

Expect
some last
minute
edits

Schema evolution for traditional databases and data warehouses

Panos Vassiliadis

also: Apostolos Zarras, Petros Manousis, Ioannis
Skoulis, George Papastefanatos

Department of Computer Science and Engineering
University of Ioannina, Hellas



Univ. of Ioannina

Database Evolution: why and what

- Software systems and, thus, databases are dynamic environments and can evolve due
 - Changes of requirements
 - Internal restructuring due to performance reasons
 - migration / integration of data from another system
 - ...
- Database evolution concerns
 - changes in the content (**data**) of the databases as time passes by
 - changes in the internal structure, or **schema**, of the database
 - changes in the operational environment of the database

What evolves in DBMS...

- Data

EMP_ID	SALARY
100	1500



EMP_ID	SALARY
100	1650

```
UPDATE EMP  
SET SALARY = SALARY *1.10  
WHERE...
```

- Metadata – Schemata – Models

```
ALTER TABLE EMP  
ADD COLUMN PHONE VARCHAR ...
```

EMP_ID	SALARY
100	1500



EMP_ID	SALARY	PHONE
100	1650	210777777

Why is (schema) evolution so important?

- Software and DB **maintenance** makes up for **at least 50% of all resources spent in a project.**
- Changes are more frequent than you think
- **Databases are rarely stand-alone: typically, an entire ecosystem of applications is structured around them**
=>
- **Changes in the schema can impact a large (typically, not traced) number of surrounding app's, without explicit identification of the impact**

Evolution taxonomy

- **Schema evolution**, itself, can be addressed at
 - the conceptual level (req's, goals, conc. model, Evolve)
 - the logical level, where the main constructs of the database structure evolve
 - E.g.,: relations and views in the relational area, classes in the object-oriented database area, or (XML) elements in the XML/semi-structured area),
 - the physical level, involving data placement and partitioning, indexing, compression, archiving etc.

Evolution taxonomy: areas

- Relational databases
- Object Oriented db's
- Conceptual models
- XML
- Ontologies
- ...

- Special case of relational: data warehouses

... To probe further ...

- Michael Hartung, James F. Terwilliger, Erhard Rahm: Recent Advances in Schema and Ontology Evolution. In Schema Matching and Mapping (Zohra Bellahsene, Angela Bonifati, Erhard Rahm), 149-190, Springer 2011, ISBN 978-3-642-16517-7
- Matteo Golfarelli, Stefano Rizzi: A Survey on Temporal Data Warehousing. IJDWM 5(1): 1-17 (2009)
- Robert Wrembel: A Survey of Managing the Evolution of Data Warehouses. IJDWM 5(2): 24-56 (2009)

Roadmap

- Evolution of views
- Data warehouses
- Impact assessment in ecosystems
- Empirical studies

What views and mat. views are

Traditional research problems related to views

View adaptation

Significant works

VIEW ADAPTATION

Views

- **Virtual** views: macros that allow the developers to construct queries easier by using them as tables in subsequent queries

```
CREATE VIEW sales_vv AS
SELECT t.calendar_year, p.prod_id, SUM(s.amount_sold) AS
       sum_sales
FROM times t, products p, sales s
WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
GROUP BY t.calendar_year, p.prod_id;
```

Query:

```
SELECT * FROM sales_vv WHERE calendar_year > 2012;
```

Views

- **Materialized** views are not macros, however, as they actually store (precompute) the result in persistent storage

```
CREATE MATERIALIZED VIEW sales_mv
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT t.calendar_year, p.prod_id,
       SUM(s.amount_sold) AS sum_sales
FROM times t, products p, sales s
WHERE t.time_id = s.time_id AND p.prod_id =
       s.prod_id
GROUP BY t.calendar_year, p.prod_id;
```

Traditional research problems with views

- **Query answering:** how to integrate views (of all kinds) in the optimizer's plan?
- **View selection:** which views to materialized given query and update workloads?
- **View maintenance:** how to update the stored extent of the mat. view when changes occur at the sources?
 - For which views can I do it? (query class)
 - How: Full or Incremental?
 - When: On update? On demand? Periodically?
 - Available info: deltas only? Int. constraints?

Oracle 11g and Materialized Views

CREATE MATERIALIZED VIEW view-name

BUILD [IMMEDIATE | DEFERRED]

- Compute extent at view definition or at query time

REFRESH [FAST | COMPLETE | FORCE]

- FAST: incremental (needs `log` def. on source tables); COMPLETE: full; FORCE: if FAST fails, then COMPLETE

ON [COMMIT | DEMAND]

- Trigger refresh when sources are updated, or on-demand

[[ENABLE | DISABLE] QUERY REWRITE]

- Used by the optimizer during Query Optimization

AS SELECT ... query definition ...;

```
CREATE MATERIALIZED VIEW LOG ON times
WITH ROWID, SEQUENCE (time_id,
calendar_year)
INCLUDING NEW VALUES;
```

```
CREATE MATERIALIZED VIEW LOG ON products
WITH ROWID, SEQUENCE (prod_id)
INCLUDING NEW VALUES;
```

View adaptation

- What if there is a change in
 - the view definition?
 - the schema of the sources?
- Can we maintain the view's
 - definition
 - extent
- correctly and efficiently?

Gupta et al @ Inf. Systems, 26(5), 2001

- **Assume the view definition changes**
- **Given**
 - the old and the new view definition
 - the existing data that are stored in the view
 - the source tables
 - (when needed: auxiliary information, like indexes on PK's, aux. relations, ...)
- **Produce the extent corresponding to the new view definition**
- **Such that**
 - It is done incrementally rather than via a complete recomputation

A “taxonomy” of atomic changes to SPJ and SPJG+ views

Method: The authors assume a comprehensive set of potential **atomic changes**.

- Addition or deletion of an attribute in the SELECT clause.
- Addition, deletion, or modification of a predicate in the WHERE clause (with and without aggregation).
- Addition or deletion of a join operand (in the FROM clause), with associated equijoin predicates and attributes in the SELECT clause.
- Addition or deletion of an attribute from the GROUPBY list.
- Addition or deletion of an aggregation function to a GROUPBY view.
- Addition, deletion, or modification of a predicate in the HAVING clause. Addition of the first predicate or deletion of the last predicate corresponds to addition and deletion of the HAVING clause itself.
- Addition or deletion of an operand to the UNION and EXCEPT operators.
- Addition or deletion of the DISTINCT operator.

For each type of change the authors propose a set of steps required to maintain the view's extent

Example: Adding an atomic selection to the WHERE clause

Assume we add a filter Q to a view V0

```
CREATE VIEW V0 AS
SELECT A1, ..., An
FROM R1 & ... & Rm
WHERE Q AND C1 AND ... AND Ck
```

We want to maintain V0 given its old extent and the source relations.

Algebraically: $V' = V - V^- \cup V^+$

where V^+ are the tuples that should be inserted in the view and V^- are the tuples to be removed

```
DELETE FROM V WHERE NOT Q           //delete V-
INSERT INTO V ( SELECT A1, ..., An   //add V+
FROM R1 & ... & Rm
WHERE Q1 AND NOT C1 AND ... AND NOT Ck )
```

Important notes

- Maintenance is incremental: you try to recompute V by checking out only the existing data
- “Taxonomy” of atomic changes with locality principle: if you are given a complex redefinition, you can process it one change at a time (atomic changes are composable)

Nica et al., EDBT 1998

- What if the schema in one of the relations participating to the view definition changes?
- The method by Nica et al., proposes an algorithm (heavily oriented towards handling **deletions**) for rewriting the view to address the change
- Two pillars:
 - A Meta Knowledge Base keeping semantic properties of the database
 - The annotation of views with directives on how to respond to changes

Meta Knowledge Base

- Information on
 - Available relations and views
 - Implicit join conditions
 - Semantic equivalences: which attribute/relation can be regarded as a potential replacement for another
- For example:
 - Join conditions:
 - `product.prod_id = sales.prod_id`
 - Equivalence assertions:
 - `sales.prod_id = product.prod_id`
 - `times.calendar_year = year(sales.time_id)`

View annotation

- E-SQL: language to annotate parts of a view (exported attributes, underlying relations and filters) wrt:
 - Dispensability: if the part can be removed from the view definition completely
 - Replaceability with an another equivalent part.

```
CREATE VIEW empProj_VV AS
SELECT e.ENAME, e.EPHONE (AD true, AR true) p.PNAME,
       w.PDURATION
FROM EMP e (RR true), PROJECT p, WORKS w
WHERE (e.EID = w.EID) AND (p.PID = w.PID) AND
      (p.PLOCATION=Barcelona) (CD true)
```

```
//assuming a relation EMP_ContactInfo duplicating id, name,
      phone of EMP's, possibly with other contact info means
```

Complex View Synchronization algorithm

- Input : (0) an SPJ view V , (1) a change in a relation, (2) old MKB entities, and, (3) new MKB entities.
- Output: view rewritings to adapt to new MKB providing the same result
- Means: model that represents attributes as hyper-nodes and (i) relations, (ii) join cond., and (iii) equivalence assertions as hyper-edges
- Steps:
 - find all entities affected for Old MKB to become New MKB,
 - for each one of these entities find a replacement from Old MKB,
 - rewrite the view over these replacements.

- DWs as Views
- Evolving dimensions & SCD
- Multiversion

DATA WAREHOUSE EVOLUTION

Early days (late '90s)

- Back then, people continued to think that DWs were **collections of materialized views**, defined over sources.
- In this case, evolution is mostly an issue of adapting the views' definitions whenever sources changes

Bellahsene (DEXA'98, KAIS02)

- Annotate views with a **HIDE** clause that works oppositely to SELECT (i.e., you project all attributes except for the hidden ones)
- What if **sources** change? The author considers attribute/relation addition & deletion and the impact it has to view rematerialization (how to recompute the materialized extent via SQL commands)
- Cost model to estimate the cost of different options

Quix @ DMDW '99



- Context: DW schemata annotated with quality factors
- Metadata that track the history of changes and provide a set of rules to enforce when a quality factor (completeness, consistency, correctness, ...) has to be reevaluated.
- Basic **taxonomy of changes**

	Relation	View	Attribute	Constraint
Add	✓	✓	✓	✓
Delete	✓	✓	✓	✓
Rename	✓	✓	✓	
Redefine semantics		✓		

... and then came dimension buses and multidimensional models ...

- ... which treat the DW is a collection of
 - **cubes**, representing clean, undisputed facts that are to be loaded from the sources, cleaned and transformed, and eventually queried by the client applications
 - defined over **consolidated dimensions** that uniquely and commonly define the context of the facts
- ... **The idea of a central DW schema acting as reference for the back-stage loading and front-end querying completely changed the perspective of DW research ...**

Slowly Changing Dimensions

What you've probably heard for dimension updates is SCD's

- Type 0: no change allowed
- Type 1: new value overwrites old
- Type 2: new record; valid time timestamps + status columns indicate which row is current and what happened
 - New Surrogate Key (so joins with facts work as if these are different dimension records)
 - Same natural / detailed key (to be used in group by's)
 - Status attribute: Current vs Old (aka Type 6)
- Type 3: add new column "PreviousValueForAttributeXXX" and update cells with new and old values respectively

Slowly Changing Dimensions

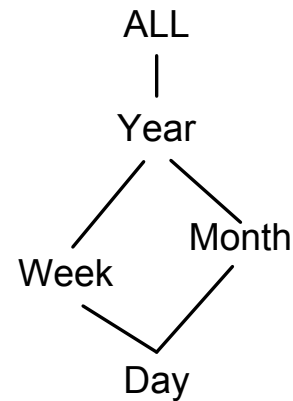
- Type 4: definitions vary
 - Split type 2 table in two tables subsets of the data set: the historical one and the current one (single row)
 - Kimball's: if some attributes of the dimension change frequently, export a new table (called "profile") just for them; facts have two FK's for the dimension, one for the dim table and another for the profile table

Hurtado, Mendelzon and Vaisman @ DOLAP99, ICDE'99

Quick guide to dimensional modeling

For each dimension:

- **Levels** for “granularity degrees” of information
- Each level L with a domain $\text{dom}(L)$ (typically isom. to integers)
- Can have attributes too
- Typically form a **lattice** with
 - a detailed level at bottom and
 - A single-valued ‘ALL’ level at the top
- **Rollup functions** between subsequent levels
- Have to be fully defined at the domain level and consistent under composition
- Drill-down relations (not functions): their inverse



- [HuMN99a,b] Set of **operators** for evolving dimensions prescribing what should be done to have both a consistent schema and a consistent set of instances

Hurtado, Mendelzon and Vaisman @ DOLAP99, ICDE'99

Generalize	Adds a new level above a preexisting one, + a rollup function
Specialize	Adds a new level below the current bottom level + a rollup function
Relate	Adds a new edge, between two parallel levels. The associated rollup function, if it exists, is determined automatically. If not possible to do so uniquely, the operator is not applicable.
Unrelate	Deletes an edge between two levels.
Delete Level	Deletes a level with the precondition that the new hierarchy must have a unique bottom level (ALL cannot be deleted).
Add Instance	Adds a value, say x, + a pair of the form (x,y) for each rollup function
Delete Instance	Deletes a value x from a level L + rollup functions
Reclassify	Update rollup-memberships (e.g., a brand moves to a new company)
Split & Merge	Czechoslovakia <-> Czechia & Slovakia + rollup functions
Update	Rename value without structural changes

Blaschka, Sapia and Höfling

@DaWaK'99

- Data model + an **evolution algebra** :
 - **Evolution operators** for multi-dimensional schemata and
 - Spec. of their effects to both schema and instances.
- Operators: **atomic evolution operations**,
- that can be used for complex operations.

Algebraic Operator

Insert level

Delete level

Insert Attribute

Delete Attribute

Connect attribute to dimension level

Disconnect attribute from dimension level

Connect attribute to fact

Disconnect attribute to fact

Insert classification relationship

Delete classification relationship

Insert fact

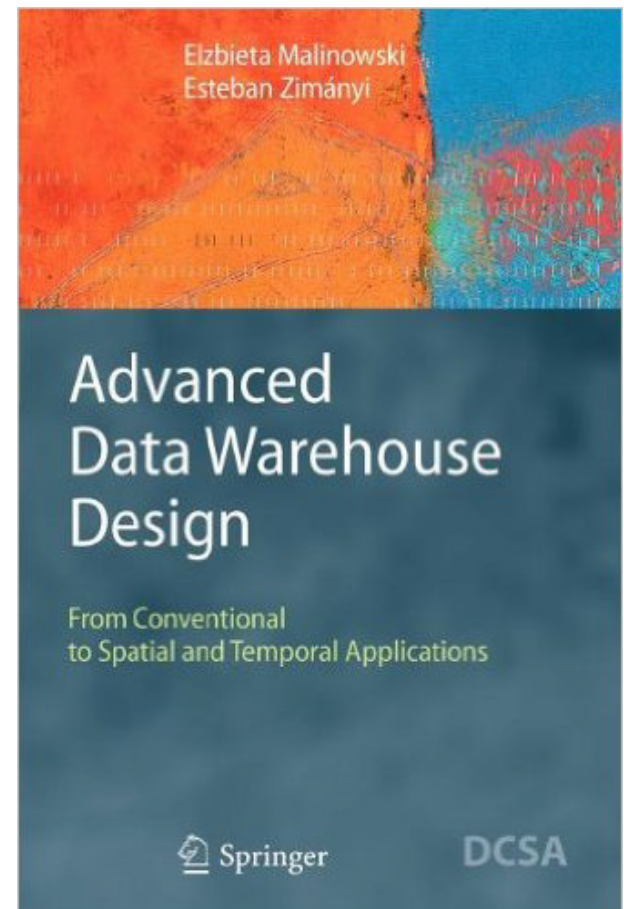
Delete fact

Insert dimension into fact

Delete Dimension

... and then came versioning...

- After we had obtained a basic understanding of how multidimensional schemata are restructured, people thought:
- “what if we keep track of the history of all the versions of a DW schema as it evolves?”
- Then, we can ask a query that span versions, get the data, transform them into a convenient schema for the query and show results to the users



Closely related to **temporal management** in DW's

See later today PhD defense by Waqas Ahmed

Eder and Koncilia @ DaWaK 2001

- Multidimensional data model that allows the registration of temporal versions of dimension data in data warehouses.
- To navigate between temporal versions: mappings as **transformation matrices**. Each matrix is a mapping of data from structure V_i to V_{i+1} for a dimension D . For example, table T describes a split of value a into values a_1 and a_2 respectively. There is an mapping function that describes that the 30% of the fact –values for A should be placed to a_1 and the remaining should be placed in a_2 .

T	A1	A2
A	30%	70%

- This mapping function is described in a transformation matrix T that says exactly that in order to go from A to A_1 we need to take 30% of the tuples of table A and what remains goes to table A_2 .
- Queries are posed over snapshots of the database. For each query the appropriate snapshots are computed.

Eder and Koncilia @ DaWaK 2001

- We can transform each cuboid C (with facts) over a set of dimensions from version V_i to version V_{i+1} , by sequentially transforming each of its dimensions one at a time.

- Original cube** (dimension values on the side)

$$C = \begin{matrix} & a_1 & a_2 & a_3 \\ b_1 & \left(\begin{array}{ccc} 3 & 7 & 5 \end{array} \right) \\ b_2 & \left(\begin{array}{ccc} 10 & 8 & 6 \end{array} \right) \\ b_3 & \left(\begin{array}{ccc} 20 & 13 & 5 \end{array} \right) \end{matrix}$$

- Transformation matrices for dimensions**

$$T_A = \begin{matrix} & a_{11} & a_{12} & a_2 & a_3 \\ a_1 & \left(\begin{array}{cccc} 0.3 & 0.7 & 0 & 0 \end{array} \right) \\ a_2 & \left(\begin{array}{cccc} 0 & 0 & 1 & 0 \end{array} \right) \\ a_3 & \left(\begin{array}{cccc} 0 & 0 & 0 & 1 \end{array} \right) \end{matrix} \quad T_B = \begin{matrix} & b_1 & b_2 & b_3 \\ b_{12} & \left(\begin{array}{ccc} 1 & 1 & 0 \end{array} \right) \\ b_3 & \left(\begin{array}{ccc} 0 & 0 & 1 \end{array} \right) \end{matrix}$$

- Final cube with the values of the original version over the structure of the new version

$$\begin{matrix} & a_{11} & a_{12} & a_2 & a_3 \\ b_{12} & \left(\begin{array}{cccc} 3.9 & 9.1 & 15 & 11 \end{array} \right) \\ b_3 & \left(\begin{array}{cccc} 6 & 14 & 13 & 5 \end{array} \right) \end{matrix}$$

Eder, Koncilia and Mitsche @ DaWaK'03, CAiSE'04

- Making use of three basic operations (INSERT, UPDATE and DELETE), the authors are able to represent more complex operations on dimension values such as: SPLIT, MERGE, CHANGE, MOVE, NEW-MEMBER, and DELETE-MEMBER.
- Also: data mining techniques for the detection of structural changes in data warehouses.

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

- How to facilitate cross-version queries?
- A graph model for DW multidimensional schemata
- Nodes : (i) fact tables and (ii) their attributes of fact tables (including properties and measures),
- Edges: functional dependencies (aka dimension hierarchies) defined over the nodes of the schema.

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

- Taxonomy of changes:
 - Add / delete node (i.e., tables and attributes)
 - Add / delete edge (i.e., restructure dimensions)
- Transactions = sequences of atomic changes

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

Augmented schema of a previous version

- Assume a version S_k
- Assume a set of changes M_1, \dots, M_n
- Then you get to a version S_{k+n}
- The augmented version of S_k wrt S_{k+n} is the schema and data of S_k , along with all the extra attributes and FD's added at S_{k+n}
- So basically, we are adapting the previous schema+data to the structure of the new version
- This might require aggregations or disaggregations (and estimations of the necessary values), addition of default values, ...

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

- Assume a fact
 - **SALES(ProdID, DayId, CustId, Price, Qty)**
- With a set of dimensions
 - **Product (Product, Type, Family)**
 - **Customer(Customer, CustGroup)**
 - **Time(Day, Month, Year)**
- and a set of changes
 - Add attribute Salesman and a hierarchy Salesman -> Store
 - Remove day from the time hierarchy and replace it with Month
 - $\text{SumSales} = \text{Qty} * \text{Price}$
- Then, the new fact is
SALES' (ProdID, MonthId, CustId, SalesmanId, Price, Qty, SumSales)

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

SALES(ProdID, DayId, CustId, Price, Qty)

SALES' (ProdID, MonthId, CustId, SalesmanId, Price, Qty, SumSales)

- We can compute the augmented version of the OLD schema

SALES^{Aug}(ProdID, DayId, MonthId, CustId, SalesmanId, Price, Qty, SumSales)

- ...that includes @ schema level
 - The old attributes & FD's
 - The new attributes & FD's added during evolution
 - ... hoping that all FD's hold (otherwise there is no augmentation)
- ... and at data level: values of SALES (the old v.) with interpolation for the measures due to dimension addition

Golfarelli, Lechtenbörger, Rizzi and Vossen @ DKE 2006

- History is a sequence of versions $H = (v_1, \dots, v_n)$. Each version has
 - Its own schema
 - The augmented schema wrt v_n //needs modification if v_{n+1} comes
 - The timestamp of change
- Why bother?
- Because at query time, we can transform the old schema and data to the last one.
- Then we can pose queries to the old data based on the structure of the new one and a get a uniform result under the last known schema.
- If differences (e.g., because of attribute deletions), we retain the common set of attributes

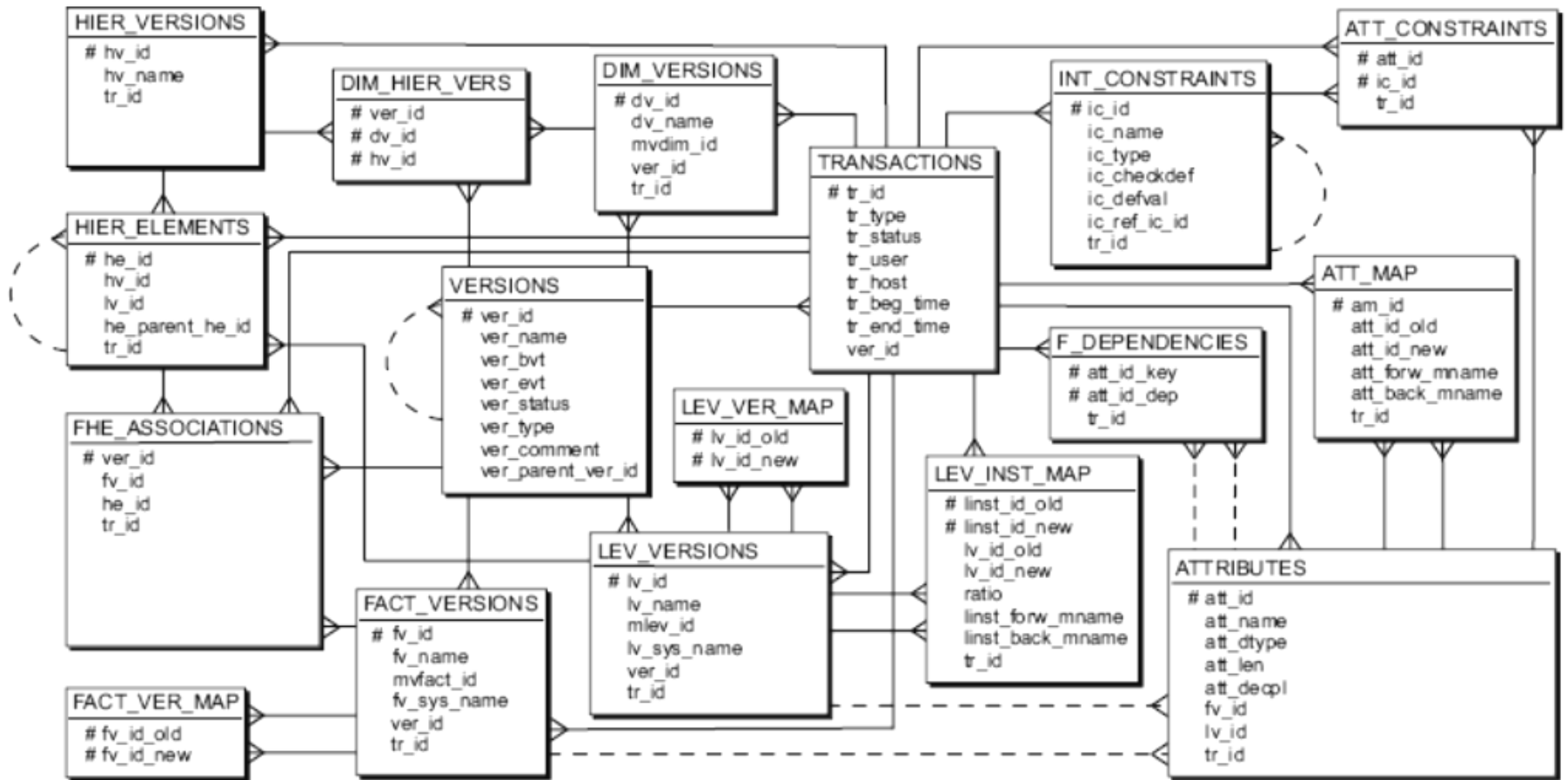
Wrembel and Bebel @ JoDS'07

- How to handle changes that come up on the external data sources (EDS) of a data warehouse?
- Deal with it via a multiversion technique!
- Everything has a version
 - Dimensions, levels and hierarchies
 - Facts
 - Attributes
 - Integrity constraints
- Mappings are between versioned objects. E.g.,
 - level versions are mapped to dimension versions
 - Fact versions to level versions
 - ...
- Both real and alternative (for simulation) versions are supported

Wrembel and Bebel @ JoDS'07

- How to handle changes that come up on the external data sources (EDS) of a data warehouse?
- Deal with it via a multiversion technique!
- Everything has a version (each with a valid time):
 - Dimensions, levels and hierarchies
 - Facts
 - Attributes
 - Integrity constraints
- Mappings are between versioned objects. E.g.,
 - level versions are mapped to dimension versions
 - Fact versions to level versions
 - ...
- Both real and alternative (for simulation) versions are supported

Wrembel and Bebel @ JoDS'07



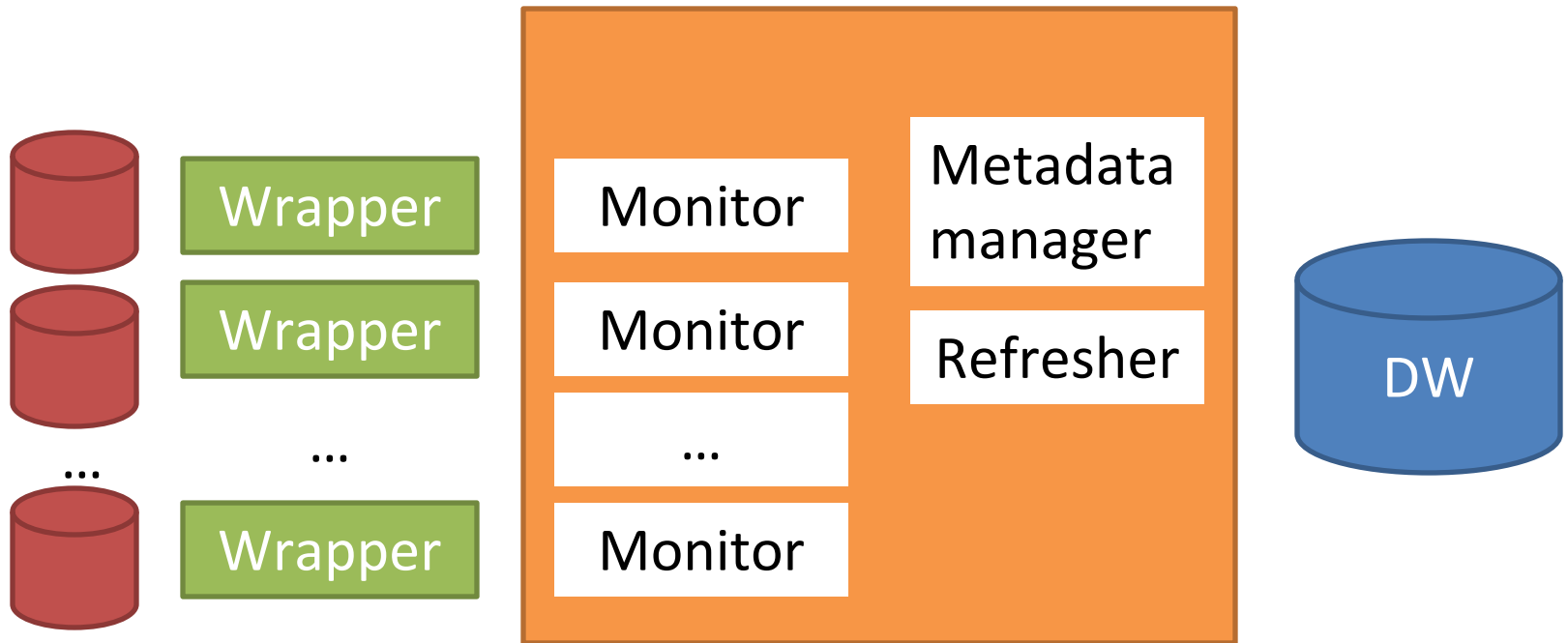
Wrembel and Bebel @ JoDS'07

- Schema Change Operations
 - the addition / deletion of attribute @ dimension level table,
 - the creation of a new fact table + the association of a fact with a dimension
 - the renaming of a table,
 - snowflake changes:
 - the creation of a new dimension level table with a given structure
 - the inclusion of a parent dimension level table into its child dimension level table,
 - the creation of a parent dimension level table based on its child level table.
- Instance change operations
 - Add/del level instance
 - Change parent of a level
 - Merge many instances of a level into a single one / split(inverse)

Wrembel and Bebel @ JoDS'07

- Querying multiple versions
- Split original query to a set of single version queries
- For each single version query, do a best-effort approach:
 - if attributes are missing, omit them;
 - use metadata for renames
 - ignore v. if a grouping is impossible
 - ...
- If possible, the collected results are integrated under the intersection of attributes common to all versions (if this is the case of the query);
- Else they are presented as a set of results, each with its own metadata

Wrembel and Bebel @ JoDS'07



monitored
External
Data
Sources

George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, Yannis Vassiliou.
Metrics for the Prediction of Evolution Impact in ETL Ecosystems: A Case
Study. Journal on Data Semantics, August 2012, Volume 1, Issue 2, pp 75-
97

A CASE STUDY OF DW EVOLUTION

Context of the Study

- We have studied a data warehouse scenario from a Greek public sector's data warehouse maintaining information for farming and agricultural statistics.
- The warehouse maintains statistical information collected from surveys, held once per year via questionnaires.
- Our study is based on the evolution of the source tables and their accompanying ETL flows, which has happened in the *context of maintenance due to the change of requirements at the real world*.
- Practically this is due to the update of the questionnaires from year to year

Internals of the monitored scenario

- The environment involves a set of **7 ETL workflows**:
 - 7 source tables, (S1 to S7)
 - 3 lookup tables(L1 to L3),
 - 7 target tables, (T1 to T7), stored in the data warehouse.
 - 7 temporary tables (each target table has a temporary replica) for keeping data in the data staging area,
 - **58 ETL activities in total** for all the 7 workflows.

