AALBORG UNIVERSITY
DENMARK

Programmatic ETL

Christian Thomsen, Aalborg University

eBISS 2017

daisy
Center for Data-intensive Systems

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- Open-sourcing pygrametl
- ETLMR
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

The ETL Process

- Extract-Transform-Load (ETL)
- The **most underestimated** process in DW development
- The **most time-consuming** process in DW development
  - Up to 70% of the development time is spent on ETL!
- Extract
  - Extract relevant data (from different kinds of sources)
- Transform
  - Transform data to DW format
  - Build DW keys, etc.
  - Cleansing of data
- Load
  - Load data into DW (time consuming)

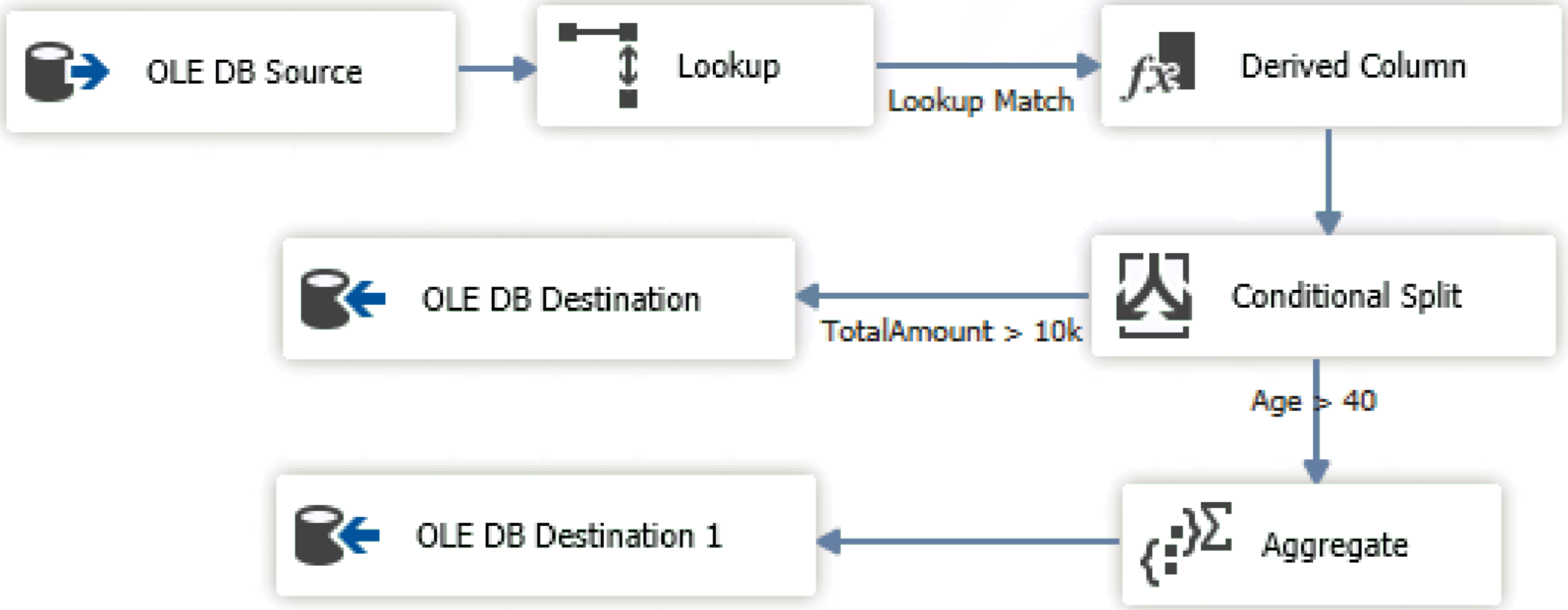
pygrametl's history

- We started the development of **pygrametl** in early 2009
- In a collaboration with industrial partners, we were using an existing GUI-based ETL tool on a real-life data set to be loaded into a snowflake schema
- Required ***a lot of clicking*** and ***tedious work***!

- In an earlier project, we could not find an ETL tool that fitted with the requirements and source data. Instead we wrote the ETL flow in Python code, but not in a reusable, general way

- We thought that there had to be an easier way ☺

GUI-based ETL flow



Motivation

- The Extract-Transform-Load (ETL) process is a crucial part for a data warehouse (DW) project
- Many commercial and open source ETL tools exist
- The dominating tools use graphical user interfaces (GUIs)
  - Pros: Easy overview, understood by non-experts, easy to use (?)
  - Cons: A lot of drawing/clicking, missing constructs, inefficient (?)
- GUIs do not automatically lead to high(er) productivity
  - A company experienced similar productivity with coding ETL in C
- Trained specialists use text efficiently
- ETL *developers* are (in our experience) trained specialists

Motivation – cont.

- We wish to challenge the idea that GUIs are always best for ETL
- For some ETL projects, a code-based solution is the right choice
  - “Non-standard” scenarios when …
    - fine-grained control is needed
    - required functionality not available in existing ETL tool
    - doing experimentation
  - Prototyping
  - Teams with limited resources
- Redundancy if each ETL program is coded from scratch
- A framework with common functionality is needed
- ***pygrametl***
  - a Python-based framework for ETL programming

Agenda

- Introduction to pygrametl
  - Motivation
  - Why Python?
  - Example
  - Dimension support
  - Fact table support
  - Flow support
  - Evaluation
  - Conclusion
- …

Why Python?

- Designed to support programmer productivity
  - Less typing – a Python program is often 2-10X shorter than a similar Java program
- Good connectivity
- Runs on many platforms (also .NET and Java)
- “Batteries included” – comprehensive standard libraries
- Object-oriented, but also support for functional programming
- Dynamically and strongly typed
- Duck typing

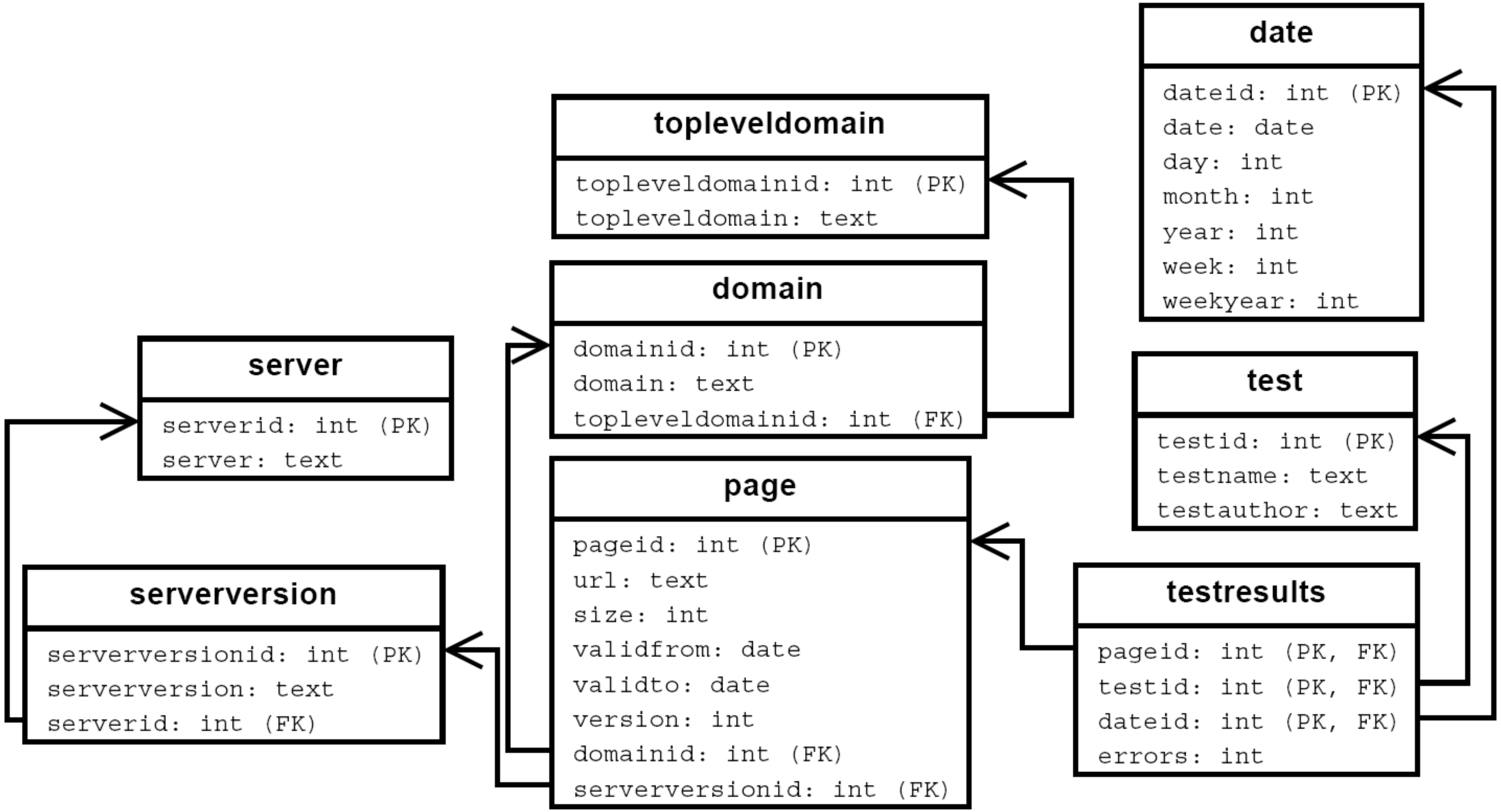
Rows in pygrametl

- We need to be able to handle *rows*. We just use Python's dict type:

