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
* Research Center “Athena”, IMIS, Athens
The nature that needs change is vicious; for it is not simple nor good...

*Nicomachean Ethics, Book VII, Aristotle*
SWEBOK Maintenance

- **Corrective** maintenance: reactive modification (or repairs) of a software product performed after delivery to correct discovered problems.

- **Adaptive** maintenance: modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.

- **Perfective** maintenance: modification of a software product after delivery to provide enhancements for users, improvement of program documentation, and recoding to improve software performance, maintainability, or other software attributes.

- **Preventive** maintenance: modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.
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**

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

```
EMP_ID    SALARY
100       1500
```

```
EMP_ID    SALARY
100       1650
```

• **Metadata – Schemata – Models**

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

```
EMP_ID    SALARY
100       1500
```

```
EMP_ID    SALARY    PHONE
100       1650    2107777777
```
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
Some exemplary code [Maule+08]

```csharp
10  public static IEnumerable<Experiment> Q1(DateTime d) {
11      DBParams dbParams = new DBParams();
12      DBRecordSet queryResult;
13      List<Experiment> exps = new List<Experiment>();
14
15      dbParams.Add("@ExpDate", d);
16
17      queryResult = QueryRunner.Run(
18          "SELECT Experiments.Name, Experiments.ExperimentId" +
19          " FROM Experiments" +
20          " WHERE Experiments.Date=@ExpDate",
21          dbParams);
22
23      while (queryResult.MoveNext()) {
24          exps.Add(new Experiment(queryResult.Record));
25      }
26
27      return exps;
28  }
```
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 ...


Roadmap

• Evolution of views
• Data warehouse Evolution
  – A case study if time
• Impact assessment in ecosystems
• Empirical studies for database evolution
• Open Issues and discussions
Roadmap

- Evolution of views
- Data warehouse Evolution
- A case study (if time)
- Impact assessment in ecosystems
- Empirical studies concerning database evolution
- Open Issues and discussions

- 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?
  – 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?
• 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 $V$ which becomes $V'$

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

We want to maintain $V_0$ given its old extent and the source relations.