PL/SQL to graph transformation

- All ETL scenarios were source coded as PL\SQL stored procedures in the data warehouse.
 - We extracted embedded SQL code (e.g., cursor definitions, DML statements, SQL queries) from activity stored procedures
 - Each activity was represented in our graph model as a view defined over the previous activities
 - Table definitions were represented as relation graphs.

Method of assessment

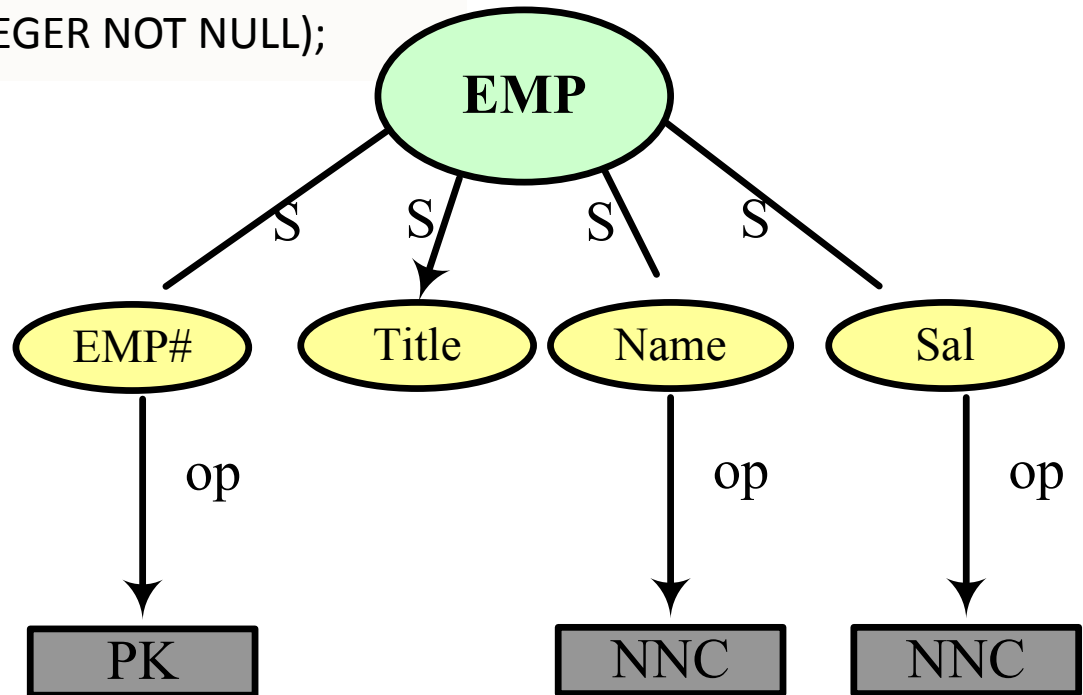
- We have represented the ETL workflows in our graph model
- We have recorded evolution events on the nodes of the source, lookup and temporary tables.
- We have applied each event sequentially on the graph and monitored the impact of the change towards the rest of the graph by recording the times that a node has been affected by each change

Graph modeling of a data-intensive ecosystem

- The **entire data-intensive ecosystem**, comprising **databases** and their internals, as well as **applications** and their data-intensive parts, is modeled via a graph that we call **Architecture Graph**
- Why Graph modeling?
 - Completeness: graphs can model everything
 - Uniformity: we would like to model everything **uniform** manner
 - Detail and Grand-View: we would like to capture **parts** and **dependencies** at the very **finest level**; at same time, we would like to have the ability to **zoom-out** at higher levels of abstraction
 - Exploit graph management techniques and toolkits

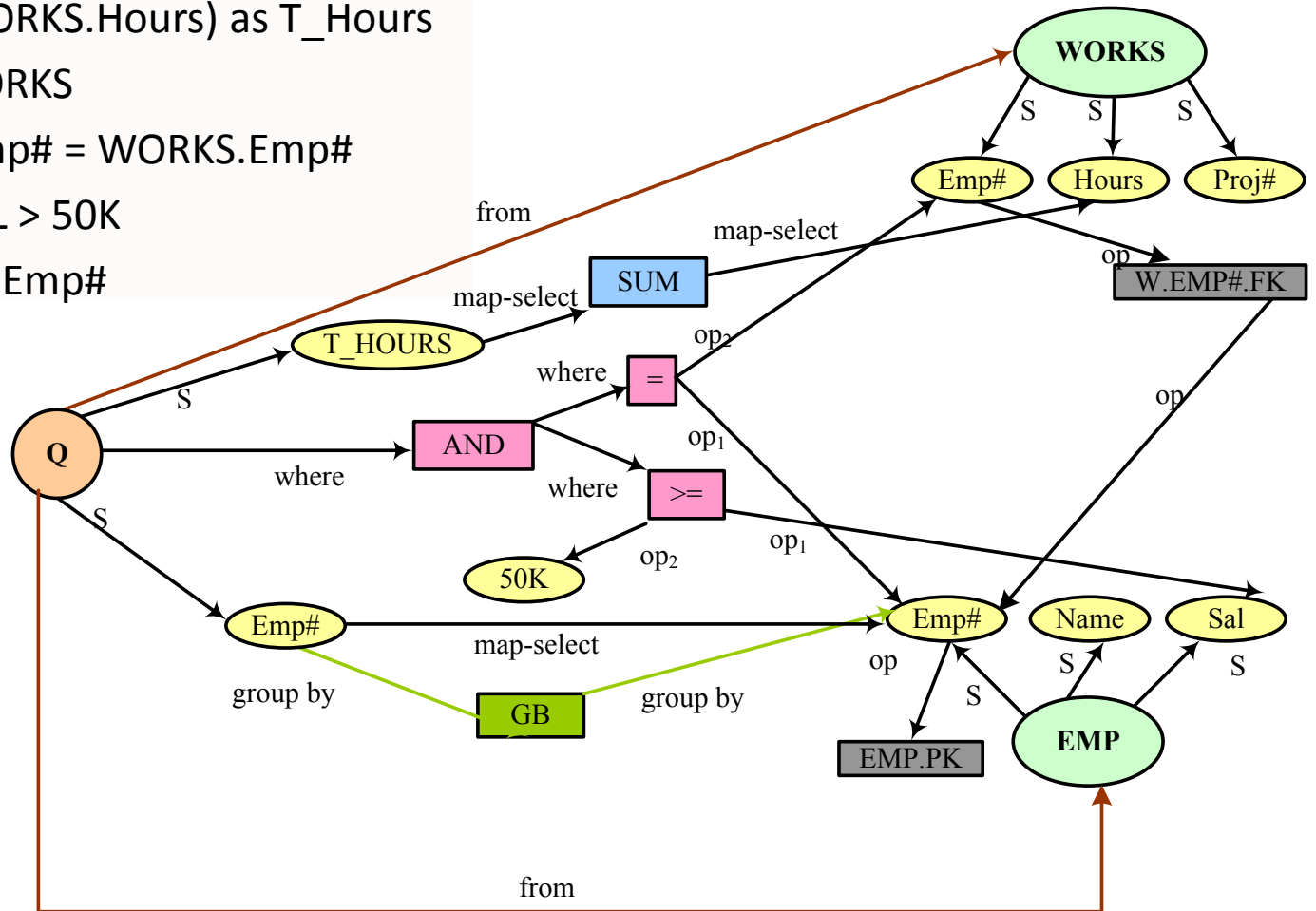
Relations – Attributes - Constraints

```
CREATE TABLE EMP (EMP# INTEGER PRIMARY KEY,  
NAME VARCHAR(25) NOT NULL,  
TITLE VARCHAR(10),  
SAL INTEGER NOT NULL);
```



Queries & Views

Q: SELECT EMP.Emp# as Emp#,
 Sum(WORKS.Hours) as T_Hours
 FROM EMP, WORKS
 WHERE EMP.Emp# = WORKS.Emp#
 AND EMP.SAL > 50K
 GROUP BY EMP.Emp#

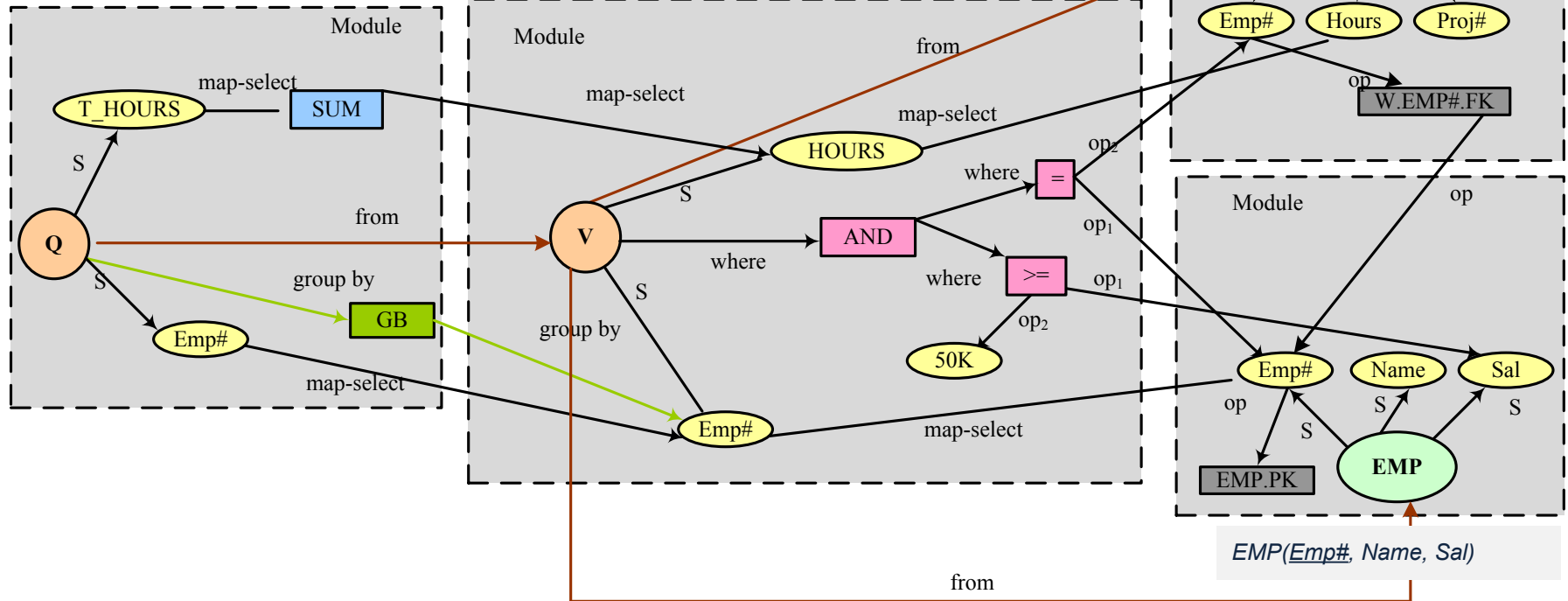


Modules: relations, queries, views

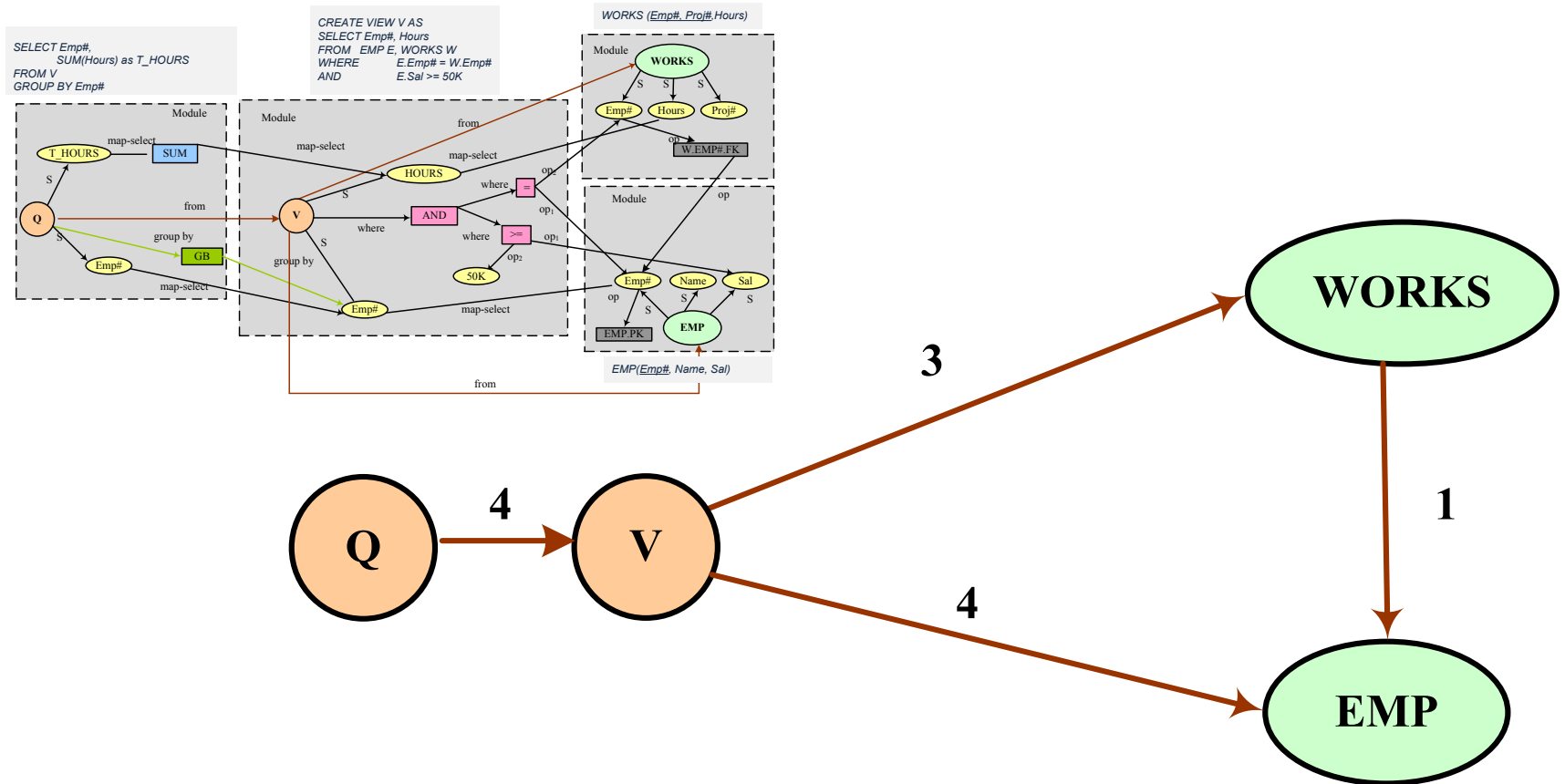
```
SELECT Emp#,
       SUM(Hours) as T_HOURS
FROM V
GROUP BY Emp#
```

```
CREATE VIEW V AS
SELECT Emp#, Hours
FROM EMP E, WORKS W
WHERE E.Emp# = W.Emp#
AND E.Sal >= 50K
```

WORKS (Emp#, Proj#, Hours)



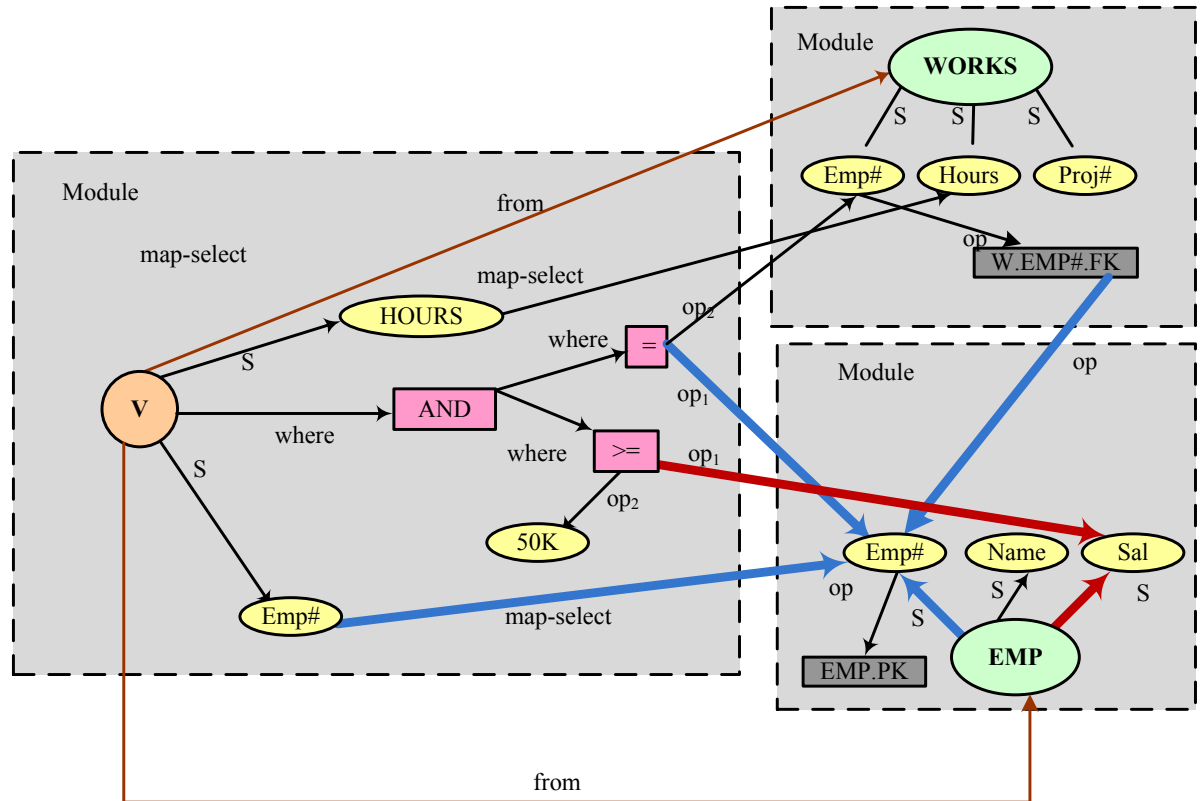
Zooming out to top-level nodes (modules)



Metrics: Node Degree

Simple metrics:
in-degree, out-degree, degree

EMP.Emp# is the most important attribute of EMP.SAL, if one considers how many nodes depend on it.

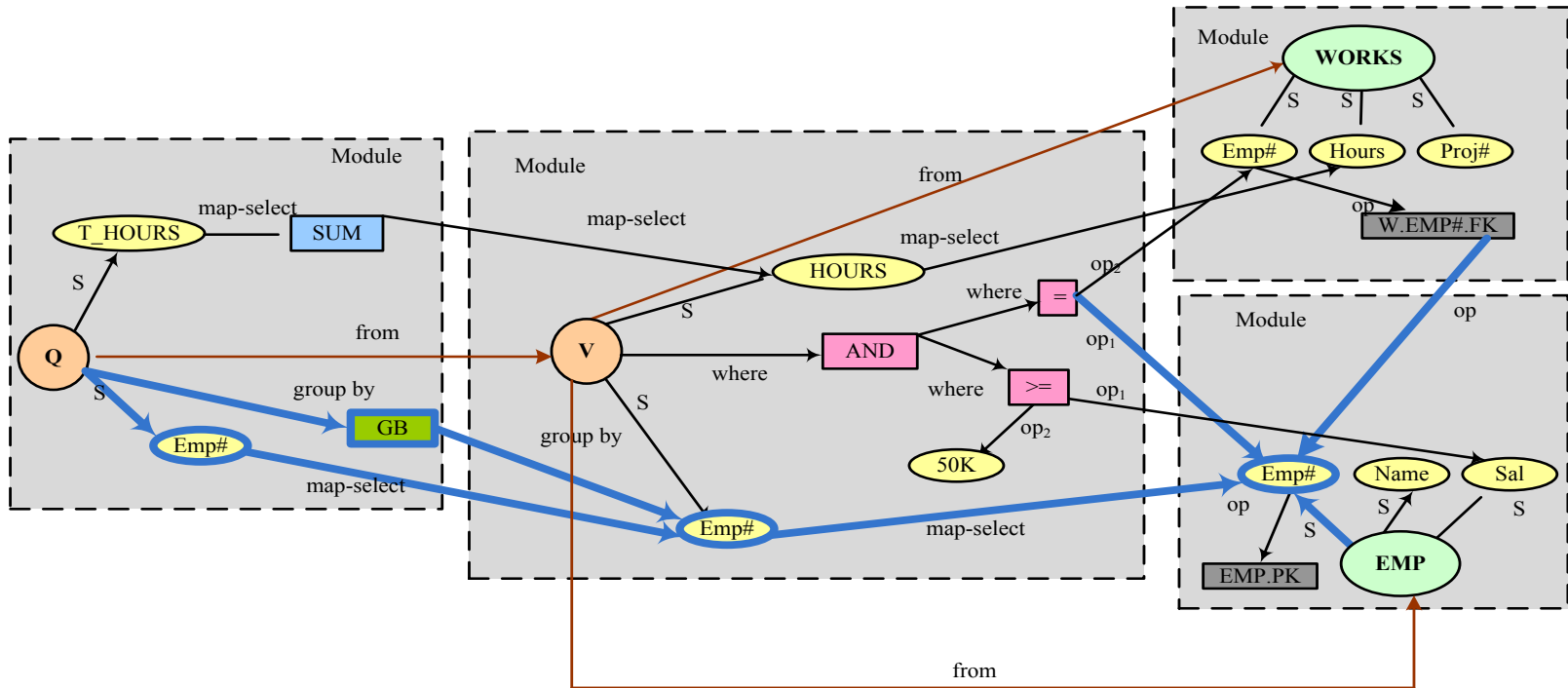


Edge direction:
from dependant
to depended upon

Metrics: Transitive Node Degree

Observe that there is both a **view** and a **query** with nodes dependent upon attribute *EMP.Emp#*.

Transitive Metrics:
in-degree, out-degree, degree

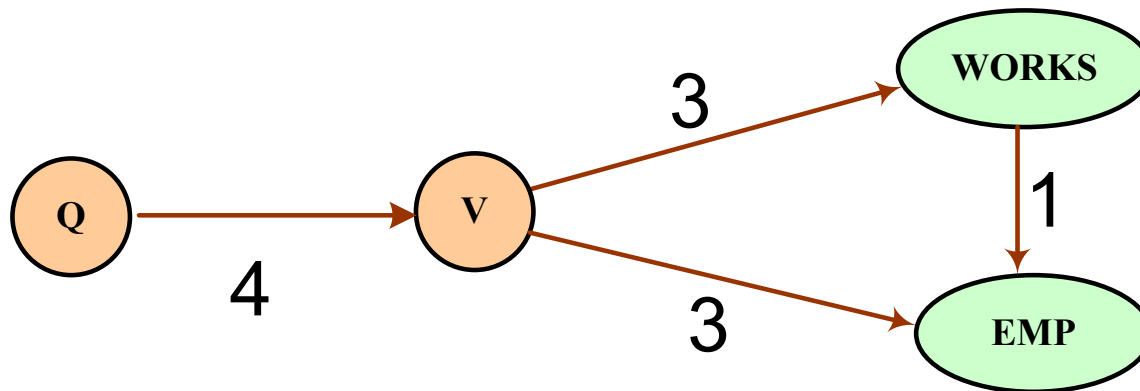


Strength: Zooming out to modules

A zoomed out graph highlights the dependence between modules (relations, queries, views), incorporating the detailed dependencies as the weight of the edges

Again, for modules, we can have both:

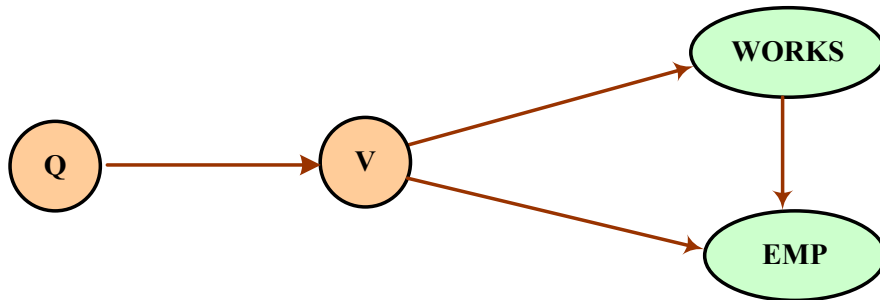
- Simple **strength**
- Transitive **strength**



Metrics: Node Entropy

The probability a node v being affected by an evolution event on node y_i :

$$P(v|y_k) = \frac{\text{paths}(v, y_k)}{\sum_{y_i \in V} \text{paths}(v, y_i)}, \text{ for all nodes } y_i \in V.$$



Examples

$$P(Q|V) = 1/4,$$

$$P(Q|EMP) = 2/4,$$

$$P(V|WORKS) = 1/3$$

Entropy of a node v : How sensitive the node v is by an arbitrary event on the graph.

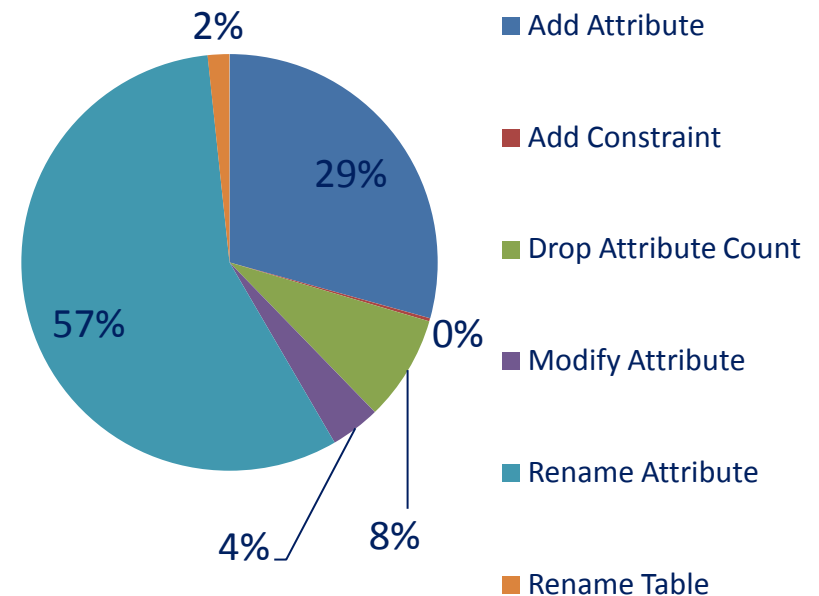
$$H(v) = - \sum_{y_i \in V} P(v | y_i) \log_2 P(v | y_i), \text{ for all nodes } y_i \in V.$$

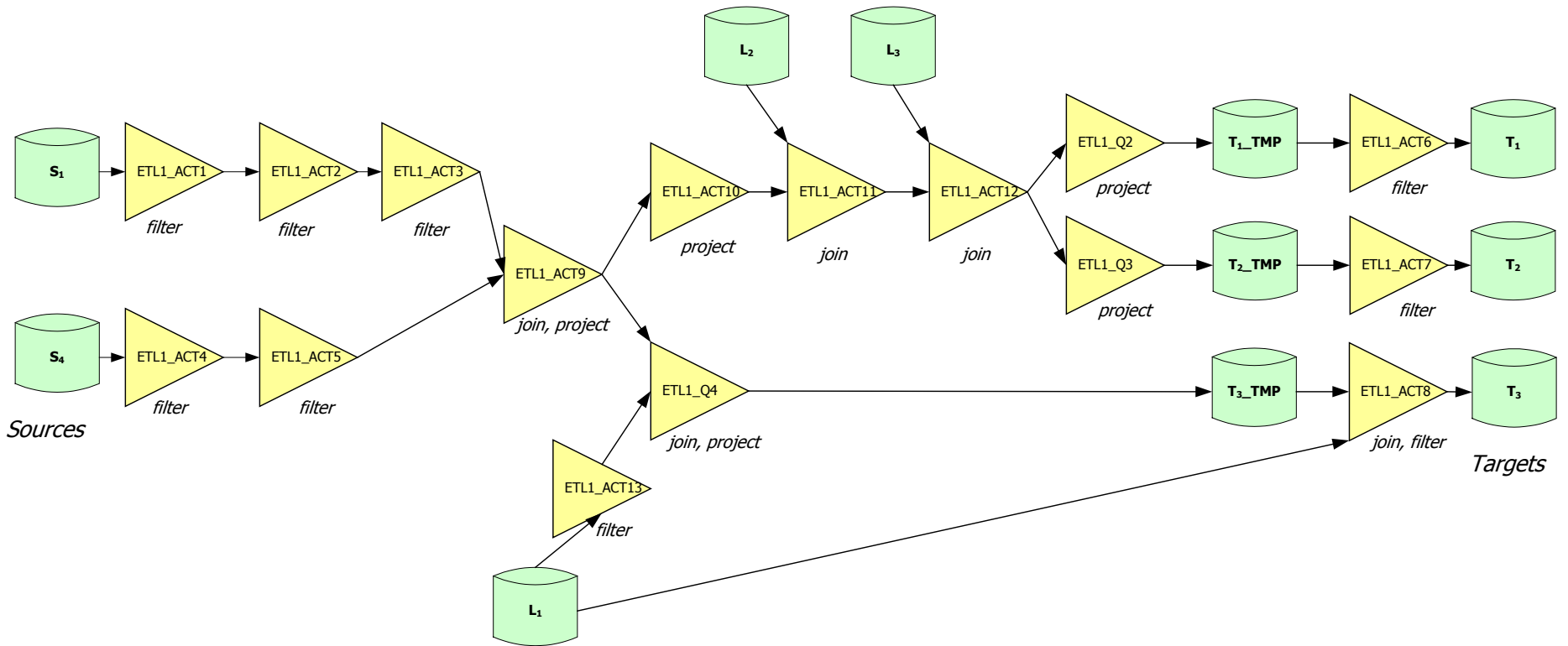
Macroscopic view

ATTN: change of requirements at the real world determines pct breakdown!!