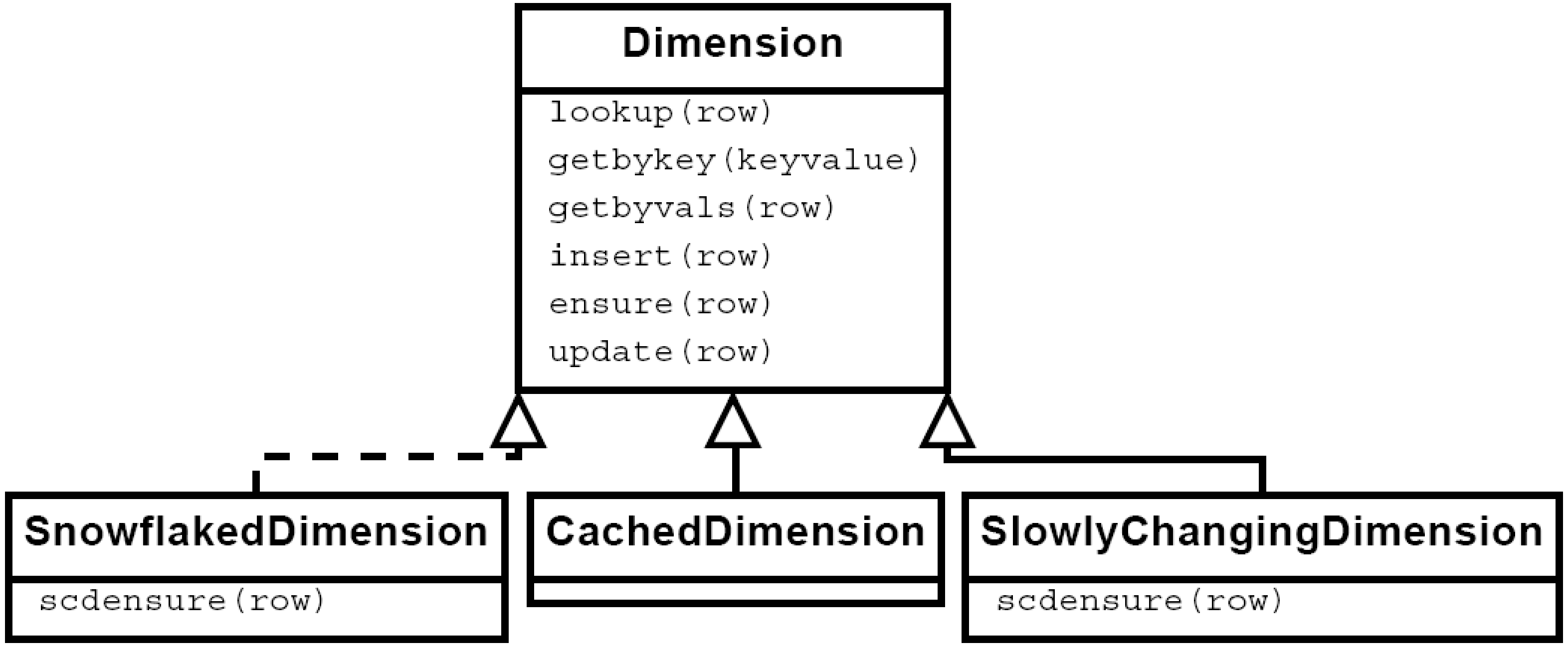
```
{'greeting':'Hello World', 'somevalue':42,  'status':'ok'}
{'greeting':'Hi',          'somevalue':3.14}
{'greeting':'Bonjour',     'somevalue':'?', 'status':'bad'}
```

- GUI-based toools can require that different rows entering a given ”step” have the same structure
- In pygrametl, we don't require this. The only requirement is that the needed data is available in a given row
  - Some of the needed data doesn't even have to be there if we can compute it on-demand
  - Data types can also differ, but should be of a ”usable” type
  - If this does not hold, an exception is raised at runtime

Example


date
dateid: int (PK)
date: date
day: int
month: int
year: int
week: int
weekyear: int
topleveldomain
topleveldomainid: int (PK)
topleveldomain: text
domain
domainid: int (PK)
domain: text
topleveldomainid: int (FK)
server
serverid: int (PK)
server: text
test
testid: int (PK)
testname: text
testauthor: text
page
pageid: int (PK)
url: text
size: int
validfrom: date
validto: date
version: int
domainid: int (FK)
serverversionid: int (FK)
serverversion
serverversionid: int (PK)
serverversion: text
serverid: int (FK)
testresults
pageid: int (PK, FK)
testid: int (PK, FK)
dateid: int (PK, FK)
errors: int

Dimension support



- The general idea is to create one `Dimension` instance for each dimension in the DW and then operate on that instance: `dimobject.insert(row)`

Dimension

```
testdim = Dimension(
```

Required

```
name="test", key="testid",
attributes=["testname", "testauthor"],
```

Optional

```
lookupatts=["testname"],
defaultidvalue=-1)
```

Further, we could have set

- `idfinder=somefunction`
  to find key values on-demand when inserting a new member
- `rowexpander=anotherfunction`
  to expand rows on-demand

Dimension's methods

- `lookup(row, namemapping={})`
  Uses the lookup attributes and returns the key value
- `getbykey(keyvalue)`
  Uses the key value and returns the full row
- `getbyvals(row, namemapping={})`
  Uses a subset of the attributes and returns the full row(s)
- `insert(row, namemapping={})`
  Inserts the row (calculates the key value if it is missing)
- `ensure(row, namemapping={})`
  Uses `lookup`. If no result is found, `insert` is used after the optional `rowexpander` has been applied
- `update(row, namemapping={})`
  Updates the row with the given key value to the given values

CachedDimension

- Like a Dimension but with caching

```
testdim = Dimension(
        name="test", key="testid",
        attributes=["testname", "testauthor"],
        lookupatts=["testname"],
        defaultidvalue=-1
        cachesize=500,
        prefill=True
        cachefullrows=True )
```

SlowlyChangingDimension

- Supports Slowly Changing Dimensions (type 2 (and 1))

```
pagedim = SCDimension(name="page", key="pageid",
  attributes=["url", "size", …],
  lookupatts=["url"],
  fromatt="validfrom",
  fromfinder=pygrametl.datereader("lastmoddate"),
  toatt="validto", versionatt="version")
```

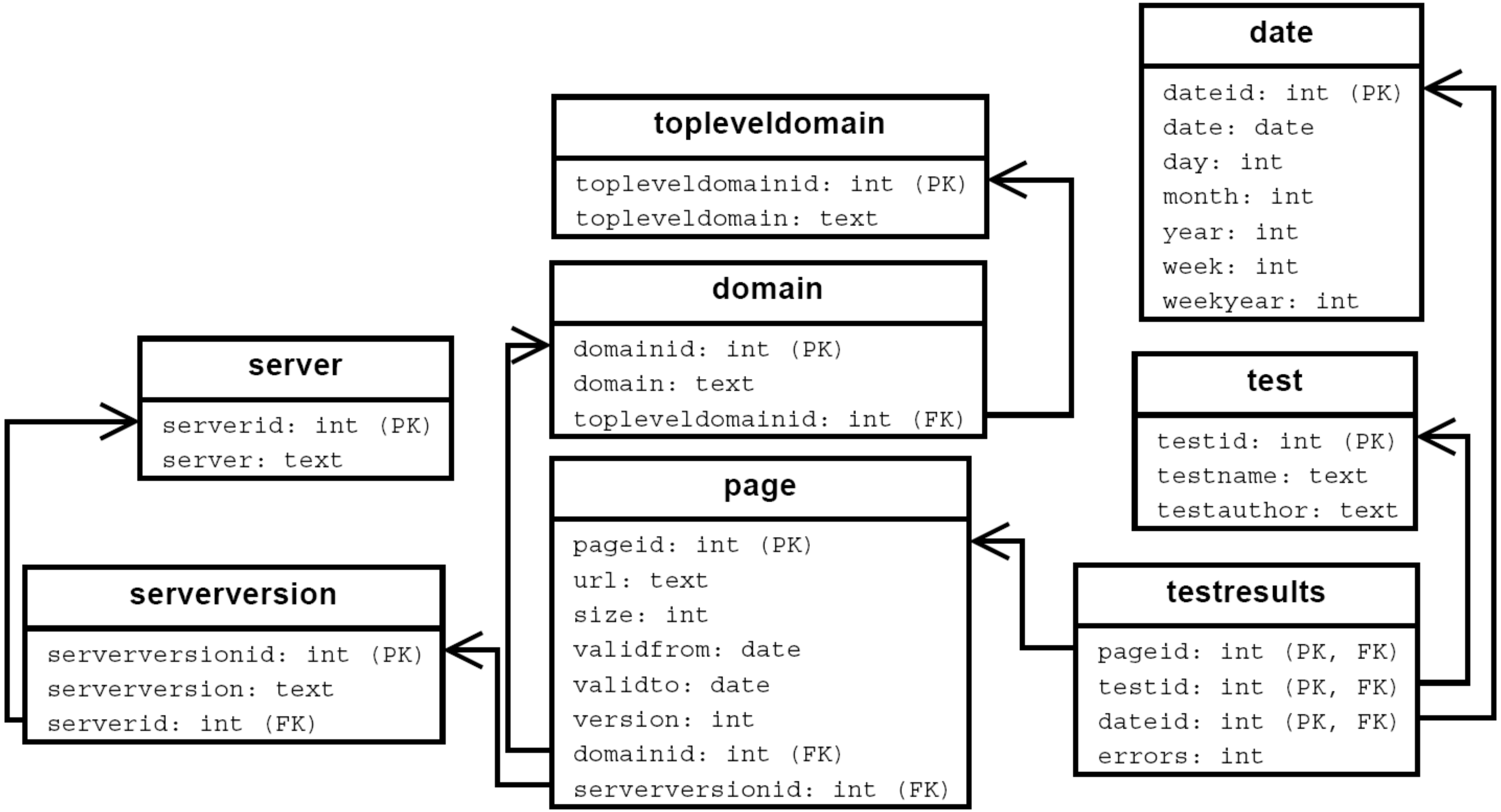
- We could also have given a list of "type 1 attributes", set a `tofinder`, and configured the caching
- Methods like `Dimension` plus `scdensure(row, namemapping={})` that is similar to `ensure` but detects changes and creates new versions if needed

SnowflakedDimension

- For filli
- Enoug
  objects

nsion

```
pagesf
([   (pa
     (se
     (dc
 ])
```



- We ca
  lookup
- Likewi

Fact table support

- `FactTable` – a basic representation
  - `insert(row, namemapping={})`
  - `lookup(row, namemapping={})`
  - `ensure(row, namemapping={})`
- `BatchFactTable`
  - Like `FactTable`, but inserts in *batches*.
- `BulkFactTable`
  - Only `insert(row, namemapping={})`
  - Does bulk loading by calling a user-provided function

```
facttbl = BulkFactTable(name="testresults",
    keyrefs=["pageid", "testid", "dateid"],
    measures=["errors"], bulksize=5000000,
    bulkloader=mybulkfunction)
```

Putting it all together

- The ETL program for our example:

```
[Declarations of Dimensions etc.]
 …
def main():
    for row in inputdata:
        extractdomaininfo(row)
        extractserverinfo(row)
        row["size"] = pygrametl.getint(row["size"])
        row["pageid"] = pagesf.scdensure(row)
        row["dateid"] = datedim.ensure(row)
        row["testid"] = testdim.lookup(row)
        facttbl.insert(row)
    connection.commit()
```

Flow support

- A good aspect of GUI-based ETL programming is that it is easy to keep different tasks separated
- pygrametl borrows this idea and supports *Steps* (with encapsulated functionality) and flows between them
- A `Step` can have a following `Step`
- The basic class `Step` offers (among other) the methods
  - `defaultworker(row)`
  - `_redirect(row, target)`
  - `_inject(row, target=None)`

- pygrametl has some predefined `Step`s:
  `MappingStep, ValueMappingStep, ConditionalStep,` …

Flow support – experiences

- It turns out that `Step`s are not used often
- Nearly no questions/comments received about them

- Do users not want to be limited to express their ETL process in terms of steps and connections when they decide to use code for the ETL process?