Algebraically: $V' = V - V^- U V^+$

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
FROM R1 & ... & Rm
WHERE Q1 AND NOT C1 AND ... AND NOT Ck )
//add $V^+$ (here: empty)
```
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 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.
**Roadmap**

- Evolution of views
- **Data warehouse Evolution**
- A case study (if time)
- Impact assessment in ecosystems
- Empirical studies concerning database evolution
- Open Issues and discussions

- DWs as Views
- Evolving dimensions & SCD
- Multiversion DWs & cross-version queries
- Bonus: a case study, if time permits

**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.
**DW = set of mat. views?**

Sources

\[ \begin{align*}
S_1_{\text{PARTSU}}_{\text{PP}} \\
S_2_{\text{PARTSU}}_{\text{PP}}
\end{align*} \]

\[ U \]

**DW**

\[ \begin{align*}
\text{DW.PARTSUPP.DATE, DAY} & \quad \text{PKEY, MONTH AVG(COST)} \\
\text{TIME} & \quad \text{PKEY, DAY MIN(COST)}
\end{align*} \]
NO!! DW ≠ Mat. views!
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)

• Also: **ADD ATTRIBUTE** to equip views with attributes not present in the sources (e.g., timestamps, calculations)

• 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
[Create] view <view name> As <query expression>;
[Add Attribute <attribute name>:<query expression>=<type>]
[Hide Attribute <attribute name>];

Used for schema evolution too. See how to change the type of an attribute

Create View V’ As Select From V;
Hide attribute A;
Add attribute A : T /* new type for A in V’ */
Drop view V;
Rename V’ as V;
• 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**

<table>
<thead>
<tr>
<th></th>
<th>Relation</th>
<th>View</th>
<th>Attribute</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Delete</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rename</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Redefine semantics</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
... and then came dimension buses and multidimensional models ...

• ... which treat the DW as 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 unambiguous, 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 ...
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
Quick guide to dimensional modeling

For each dimension:

- **Levels** for “granularity degrees” of information
- Each level L with a domain 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
Research issue:

- What is the **algebra** of operators to change the dimensions of a DW?
- What are the **operators**?
- How do they affect the **schema and data** of dimensions and cubes?

[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.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalize</td>
<td>Adds a new level above a preexisting one, + a rollup function</td>
</tr>
<tr>
<td>Specialize</td>
<td>Adds a new level below the current bottom level + a rollup function</td>
</tr>
<tr>
<td>Relate</td>
<td>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.</td>
</tr>
<tr>
<td>Unrelate</td>
<td>Deletes an edge between two levels.</td>
</tr>
<tr>
<td>Delete Level</td>
<td>Deletes a level with the precondition that the new hierarchy must have a unique bottom level (ALL cannot be deleted).</td>
</tr>
<tr>
<td>Add Instance</td>
<td>Adds a value, say x, + a pair of the form (x,y) for each rollup function</td>
</tr>
<tr>
<td>Delete Instance</td>
<td>Deletes a value x from a level L + rollup functions</td>
</tr>
<tr>
<td>Reclassify</td>
<td>Update rollup-memberships (e.g., a brand moves to a new company)</td>
</tr>
<tr>
<td>Split &amp; Merge</td>
<td>Czechoslovakia &lt;-&gt; Czechia &amp; Slovakia + rollup functions</td>
</tr>
<tr>
<td>Update</td>
<td>Rename value without structural changes</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Algebraic Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert level</td>
</tr>
<tr>
<td>Delete level</td>
</tr>
<tr>
<td>Insert Attribute</td>
</tr>
<tr>
<td>Delete Attribute</td>
</tr>
<tr>
<td>Connect attribute to dimension level</td>
</tr>
<tr>
<td>Disconnect attribute from dimension level</td>
</tr>
<tr>
<td>Connect attribute to fact</td>
</tr>
<tr>
<td>Disconnect attribute to fact</td>
</tr>
<tr>
<td>Insert classification relationship</td>
</tr>
<tr>
<td>Delete classification relationship</td>
</tr>
<tr>
<td>Insert fact</td>
</tr>
<tr>
<td>Delete fact</td>
</tr>
<tr>
<td>Insert dimension into fact</td>
</tr>
<tr>
<td>Delete Dimension</td>
</tr>
</tbody>
</table>
... 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 spans versions, and transform its results into a convenient schema to show to the users.

Closely related to temporal management in DW’s