	# tables affected	Occurrences	pct
Add Attribute	8	122	29%
Add Constraint	1	1	0%
Drop Attribute Count	5	34	8%
Modify Attribute	9	16	4%
Rename Attribute	5	236	57%
Rename Table	7	7	2%
		416	

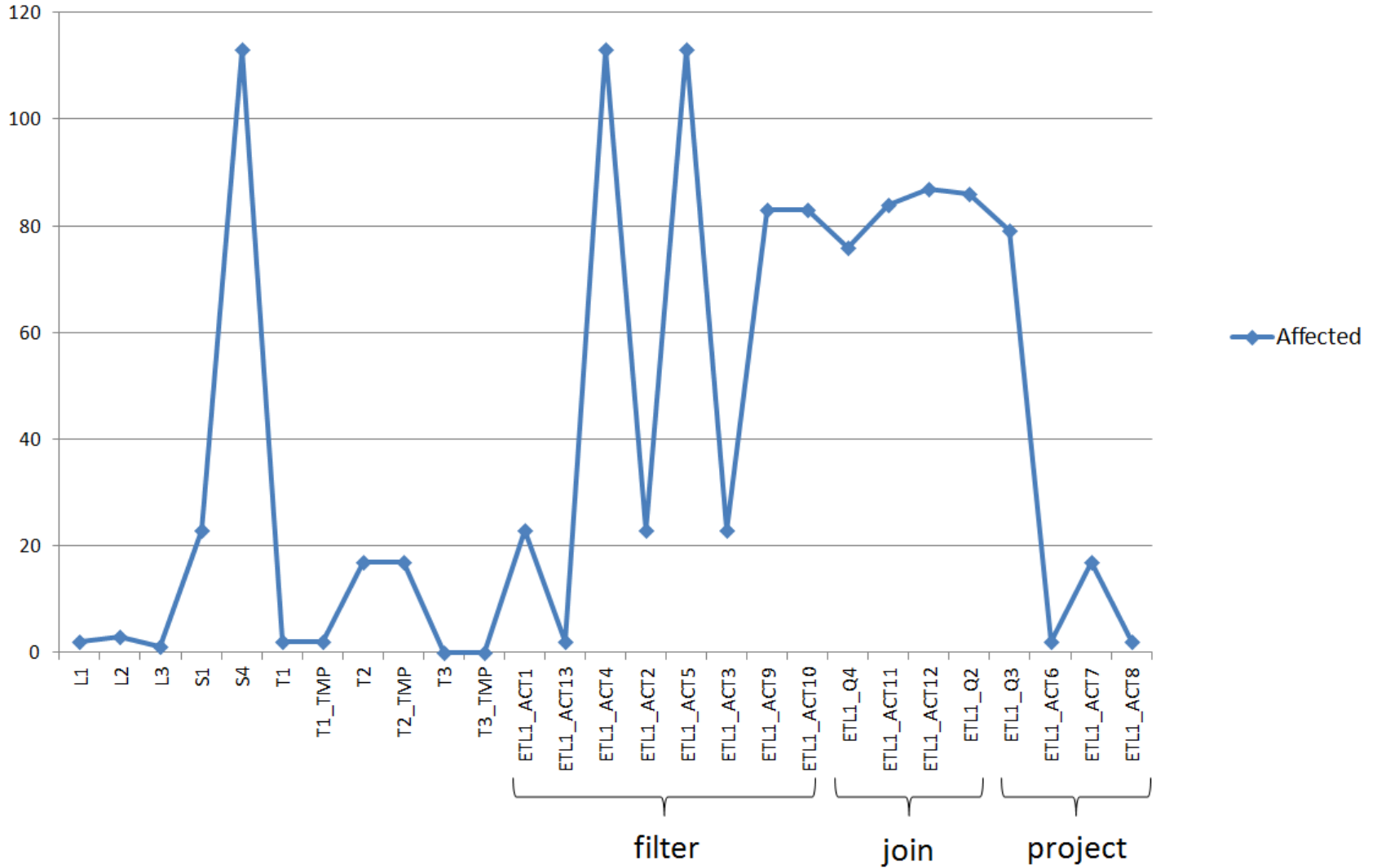
Breakdown per event type



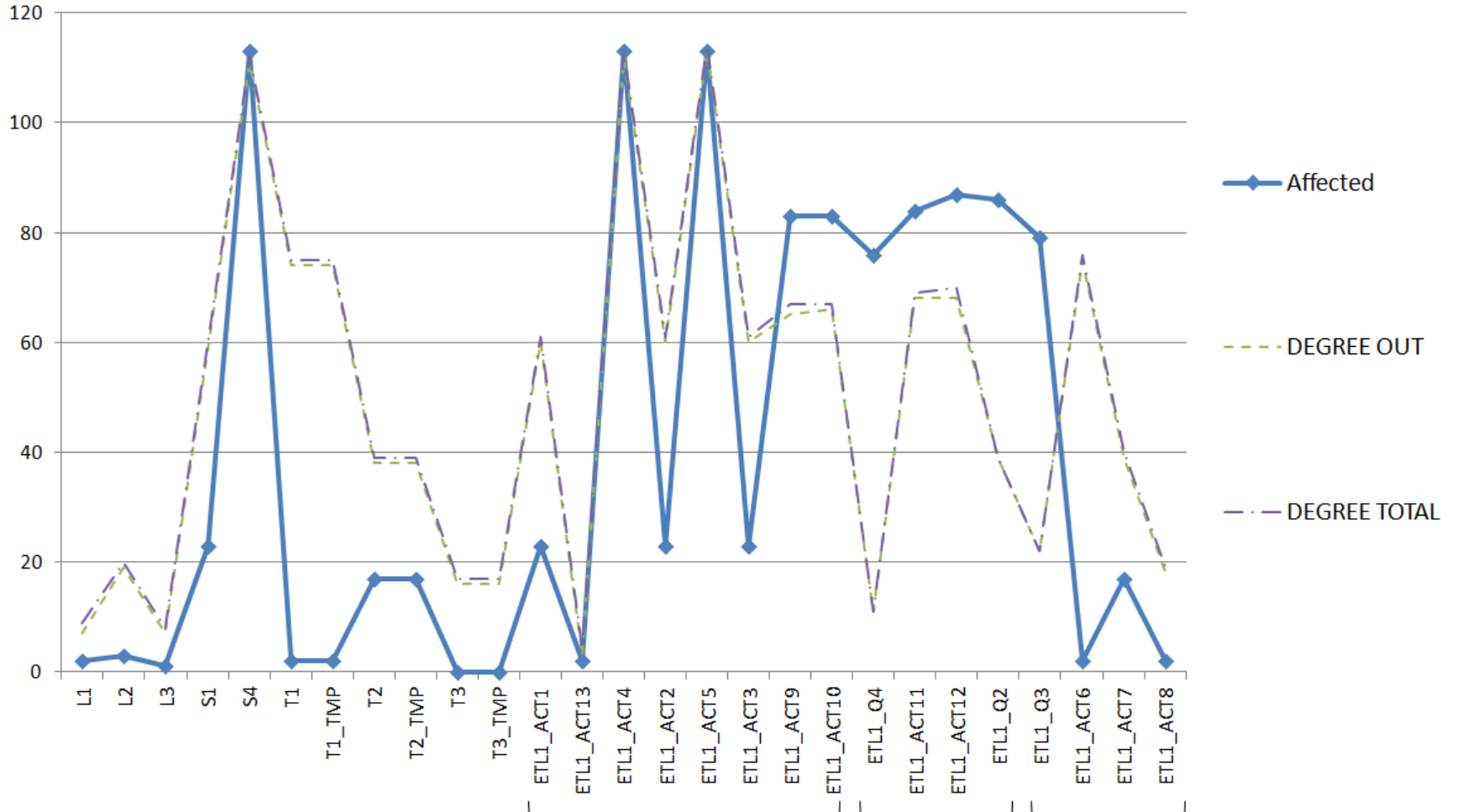


Workflow of the first ETL scenario, ETL1

ETL1 -- Actual # affected nodes vs Graph Metrics



ETL1 -- Actual # affected nodes vs Graph Metrics



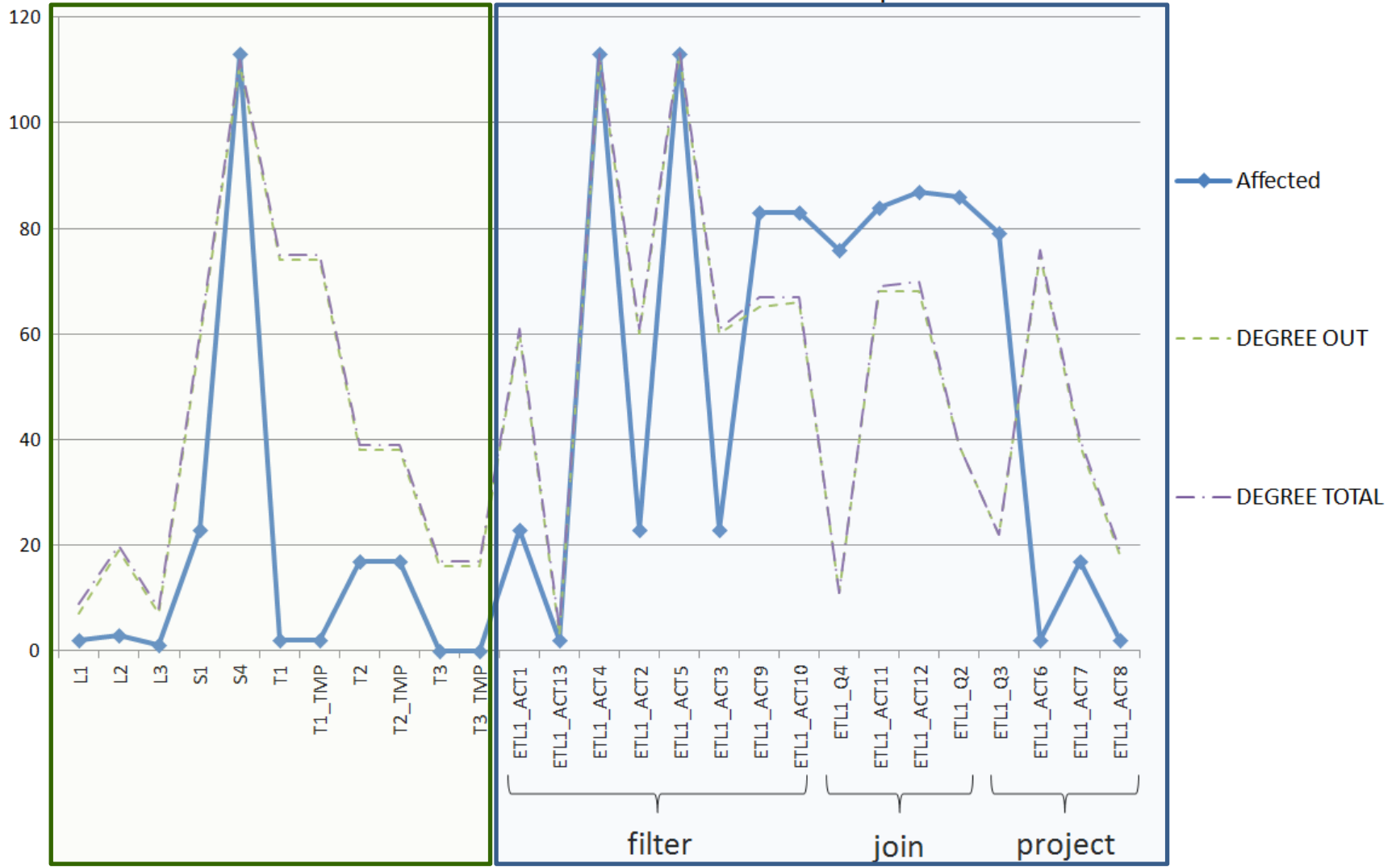
Out – degree
 - Schema size for tables
 - Output schema size for activities

filter

join

project

ETL1 -- Actual # affected nodes vs Graph Metrics

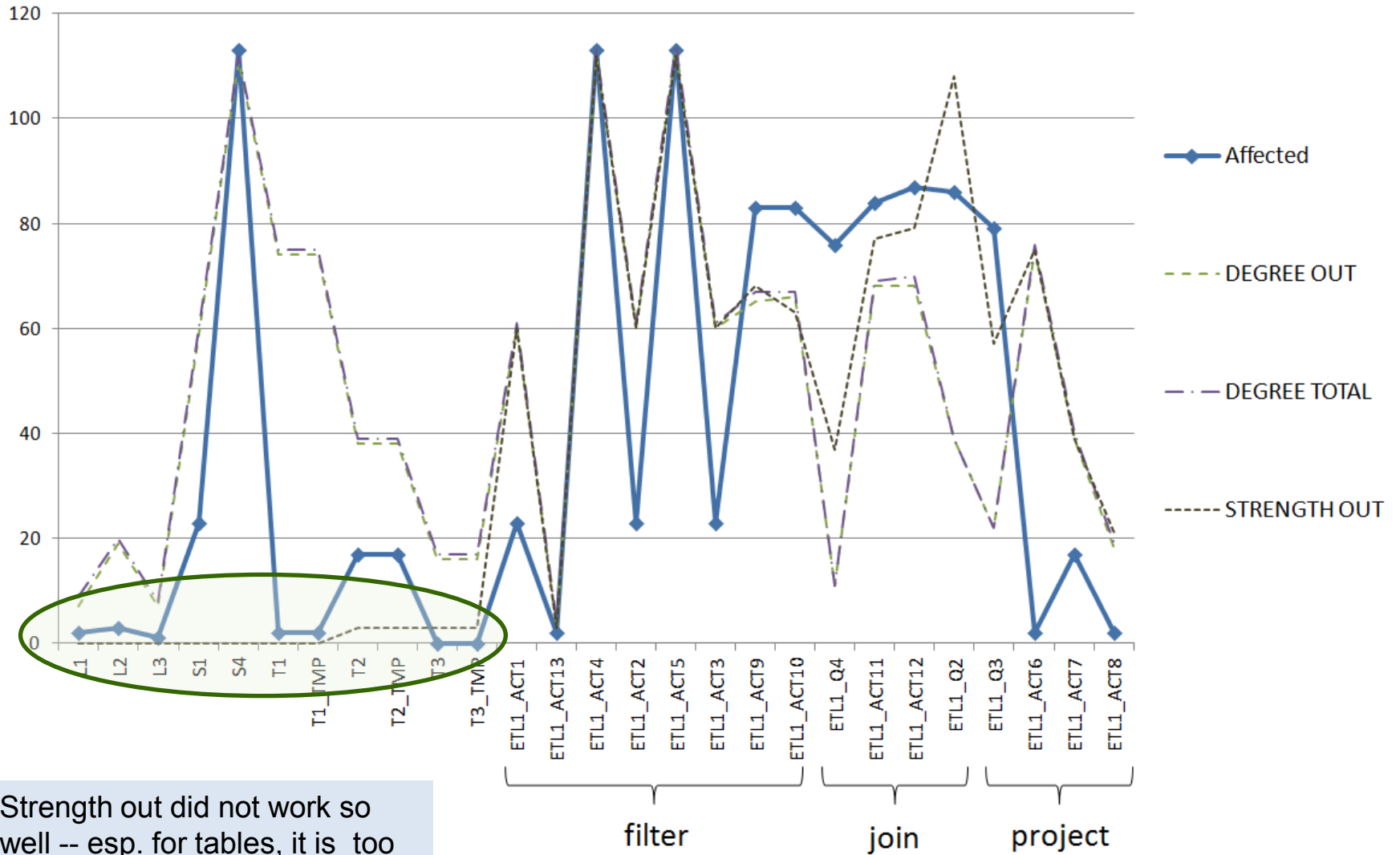


Pretty good job for tables

Decent job for filters and joins

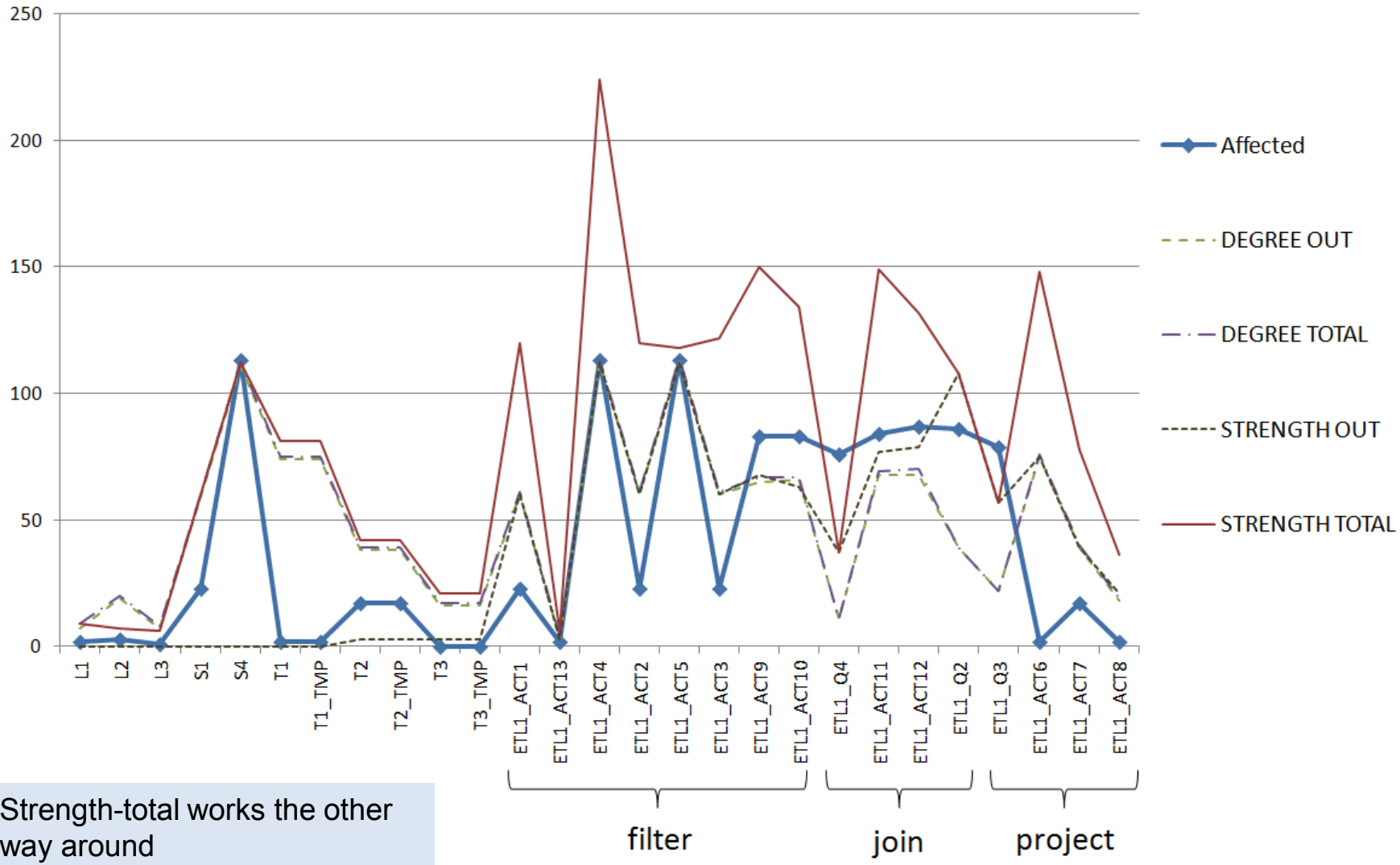
Not so good for projection activities

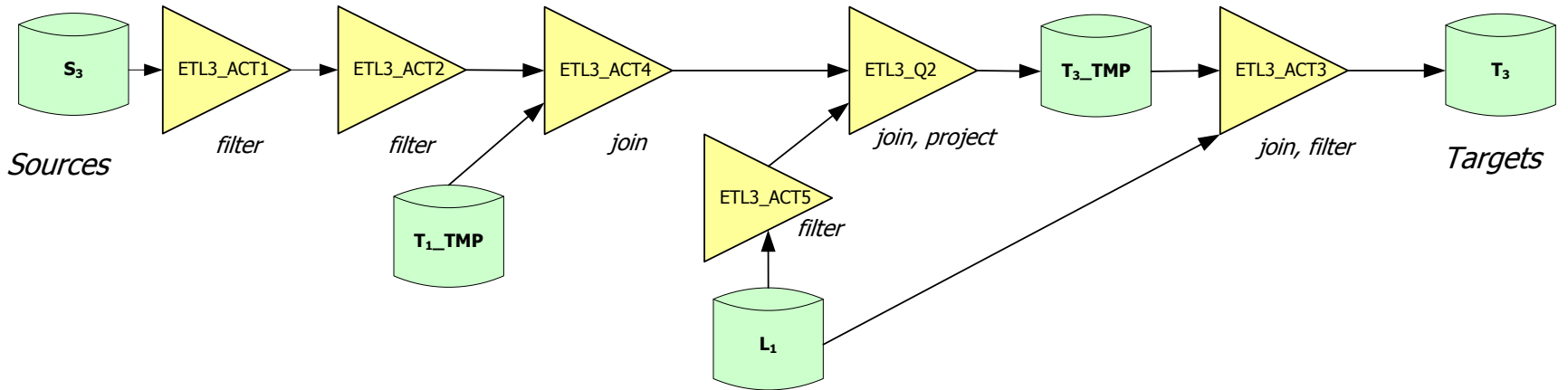
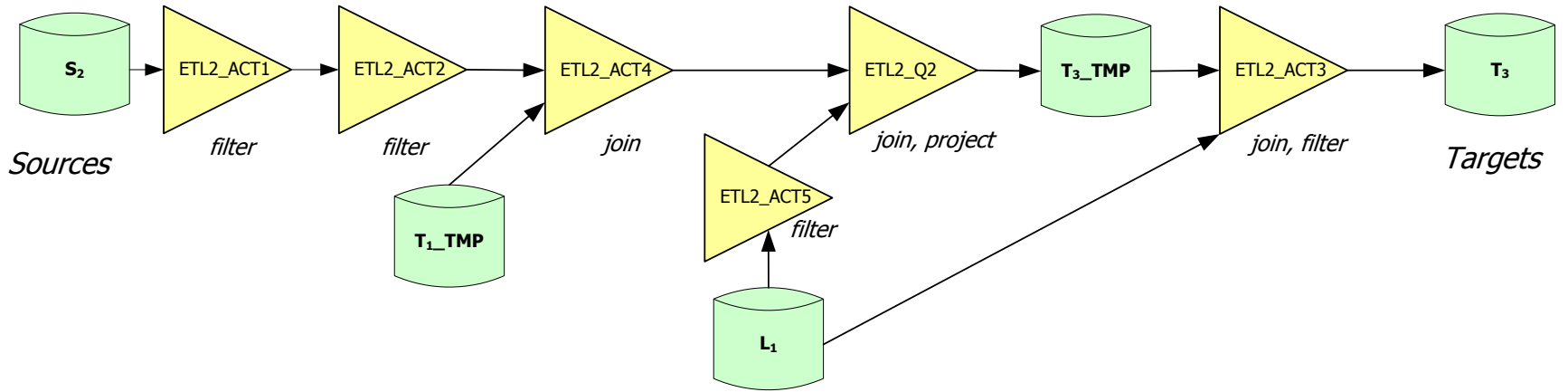
ETL1 -- Actual # affected nodes vs Graph Metrics



Strength out did not work so well -- esp. for tables, it is too bad

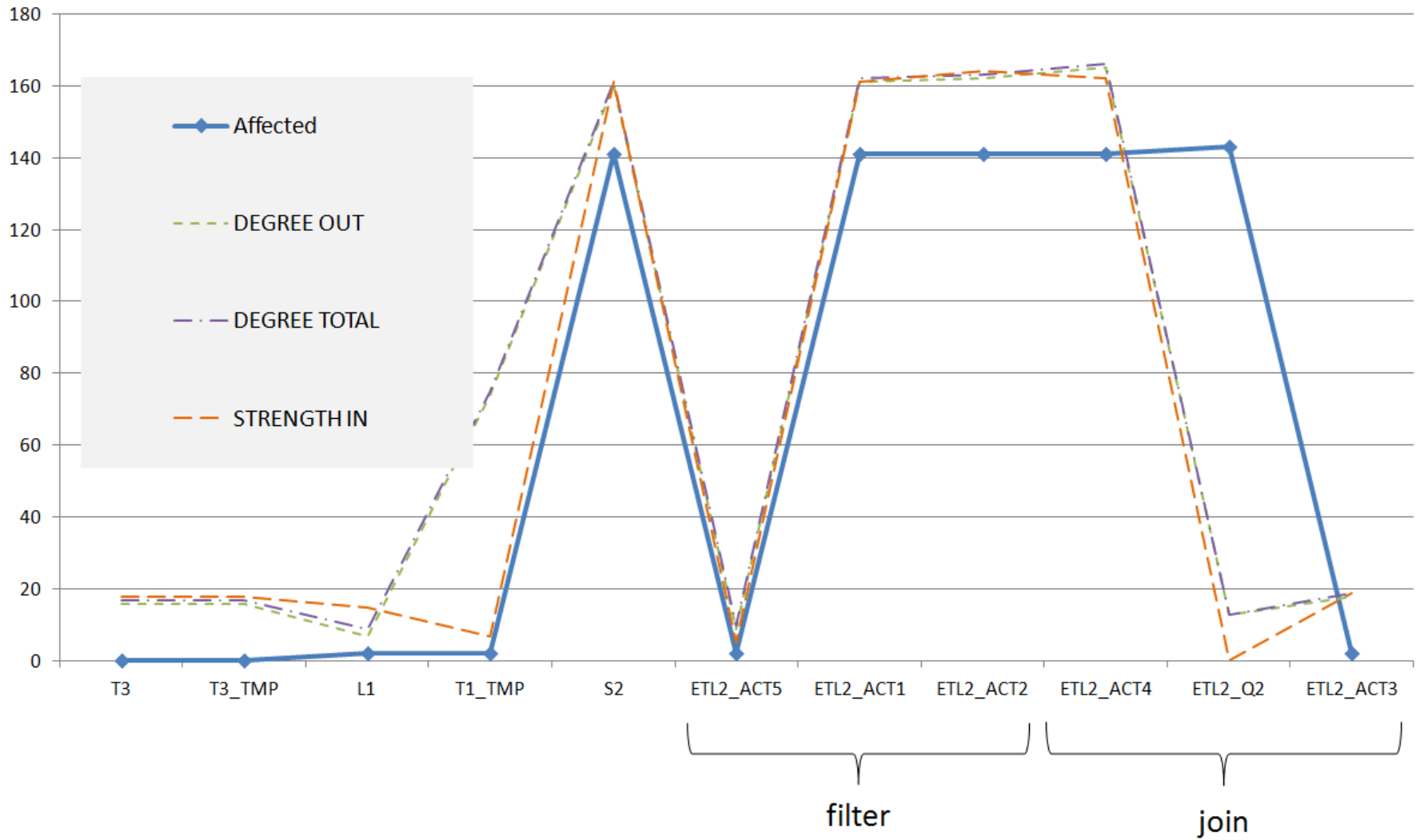
ETL1 -- Actual # affected nodes vs Graph Metrics