Evaluation

- We implemented ETL solutions for the example in pygrametl and Pentaho Data Integration (PDI)
  - PDI is a leading open source GUI-based ETL tool
  - Ideally, commercial tools should also have been used but commercial licenses often forbid publication of performance results
- Difficult to make a complete comparison…
  - We have experience with PDI but we wrote pygrametl
  - A full-scale test would require teams with fully trained developers
- We evaluated development time
  - each tool was used twice – in the first use, we had to find a strategy, in the latter use we only found the interaction time
- … and performance
  - on generated data with 100 million facts

Comparison

pygrametl

- 142 lines (incl. whitespace and comments),
  56 statements

- 1st use: 1 hour
- 2nd use: 24 minutes

PDI

- 19 boxes and 19 arrows

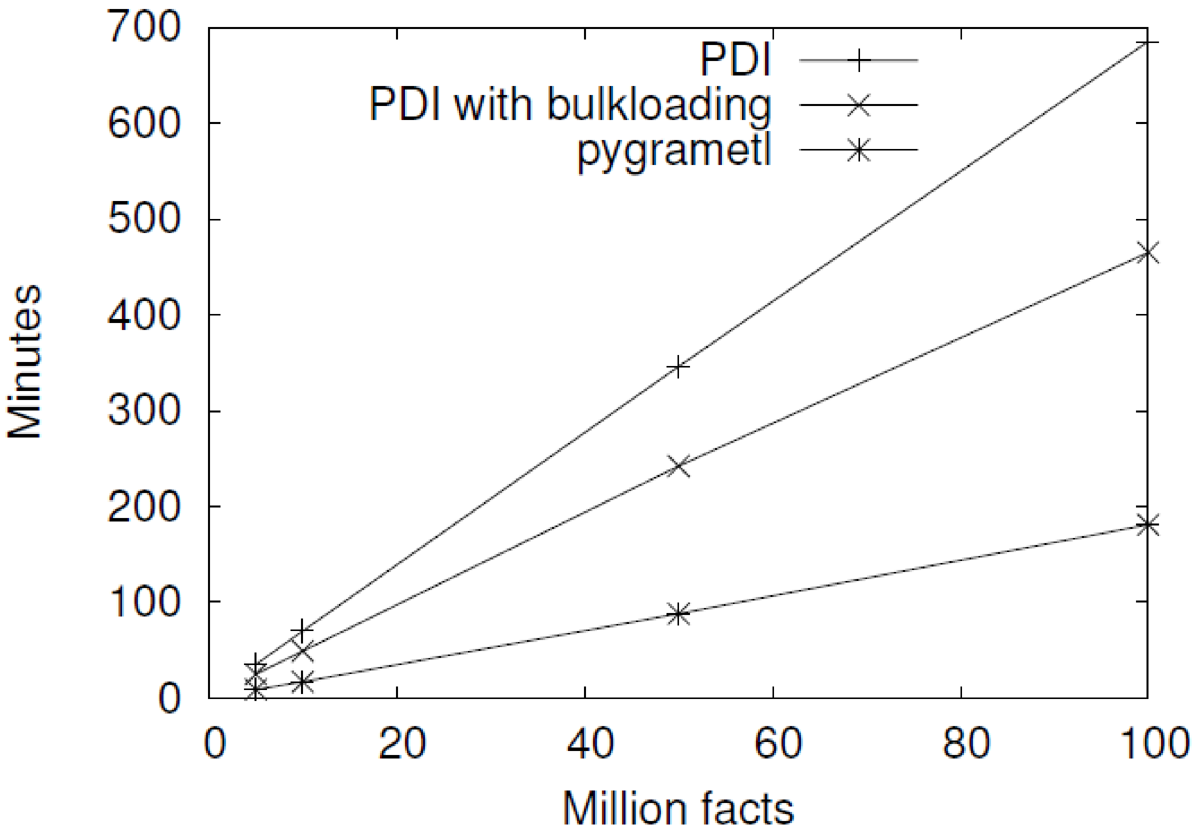
- 1st use: 2 hours
- 2nd use: 28 minutes

Performance test

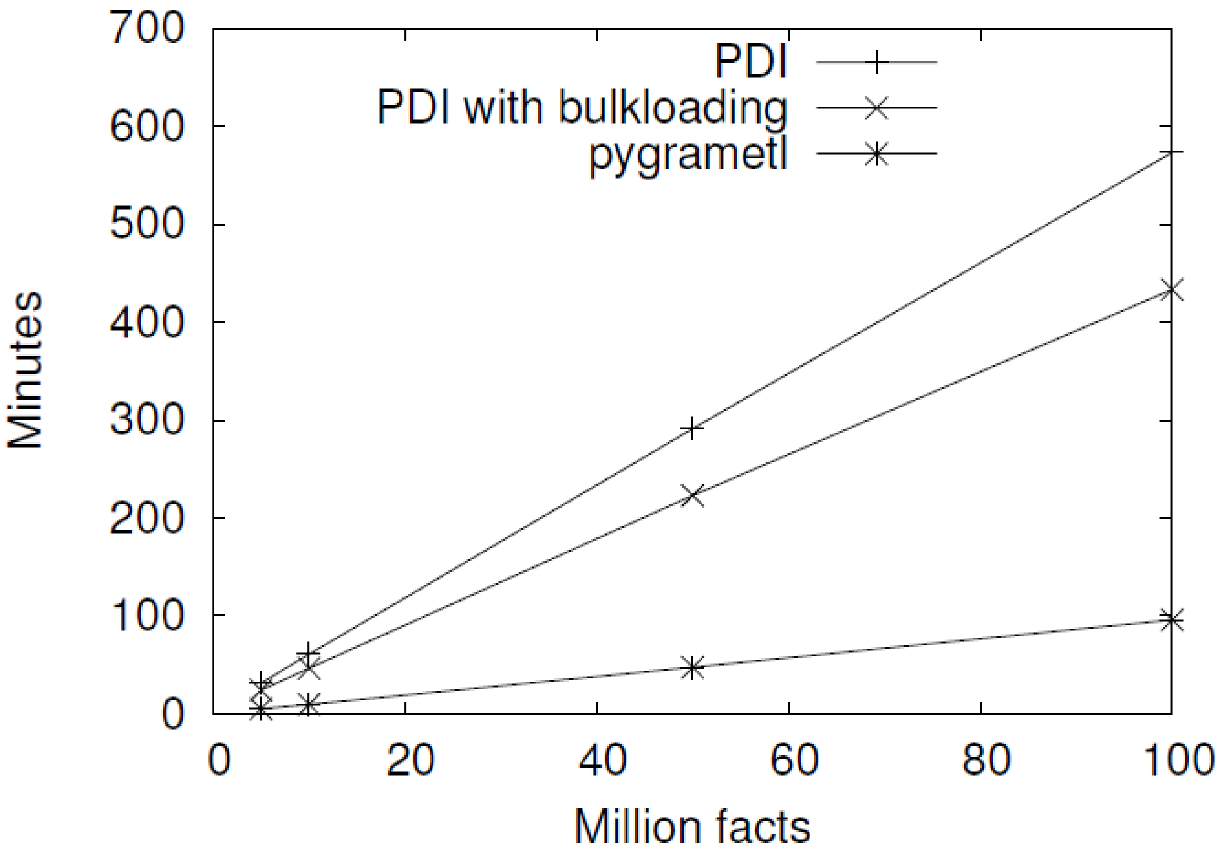
- [Experiment from DOLAP'09 but on new hardware and newer software versions]
- Uses the running example
  - 2,000 domains each with 100 pages and 5 tests
  - ➔ One considered month gives 1 million facts
  - We get ~100,000 new page versions per month after the 1st month

- VirtualBox with 3 virtual CPUs; host has 2.70GHz i7 with 4 cores and hyperthreading
- VirtualBox with 16GB RAM; host has 32GB
- Host has SSD disk
- Linux as guest OS; host runs Windows 10
- Python 3.6, OpenJDK 8, PostgreSQL 9.4
- pygrametl 2.5, PDI 7.1
- Both pygrametl and PDI were allowed to cache all dimension data

Performance



Wall-clock time



CPU time

Conclusion and future work

- We challenge the conviction that ETL is always best done by means of a GUI
- We propose to let ETL developers do ETL programming by writing code
- To make this easy, we provide *pygrametl*
  - a Python-based framework for ETL programming
- Some persons prefer a graphical overview of the ETL process
  - The optimal solution includes both a GUI and code
  - It would be interesting to make a GUI for creating and connecting steps
  - Updates in code should be visible in GUI and vice versa
    - “Reverse engineering” & “roundtrip engineering”
- Next, we will consider a simple and efficient way to create parallel ETL programs in pygrametl

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- **Explicit parallelism in pygrametl**
- A case-study
- Open-sourcing pygrametl
- ETLMR
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