See later today presentation by Waqas Ahmed
Cross-Version Querying

Schema

Data

t1

t2

Q

Q
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 $a_2$.

<table>
<thead>
<tr>
<th>T</th>
<th>A1</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30%</td>
<td>70%</td>
</tr>
</tbody>
</table>

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.
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)

$$C = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ b_2 & b_3 & b_3 \end{bmatrix} = \begin{bmatrix} 3 & 7 & 5 \\ 10 & 8 & 6 \\ 20 & 13 & 5 \end{bmatrix}$$

**Transformation matrices for dimensions**

$$T_A = \begin{bmatrix} a_{11} & a_{12} & a_2 & a_3 \\ a_1 & 0.3 & 0.7 & 0 & 0 \\ a_2 & 0 & 0 & 1 & 0 \\ a_3 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \quad T_B = \begin{bmatrix} b_{12} & b_1 & b_2 & b_3 \\ b_1 & 1 & 1 & 0 \\ b_2 & 0 & 0 & 1 \end{bmatrix} \quad \quad \begin{bmatrix} a_{11} & a_{12} & a_2 & a_3 \\ b_{12} & 3.9 & 9.1 & 15 & 11 \\ b_3 & 6 & 14 & 13 & 5 \end{bmatrix}$$

**Final cube** with the **values** of the **original version** over the “**structure**” of the **new version**

Photos stolen from the paper
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.
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
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
Augmented schema of a previous version

- Assume a version $S_k$
- Assume a set of changes $M_1, ..., M_n$
- 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, ...
• Assume a fact
  – \texttt{SALES(ProdID, DayId, CustId, Price, Qty)}
• With a set of dimensions
  – \texttt{Product (Product, Type, Family)}
  – \texttt{Customer(Customer, CustGroup)}
  – \texttt{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
  – SumSales = Qty*Price
• Then, the new fact is
\texttt{SALES'(ProdID,MonthId,CustId,SalesmanId,Price,Qty,SumSales)}
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
History is a sequence of versions \(H = (v_1, ..., v_n)\). Each version has
- Its own schema
- The augmented schema wrt \(v_n\) \(\text{\ needs modification if } v_{n+1} \text{ comes}\)
- 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 get a uniform result under the last known schema.

If differences (e.g., because of attribute deletions), we retain the common set of attributes
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

• 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)
• 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

Wrapper
Wrapper
Wrapper

Monitor
Monitor
Monitor

Metadata manager
Refresher

DW
A CASE STUDY OF DW EVOLUTION

Roadmap
• Evolution of views
• Data warehouse Evolution
• A case study (if time)
• Impact assessment in ecosystems
• Empirical studies concerning database evolution
• Open Issues and discussions


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 module 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.
CREATE TABLE EMP (EMP# INTEGER PRIMARY KEY, 
NAME VARCHAR(25) NOT NULL, 
TITLE VARCHAR(10), 
SAL INTEGER NOT NULL);
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#
CREATE VIEW V AS
SELECT Emp#, Hours
FROM EMP E, WORKS W
WHERE E.Emp# = W.Emp#
AND E.Sal >= 50K

SELECT Emp#, SUM(Hours) as T_HOURS
FROM V
GROUP BY Emp#
Zooming out to top-level nodes (modules)
**Metrics: Node Degree**

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

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

Edge direction: from dependant to depended upon
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

![Diagram](image-url)
Metrics: Node Entropy

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

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

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

<table>
<thead>
<tr>
<th>Event Type</th>
<th># tables affected</th>
<th>Occurrences</th>
<th>pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Attribute</td>
<td>8</td>
<td>122</td>
<td>29%</td>
</tr>
<tr>
<td>Add Constraint</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Drop Attribute Count</td>
<td>5</td>
<td>34</td>
<td>8%</td>
</tr>
<tr>
<td>Modify Attribute</td>
<td>9</td>
<td>16</td>
<td>4%</td>
</tr>
<tr>
<td>Rename Attribute</td>
<td>5</td>
<td>236</td>
<td>57%</td>
</tr>
<tr>
<td>Rename Table</td>
<td>7</td>
<td>7</td>
<td>2%</td>
</tr>
</tbody>
</table>

Breakdown per event type

- Add Attribute: 29%
- Add Constraint: 0%
- Drop Attribute Count: 8%
- Modify Attribute: 4%
- Rename Attribute: 57%
- Rename Table: 2%

**ATTN: change of requirements at the real world determines pct breakdown!!**
Workflow of the first ETL scenario, ETL1
ETL1 -- Actual # affected nodes vs Graph Metrics

- Filter
- Join
- Project
Out – degree