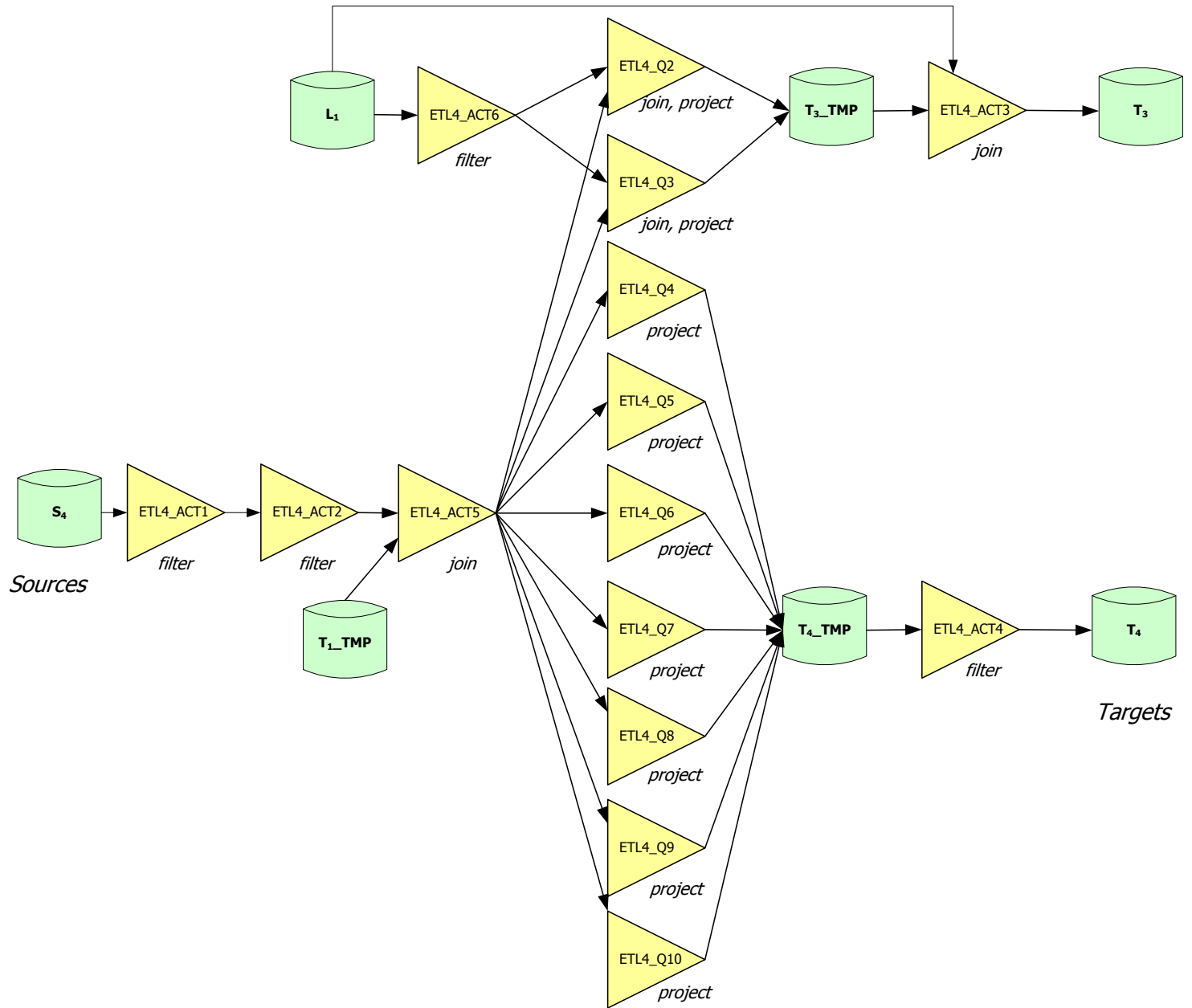




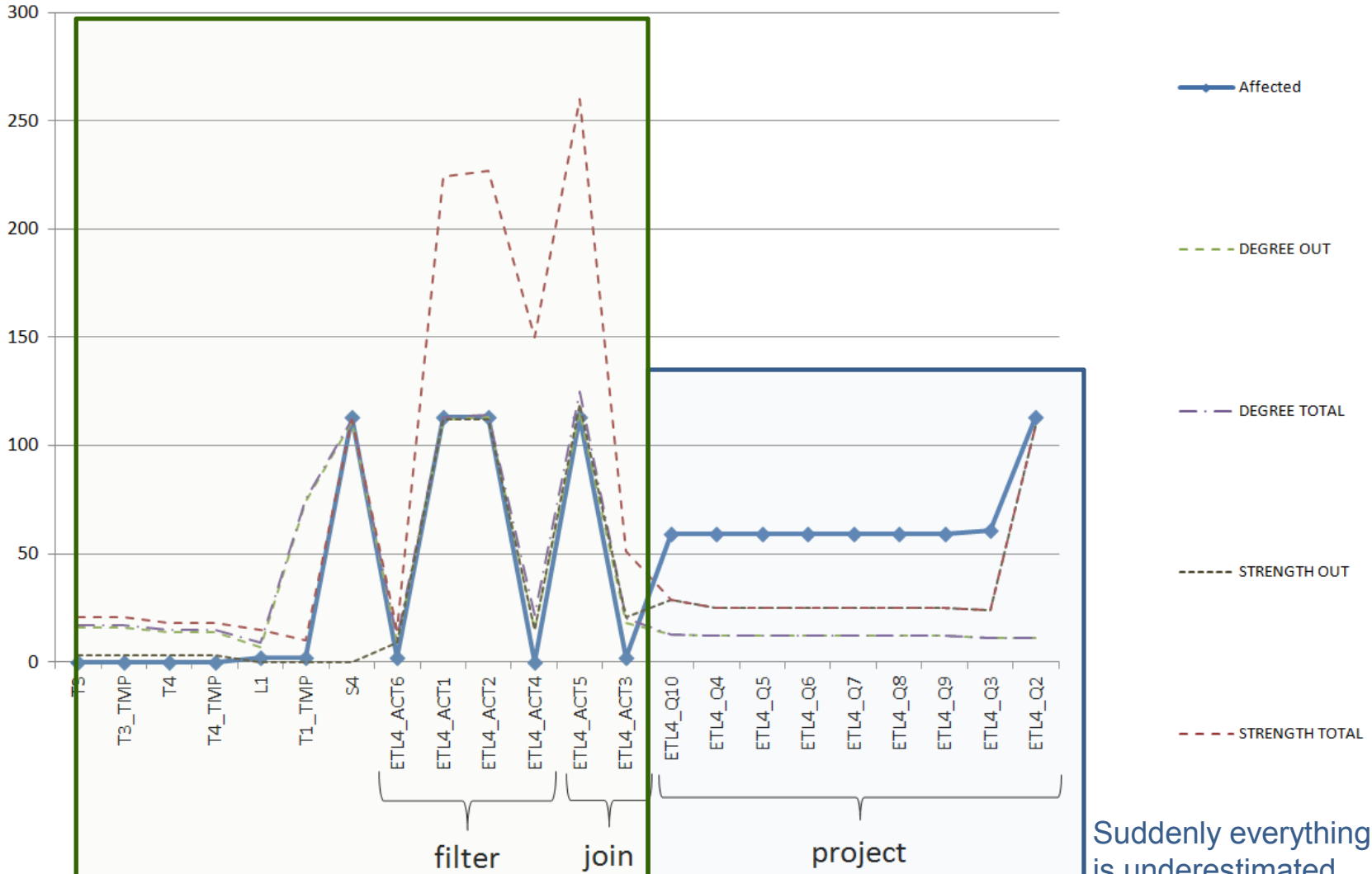
Workflows of the second & third ETL scenarios, ETL2 – ETL3

ETL2 -- Actual # affected nodes vs Graph Metrics





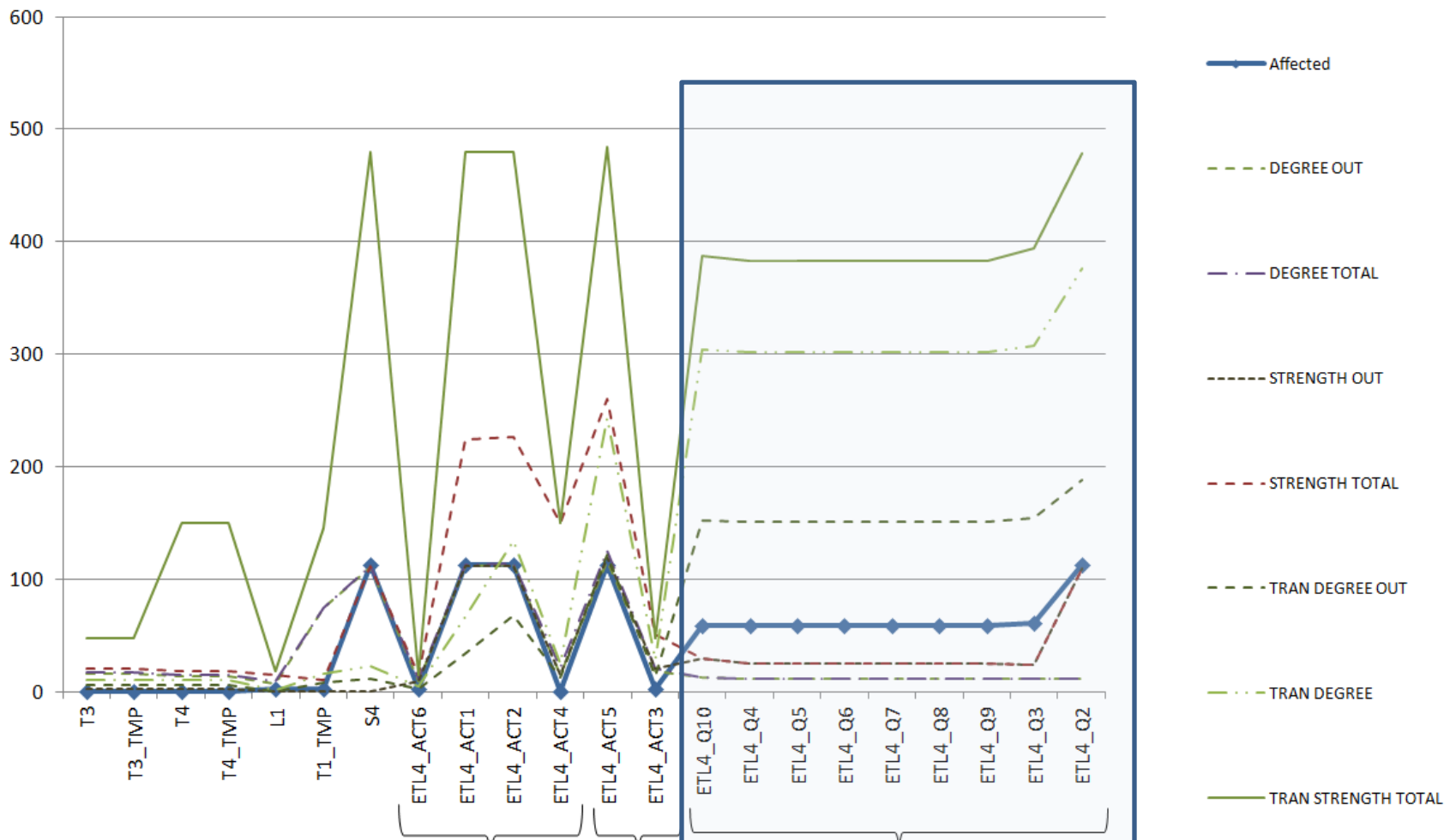
ETL4 -- Actual # affected nodes vs Graph Metrics



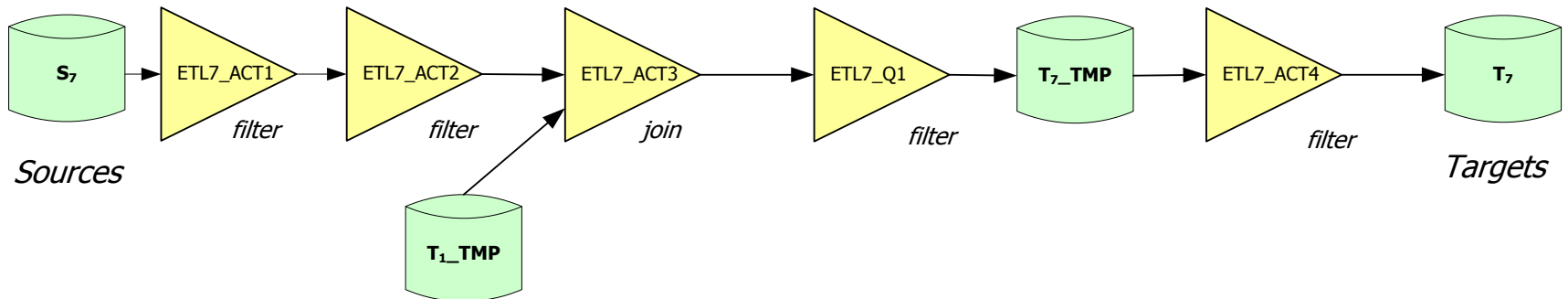
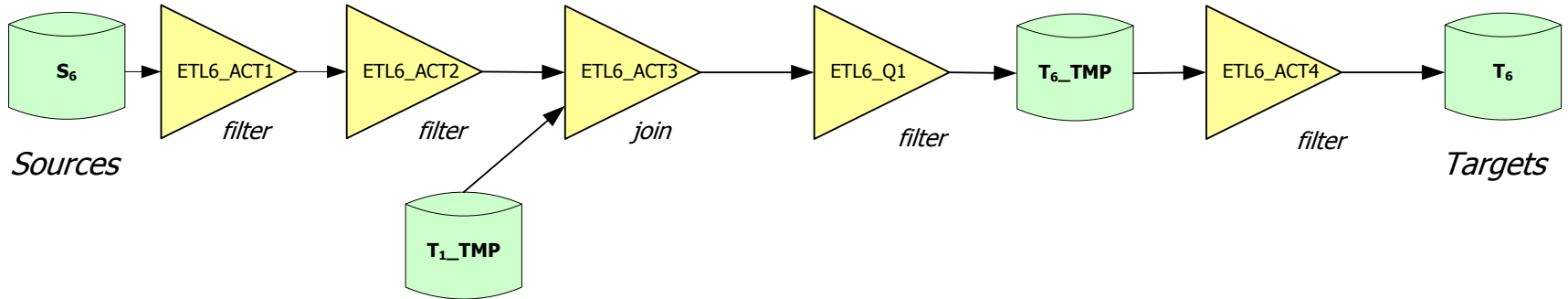
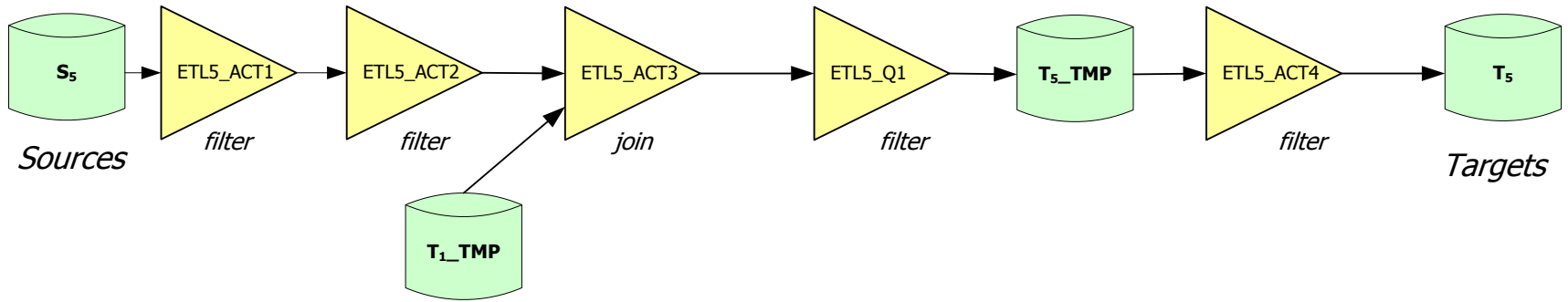
Suddenly everything is underestimated

Pretty good job in the left part

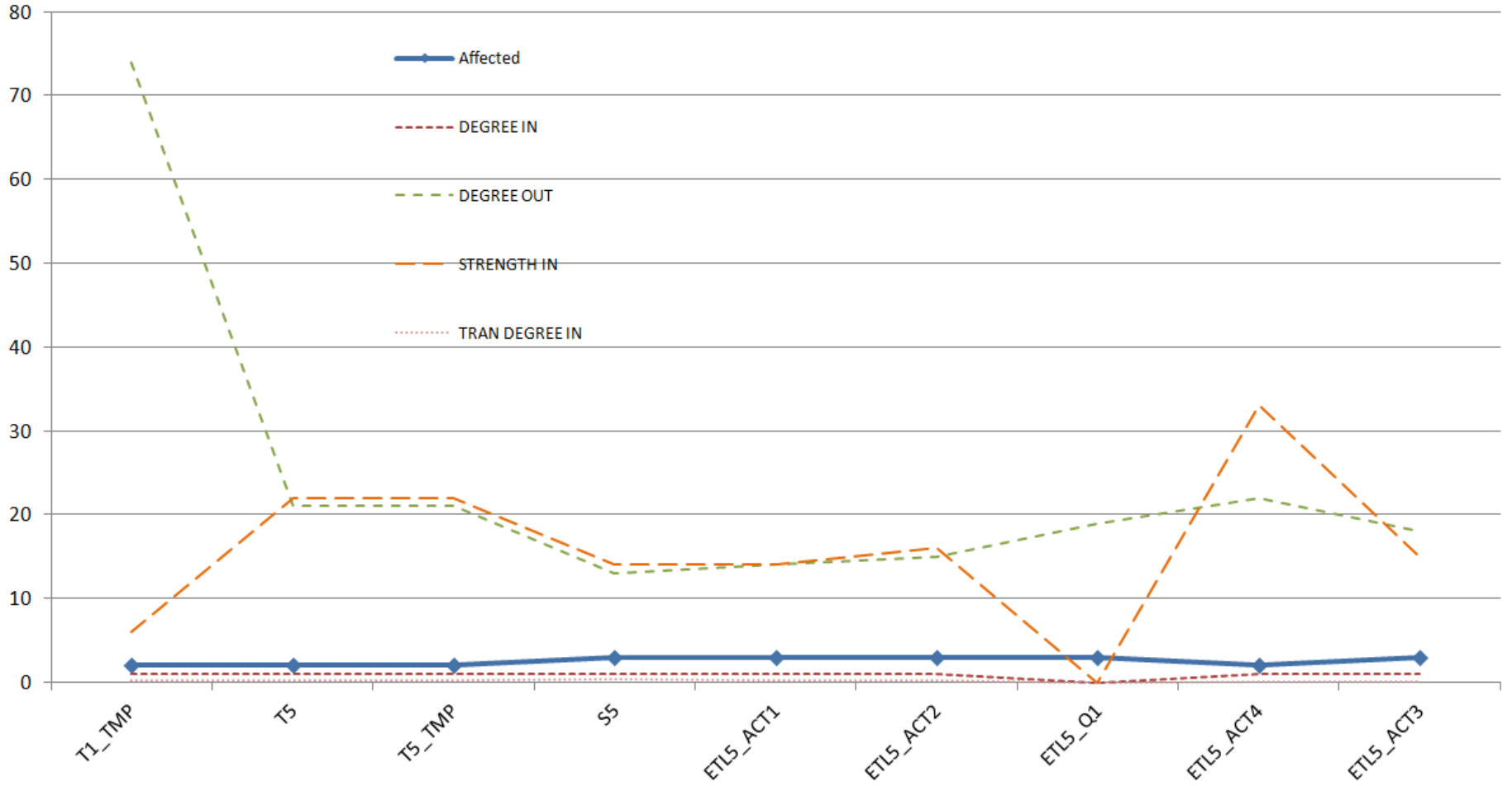
ETL4 -- Actual # affected nodes vs Graph Metrics



Transitive metrics to the rescue



ETL5 -- Actual # affected nodes vs Graph Metrics



Schema size and module complexity as predictors for the vulnerability of a system

- The **size of the schemas** involved in an ETL design significantly affects the design vulnerability to evolution events.
 - For example, source or intermediate tables with many attributes are more vulnerable to changes at the attribute level.
 - The **out-degree** captures the **projected attributes** by an activity, whereas the **out-strength** captures the total number of dependencies between an activity and its sources.
- The **internal structure of an ETL activity** plays a significant role for the impact of evolution events on it.
 - Activities with **high out-degree and out-strengths** tend to be more vulnerable to evolution
 - Activities performing **attribute reduction** (e.g., through either a group-by or a projection operation) are in general, less vulnerable to evolution events.
 - **Transitive degree and entropy metrics** capture the dependencies of a module with its various non-adjacent sources. Useful for activities which act as “hubs” of various different paths from sources in complex workflows.
- The **module-level design** of an ETL flow also affects the overall evolution impact on the flow.
 - For example, it might be worthy to **place schema reduction activities early in an ETL flow** to restrain the flooding of evolution events.

Summary & Guidelines

ETL Construct	Most suitable Metric	Heuristic
Source Tables	<i>out-degree</i>	<i>Retain small schema size</i>
Intermediate & Target Tables	<i>out-degree</i>	<i>Retain small schema size in intermediate tables</i>
Filtering activities	<i>out-degree, out-strength</i>	<i>Retain small number of conditions</i>
Join Activities	<i>out-degree, out-strength, trans. out-degree, trans. out-strength, entropy</i>	<i>Move to early stages of the workflow</i>
Project Activities	<i>out-degree, out-strength, trans. out-degree, trans. out-strength, entropy</i>	<i>Move attribute reduction activities to early stages of the workflow and attribute increase activities to later stages</i>

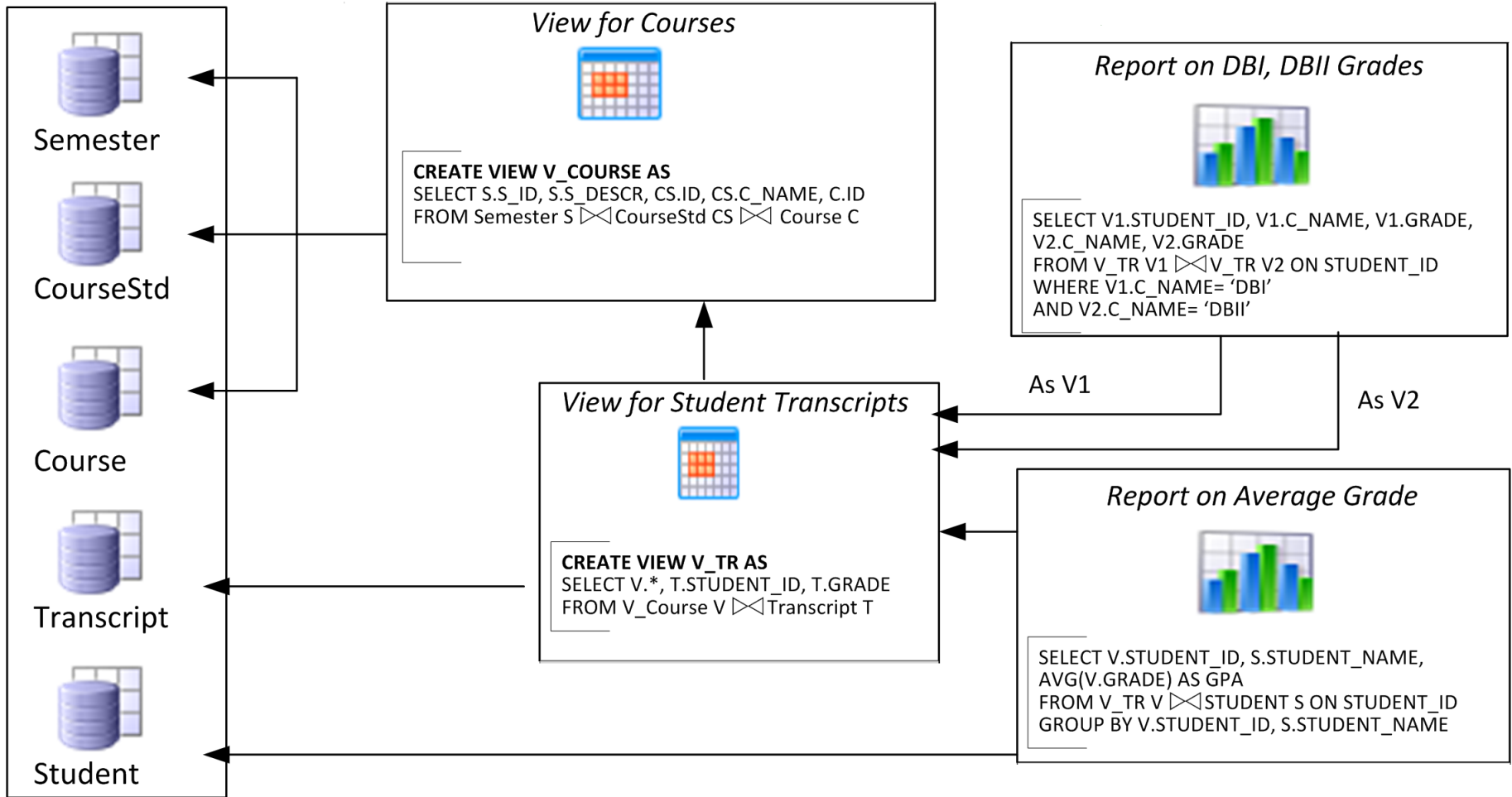
... and data intensive ecosystems...

IMPACT ASSESSMENT

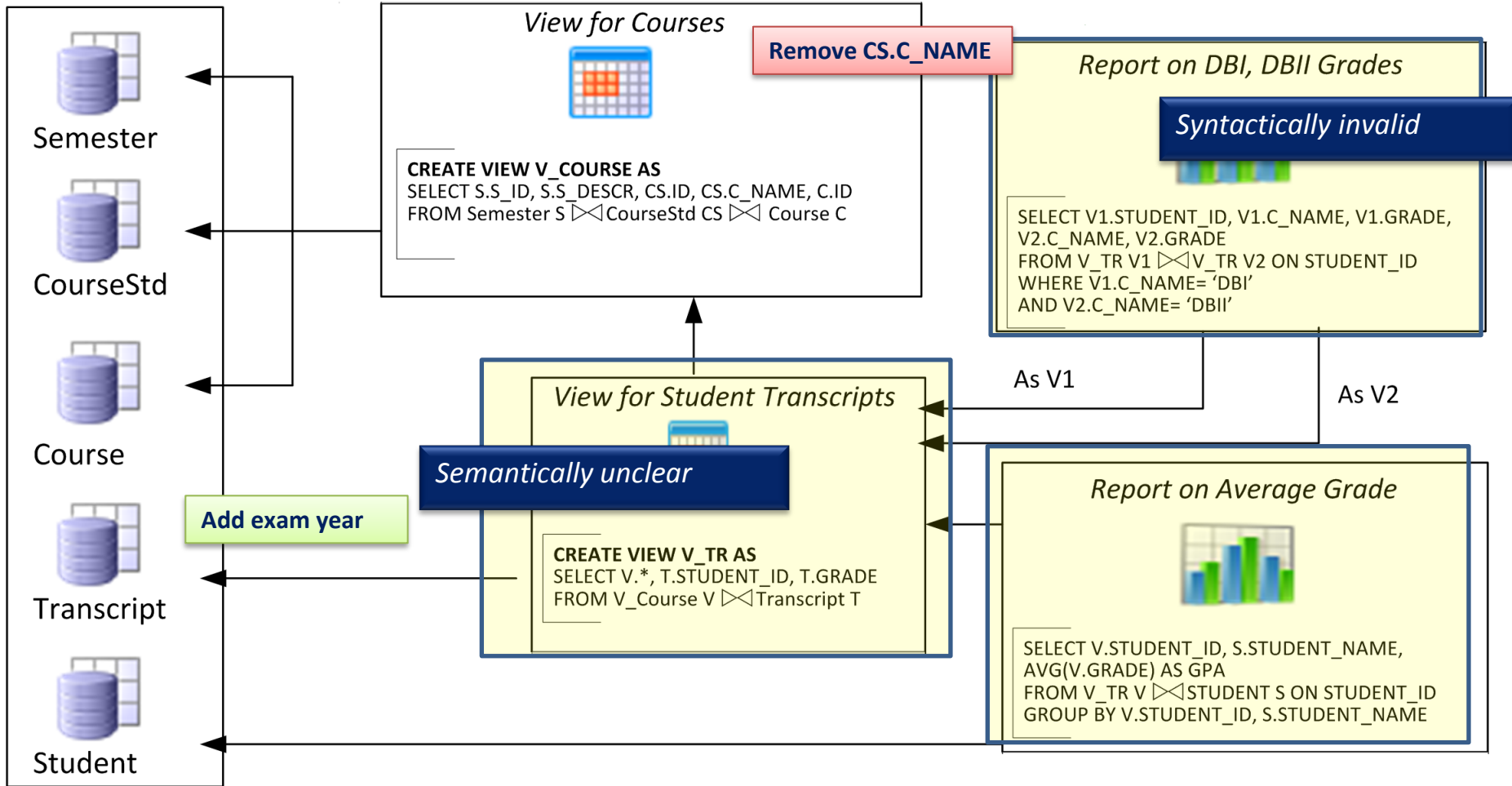
Data intensive ecosystems

- Ecosystems of **applications**, built on top of one or more **databases** and strongly dependent upon them
- When the database changes, the applications are affected
 - Syntactically
 - Semantically

Evolving data-intensive ecosystem



Evolving data-intensive ecosystem



The impact can be **syntactical** (causing crashes), **semantic** (causing info loss or inconsistencies) and related to the performance

What happens if I modify table search_index? Who are the neighbors?

The screenshot displays the HECATAEUS application interface for a Drupal project. The main window is divided into several panels:

- Static Panel:** Shows a "Summary Graph" of the network structure.
- Visual Panel:** Shows a zoomed-in view of the network graph. A dialog box titled "Input" is overlaid on this panel, containing the text "The name of nodes to find (separated with .):" and a text input field with the value "search_index". The dialog has "Cancel" and "OK" buttons.
- File system Panel:** Shows a tree view of the project's file system, including folders like "includes" and "modules", and files like "actions.inc", "batch.inc", etc.
- Policy Panel:** Contains tabs for "Policy" and "Event". It shows a list of policies: "POLICIES/DefaultPolicy.plc" and "POLICIES/other.plc". Below this are buttons for "Load", "New", and "Save". There are also fields for "Set the event type:" and "Set the policy type:" with dropdown menus, and "Add policy" and "Delete policy" buttons.

What happens if I modify table search_index? Who are the neighbors?