Introduction

- Parallelization is often needed to handle the data volumes
  - Task parallelism
  - Data parallelism

- Parallelization makes the development more complex

- We present general functionality for performing parallel ETL
- Parallelization should fit with pygrametl's simplicity
- It should be easy to turn a non-parallel program into a parallel one

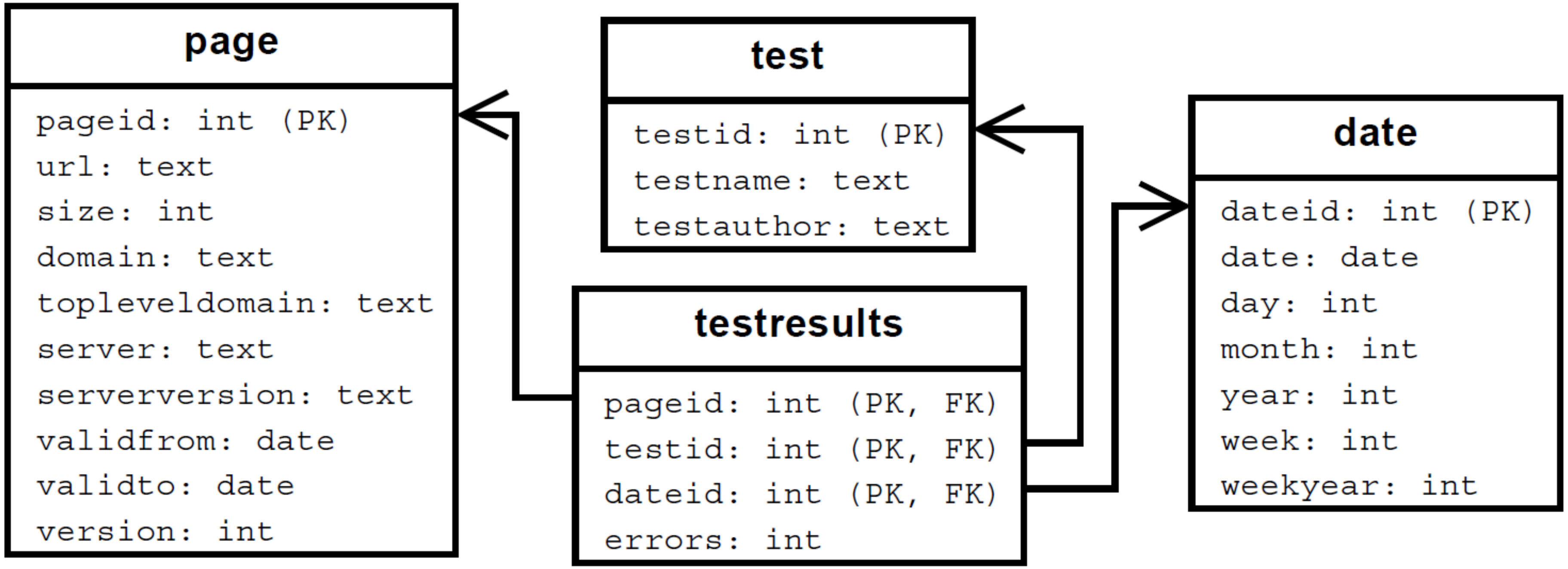
- With little more CPU time, much less wall-clock time is used

Extract

- It is often time-consuming to extract data from sources
- Do it in another process/thread
  - Do also transformations in the other process in this example

```
• downloadlog = CSVSource(open(…))
• testresuls = CSVSource(open(…))

• joineddata = MergeJoiningSource(downloadlog, ‘localfile’,
                                  testresults, ‘localfile’)

• transformeddata = TransformingSource(joineddata, sizetoint,
                         extractdomaininfo, extractserverinfo)

• inputdata = ProcessSource(transformeddata)
```