- Schema size for tables
- Output schema size for activities
Pretty good job for tables
Decent job for filters and joins
Not so good for projection activities
Strength out did not work so well -- esp. for tables, it is too bad
Strength-total works the other way around
Workflows of the second & third ETL scenarios, ETL2 – ETL3
ETL2 -- Actual # affected nodes vs Graph Metrics

- Affected
- DEGREE OUT
- DEGREE TOTAL
- STRENGTH IN

Diagram showing the relationship between various metrics and nodes in a graph.
Suddenly everything is underestimated.

Pretty good job in the left part.
ETL4 -- Actual # affected nodes vs Graph Metrics

Transitive metrics to the rescue
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

<table>
<thead>
<tr>
<th>ETL Construct</th>
<th>Most suitable Metric</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Tables</td>
<td><code>out-degree</code></td>
<td>Retain small schema size</td>
</tr>
<tr>
<td>Intermediate &amp; Target Tables</td>
<td><code>out-degree</code></td>
<td>Retain small schema size in intermediate tables</td>
</tr>
<tr>
<td>Filtering activities</td>
<td><code>out-degree, out-strength</code></td>
<td>Retain small number of conditions</td>
</tr>
<tr>
<td>Join Activities</td>
<td><code>out-degree, out-strength</code>, <code>trans. out-degree, trans. out-strength, entropy</code></td>
<td>Move to early stages of the workflow</td>
</tr>
<tr>
<td>Project Activities</td>
<td><code>out-degree, out-strength</code>, <code>trans. out-degree, trans. out-strength, entropy</code></td>
<td>Move attribute reduction activities to early stages of the workflow and attribute increase activities to later stages</td>
</tr>
</tbody>
</table>
Roadmap
• Evolution of views
• Data warehouse Evolution
• A case study (if time)
• Impact assessment in ecosystems
• Empirical studies concerning database evolution
• Open Issues and discussions

... 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

• Like all software systems, they too change...
Evolving data-intensive ecosystem

**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

**Report on DBI, DBII Grades**

FROM V_TR V1 ⟷ V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME = 'DBI'
AND V2.C_NAME = 'DBII'

**View for Student Transcripts**

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

**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-intensive ecosystem

The impact can be **syntactical** (causing crashes), **semantic** (causing info loss or inconsistencies) and related to the performance
The impact of changes & a wish-list

- **Syntactic**: scripts & reports simply crash
- **Semantic**: views and applications can become inconsistent or information losing
- **Performance**: can vary a lot

We would like: *evolution predictability*

i.e., control of **what will be affected**

before changes happen

- Learn what changes & how
- Find ways to quarantine effects
The **Hecataeus** tool & method. 
Here: a first map of Drupal

What happens if I modify table search_index? Who are the neighbors?
What happens if I modify table search_index? Who are the neighbors?
In the file structure too...
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)
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

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.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;
How to handle evolution?

Propagate ALL possible evolution events

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 JOIN CourseStd CS JOIN Course C

Remove CS.C_NAME

Propagate ALL possible evolution events

Block Deletion for V_TR.C_NAME

Propagate all other possible evolution events

Report on DBI, DBII Grades

FROM V_TR V1 JOIN V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME = 'DBI'
AND V2.C_NAME = 'DBII'

As V1

As V2

Report on Average Grade

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

Propagate ALL possible evolution events

// 2 policy rules suffice to annotate the entire ecosystem:
NODE: ON * THEN PROPAGATE;
Q_pass2courses_IN_V1.C_SNAME ON DELETE_SELF THEN BLOCK;
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.
Impact assessment & rewriting
Conflicts: what they are and how to handle them (more than flooding)

BEFORE

- View0 initiates a change
- View1 and View2 accept the change
- Query2 rejects the change
- Query1 accepts the change

AFTER

- 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

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

2. Queries Q215 and Q216 vetoed
Other efforts

• Maule et al @ ICSE 2008
• The Prism/Prism++ line of research
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.
[On the side] code assumed to be of the form:

```csharp
10  public static IEnumerable<Experiment> Q1(DateTime d) {
11     DBParams dbParams = new DBParams();
12     DBRecordSet queryResult;
13     List<Experiment> exps = new List<Experiment>();
14
15     dbParams.Add("@ExpDate", d);
16
17     queryResult = QueryRunner.Run(
18         "SELECT Experiments.Name,Experiments.ExperimentId"
19         + " FROM Experiments" +
20         " WHERE Experiments.Date=@ExpDate",
21         dbParams);
22
23     while (queryResult.MoveNext()) {
24         exps.Add(new Experiment(queryResult.Record));
25     }
26
27     return exps;
28 }
```
... whereas what you can get is ...

```php
$query = db_select('comment', 'c1');
$query->innerJoin('comment', 'c2', 'c2.nid = c1.nid');
$query->addExpression('COUNT(*), 'count');
$query->condition('c2.cid', $cid);
if (!user_access('administer comments')) {
    $query->condition('c1.status', COMMENT_PUBLISHED);
}
$mode = variable_get('comment_default_mode', $node_type, COMMENT_MODE_THREADED);
if ($mode == COMMENT_MODE_FLAT) {
    $query->condition('c1.cid', $cid, '<');
} else {
    $query->where('SUBSTRING(c1.thread, 1, (LENGTH(c1.thread) - 1)) < SUBSTRING(c2.thread, 1, (LENGTH(c2.thread) - 1))');
}
return $query->execute()->fetchField();
```
Prism/Prism++

- Series of works from the same authors
Prism/Prism++ motivation

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

> Automatically migrate schema + data+ surrounding queries

- Schema Modification Operators (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.

<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 \( S.ID = T.ID \)
Roadmap

• Evolution of views
• Data warehouse Evolution
• A case study (if time)
• Impact assessment in ecosystems
• Empirical studies concerning database evolution
• Open Issues and discussions

A timeline of efforts concerning empirical studies
Results by Univ. Ioannina

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

Sjoberg
IST 93
1993

Curino+
ICEIS08
2008

Univ. Riverside
IWPSE09, ICDEW11
2009

Qiu, Li, Su
FSE’13
2011

Univ. Ioannina
CAiSE14, ER15
2015
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 where balanced.
Timeline of empirical studies

Curino+ @ ICEIS08: Mediawiki for 4.5 years
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

- Sjoberg: IST 93
- Curino+: ICEIS08
- Univ. Riverside: IWPSE09, ICDEW11
- Qiu, Li, Su: FSE’13
- Univ. Ioannina: CAiSE14, ER15

To be detailed next.
CAiSE14: DB level
ER’15: Table level
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 graphical representations and statistically studied the measures, both at the macro (database) and at the micro (table) level
.. What do we see if we observe the evolution of the entire schema?


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

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

- Parses DDL files
- Creates a model for the parsed SQL elements
- Compares two versions 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
Hecate: SQL schema diff viewer
Schema Size (relations)

Schema Growth (diff in #tables)

Change over time

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

What do we see if we observe the evolution of individual tables?


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).

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.

<table>
<thead>
<tr>
<th></th>
<th>&lt;5</th>
<th>5-10</th>
<th>&gt;10</th>
</tr>
</thead>
<tbody>
<tr>
<td>atlas</td>
<td>10,23%</td>
<td>68,18%</td>
<td>21,59%</td>
</tr>
<tr>
<td>biosql</td>
<td>75,56%</td>
<td>24,44%</td>
<td>0,00%</td>
</tr>
<tr>
<td>coppermine</td>
<td>52,17%</td>
<td>30,43%</td>
<td>17,39%</td>
</tr>
<tr>
<td>ensembl</td>
<td>54,84%</td>
<td>38,06%</td>
<td>7,10%</td>
</tr>
<tr>
<td>mediawiki</td>
<td>61,97%</td>
<td>19,72%</td>
<td>18,31%</td>
</tr>
<tr>
<td>phpbb</td>
<td>40,00%</td>
<td>44,29%</td>
<td>15,71%</td>
</tr>
<tr>
<td>typo3</td>
<td>21,88%</td>
<td>31,25%</td>
<td>46,88%</td>
</tr>
<tr>
<td>opencart</td>
<td>57,20%</td>
<td>33,05%</td>
<td>9,75%</td>
</tr>
<tr>
<td>Average</td>
<td>46,73%</td>
<td>36,18%</td>
<td>17,09%</td>
</tr>
</tbody>
</table>
The $\Gamma$ Pattern: "if you 're wide, you survive"

- The $\Gamma$ 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.
These exceptions are due to the fact that they do not exceed 10 attributes.
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 whose schema is on average one standard deviation above the mean