The screenshot displays the HECATAEUS interface for a Drupal project. The main window shows a network graph with nodes representing various components. The 'SEARCH_INDEX' node is selected, and a tooltip provides the following information:

```
MODULE
From file /home/pmanousi/git/Hecataeus/AppData/drupalProject/SQLS/modules/search/search.module
SQL Definition
SELECT SUM(SCORE) FROM SEARCH_INDEX WHERE WORD = 0;
Line: 5
Status: NO_STATUS
```

The 'Information Area' on the right side of the interface shows the following information:

```
Scripts using relation SEARCH_INDEX:
/modules/search/search.module
```

Red arrows point to the tooltip and the 'Information Area' respectively.

Tooltips with info on the script & query

In the file structure too...

The screenshot displays the HECATAEUS - drupalProject interface. The main window is divided into three sections:

- Static:** Contains a 'Summary Graph' showing a complex network of nodes and edges. Below it are 'Policy' and 'Event' tabs, a 'Pick a node' input field, and a 'Selected Node: SEARCH_INDEX_SCHEMA.WORD' dropdown. At the bottom are buttons for 'Highlight Impact', 'Pick file of events', and 'Play events'.
- Visual:** Shows a zoomed-in view of the 'SEARCH_INDEX' node. The graph includes nodes labeled 'SYSTEM', 'MYNODE', 'SEARCH_TOTAL', 'SEARCH_INDEX', 'REGISTRY_FILE', 'ACTIONS', 'MYNODE_TYPE', and 'MYBLOC_R'. The 'SEARCH_INDEX' node is highlighted with a red arrow.
- File system:** A tree view of the project's file structure. A red arrow points to the 'search' directory, which contains 'search.api.php' and 'search.module'. The 'search.module' file is highlighted.

The 'Information Area' at the bottom right is currently empty.

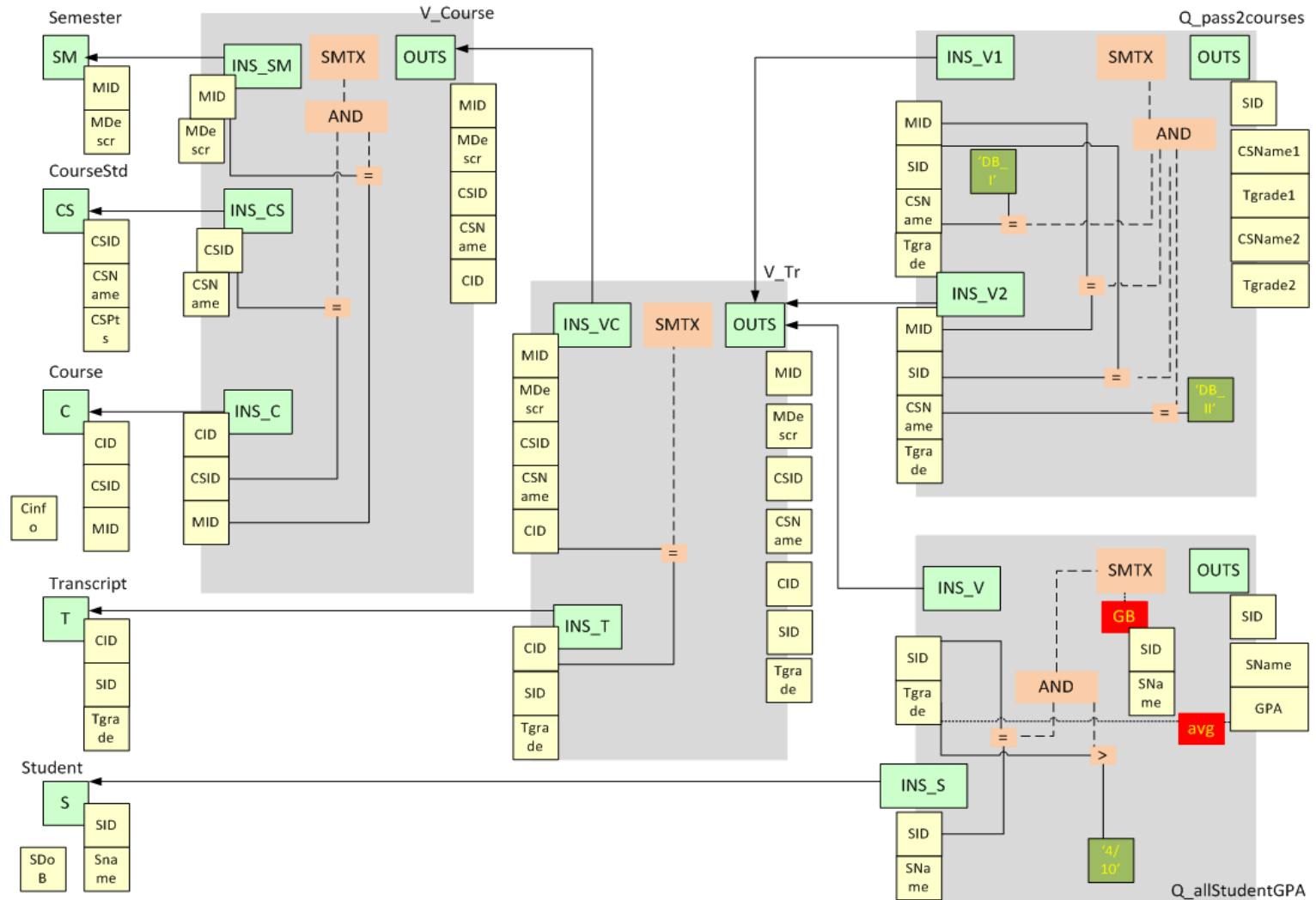
How to handle evolution?



- **Architecture Graphs**: graph with the data flow between modules (i.e., relations, views or queries) at the detailed (attribute) level; module internals are also modeled as subgraphs of the Architecture Graph
- **Policies**, that annotate a module with a reaction for each possible event that it can withstand, in one of two possible modes:
 - (a) **block**, to veto the event and demand that the module retains its previous structure and semantics, or,
 - (b) **propagate**, to allow the event and adapt the module to a new internal structure.
- Given a potential change in the ecosystem
 - we **identify which parts of the ecosystem are affected** via a “change propagation” algorithm
 - we **rewrite the ecosystem to reflect the new version** in the parts that are affected and do not veto the change via a rewriting algorithm
 - Within this task, we **resolve conflicts** (different modules dictate conflicting reactions) via a conflict resolution algorithm

Manousis+ @ ER 2013 for the details of impact analysis (summary coming)
ER 2014 for the visualization (not here)

University E/S Architecture Graph

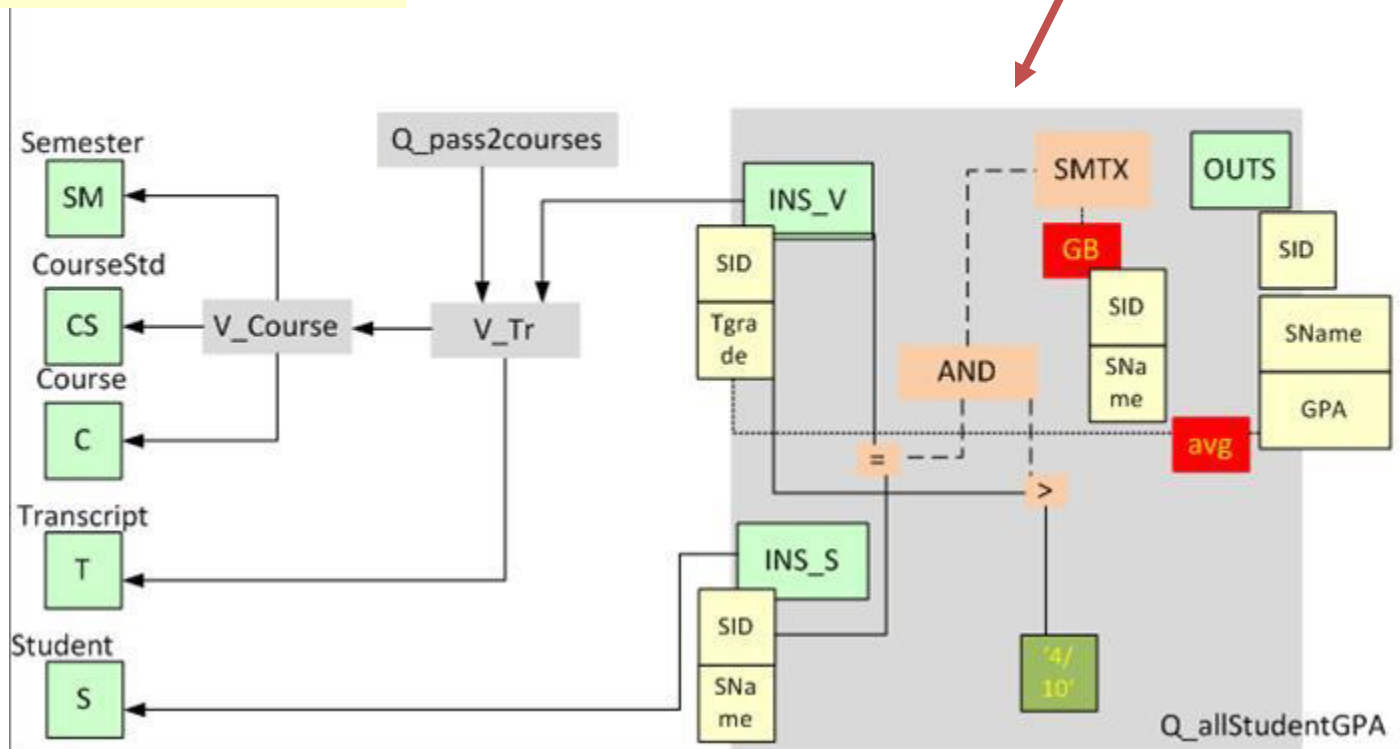


Architecture Graph

Modules and Module Encapsulation

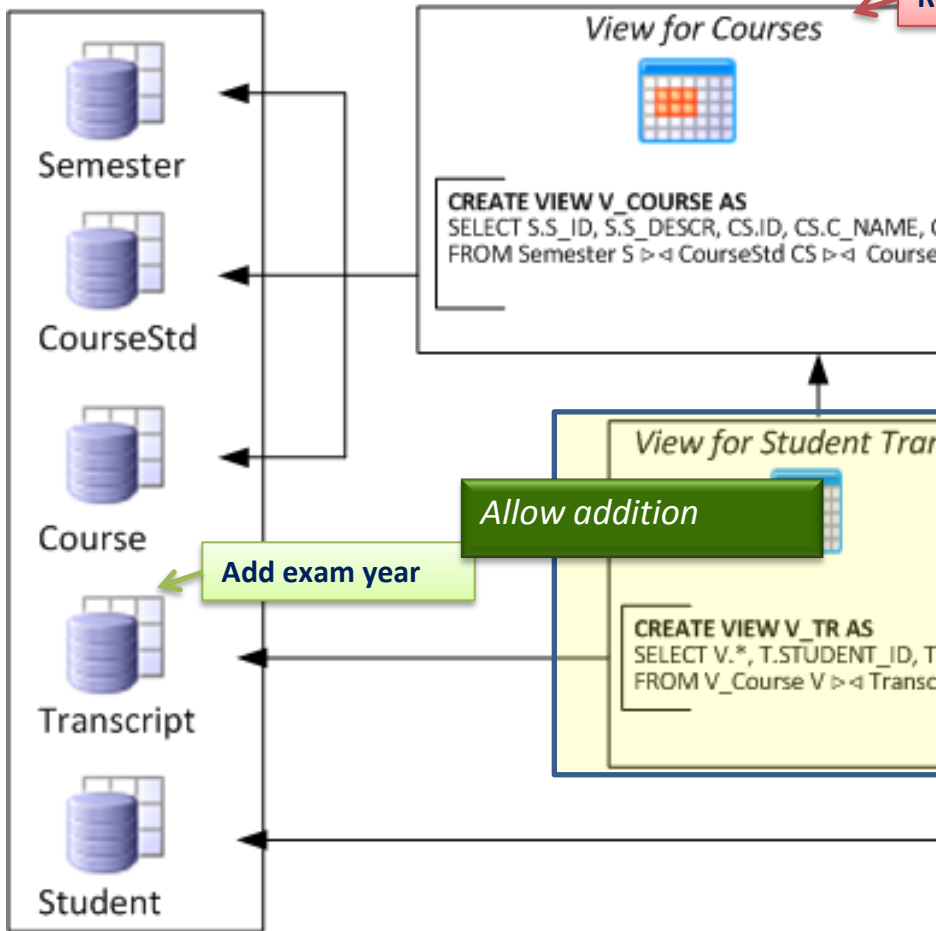
Observe the input and output schemata!!

```
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
```



Policies to predetermine reactions

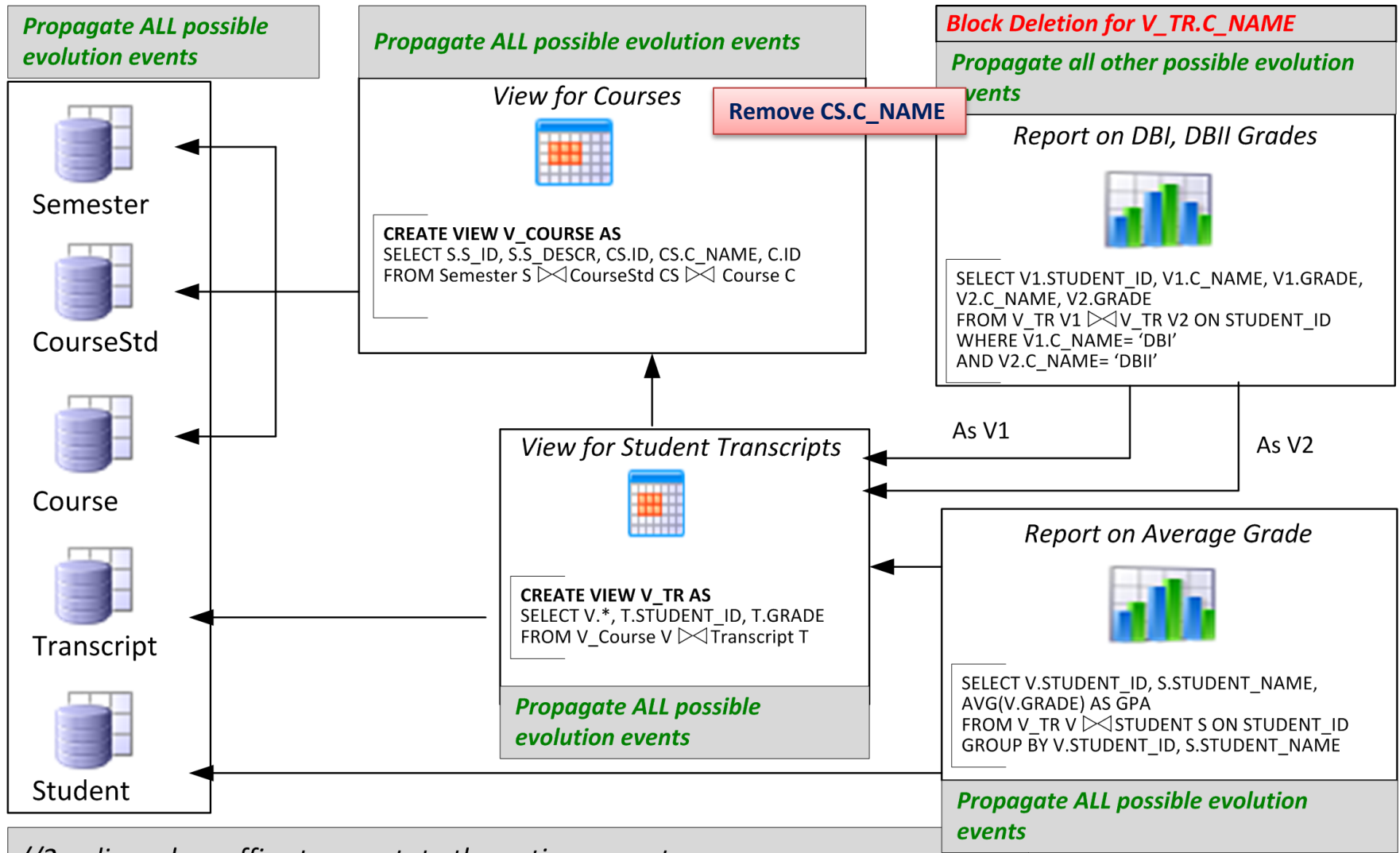
University DB



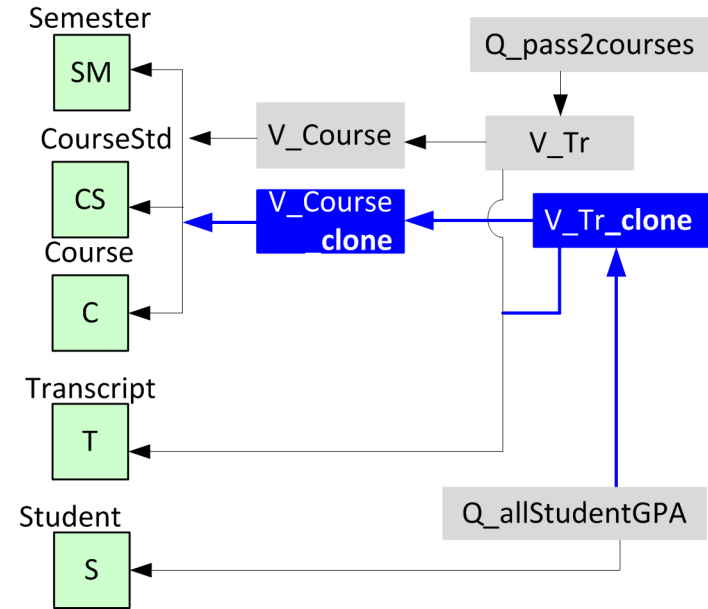
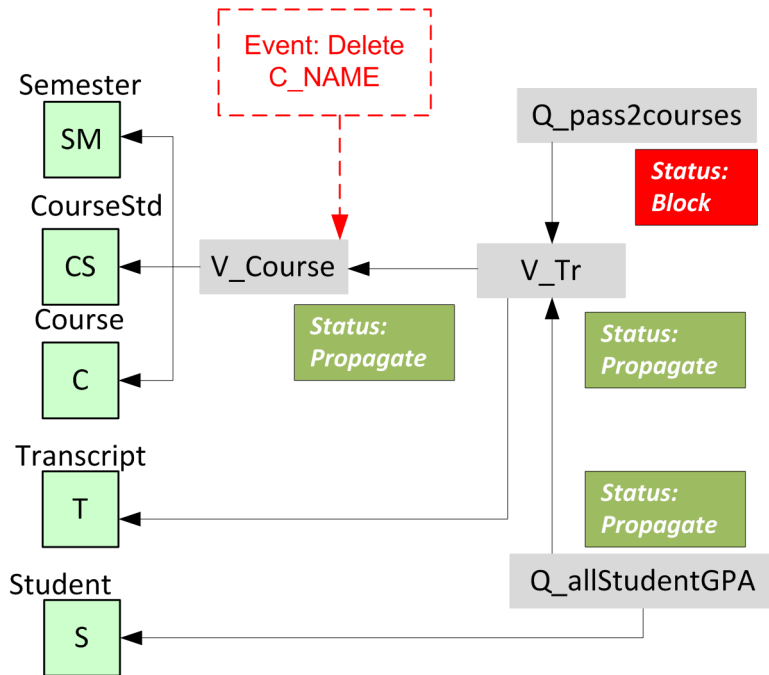
RELATION.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
 RELATION.OUT.SELF: on DELETE_SELF then PROPAGATE;
 RELATION.OUT.SELF: on RENAME_SELF then PROPAGATE;
 RELATION.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
 RELATION.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
 VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
 VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
 VIEW.OUT.SELF: on DELETE_SELF then PROPAGATE;
 VIEW.OUT.SELF: on RENAME_SELF then PROPAGATE;
 VIEW.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
 VIEW.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
 VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
 VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
 VIEW.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
 VIEW.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
 VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
 VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
 VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
 VIEW.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
 QUERY.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
 QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
 QUERY.OUT.SELF: on DELETE_SELF then PROPAGATE;
 QUERY.OUT.SELF: on RENAME_SELF then PROPAGATE;
 QUERY.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
 QUERY.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
 QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
 QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
 QUERY.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
 QUERY.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
 QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
 QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
 QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
 QUERY.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;

Policies to predetermine the modules' reaction to a hypothetical event?

How to handle evolution?



Impact assessment & rewriting

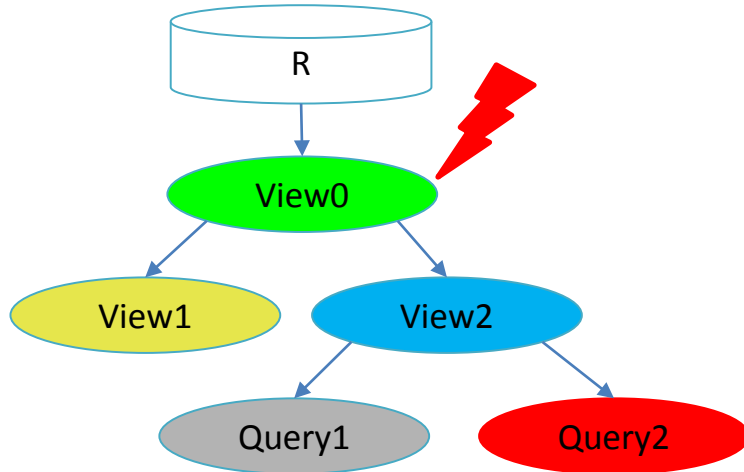


Internals of impact assess. & rewriting

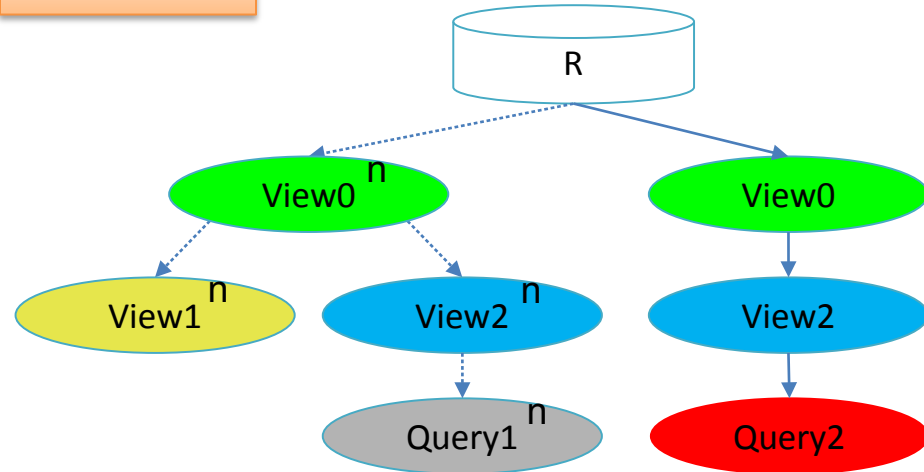
- 1. Impact assessment.** Given a potential event, a status determination algorithm makes sure that the nodes of the ecosystem are assigned a status concerning (a) whether they are affected by the event or not and (b) what their reaction to the event is (block or propagate).
- 2. Conflict resolution and calculation of variants.** Algorithm that checks the affected parts of the graph in order to highlight affected nodes with whether they will adapt to a new version or retain both their old and new variants.
- 3. Module Rewriting.** Our algorithm visits affected modules sequentially and performs the appropriate restructuring of nodes and edges.

Conflicts: what they are and how to handle them (more than flooding)

BEFORE



AFTER



- View0 initiates a change
- View1 and View 2 accept the change
- Query2 rejects the change
- Query1 accepts the change

- The path to Query2 is left intact, so that it retains its semantics
- View1 and Query1 are adapted
- View0 and View2 are adapted too, however, we need two versions for each: one to serve Query2 and another to serve View1 and Query1

Played an impact analysis scenario: delete attr. 'word' from search_index

The screenshot displays the HECATAEUS impact analysis tool interface. The main window is titled "Visual" and shows a query graph with nodes and connections. A red arrow points to the text "1. The table allowed the deletion, but..." and a black arrow points to the text "2. Queries Q215 and Q216 vetoed".

1. The table allowed the deletion, but...

2. Queries Q215 and Q216 vetoed

The interface includes a "Static" panel on the left with a "Summary Graph" and a "Policy" tab. The "Visual" panel has tabs for "SEARCH_INDEX" and "Impact analysis". The "File system" panel on the right shows a directory structure with "search.module" selected. The "Information Area" at the bottom right lists nodes affected by the change:

```
Nodes that were affected by the change:  
AND . =  
Q215_SMTX. AND  
SEARCH_INDEX_SCHEMA.WORD  
Q215  
SEARCH_INDEX.SEARCH_INDEX_SCHEMA  
Q215.Q215_SMTX  
AND . =  
Q216_IN_SEARCH_INDEX.WORD  
Q216.Q216_IN_SEARCH_INDEX  
Q216.Q216_SMTX  
Q216_SMTX. =  
Q216  
Q215_IN_SEARCH_INDEX.WORD  
Q215.Q215_OUT  
SEARCH_INDEX  
Q215.Q215_IN_SEARCH_INDEX  
Q215_OUT.WORD
```

Other efforts

- Maule et al @ ICSE 2008
- The Prism/Prism++ line of research

Maule et al. @ ICSE'08

- Given an OO app. built on top of a relational db schema and a change type
- Produce the locations of the code that are affected
- Method:
 1. Slicing. A prototype slicing implementation to identify the database queries of the program.
 2. A data-flow analysis algorithm to estimate all the possible runtime values for the parameters of the query.
 3. Use an impact assessment tool, Crocopat, with a reasoning language (RML). Depending on the type of change, a different RML program that assesses impact over the stored data of the previous step is run: this isolates the lines of code affected by the change.

Prism/Prism++

- Series of works from the same authors
- Carlo Curino, Hyun Jin Moon, Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *PVLDB* 1(1): 761-772 (2008)
- Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128, 2010.
- Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.

Prism/Prism++ motivation

- Evolution happens all the time => can be viewed as a sequence of changes



Automatically migrate schema + data+ surrounding queries

- SMO's are a principled set of operators to describe evolution steps, s.t.:
 - The evolution DDL is implied by the SMO's
 - The DML for data migration can be automatically produced from the SMO's
 - The surrounding queries can be rewritten to the new schema

SMO's and ICSMO's

- CREATE TABLE $R(a, b, c)$
- DROP TABLE R
- RENAME TABLE R INTO T
- COPY TABLE R INTO T
- MERGE TABLE R, S INTO T
- PARTITION TABLE R INTO S WITH condition, T
- DECOMPOSE TABLE R INTO $S(a, b) T(a, c)$
- JOIN TABLE R, S INTO T WHERE condition
- ADD COLUMN d [AS constant | function(a, b, c)] INTO R
- DROP COLUMN r FROM R
- RENAME COLUMN b IN R TO d

Automatic creation of
- DDL (schema evo)
- DML (data migration)
is feasible

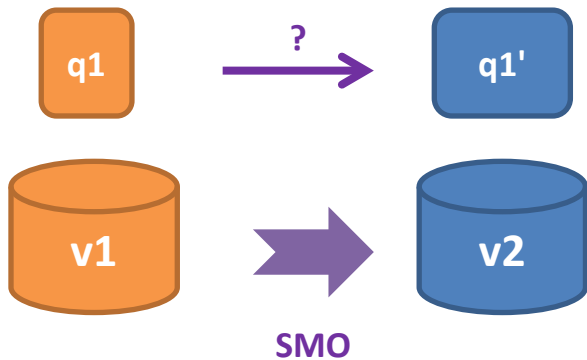
ICSMO's: the technique is
extended to cover
Integrity Constraints too.

- ALTER TABLE R ADD PRIMARY KEY $pk1(a, b)$ <policy>
- ALTER TABLE R ADD FOREIGN KEY $fk1(c, d)$ REFERENCES $T(a, b)$
<policy>
- ALTER TABLE R ADD VALUE CONSTRAINT $vc1(c, d)$ AS $R.e="0"$ <policy>
- ALTER TABLE R DROP PRIMARY KEY $pk1$
- ALTER TABLE R DROP FOREIGN KEY $fk1$
- ALTER TABLE R DROP VALUE CONSTRAINT $vc1$

<policy>

- (i) CHECK if current db satisfies the constraint, else ICMO is rolled back,
- (ii) ENFORCE the removal of all data violating the constraint,
- (iii) IGNORE violating tuples + informs the user about this.

Answering old queries to new schemata without user noticing it



- Assume we migrate the schema + data from v1 to v2
- Can we rewrite the query q1 to q1' s.t. we get the same result, as if we were still in v1?

- SMO invertibility:
- $q1/v1 = q1 / \text{SMO}^{-1}(v2) = q1' / v2$

V1: R(...)

Q1: SELECT * FROM R

SMO: PARTITION R in S(...), T(...)

Q1': SELECT * FROM S,T WHERE S.ID =T.ID

EMPIRICAL STUDIES

WHAT ARE THE “LAWS” OF DATABASE SCHEMA EVOLUTION?



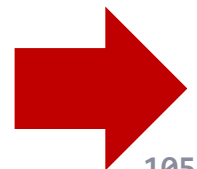
What are the “laws” of database (schema) evolution?

- How do databases change?
- In particular, **how does the schema of a database evolve over time?**
- Long term research goals:
 - Are there any “**invariant properties**” (e.g., patterns of repeating behavior) on the way database (schemata) change?
 - Is there a **theory / model** to explain them?



Why care for the “laws”/patterns of schema evolution?