Example



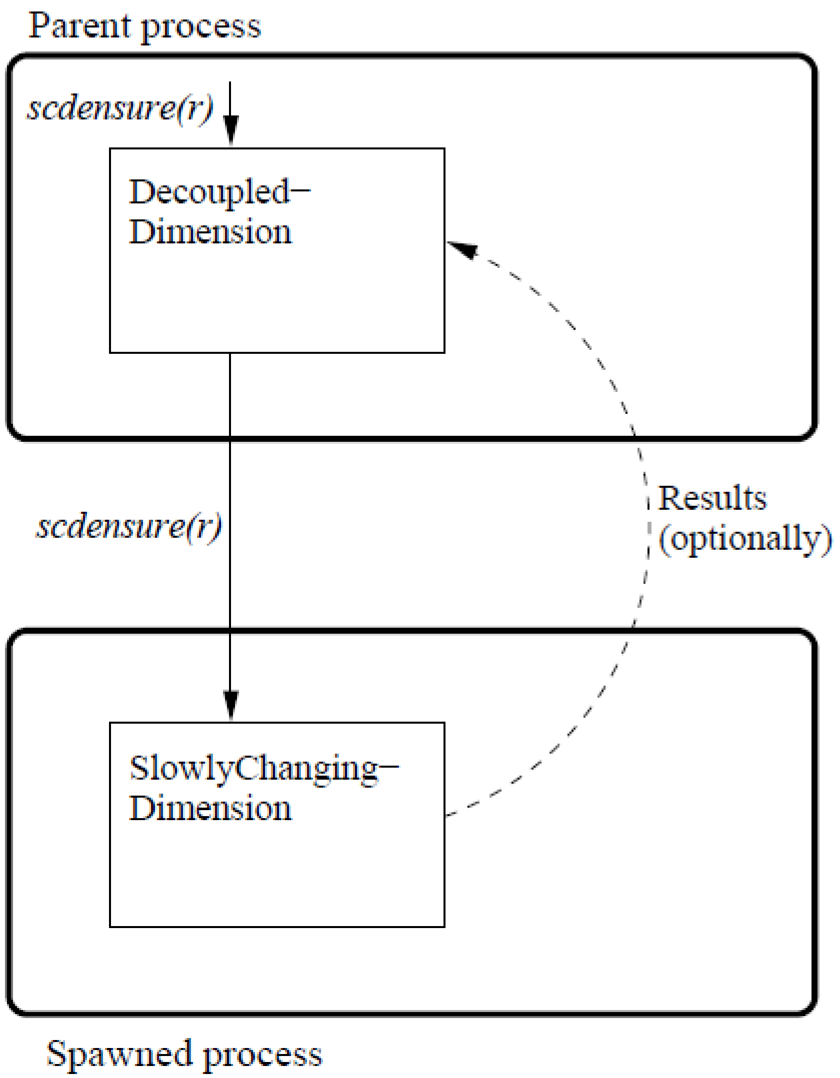
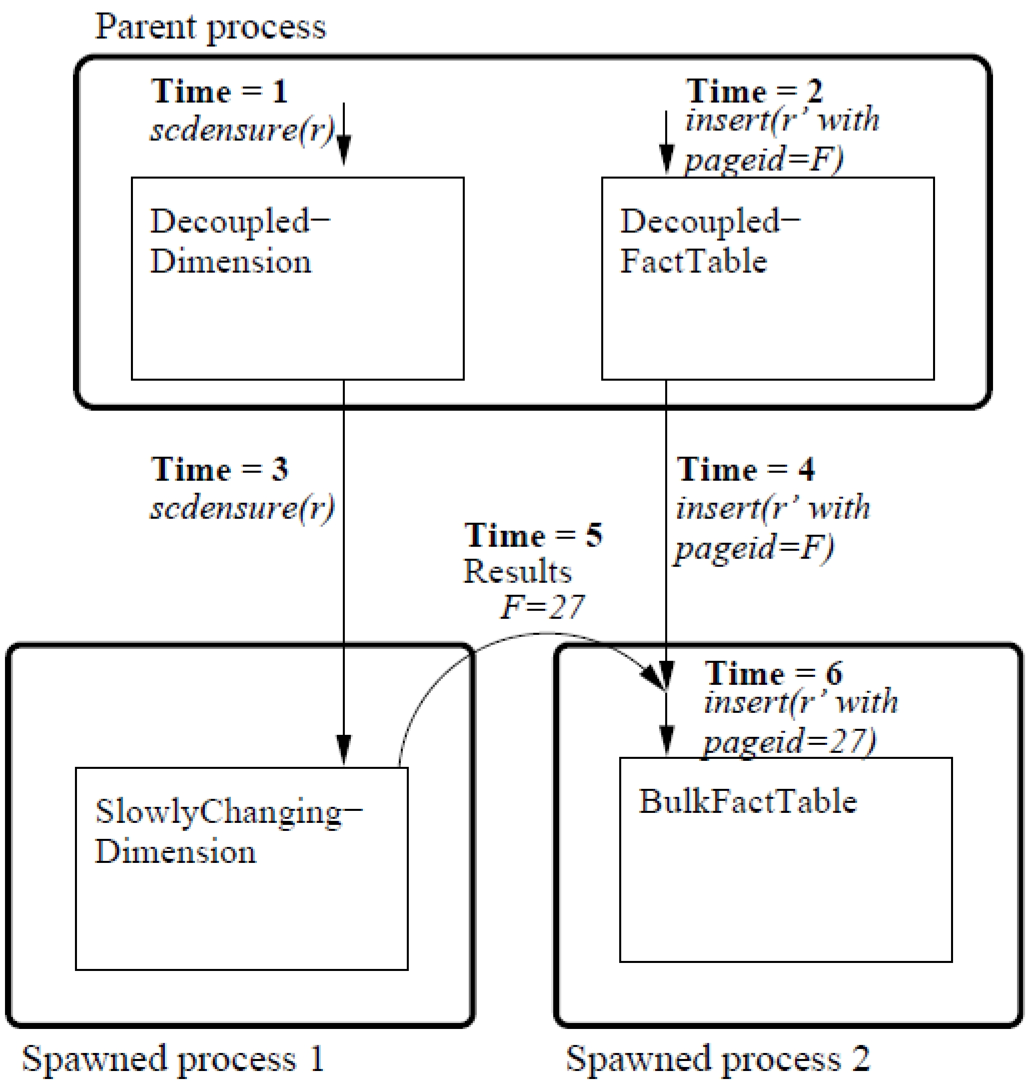
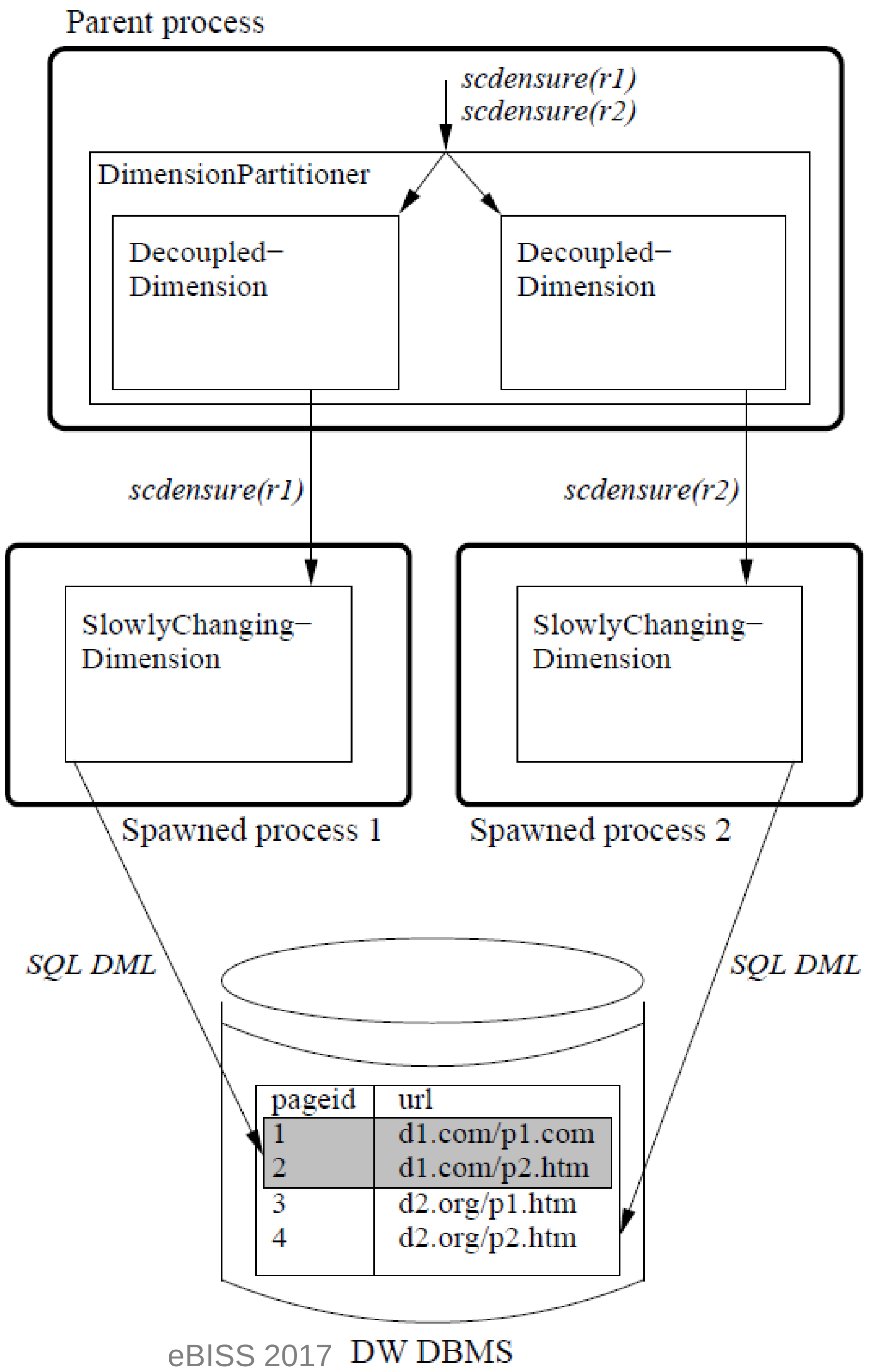
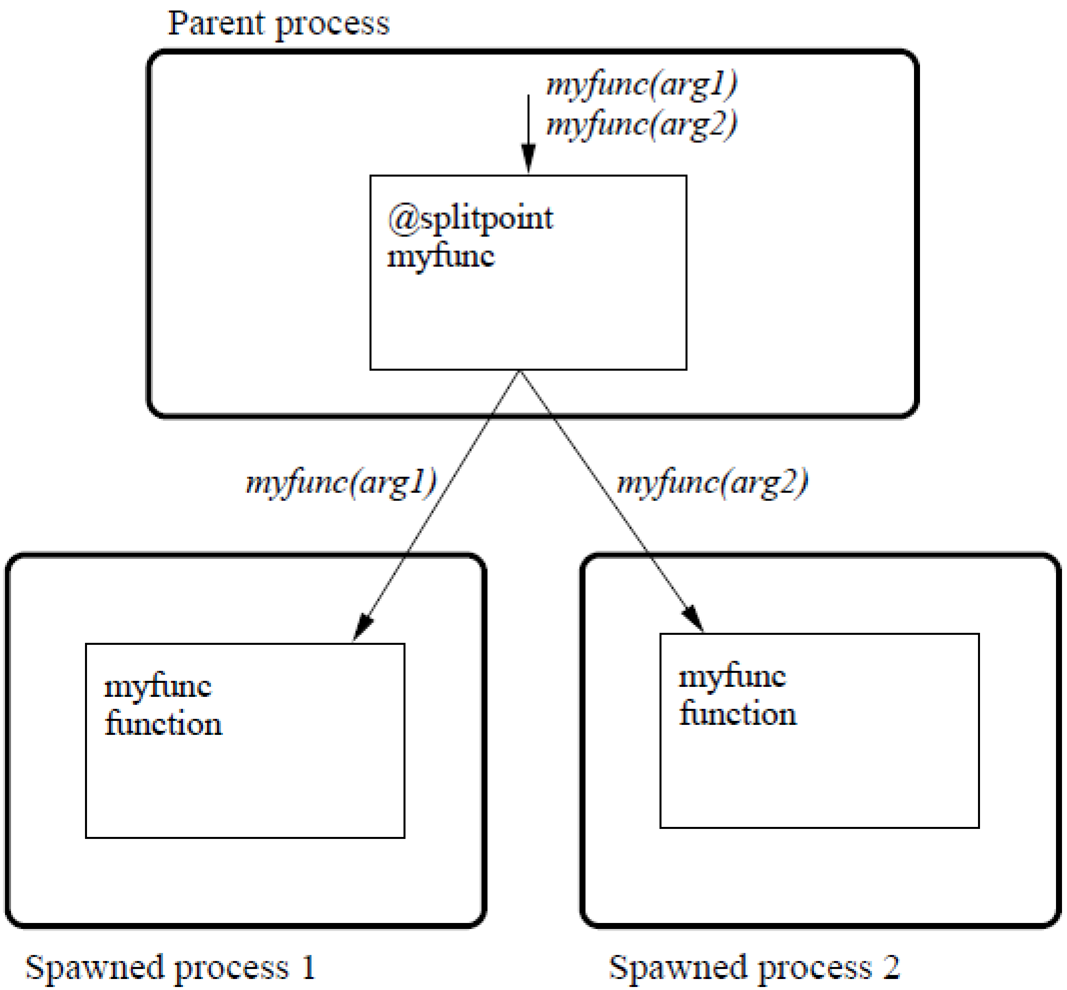
A type-2 Slowly Changing Dimension with much data

```
[Declarations not shown]

for row in inputdata:
   row['testid'] = testdim.lookup(row)
   row['dateid'] = datedim.ensure(row)
   row['pageid'] = pagedim.scdensure(row)
   facttbl.insert(row)
```

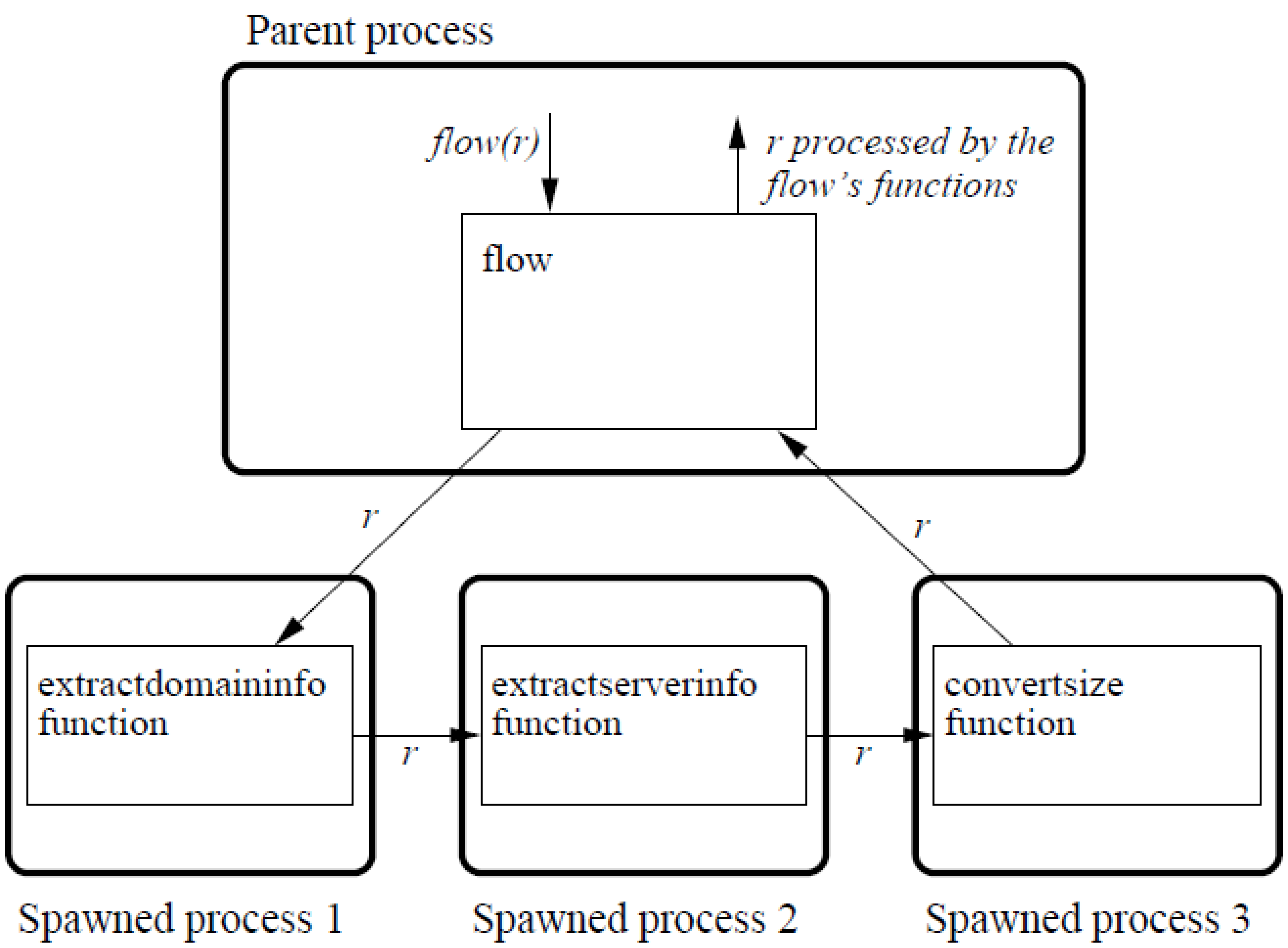
Decoupled objects

- Much time is spent on dimension and fact table operations
- Do these in parallel with other things (task parallelism)
- Push them to other processes/threads