• wide tables with large schema sizes demonstrating small to medium (typically around the middle of the y-axis) amount of change.
The inverse $\Gamma$ 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 $\Gamma$).
Quiet tables rule, esp. for mature db’s

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

<table>
<thead>
<tr>
<th></th>
<th>DIED</th>
<th>SURVIVED</th>
<th>Aggregate per update type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#tables</td>
<td>No change</td>
<td>Quiet (0-0.1)</td>
</tr>
<tr>
<td>atlas</td>
<td>88</td>
<td>8%</td>
<td>7%</td>
</tr>
<tr>
<td>biosql</td>
<td>45</td>
<td>20%</td>
<td>13%</td>
</tr>
<tr>
<td>phpbb</td>
<td>70</td>
<td>0%</td>
<td>3%</td>
</tr>
<tr>
<td>typo3</td>
<td>32</td>
<td>16%</td>
<td>6%</td>
</tr>
<tr>
<td>coppermine</td>
<td>23</td>
<td>4%</td>
<td>0%</td>
</tr>
<tr>
<td>ensembl</td>
<td>155</td>
<td>24%</td>
<td>23%</td>
</tr>
<tr>
<td>mwiki</td>
<td>71</td>
<td>14%</td>
<td>13%</td>
</tr>
<tr>
<td>opencart*</td>
<td>128</td>
<td>9%</td>
<td>2%</td>
</tr>
</tbody>
</table>

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
Longevity and update activity correlate!!

- Remember: top changers are defined as such wrt ATU (AvgTrxnUpdate), not wrt sum(changes)

- Still, they dominate the sum(changes) too! (see top of inverse $\Gamma$)

- See also upper right blue part of diagonal: too many of them are born early and survive => live long!
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)
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).
Few short lived tables are born and die in the mature life of the db.

Deleted tables are born early & last short & do not change a lot.

Die young and suddenly

[Early life of the db] 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.

[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.
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)!
All in one

- 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.
$\text{Gamma} \quad \text{Inv. } \text{Gamma, Empty Triangle}$

- **atlas: duration / size**
  - Duration vs. Schema size at birth for different categories of users.
  - Categories include: Sudden Death, Quiet, Dead, Active, Dead, Rigid, Quiet Survivor, Active Survivor.
  - Size: 266

- **coppermine: updates / duration**
  - Updates vs. Duration for different categories of users.
  - Categories include: Sudden Death, Rigid, Quiet Survivor, Active Survivor.

- **phpBB: updates / size**
  - Updates vs. Schema size at birth for different categories of users.
  - Categories include: Quiet, Dead, Active, Dead, Rigid, Quiet Survivor, Active Survivor.

- **mwiki: duration / birth**
  - Duration vs. Birth for different categories of users.
  - Categories include: Sudden Death, Quiet, Dead, Active, Dead, Rigid, Quiet Survivor, Active Survivor.
Where we stand
Open issues
... and discussions ...

OPEN ISSUES

Roadmap
• Evolution of views
• Data warehouse Evolution
• A case study (if time)
• Impact assessment in ecosystems
• Empirical studies concerning database evolution
• Open Issues and discussions
Where we stand

• We know to handle evolving materialized views
• And we also know to handle simple cases of DW evolution
• We have started some work on handling evolving ecosystems
• We have a first glimpse of the mechanics of database evolution
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?
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
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?

• How do current trends in DW technology relate to schema 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
Take Away Message

• Evolution is *viciously omnipresent*; due to its huge impact, it is leading to non-evolvable (rigid) data & software structures

• Practically:
  – Plan for evolution, well ahead of construction
  – So far, our solutions and tools help only so much
  – Industry not likely to help

• This is why *we can and have to do research*
  – We can do *pure scientific research* to find laws
  – We can do practical work for *tools and methods* that reduce the pain

... and don’t forget to put everything in the git ...
Thank you!  
Q&A

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

DB Schema Evolution
Data sets, Code, Results

publications/2014_CAiSE/
publications/2015_ER

Architecture Graphs && Hecataeus'
projects/hecataeus/

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