- Scientific curiosity!
- Practical Impact: DB's are **dependency magnets**. Applications have to conform to the structure of the db...
 - typically, **development waits till the “db backbone” is stable** and applications are build on top of it
 - **slight changes** to the structure of a db **can cause** several (parts of) different applications to **crash**, causing the need for **emergency repairing**



Imagine if we could predict how a schema will evolve over time...

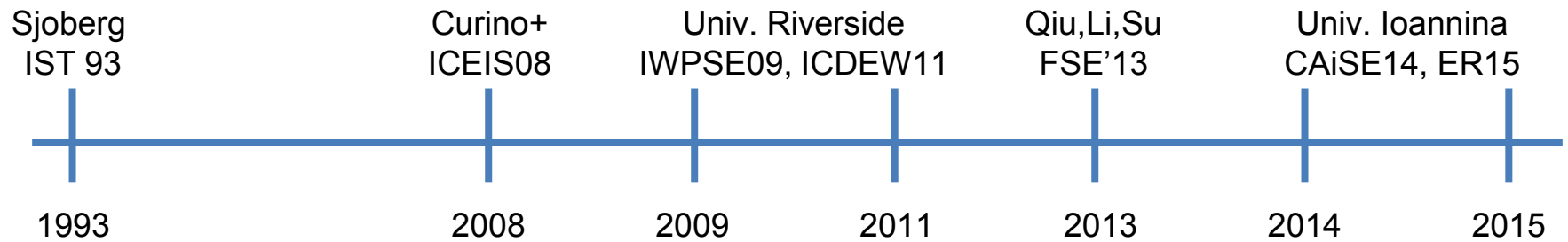
- ... we would be able to **“design for evolution”** and **minimize the impact of evolution** to the surrounding applications
 - by **applying design patterns**
 - by **avoiding anti-patterns** & complexity increase**... in both the db and the code**
- ... we would be able to **plan** administration and perfective maintenance tasks and resources, instead of responding to emergencies

Why aren't we there yet?

- Historically, nobody from the research community had access + the right to publish to version histories of database schemata
- Open source tools internally hosting databases have changed this landscape:
 - not only is the code available, but also,
 - public repositories (git, svn, ...) keep the entire history of revisions
- We are now presented with the opportunity to study the version histories of such “open source databases”



Timeline of empirical studies

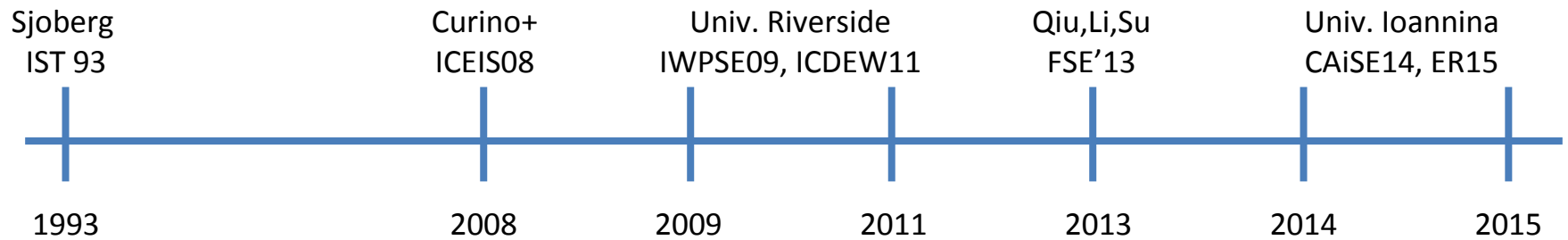


Our take on the problem



- Collected **version histories** for the schemata of 8 open-source **projects**
 - CMS's: MediaWiki, TYPO3, Coppermine, phpBB, OpenCart
 - Physics: ATLAS Trigger --- Bio: Ensemble, BioSQL
- Preprocessed them to be parsable by our **HECATE schema comparison tool** and exported the **transitions** between each two subsequent versions and **measures** for them (size, growth, changes)
- Visualized the transitions in graphs and **statistically studied the measures**, both at the **macro (database)** and at the **micro (table)** level

Timeline of empirical studies



Timeline of empirical studies

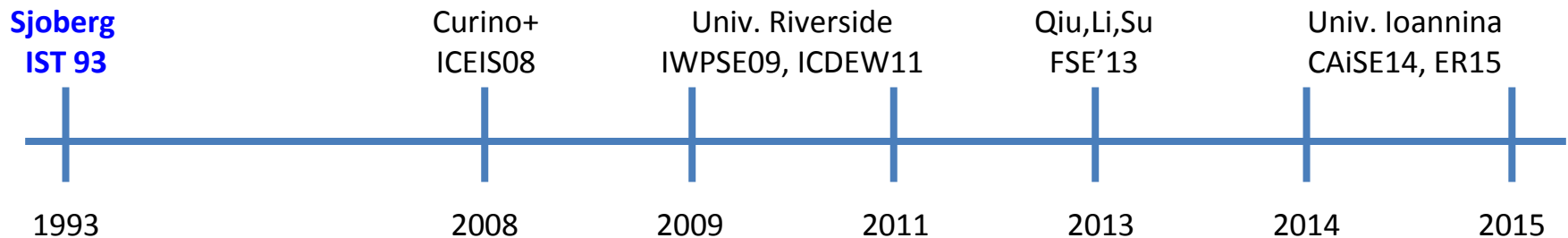
Sjoberg @ IST 93: 18 months study of a health system.

139% increase of #tables ; 274% increase of the #attributes

Changes in the code (on avg):

- relation addition: 19 changes ; attribute additions: 2 changes
- relation deletion : 59.5 changes; attribute deletions: 3.25 changes

An **inflating period** during construction where almost all changes were additions, and a **subsequent period** where additions and deletions were balanced.



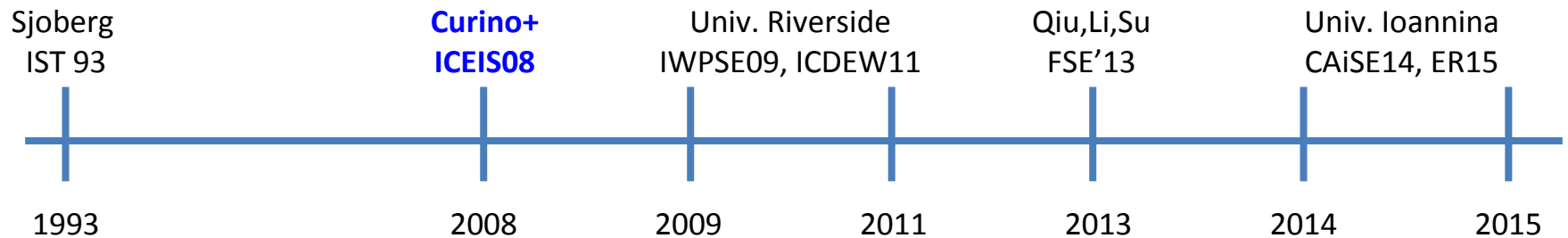
Timeline of empirical studies

Curino+ @ ICEIS08: Mediawiki

100% increase in the number of tables

142% in the number of attributes.

45% of changes do not affect the information capacity of the schema
(but are rather index adjustments, documentation, etc)



Timeline of empirical studies

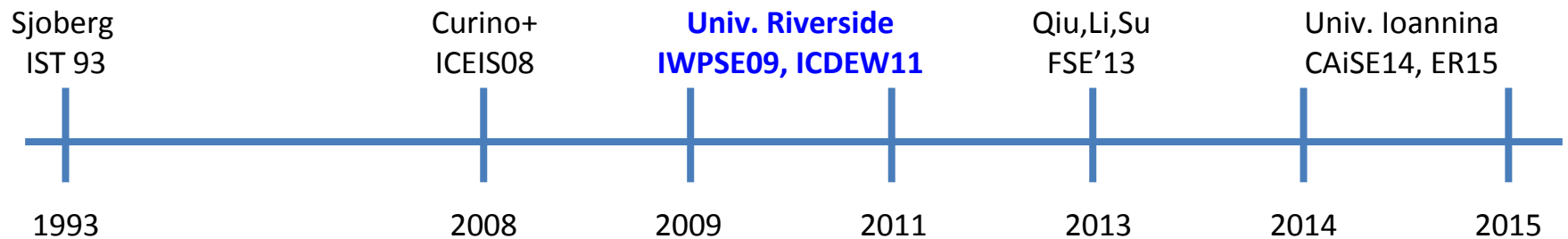
IWPSE09: Mozilla and Monotone (a version control system)

Many ways to be out of synch between code and evolving db schema

ICDEW11: Firefox, Monotone , Biblioteq (catalogue man.) , Vienna (RSS)

Similar pct of changes with previous work

Frequency and timing analysis: **db schemata tend to stabilize over time**, as there is more change at the beginning of their history, but seem to converge to a relatively fixed structure later



Timeline of empirical studies

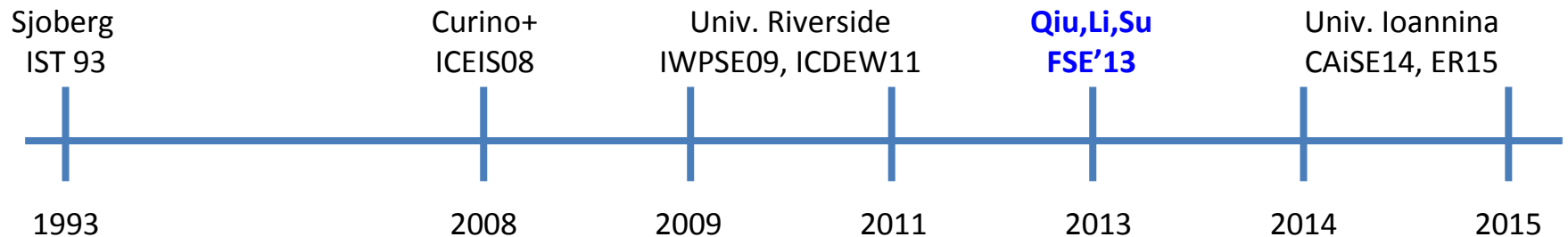
Qiu,Li,Su@ FSE 2013: 10 (!) database schemata studied.

Change is focused both (a) with respect to time and (b) with respect to the tables who change.

Timing: 7 out of 10 databases reached 60% of their schema size within 20% of their early lifetime.

Change is frequent in the early stages of the databases, with inflationary characteristics; then, the schema evolution process calms down.

Tables that change: 40% of tables do not undergo any change at all, and 60%-90% of changes pertain to 20% of the tables (in other words, 80% of the tables live quiet lives). The most frequently modified tables attract 80% of the changes.



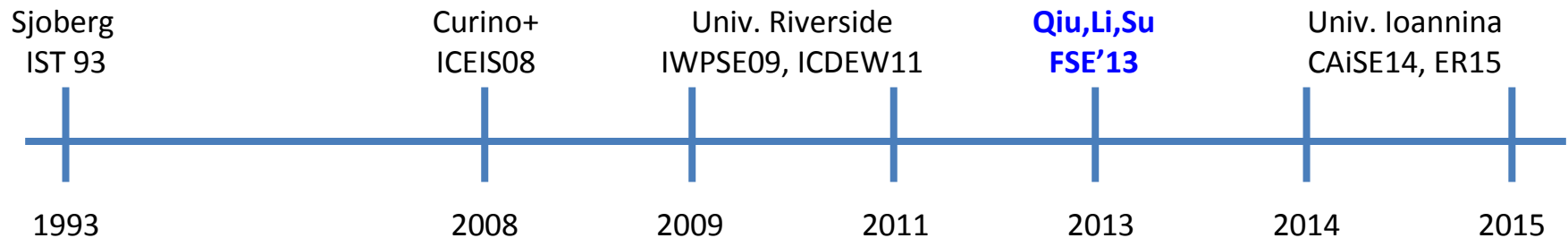
Timeline of empirical studies

Qiu,Li,Su@ FSE 2013: Code and db co-evolution, not always in synch.

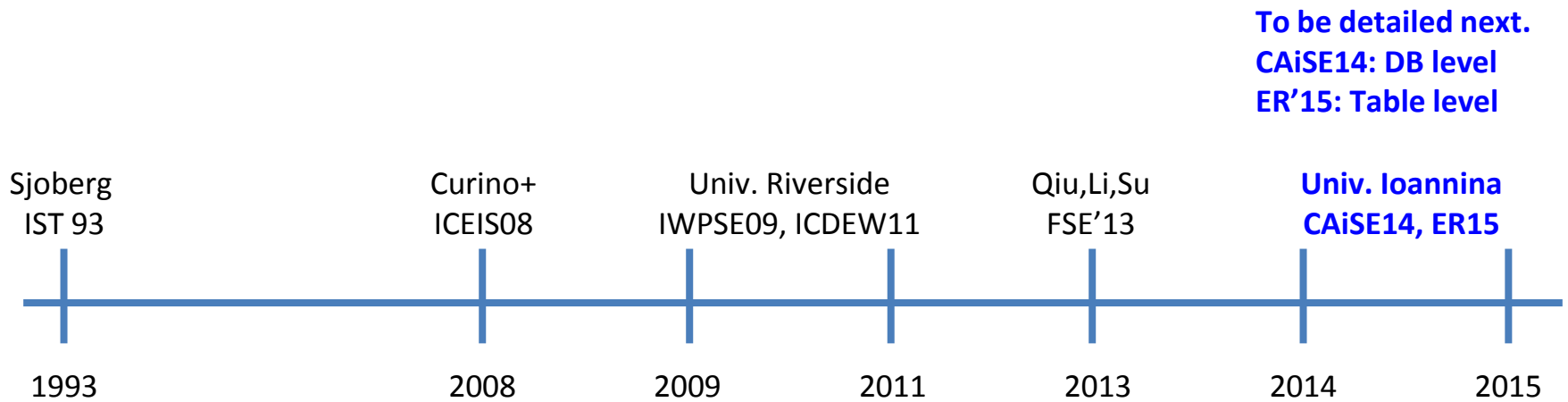
- Code and db changed in the same revision: 50.67% occasions
- Code change was in a previous/subsequent version than the one where the database schema change: 16.22% of occasions
- database changes not followed by code adaptation: 21.62{\%} of occasions
- 11.49% of code changes were unrelated to the database evolution.

Each atomic change at the schema level is estimated to result in 10 -- 100 lines of application code been updated;

A valid db revision results in 100 -- 1000 lines of application code being updated



Timeline of empirical studies



.. What do we see if we observe the evolution of the entire schema?

http://www.cs.uoi.gr/~pvassil/publications/2014_CAiSE/

Skoulis, Vassiliadis, Zarras. Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution? **CAiSE 2014**

Also: Growing up with stability: How open-source relational databases evolve. **Information Systems, Volume 53**, October–November 2015

SCHEMA EVOLUTION FOR O/S DB'S AT THE “MACRO” LEVEL

Datasets

<https://github.com/DAINTINESS-Group/EvolutionDatasets>

- Content management Systems
 - MediaWiki, TYPO3, Coppermine, phpBB, OpenCart
- Medical Databases
 - Ensemble, BioSQL
- Scientific
 - ATLAS Trigger

Data sets

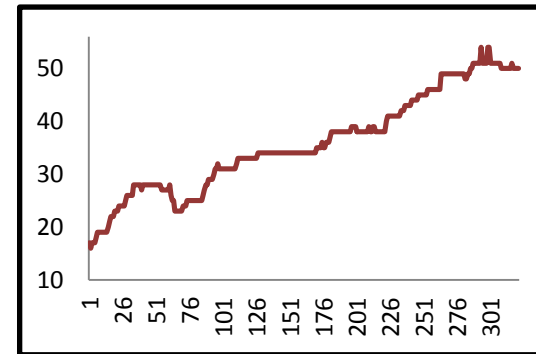
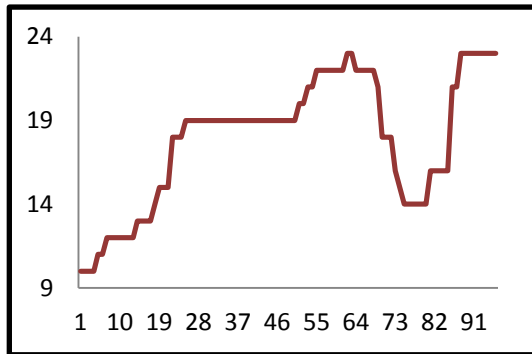
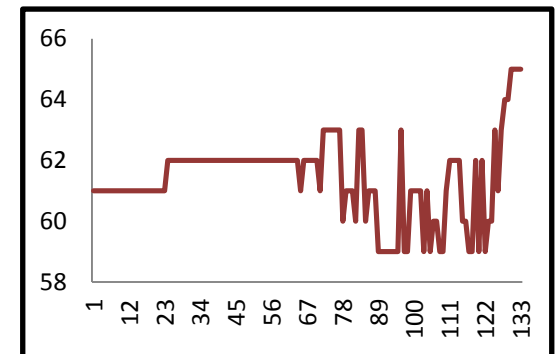
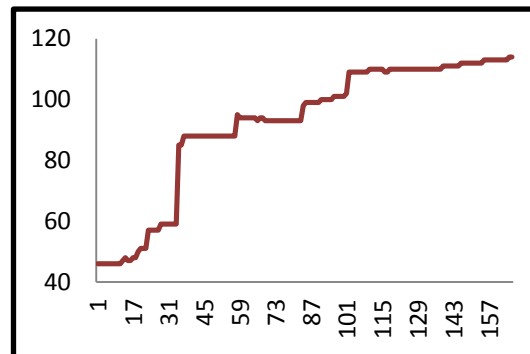
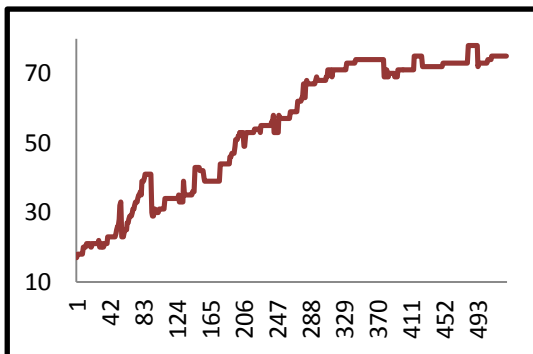
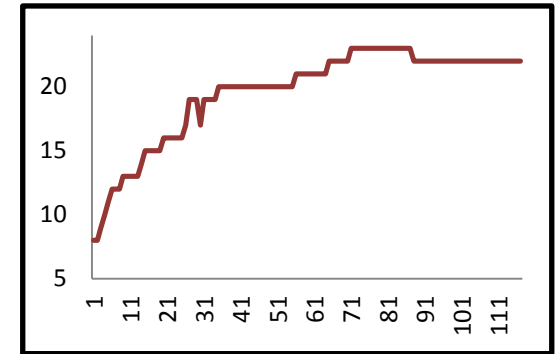
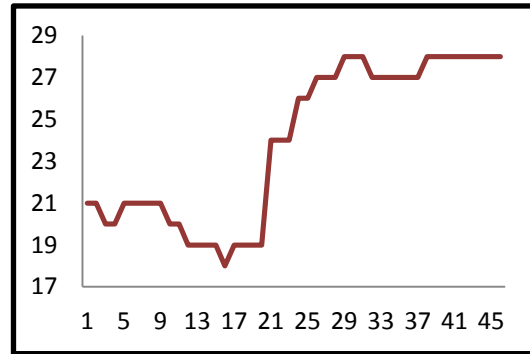
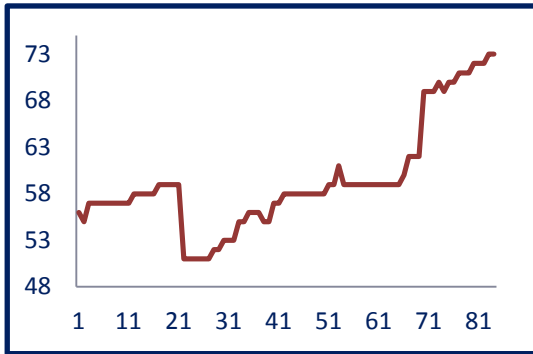
Dataset	Versions	Lifetime	Tables Start	Tables End	Attributes Start	Attributes End	Commits per Day	% commits with change	Repository URL
ATLAS Trigger	84	2 Y, 7 M, 2 D	56	73	709	858	0,089	82%	http://atdaq-sw.cern.ch/cgi-bin/viewcvs-atlas.cgi/offline/Trigger/TrigConfiguration/TrigDb/share/sql/com-bined_schema.sql
BioSQL	46	10 Y, 6 M, 19 D	21	28	74	129	0,012	63%	https://github.com/biosql/biosql/blob/master/sql/biosqldb-mysql.sql
Coppermine	117	8 Y, 6 M, 2 D	8	22	87	169	0,038	50%	http://sourceforge.net/p/coppermine/code/8581/tree/trunk/cpg1.5.x/sql/schema.sql
Ensembl	528	13 Y, 3 M, 15 D	17	75	75	486	0,109	60%	http://cvs.sanger.ac.uk/cgi-bin/viewvc.cgi/ensembl/sql/table.sql?root=ensembl&view=log
MediaWiki	322	8 Y, 10 M, 6 D	17	50	100	318	0,100	59%	https://svn.wikimedia.org/viewvc/mediawiki/trunk/phase3/maintenance/tables.sql?view=log
OpenCart	164	4 Y, 4 M, 3 D	46	114	292	731	0,104	47%	https://github.com/opencart/opencart/blob/master/upload/install/opencart.sql
phpBB	133	6 Y, 7 M, 10 D	61	65	611	565	0,055	82%	https://github.com/phpbb/phpbb3/blob/develop/phpBB/install/schemas/mysql_41_schema.sql
TYPO3	97	8 Y, 11 M, 0 D	10	23	122	414	0,030	76%	https://git.typo3.org/Packages/TYPO3.CMS.git/history/TYPO3_6-0:t3lib/stddb/tables.sql

Hecate: SQL schema diff viewer

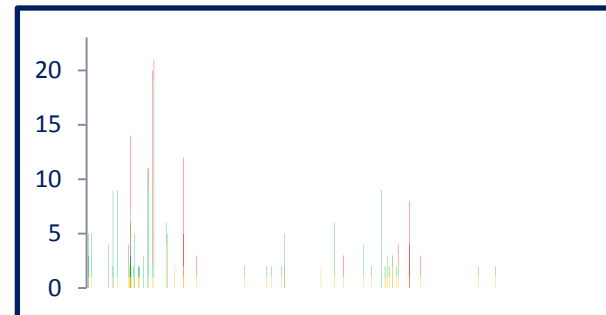
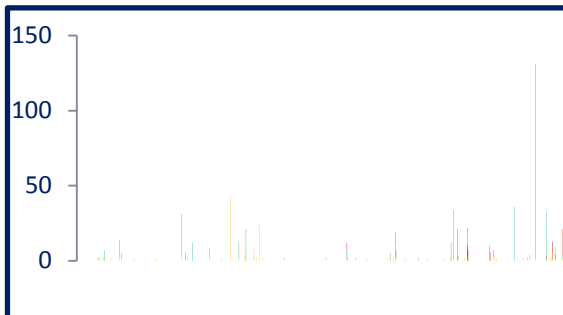
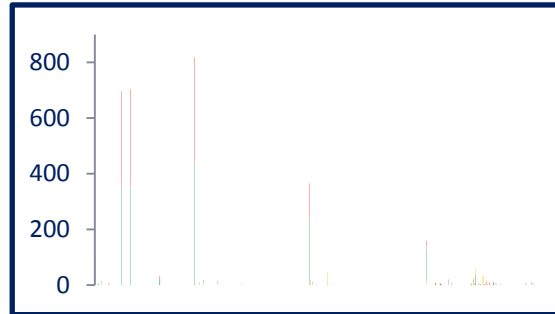
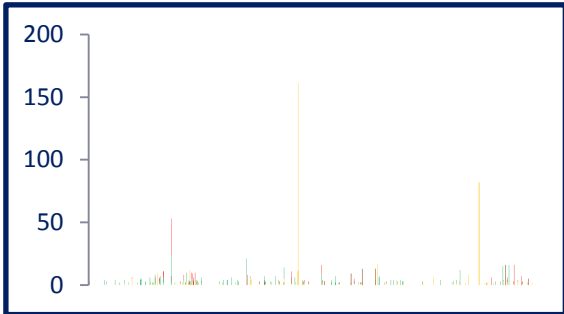
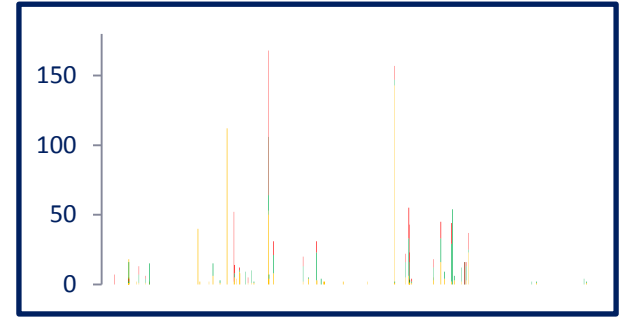
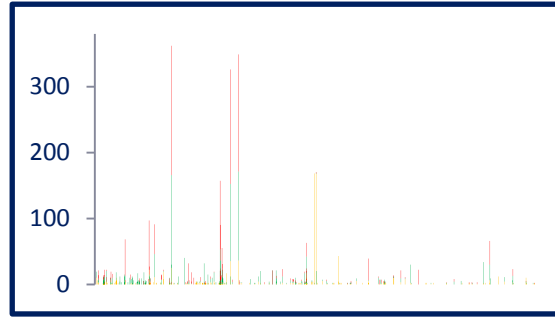
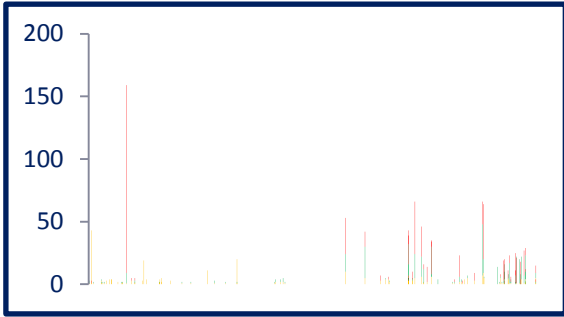
- Parses DDL files
- Creates a model for the parsed SQL elements
- Differentiates two version of the same schema
- Reports on the diff performed with a variety of metrics
- Exports the transitions that occurred in XML format

<https://github.com/DAINTINESS-Group/Hecate>

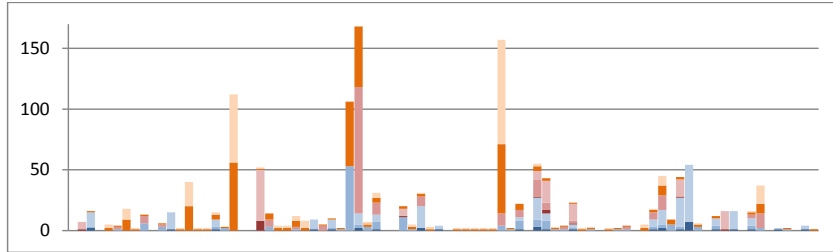
Schema Size (relations)



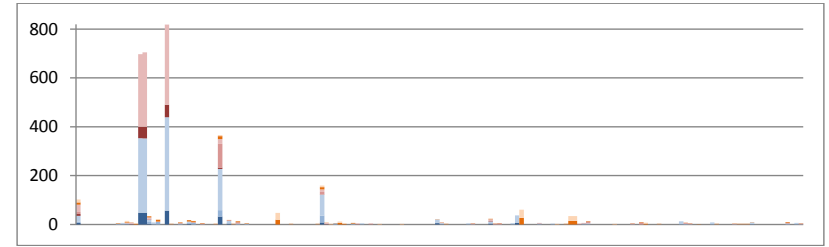
Change over time



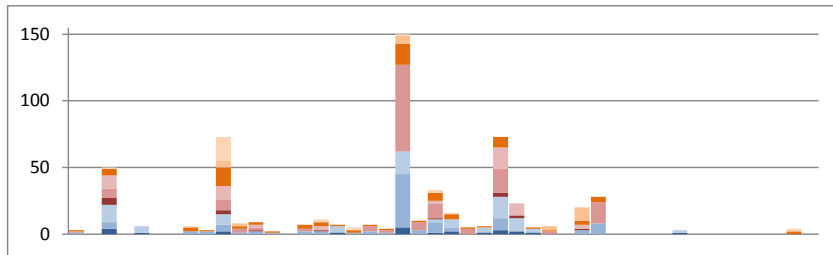
Change over version



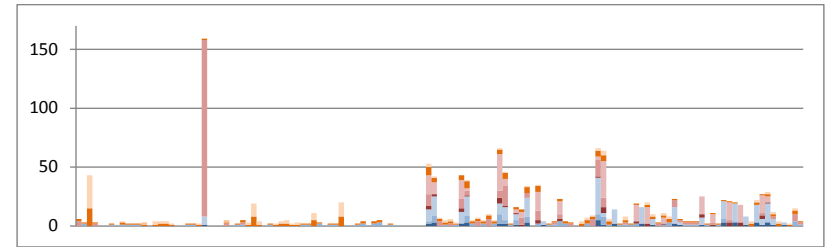
ATLAS Trigger



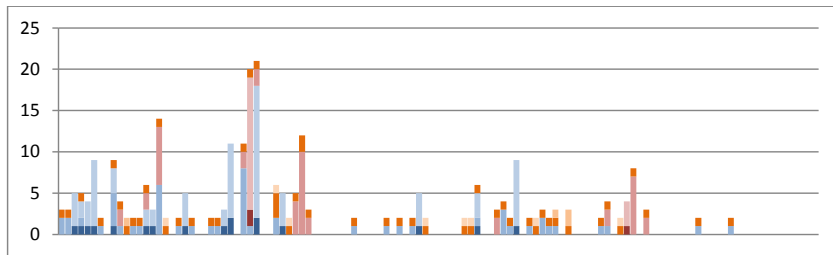
OpenCart



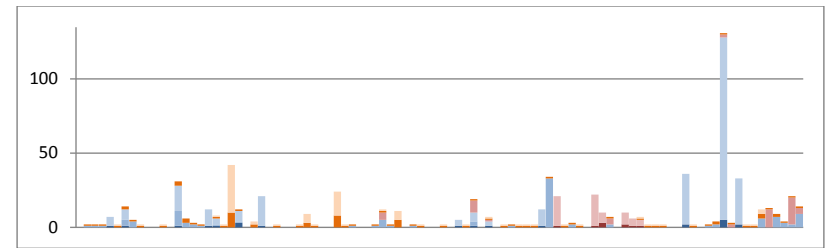
BioSQL



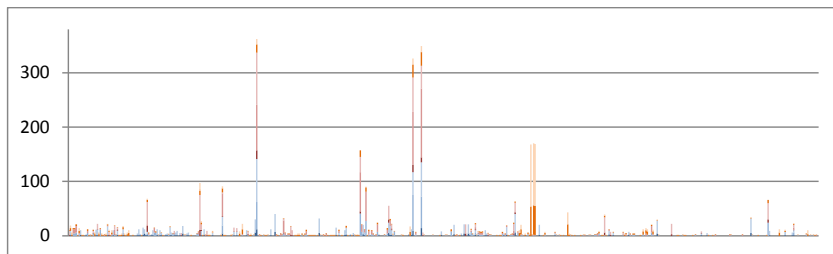
phpBB



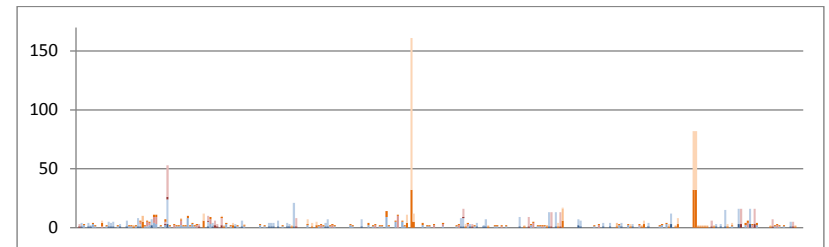
Coppermine



TYPO3



Ensembl



MediaWiki

Main results



Schema size (#tables, #attributes) supports the assumption of a feedback mechanism

- Schema size **grows over time**; not continuously, but with bursts of concentrated effort
- **Drops in schema size signifies the existence of perfective maintenance**
- Regressive formula for size estimation holds, with a quite short memory

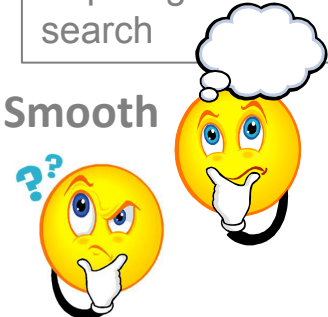
Schema Growth (diff in size between subsequent versions) is small!!!