- **`Decoupled`** spawns a new process for a given object *o* and lets *o* execute in the new process such that *o* is ***decoupled***
- In the parent process, a **`Decoupled`** acts as proxy.
  - Can return a **`FutureResult`** when a method on *o* is invoked

- ```
  pagedim = DecoupledDimension(
            SCDimension(name='page', key=...) )
  ```
  ```

Decoupled objects



Consuming decoupled objects

- We now get a **`FutureResult`** from **`pagedim`**
  - Can't be inserted into the fact table – we need the real result

```
for row in inputdata:
   row['testid'] = testdim.lookup(row)
   row['dateid'] = datedim.ensure(row)
   row['pageid'] = pagedim.scdensure(row)
   facttbl.insert(row)
```

- We could ask for the real result, but no parallelism then…

- Instead, we also decouple **`facttbl`** and let it ***consume*** **`pagedim`**
  - The **`FutureResults`** created by **`pagedim`** are then automatically replaced *in the new process* for **`facttbl`**
  - All processes can now work in parallel

Consuming decoupled objects



Partitioning

- Processing the page dimension is a bottleneck in our example
- We can create several decoupled objects for the page dimension
  - And make code to partition the data between them
  - Not very flexible – what if we add/remove decoupled objects?

- **`DimensionPartitioner`** remedies this
  - Partitions between any number of **`Dimension`** objects
    - Data parallelism when decoupled objects are used
  - Looks like a **`Dimension`** → no changes needed in the main code
  - A method invocation is redirected to the right instance
    - Based on hashing of business key **or** a user-definable *partitioner*

```
pagedim = DimensionPartitioner([pagedim1, pagedim2]),
      partitioner = lambda row: hash(row['domain']))
```

Partitioning



Parallel functions

- Often, time-consuming functions are used
- It should be easy to make a function run in parallel with other tasks

- We use ***annotations*** for this

```
@splitpoint(instances=2)
def myfunc(*args):
        # Some Python code here …
```

- Here, two new processes are spawned (instances=2)
- All calls of `myfunc` return immediately and the function instead executes in one of the new processes

@splitpoint



Parent process
myfunc(arg1)
myfunc(arg2)
@splitpoint
myfunc
myfunc(arg1)
myfunc(arg2)
myfunc
function
myfunc
function
Spawned process 1
Spawned process 2

Parallel functions in flows

- Another way to use functions in parallel is a ***flow***
- A flow *F* consists of a sequence of functions *f1, f2, …, fn* running in parallel in a number of processes
- F is callable such that *F(x)* corresponds to *f1(x)* followed by *f2(x)* followed by … followed by *fn(x)*

- ```
  flow = createflow(extractdomaininfo,
                    extractserverinfo, convertsize)
  ```

- Several functions can also be made to run in a *shared* process:
  ```
  flow = createflow(extractdomaininfo,
                    (extractserverinfo, convertsize))
  ```
  ```

Flows


Parent process
flow(r)
r processed by the flow's functions
flow
r
r
extractdomaininfo function
extractserverinfo function
convertsize function
r
r
Spawned process 1
Spawned process 2
Spawned process 3

Combining flows and splitpoints

```
flow = createflow(…)

@splitpoint
def producedata():
    for row in somesrc:
        flow(row) #Insert row into the flow

def consumedata():
    for row in flow: #Get the transformed data
        # Do something

producedata()
consumedata()
```

Implementation

- Use of threads can be really slow in CPython (the reference implementation)
  - In our example, use of four threads is slower than use of a single thread!
- Instead, pygrametl uses **`multiprocessing`** where processes are launced
  - This is better, but IPC is expensive
- pygrametl can also run on Jython (Python implemented in Java)
  - Threading works well here and pygrametl then uses threads instead of processes

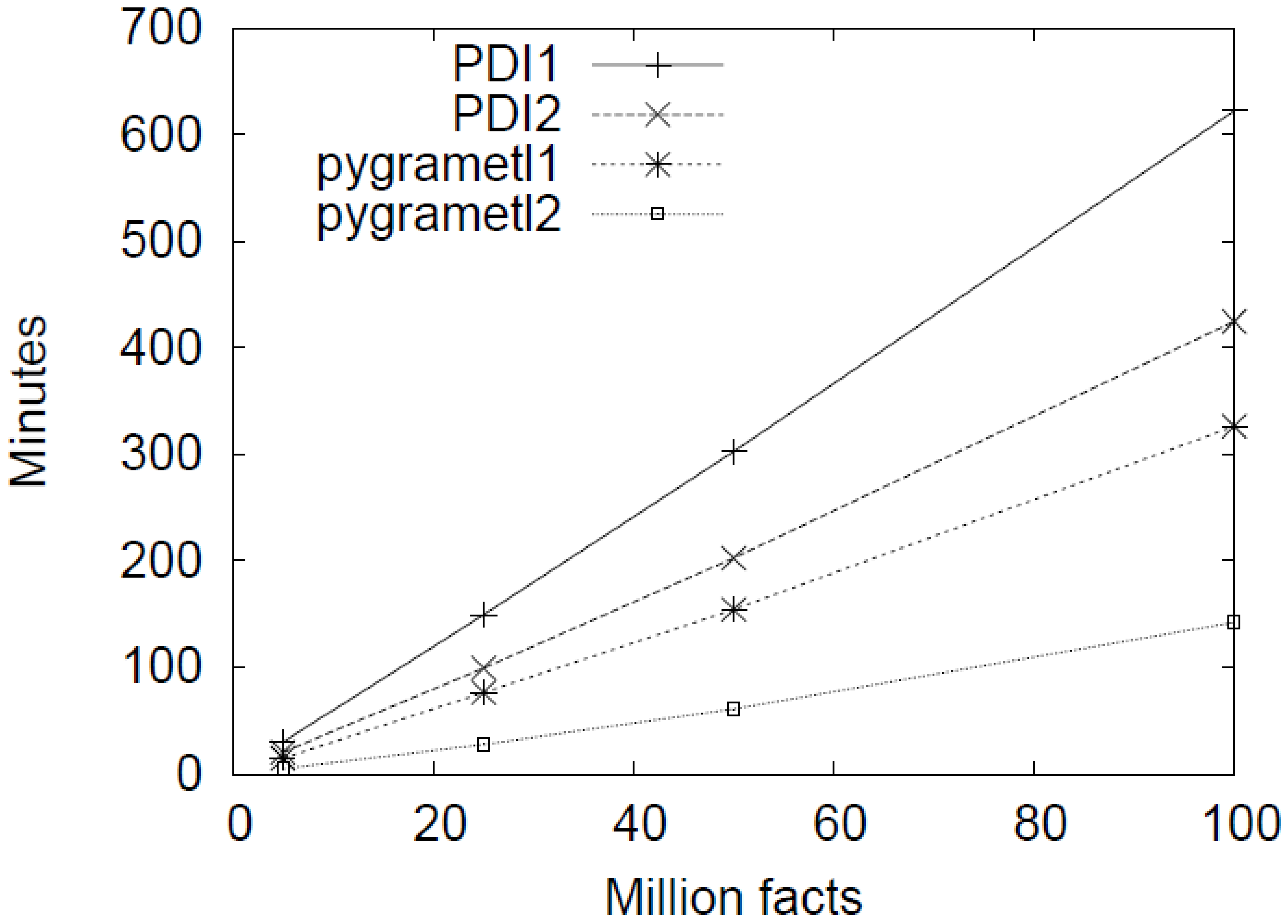
Performance

- We use the running example as test bed
- pygrametl under Jython 2.5.2/Java 6
- PDI 3.2
  - [4.0 and 4.1 existed at the time of the experiment, but were slower than 3.2]
- PostgreSQL 8.4
- 2 x Quad core 1.86GHz Xeon CPUs, 16 GB RAM

