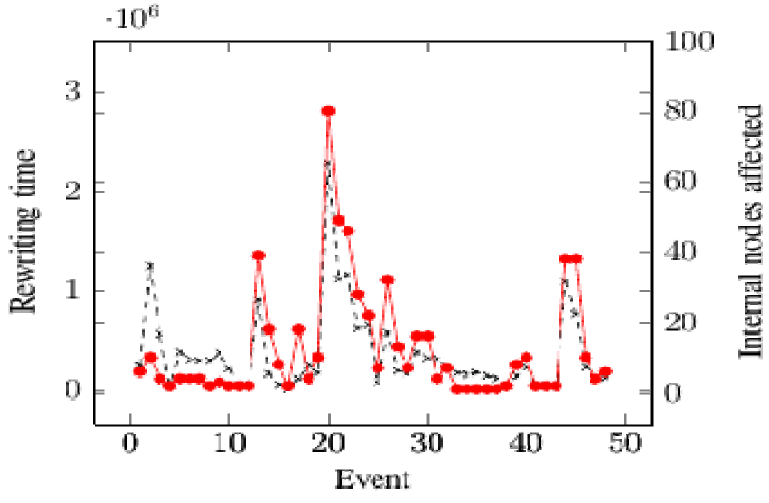
(a) Propagate



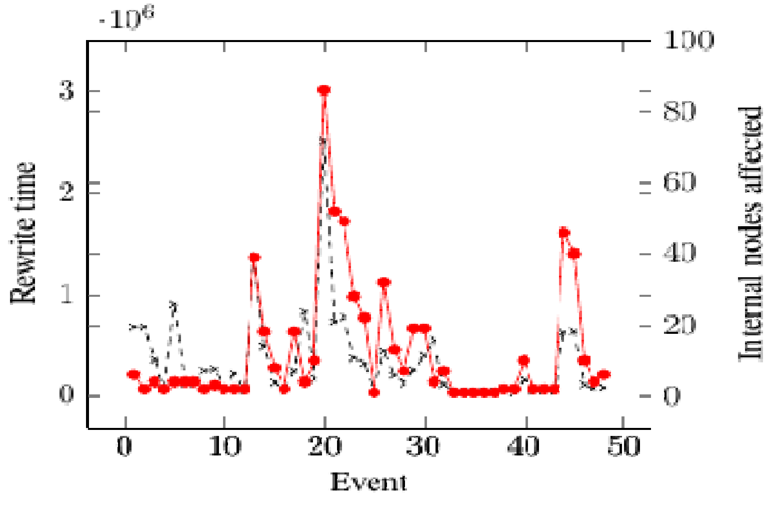
(b) Mixture

- Slightly slower time in mixture mode due to blockers.

Rewrite Cost



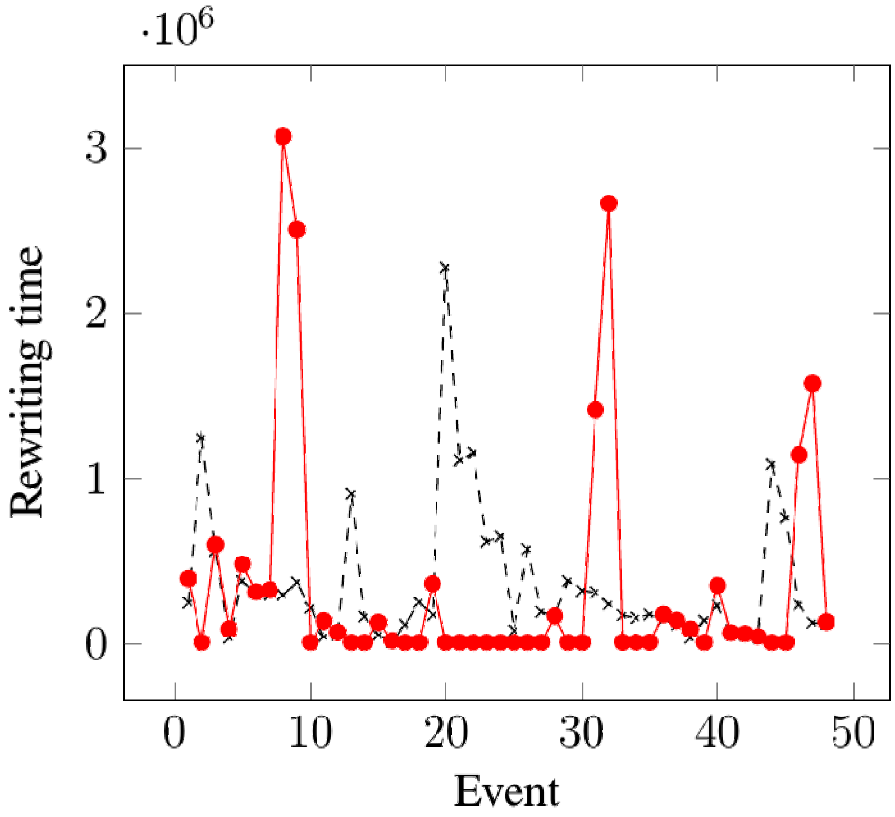
(a) Propagate



(c) Mixture

- Due to blockers and workload containing mostly relation changes, we have no rewrites in mixture mode in a set of events

Rewrite time comparison



- Picks of red are due to cloning of modules.
- Bottoms of red are due to blockers at a relation related event.

Lessons learned #1

- Users gain up to 90% of effort.
- Even in really cohesive environments users gain at least 25% of effort.
- When all modules propagate changes 3.5 modules rewrite themselves on average.

Lessons learned #2

- In “popular” modules our method takes great time compared to unpopular ones.

- Module cloning costs.

- But since the time is measured in nanoseconds this is not big deal

Thank you

Any question?