- Growth is **small**, smaller than in typical software
- The number of changes for each evolution step follows **Zipf's law** around zero
- Average growth is close (slightly higher) to zero

Patterns of change: no consistently constant behavior

- Changes **reduce in density** as databases age
- Change follows three patterns: **Stillness**, **Abrupt change** (up or down), **Smooth growth upwards**
- Change frequently follows **spike** patterns
- **Complexity** does **not** increase with age

Grey for results requiring further search



Vassiliadis, Zarras, Skoulis. **How is Life for a Table in an Evolving Relational Schema? Birth, Death & Everything in Between.**

To appear in **ER 2015**

OBSERVING THE EVOLUTION OF O/S DB SCHEMATA AT THE MICRO LEVEL

Statistical study of durations

- Short and long lived tables are practically equally proportioned
- Medium size durations are few!
- Long lived tables are mostly survivors (see on the right)

<u>Tables...</u>	<u>Range</u>	<u>#Tables</u>	<u>Pct.</u>
Short lived	< 0.33	302	41.94%
medium duration	0.33 - 0.77	149	20.69%
Long lived	> 0.77	269	37.36%
<hr/>			
<i>Long but not full dur.</i>	<i>(0.77 - 1.0)</i>	81	11.25%
<i>from v0 to v.last</i>	1.0	188	26.11%

One of the fascinating revelations of this measurement was that there is a **26.11%** fraction of tables that appeared in the beginning of the database and survived until the end. In fact, if a table is long-lived there is a 70% chance (188 over 269 occasions) that it has appeared in the beginning of the database.

Tables are mostly thin

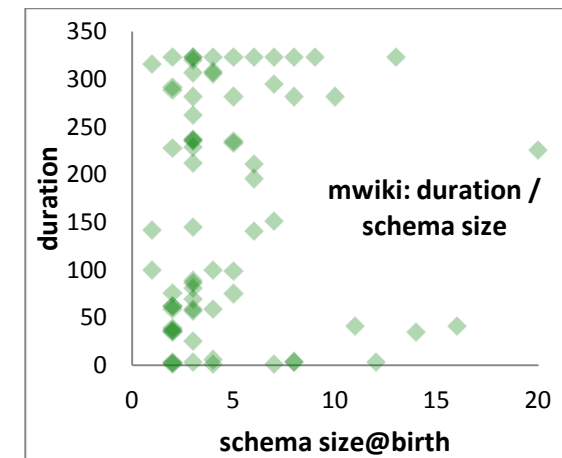
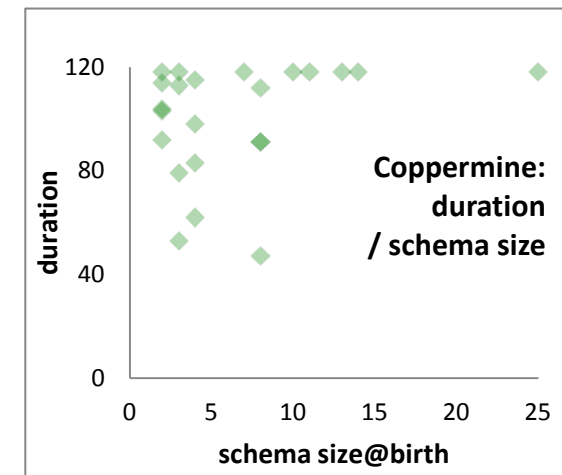
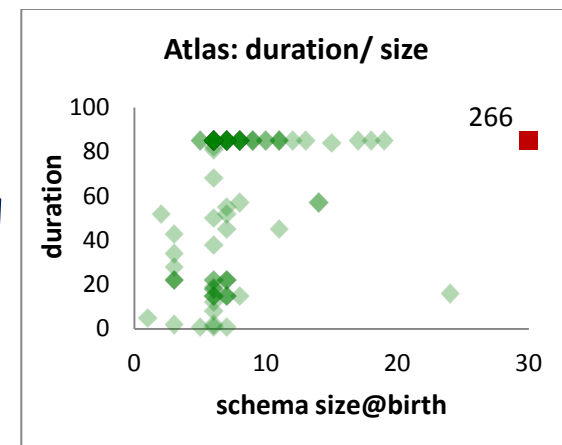
- On average, half of the tables (approx. 47%) are **thin** tables with less than 5 attributes.
- The tables with 5 to 10 attributes are approximately one third of the tables' population
- The large tables with more than 10 attributes are approximately 17% of the tables.

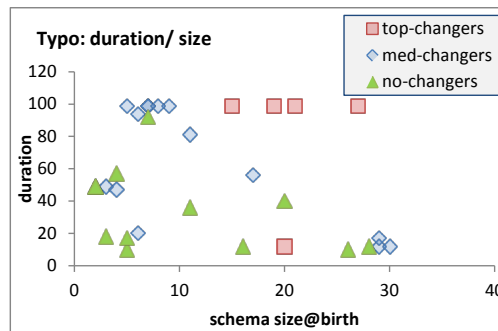
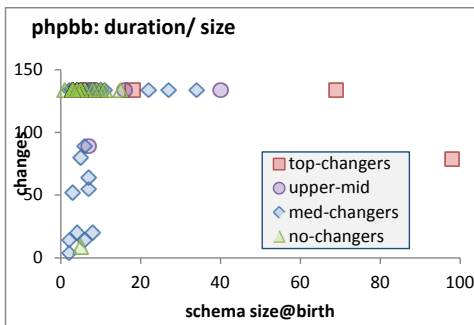
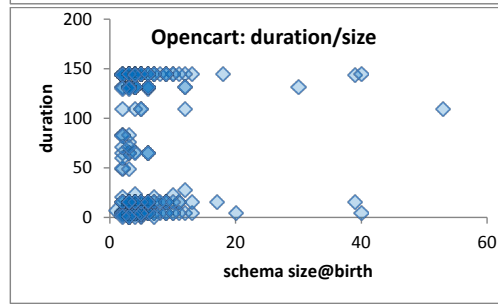
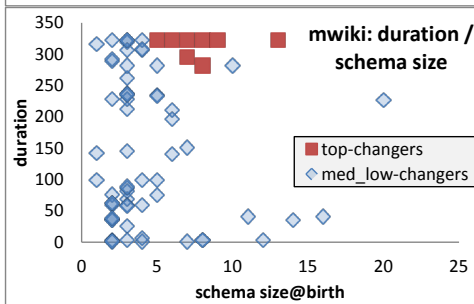
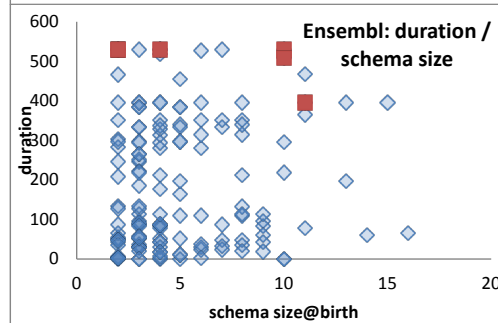
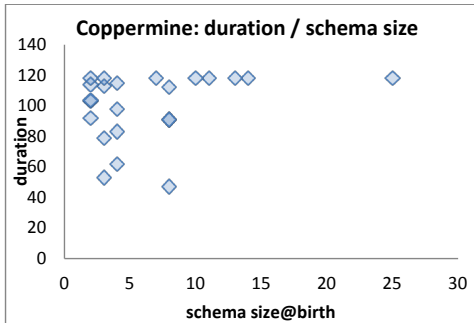
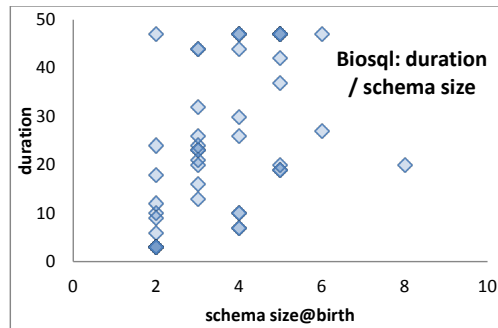
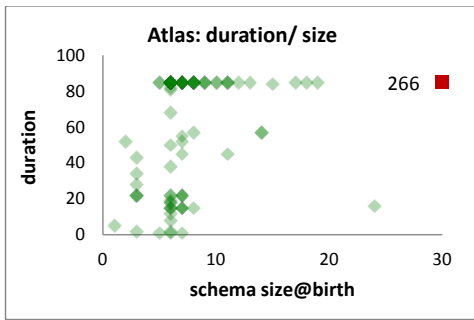
Pct of tables with num. of attributes ...

	<u><5</u>	<u>5-10</u>	<u>>10</u>
atlas	10,23%	68,18%	21,59%
biosql	75,56%	24,44%	0,00%
coppermine	52,17%	30,43%	17,39%
ensembl	54,84%	38,06%	7,10%
mediawiki	61,97%	19,72%	18,31%
phpbb	40,00%	44,29%	15,71%
typo3	21,88%	31,25%	46,88%
opencart	57,20%	33,05%	9,75%
Average	46,73%	36,18%	17,09%

The Γ Pattern: "if you 're wide, you survive"

- The Γ phenomenon:
 - tables with small schema sizes can have arbitrary durations, //small size does not determine duration
 - larger size tables last long
- Observations:
 - whenever a table exceeds the critical value of 10 attributes in its schema, its chances of surviving are high.
 - in most cases, the large tables are created early on and are not deleted afterwards.

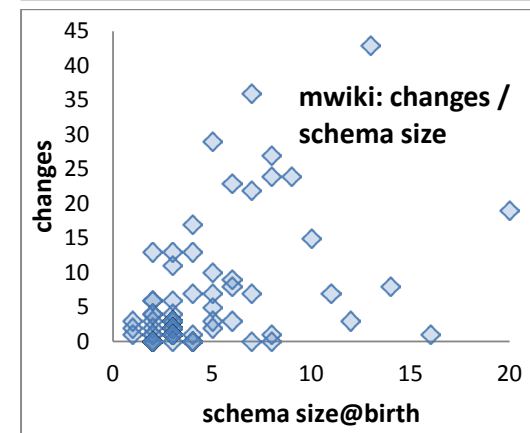
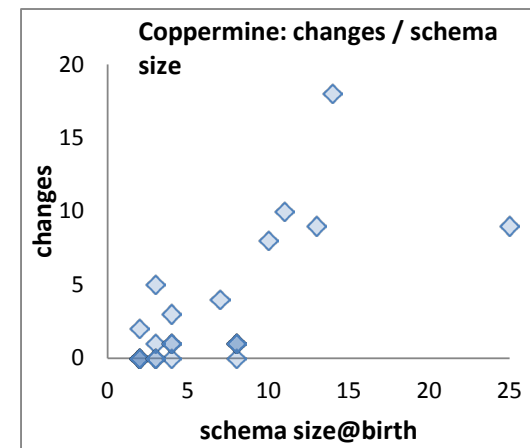
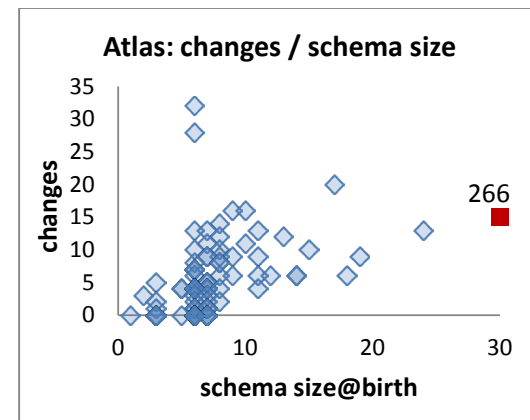


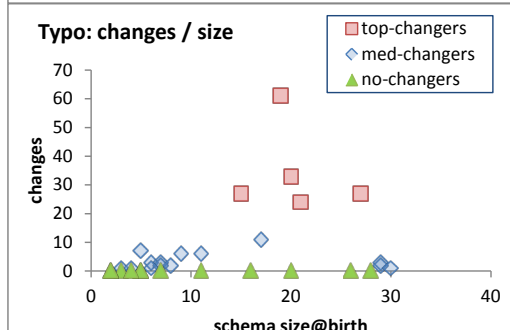
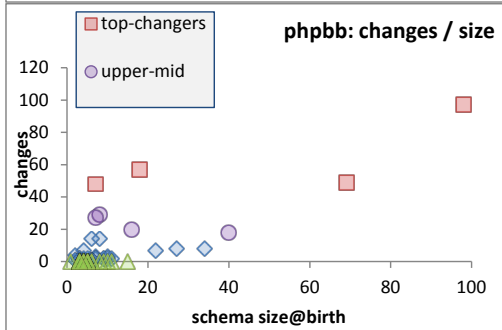
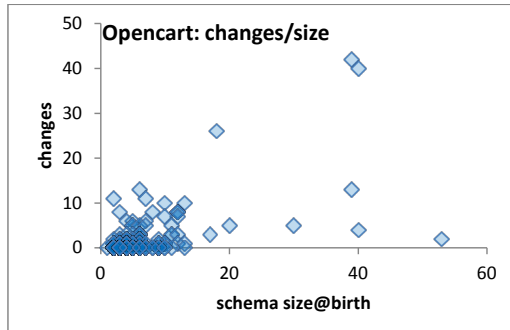
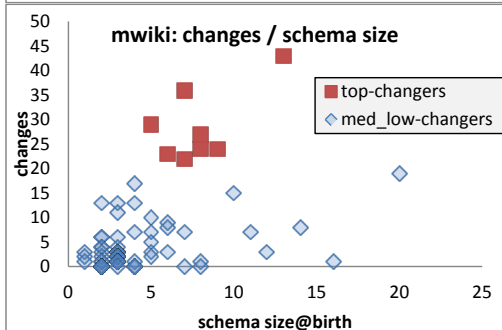
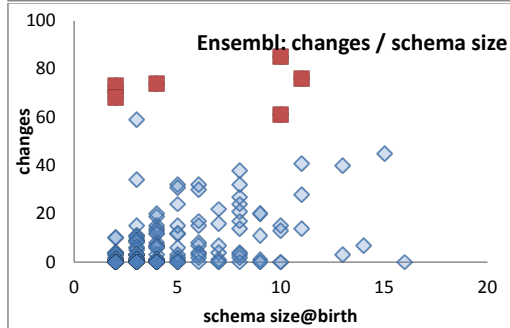
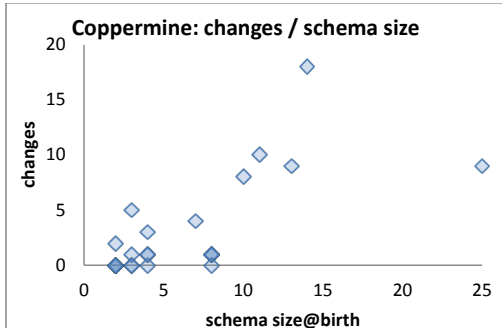
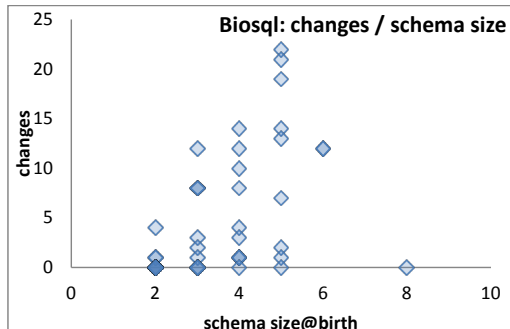
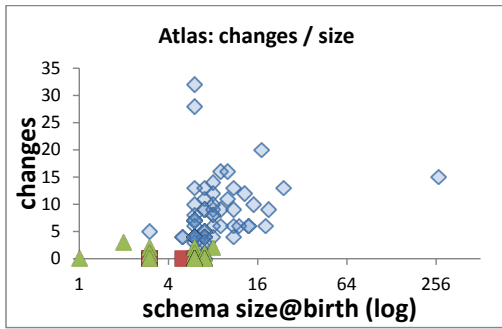


The Comet Pattern

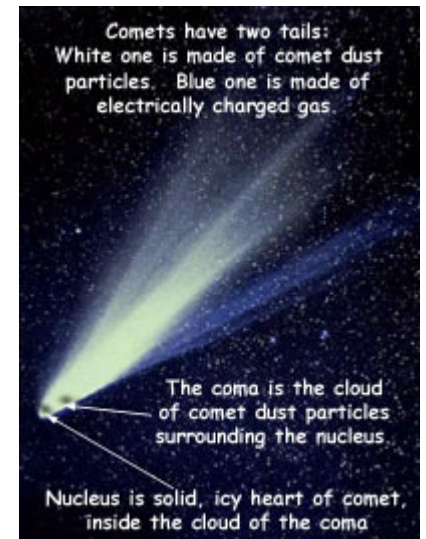
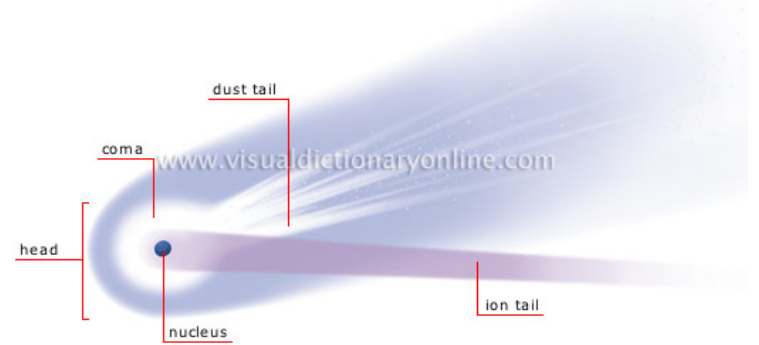
“Comet “ for change over schema size with:

- a large, dense, **nucleus** cluster close to the beginning of the axes, denoting small size and small amount of change,
- **medium** schema **size** tables typically demonstrating **medium to large change**
 - The tables with the largest amount of change are typically tables slightly higher the median value of the schema size axis
- **wide** tables with large schema sizes demonstrating **small to medium** (typically around the median of the y-axis) amount of change.





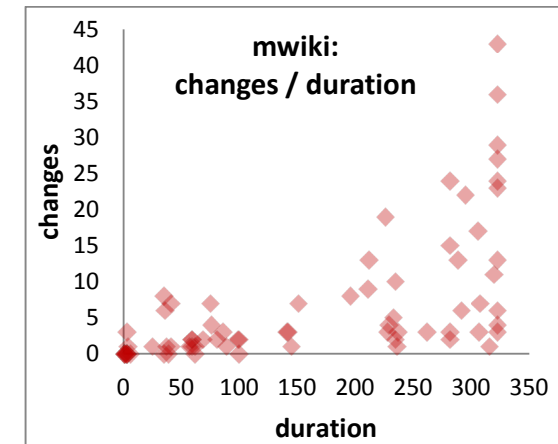
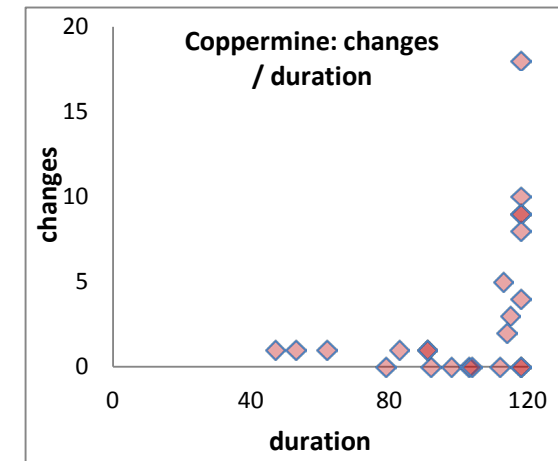
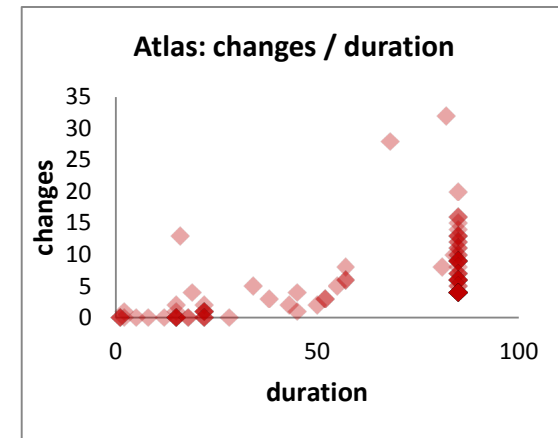
<http://visual.merriam-webster.com/astronomy/celestial-bodies/comet.php>

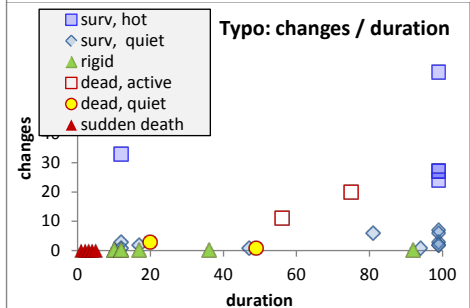
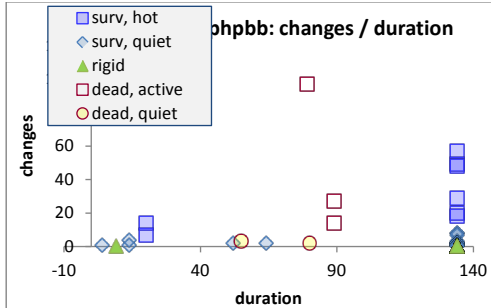
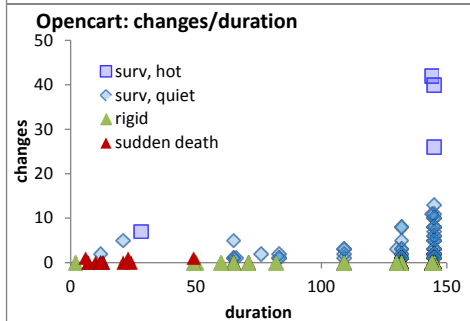
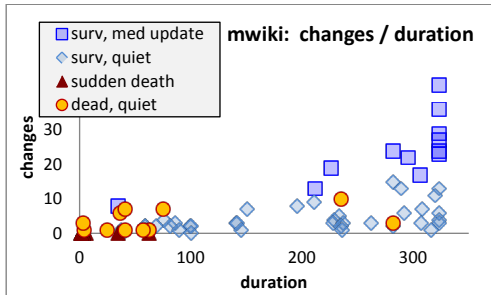
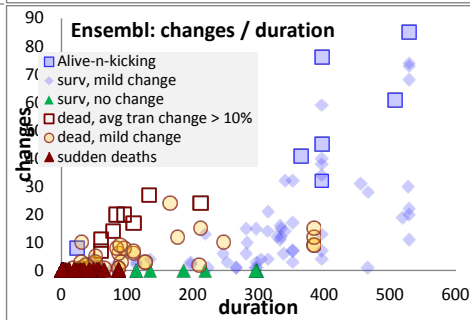
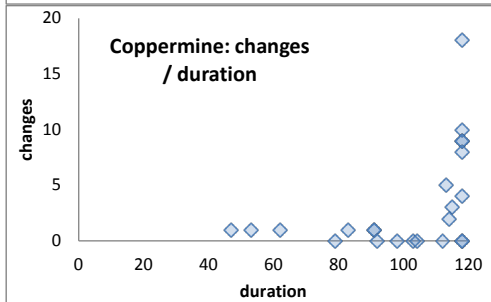
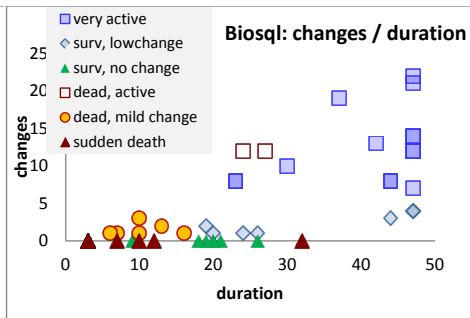
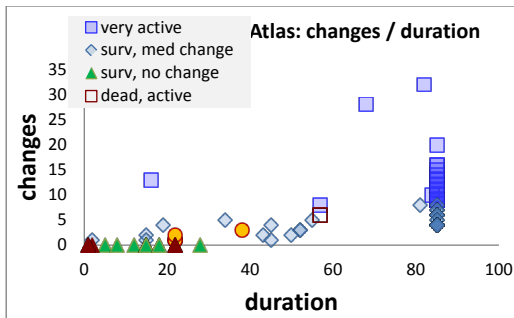


<http://spaceplace.nasa.gov/comet-nucleus/en/>

The inverse Γ pattern

- The correlation of change and duration is as follows:
 - small durations come necessarily with small change,
 - large durations come with all kinds of change activity and
 - medium sized durations come mostly with small change activity (inverse Γ).