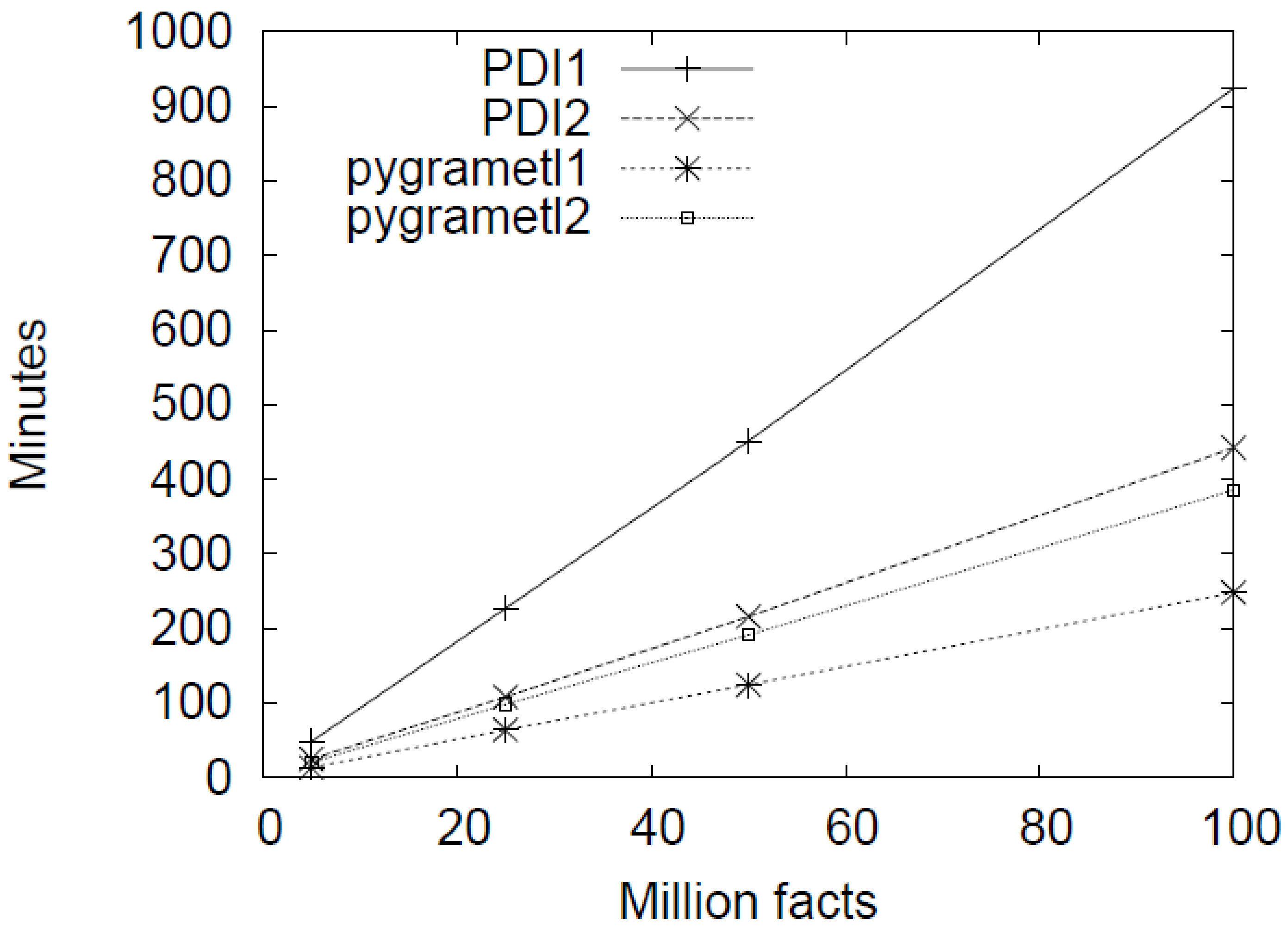
- Compare the single-threaded ”pygrametl1” program to the multi-threaded ”pygrametl2”
  - 2 `DecoupledDimension`s for the page dimension
  - a `DecoupledFactTable`
  - a separate process for extracting data and performing simple transformations

- … and compare to PDI with 1 and 2 connections

Elapsed time



CPU time



PDI1
PDI2
pygrametl1
pygrametl2
Minutes
Million facts
0
100
200
300
400
500
600
700
800
900
1000
0
20
40
60
80
100

Conclusion and future work

- Many different ways to add parallelism to an ETL program
  - Task parallelism: `Decoupled`, `@splitpoint`, flows
  - Data parallelism: `Decoupled + Partitioner`, `@splitpoint(instances=…)`

- Easy to add parallelism to a non-parallel ETL program
  - But some parts of an ETL program may be blocking

- Use a little more CPU time to reduce the elapsed (wall-clock) time a lot

- Future work:
  - Performance monitoring and hints
  - Maturing the tool and adding features

Experiences with pygrametl's parallelism

- The classes and functions for parallelism accept optional parameters:
  - **`batchsize`**: the amount of grouped method calls transferred between the processes
  - **`queuesize`**: the maximum amount of waiting batches
- If not given, default values are used

- Challenge: The values can significantly affect the performance
- Values which are good on one machine are not necessarily good on another ☹
  - Previous student project: Automatically find good values
- On Jython, a part of the explanation has to do with garbage collection

Related work

- The commercially available ETL tools use parallelism
  - Some do it simplisticly and start a thread for each step, others find groups/trees of steps to be processed by one thread
  - It is very different how different (graphically drawn) ETL definitions exploit parallelism and this should still be considered carefully
  - With the suggested approaches, the programmer has the full control of how and where to apply parallelism

- *MapReduce* for ETL
  - ETLMR (Liu, Thomsen, and Pedersen) is a modified version of pygrametl to be used with *MapReduce*
  - PDI

- PyCSP (Bjørndal et al) for parallel functions by means of annotations
  - A general framework for parallelism
  - Requires explicit input/output channels

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- **A case-study**
- Open-sourcing pygrametl
- ETLMR
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

A case study: FlexDanmark

- FlexDanmark organizes taxi trips for patients going to hospitals etc.
  - Revenue:120 million USD/year

- To do this, they make their own routing based on detailed speed maps
- GPS data from ~5,000 vehicles; ~2 million coordinates delivered every night

- A Data Warehouse represents the (cleaned) GPS data
- The ETL procedure is implemented in Python
  - transformations between different coordinate systems
  - spatial matching to the closest weather station
  - spatial matching to municipalities and zip code areas
  - map matching to roads
  - load of DW by means of pygrametl

Case study: Code-generation

- Another DW at FlexDanmark holds data about payments for trips, taxi companies, customers, …
- Integrates data from different source systems delivered in CSV dumps
- Payment details (i.e., facts) about already processed trips can be updated
- ➔ fact tables treated similarly to ”type 2 SCDs” with ValidFrom/ValidTo and Version#
- New sources and dimensions are sometimes added
- FlexDanmark has created a framework that creates Python code incl. pygrametl objects (and tables in the DW) based on metadata parameters
  - A new data source can be added with 10-15 lines of code in ½ hour
    a new dimension with 2 lines of code
  - Parallelism, versioning, etc. immediately available

Case-study: Lessons learned

- Programming gives a big freedom/many possibilities
- Complexity can grow
- ➔ A big need for good documentation

Case study: Why programmatic ETL

- Programming gives bigger flexibility
- Easy to reuse parts in different places
- Tried to implement map matching in commercial ETL GUI-based tool
  - Hard to ”fit” into the frames
  - Gave up and went for programmatic ETL in Python
  - Existing libraries could easily be reused (and replaced with others)
- Commercial ETL tools are expensive
  - Open-source GUI-based ETL tools considered, but after comparing coded and ”drawn” ETL flow examples, it was decided to go for code

Other cases

- Often, we don't know what users use pygrametl for…
- Some have told us what they use it for
- Domains include
  - health
  - advertising
  - real estate
  - public administration
  - sales
- Sometimes job adds mention pygrametl knowledge as a requirement

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- **Open-sourcing pygrametl**
- ETLMR
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

pygrametl as open source

- We published a paper about pygrametl in DOLAP’09 and put the pygrametl source code on our homepage
  - There were some downloads and comments but not too many

- Later, we moved the code to Google Code and got more attention

- When Google Code was taken out of service, we moved to GitHub and got much more attention
  - Currently 15 watchers, 85 stars, 16 forks
  - pygrametl.org has ~30 unique visitors per day (most visitors Mon-Fri)

Experiences with open sourcing pygrametl

- Avoid obstacles for the users
  - Users want easy installation: **`pip install pygrametl`**
  - When pygrametl was not on GitHub and PyPI, people created their own unofficial projects/packages – outside our control
  - Tell early what your software can/cannot do

- Make it clear how to get into contact with you
  - Avoid too many possibilities

- Make documentation and guides
  - We can see that our online Beginner's Guide and examples are popular
  - Remove old documentation
    - We forgot some outdated HTML pages which continued to be on the top in Google's results

Experiences with open sourcing pygrametl

- Engage users when they ask for help
  - How to reproduce the problem
  - How to solve it (ask them to provide code if possible)
- Users also find performance bottlenecks – and good improvements
  - For example ORDER BY vs. local sort in Python
- Some users are also willing to pay for development of a feature
  - Check with you university if you are allowed to take the job
  - Make a contract that specifies all the details incl. IPR, license, contributions back to the project, …
  - Can you be sued if something goes wrong? Specify the maximum liability
- Users are often very quiet
  - A good sign??? A bad sign???
  - Use something like Google Analytics to see how many visitors you have (does not tell no. of downloads from PyPI etc.)