Quiet tables rule, esp. for mature db's

Table distribution (pct of tables) wrt their avg transitional update rate

	#tables	DIED				SURVIVED				Aggregate per update type		
		No change	Quiet (0-0.1)	Active (>0.1)	Total	No change	Quiet (0-0.1)	Active (>0.1)	Total	No change	Quiet (0-0.1)	Active (>0.1)
atlas	88	8%	7%	2%	17%	13%	42%	28%	83%	20%	49%	31%
biosql	45	20%	13%	4%	38%	16%	16%	31%	62%	36%	29%	36%
phpbb	70	0%	3%	4%	7%	50%	31%	11%	93%	50%	34%	16%
typo3	32	16%	6%	6%	28%	22%	34%	16%	72%	38%	41%	22%
coppermine	23	4%	0%	0%	4%	30%	61%	4%	96%	35%	61%	4%
ensembl	155	24%	23%	6%	52%	6%	38%	3%	48%	30%	61%	9%
mwiki	71	14%	13%	3%	30%	3%	63%	4%	70%	17%	76%	7%
opencart*	128	9%	2%	0%	11%	42%	44%	3%	89%	51%	46%	3%

Non-survivors

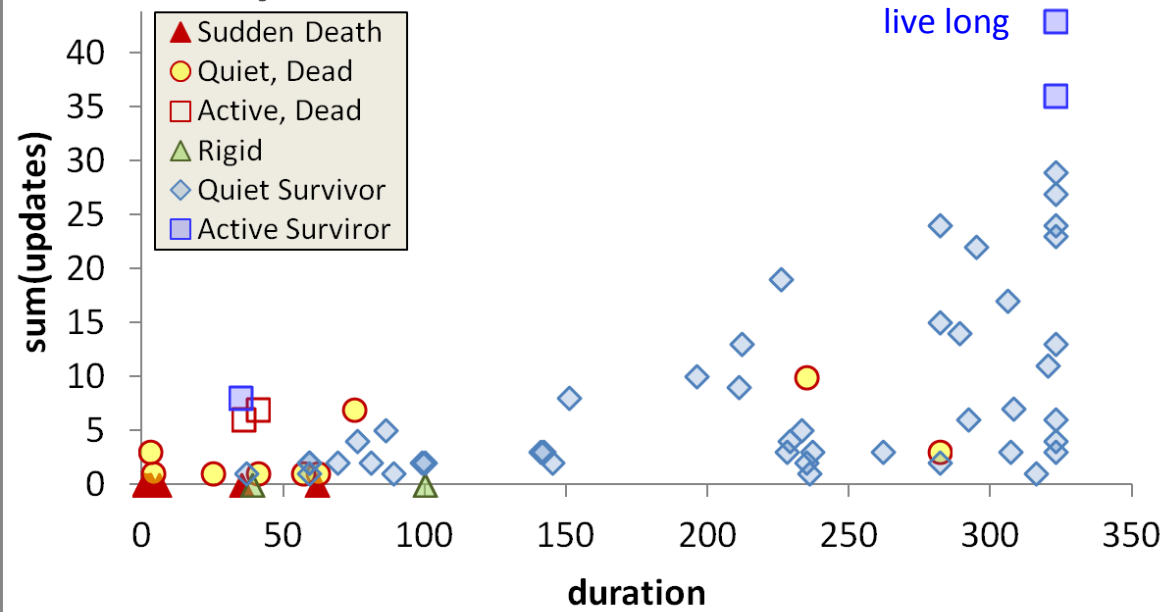
- Sudden deaths mostly
- Quiet come ~ close
- Too few active

Survivors

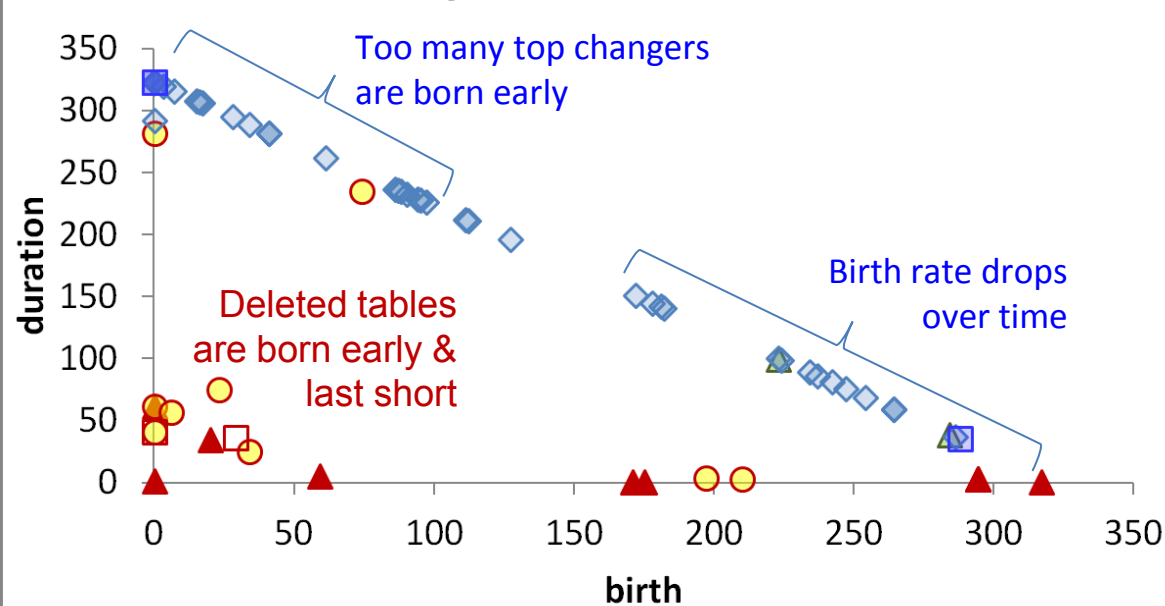
- Quiet tables rule
- Rigid and active then
- Active mostly in “new” db's

Mature DB's: the pct of active tables drops significantly

mwiki: updates / duration



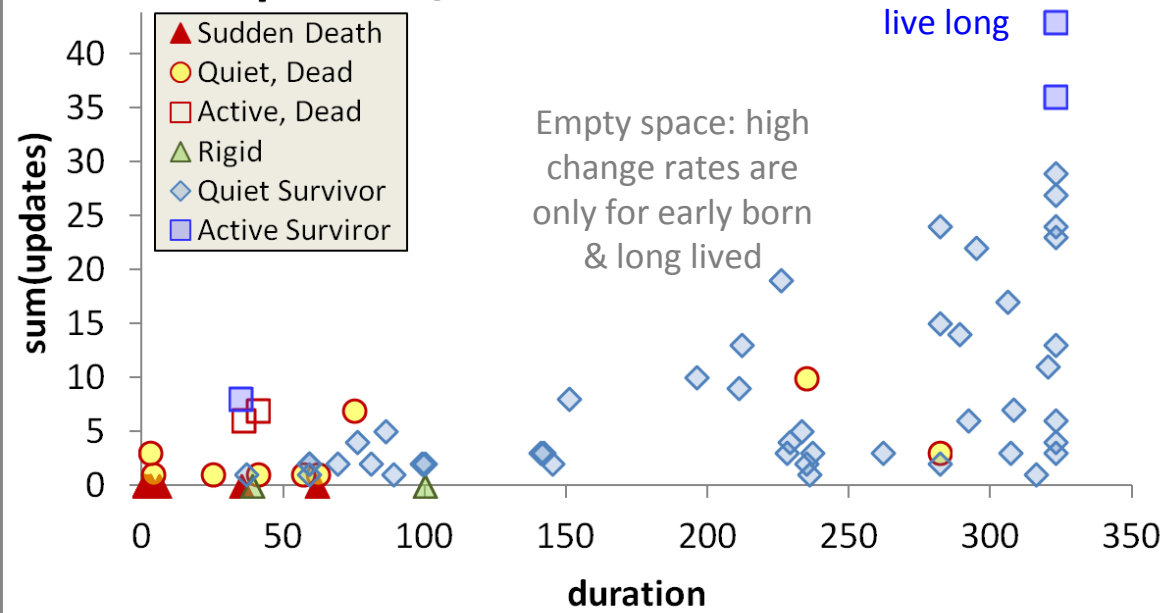
mwiki: duration / birth



Longevity and update activity correlate !!

- Remember: top changers are defined as such wrt ATU, not wrt sum(changes)
- Still, they dominate the sum(changes) too! (see top of inverse Γ)
- See also upper right blue part of diagonal: too many of them are born early and survive => live long!

mwiki: updates / duration

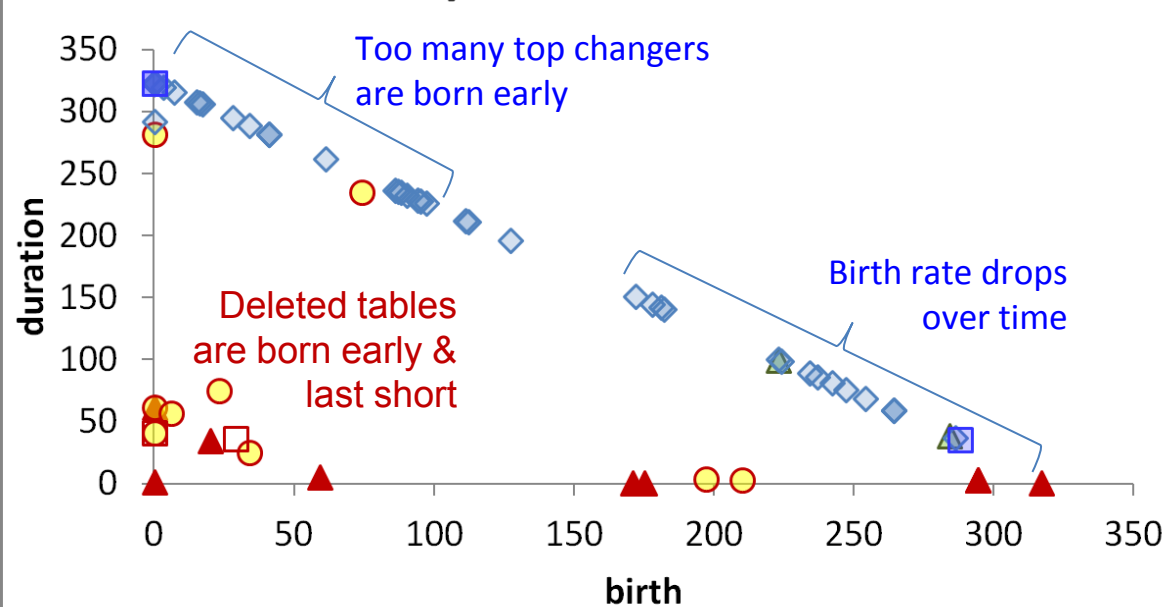


Longevity and update activity correlate !!

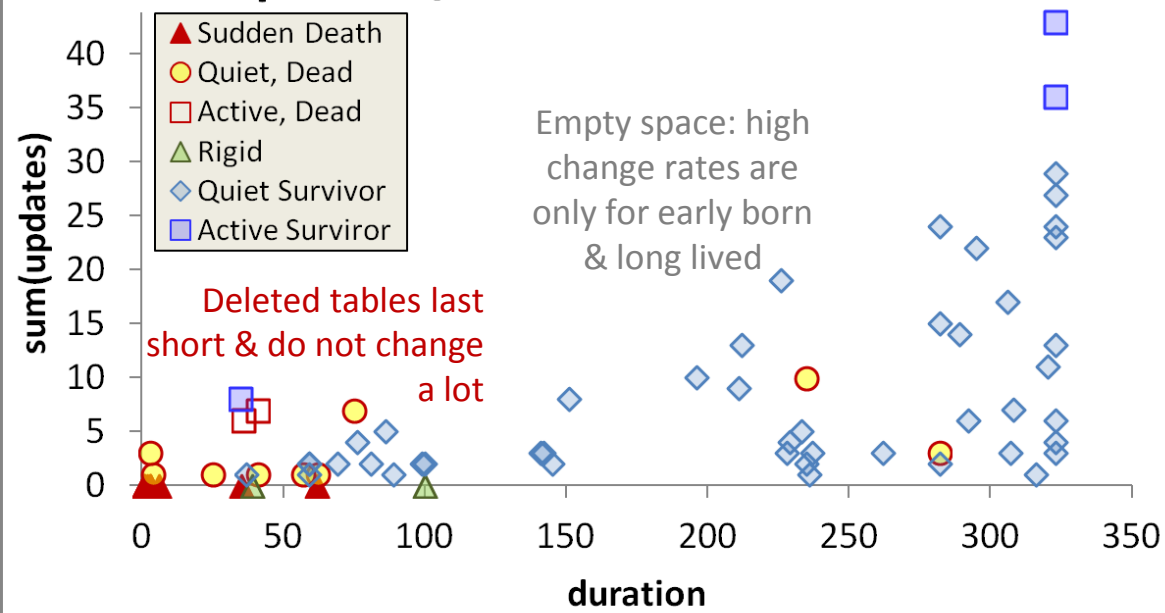
The few top-changers (in terms of ATU)

- are long lived,
- typically come from the early versions of the database
- due to the combination of high ATU and duration => they have high total amount of change, and,
- frequently start with medium schema sizes (not shown here)

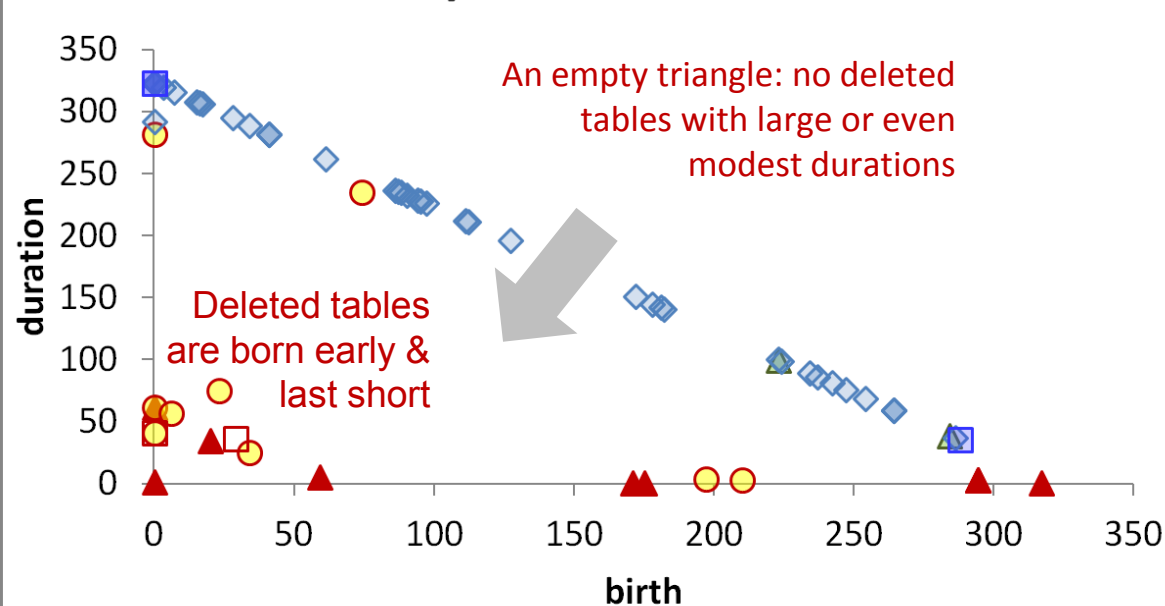
mwiki: duration / birth



mwiki: updates / duration



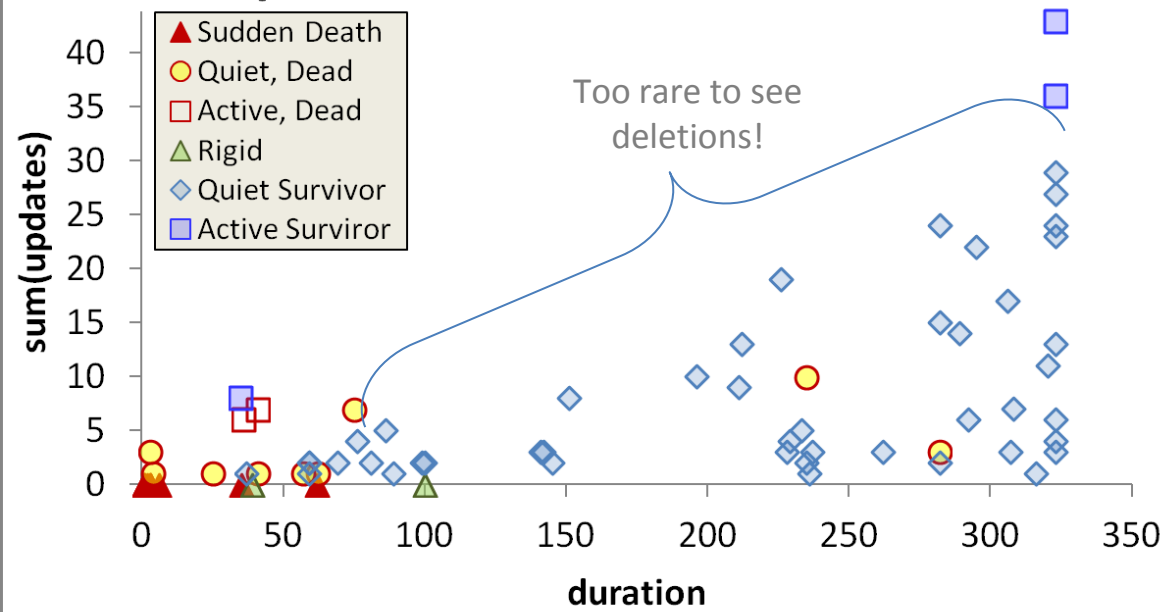
mwiki: duration / birth



Die young and suddenly

- There is a very large concentration of the deleted tables in a small range of newly born, quickly removed, with few or no updates...
- ... resulting in very low numbers of removed tables with medium or long durations (empty triangle).

mwiki: updates / duration



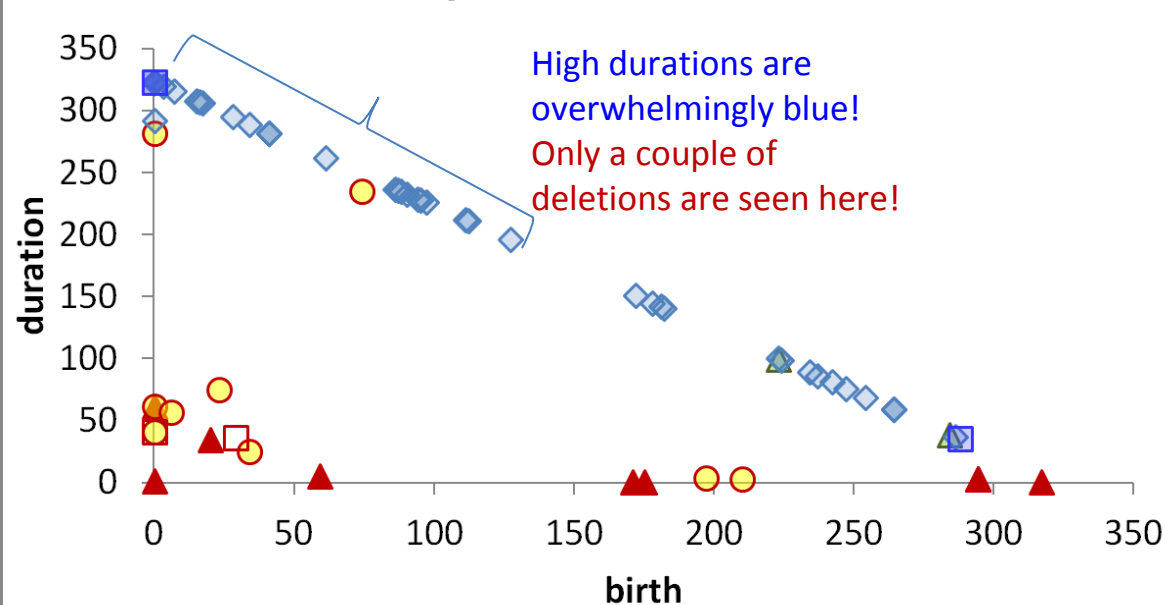
Survive long enough & you 're probably safe

It is quite rare to see tables being removed at old age

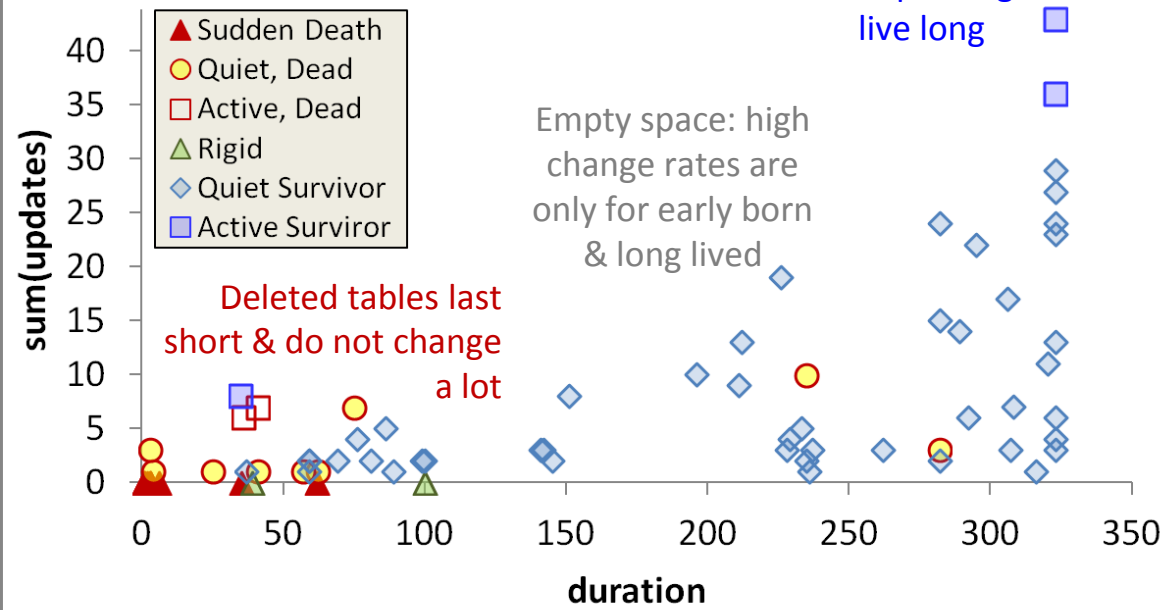
Typically, the area of high duration is overwhelmingly

inhabited by survivors (although each data set comes with a few such cases)!

mwiki: duration / birth

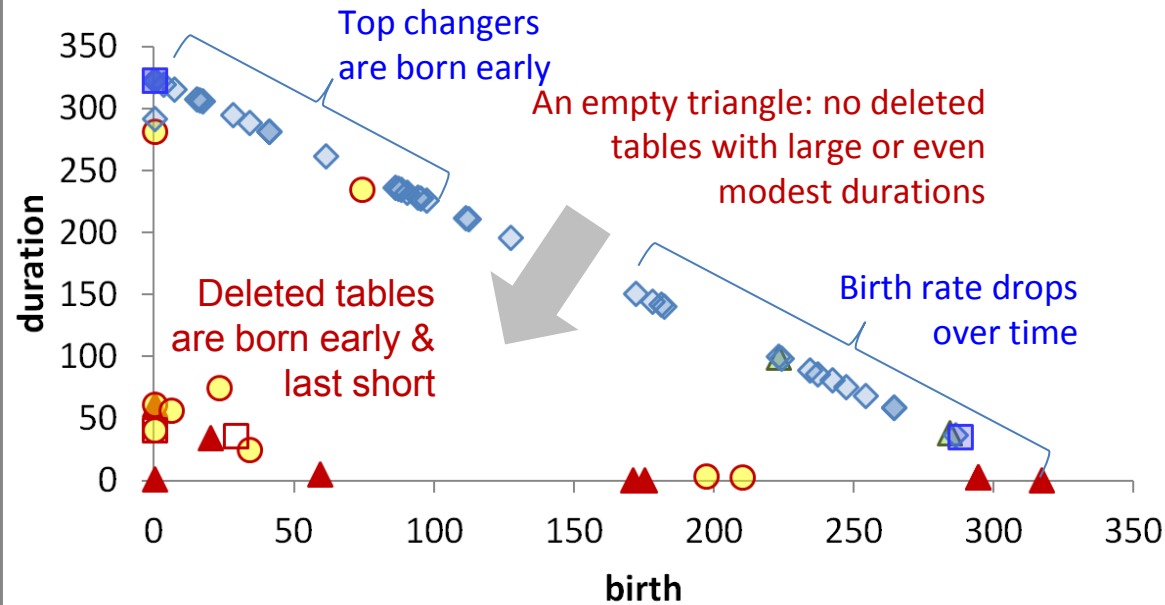


mwiki: updates / duration



All in one

mwiki: duration / birth



Sudden deaths per period

- [Early life of the db] There **a very large concentration of the deleted tables in a small range of newly born, quickly removed, with few or no updates**, resulting in very low numbers of removed tables with medium or long durations.
- [Mature db] After the early stages of the databases, we see the birth of tables who eventually get deleted, but they mostly come with very small durations and sudden deaths.

Early stages of the db life are more active

- Moreover, early stages of the database life are more "active" in terms of births, deaths and updates, and have higher chances of producing deleted tables.
- After the first major restructuring, the database continues to grow; however, we see much less removals, and maintenance activity becomes more concentrated and focused.

OPEN ISSUES

svn/git for db schemata

- The versioning tale says: keep the history of previous schemata available, as this can allow the automation of query/application migration/forward-engineering and the translation of old data to a new structure.
- When it comes to software, svn/git paradigm is the undisputed champion:
 - You make branches for concurrent development
 - Collisions are automatically detected
 - Different versions can be merged
 - You can refer to a particular version of the code easily
- How does this apply to databases and application development for databases?
- Is it really worth the trouble?

Schema curation and preservation

- Data curation and preservation is a very large topic on its own
- If we focus only at the schema part, and assuming we want to support history management for database schemata, how do we implement it?
- SMO's can be the key for altering a db schema in a way that history can be replayed backwards / forward
 - Catch: meta information and functional dependencies are key to these methods. Need to pay the price for them.
- But how can we handle the data efficiently then?

Current trends in data management

- How will the area of schema evolution be affected by the trends in the area of data management?
- First, we need to agree on how the future will look like...
- Open for discussion

What about DW evolution?

- Largely depends on how different / unique DW's will be contrasted to
 - what they look like now
 - what databases will be in the future
- Open for discussion

Are there “laws” of schema evolution?

- Collect more test cases
- Tools for the automation of the process
 - Extract changes & verify their correctness (**what** happened)
 - Link changes to expressed user req's / bugs / ... (**why** it happened & **by whom**)
 - Extract sub-histories of focused maintenance (**how** it happened & **when**)
 - Co-change of schema and code (**what is affected** in the code)
 - Visualization
- Consolidate the fundamental laws that govern evolution && forecast it (what **will** change)

Management of ecosystems' evolution

- Can we find these constructs that are most sensitive to evolution?
 - Metrics for sensitivity to evolution?
- Automation of the reaction to changes
 - self-monitoring
 - impact prediction
 - auto-regulation (policy determination)
 - self-repairing

Thank you!

Q&A

<https://github.com/DAINTINESS-Group/>

<http://www.cs.uoi.gr/~pvassil/>

DB Schema Evolution

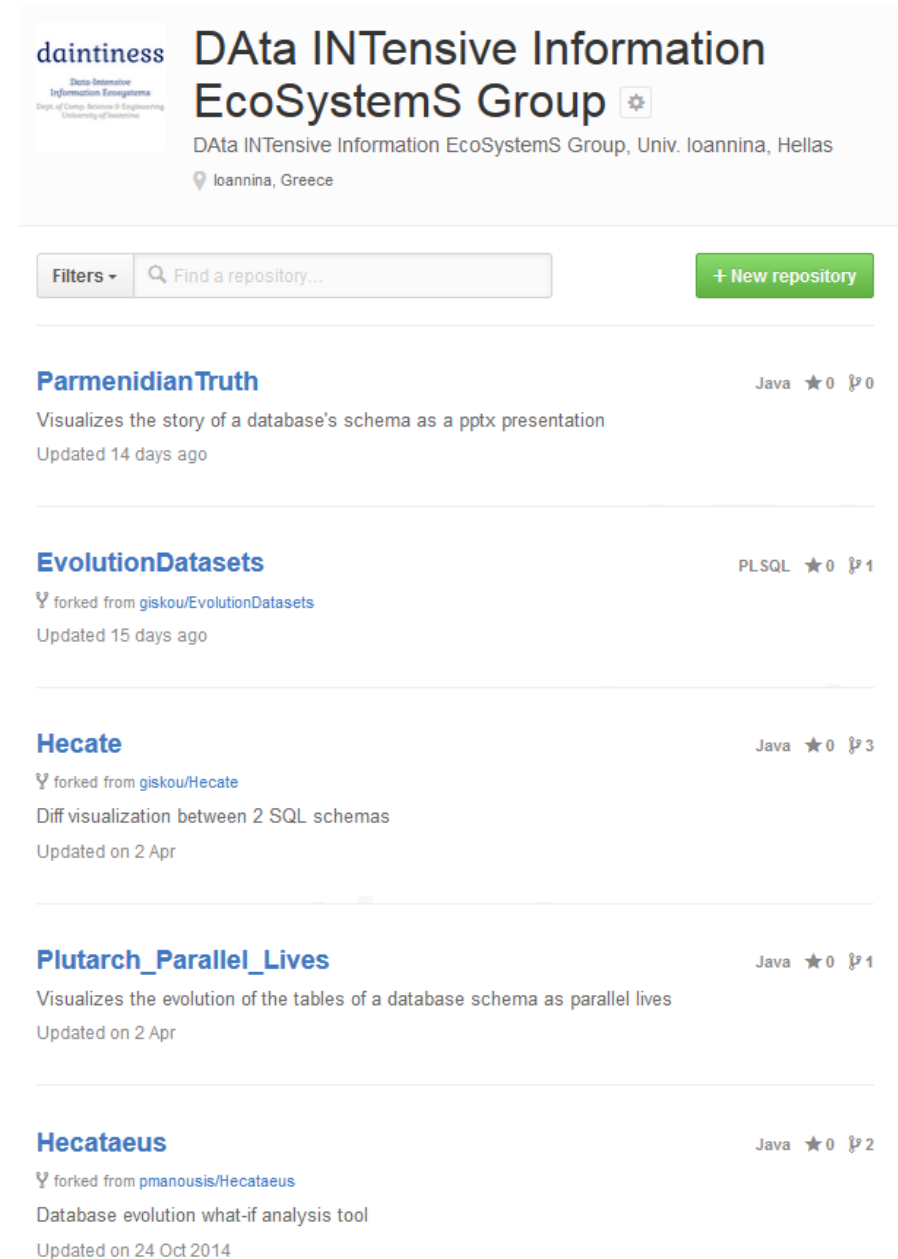
Data sets, Code, Results


[publications/2014 CAiSE/](#)

[publications/2015 ER](#)

Architecture Graphs &&
Hecataeus'

[projects/hecataeus/](#)




daintiness DATA INTensive Information EcoSystemS Group 

DATA INTensive Information EcoSystemS Group, Univ. Ioannina, Hellas


Ioannina, Greece

Filters [+ New repository](#)

ParmenidianTruth Java ★ 0  0


Visualizes the story of a database's schema as a pptx presentation

Updated 14 days ago

EvolutionDatasets PLSQL ★ 0  1

Forked from [giskou/EvolutionDatasets](#)


Updated 15 days ago

Hecate Java ★ 0  3

Forked from [giskou/Hecate](#)


Diff visualization between 2 SQL schemas

Updated on 2 Apr

Plutarch_Parallel_Lives Java ★ 0  1

Visualizes the evolution of the tables of a database schema as parallel lives

Updated on 2 Apr

Hecataeus Java ★ 0  2

Forked from [pmanousis/Hecataeus](#)

Database evolution what-if analysis tool

Updated on 24 Oct 2014