Source code

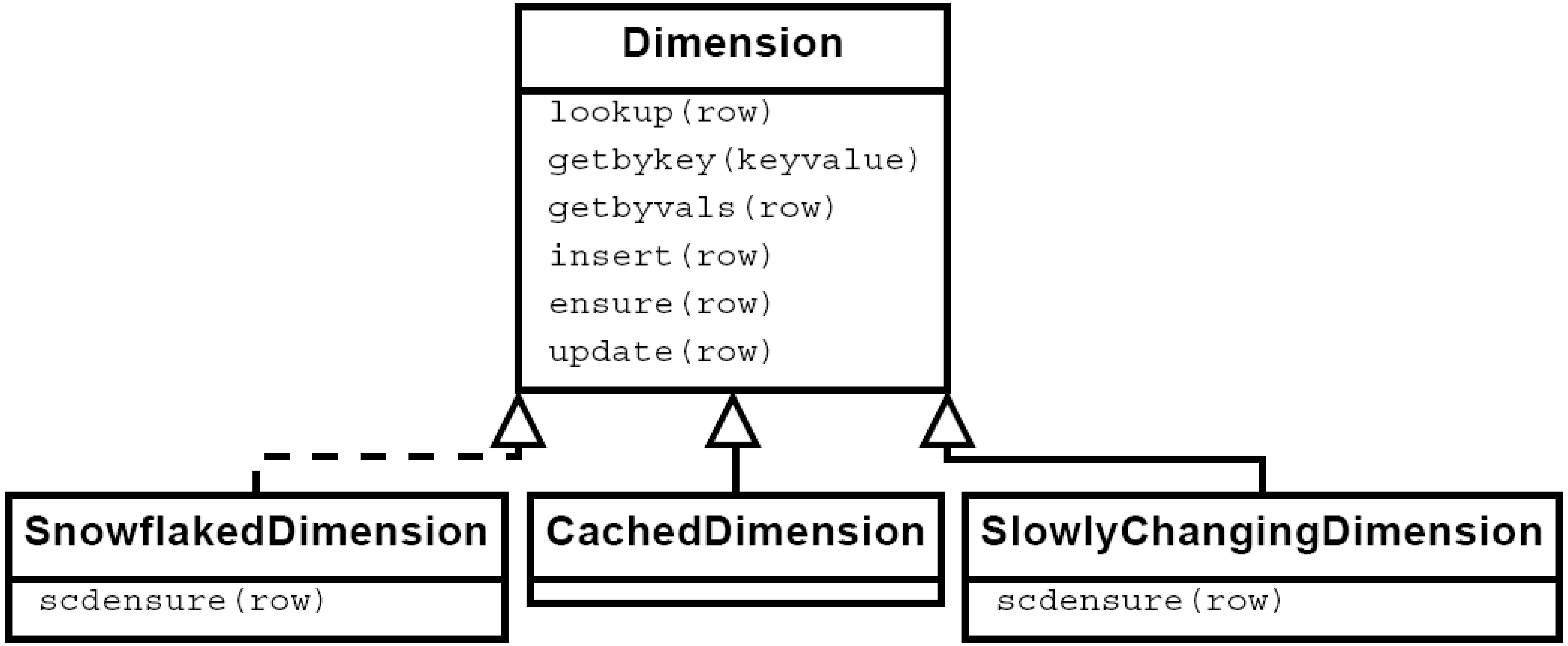
- The source code and the shown example cases can be downloaded from

  http://pygrametl.org

- The source code is maintained by Søren Kejser Jensen, Ove Andersen, and Christian Thomsen

- Thanks to all code contributors!

In the beginning…



- Plus **`FactTable`**, **`BatchFactTable`**, and **`BulkFactTable`**

Now

- But then we got new ideas/requests/needs… (Not a bad thing!)
  - Dimension(object)
  - CachedDimension(Dimension)
  - SnowflakedDimension(object)
  - SlowlyChangingDimension(Dimension) *# Supports type 2(+1) changes*
  - TypeOneSlowlyChangingDimension(CachedDimension)
  - FactTable(object)
  - BatchFactTable(FactTable)
  - _BaseBulkloadable(object)
  - BulkFactTable(_BaseBulkloadable)
  - BulkDimension(_BaseBulkloadable)
  - CachedBulkDimension(_BaseBulkloadable, CachedDimension)
  - SubprocessFactTable(object)
  - DecoupledDimension(Decoupled) *# Looks like a Dimension*
  - DecoupledFactTable(Decoupled) *# Looks like a FactTable*
  - BasePartitioner
  - DimensionPartitioner(BasePartitioner) *#Looks like a Dimension*
  - FactTablePartitioner(BasePartitioner) *# Looks like a FactTable*

Why so many classes?

- New ideas often resulted in new classes
  - with the same interface.
    - (Sometimes we used inconsistent argument names ☹)
  - We think/hope that they are easy to use
- We did not break existing functionality

- On the other hand…
  - It is not intuitive that you should not use Dimension, but rather CachedDimension – who wouldn't want caching?
  - Sometimes we implement the same thing in different variations – SlowlyChangingDimension provides its own caching
  - It would be nice to always have the possibility of using bulkloads, caching, and parallelism

What should we do in the future

- Version 2.x should remain compatible with previous versions
- Version 3.0 **could** introduce major API changes if we decide to do so
- Fewer ”table classes” with the same or more functionality
  - Caching, bulkloading, and parallelism could always be possible
  - Only one class for SCDs
  - …

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- Open-sourcing pygrametl
- **ETLMR**
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

MapReduce: Origin and purpose

- Introduced by Google in 2004
- Makes distributed computing on clusters easy
- Highly scalable: Handles TBs of data on clusters of 1000s of machines (scales out)
- The user only has to specify two functions
  - An abstraction: The two functions deal with key/value sets.
- The system can then take care of partitioning, scheduling, failures, etc. (all the tedious work)
  - The user can focus on the important computations
- MapReduce is batch processing system. Brute force!
  - To be used on large amounts of data

Programming Model

- Takes a set of key/value pairs as input
- Produces another set of key/value pairs as output
- Keys and values can be primitives or complex types

- The user provides two functions: *map* and *reduce*
- ***map***: `(k1, v1) → list(k2, v2)`
  - Takes an input pair and produces a set of intermediate key/value pairs. MapReduce groups all intermediate pairs with the same key and gives them to *reduce*
- ***reduce:*** `(k2, list(v2)) → list(k3, v3)` **(Hadoop)**
  `list(v2)` **(Google)**
  - Takes an intermediate key and the set of all values for that key. Merges the values to form a smaller set (typically empty or with a single value)

daisy

Example: WordCount

```
map(String key, String value):
  // key: document name; value: doc. contents
  foreach word in value:
     EmitIntermediate(word, 1)


reduce(String key, Iterator<int> values):
  // key: a word; values: list of counts
  int result = 0;
  foreach v in values:
     result += v;
  Emit(key, result);
```

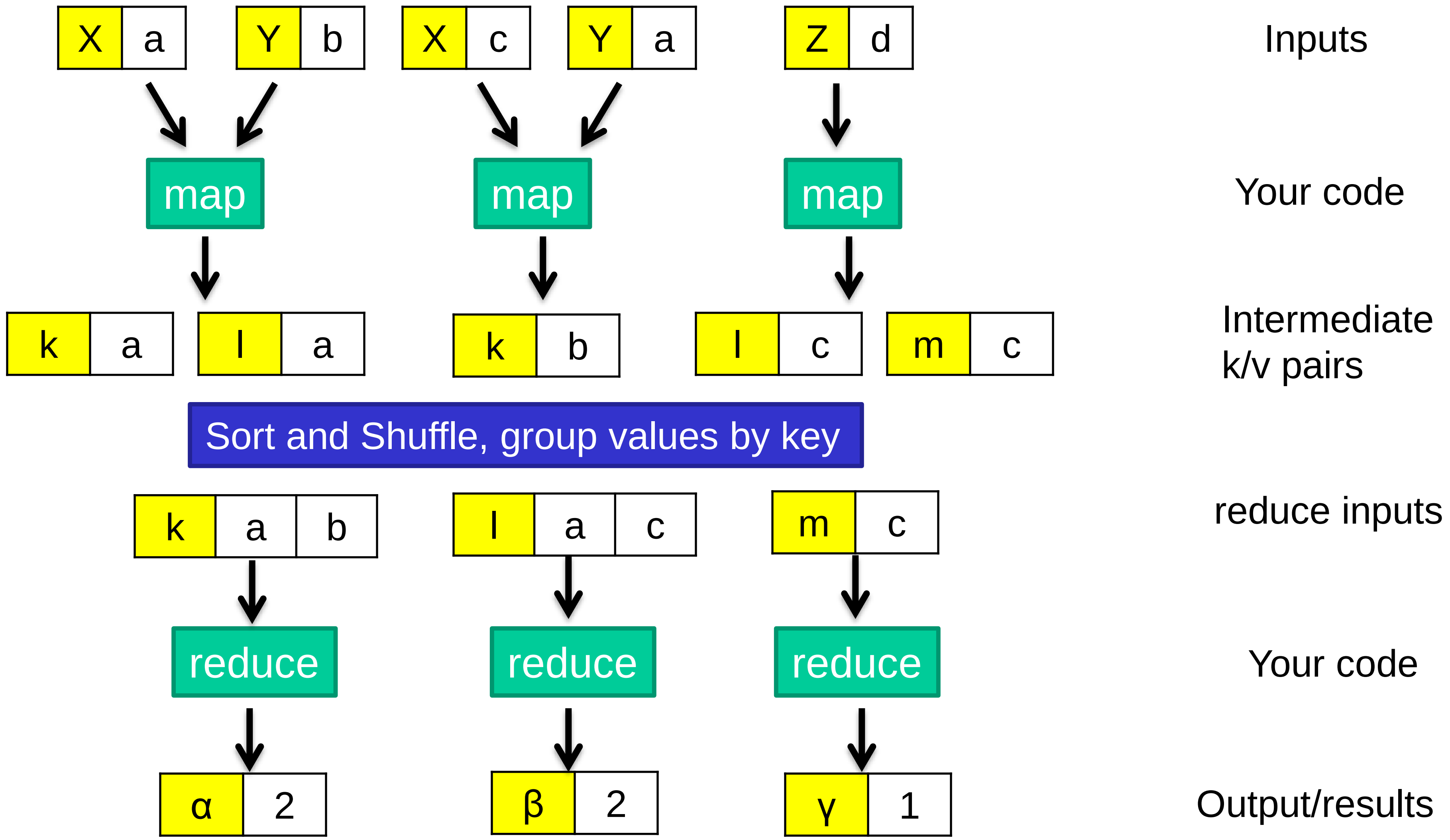
How does it work?

- The map invocations are distributed across many machines such that many map invocations run concurrently
  - Often many thousands of task to assign to hundreds or thousands of nodes
  - The input is automatically partitioned into logical *splits* which can be processed in parallel
  - The input data is stored in a *distributed file system*. The MapReduce runtime systems tries to schedule the map task to the node where its data is located. This is called *data locality*.

- The intermediate key/value pairs (map outputs) are partitioned using a ***deterministic*** partitioning function on the key:
  - By default: hash(key)

- The reduce invocations can then also be distributed across many machines
  - but not until all map tasks have finished

Conceptual view



WordCount – the actual code for Hadoop

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

Code from apache.org

ETL on MapReduce

- An ever-increasing demand for ETL tools to process very large amounts of data efficiently
- Parallelization is a key technology
- MapReduce offers high flexibility and scalability and is interesting to apply
- But MapReduce is a generic programming model and has no native support of ETL-specific constructs
  - Star, snowflake schemas, slowly changing dimensions
- Implementing a parallel ETL program on MapReduce complex, costly, error-prone, and leads to low programmer productivity

ETLMR

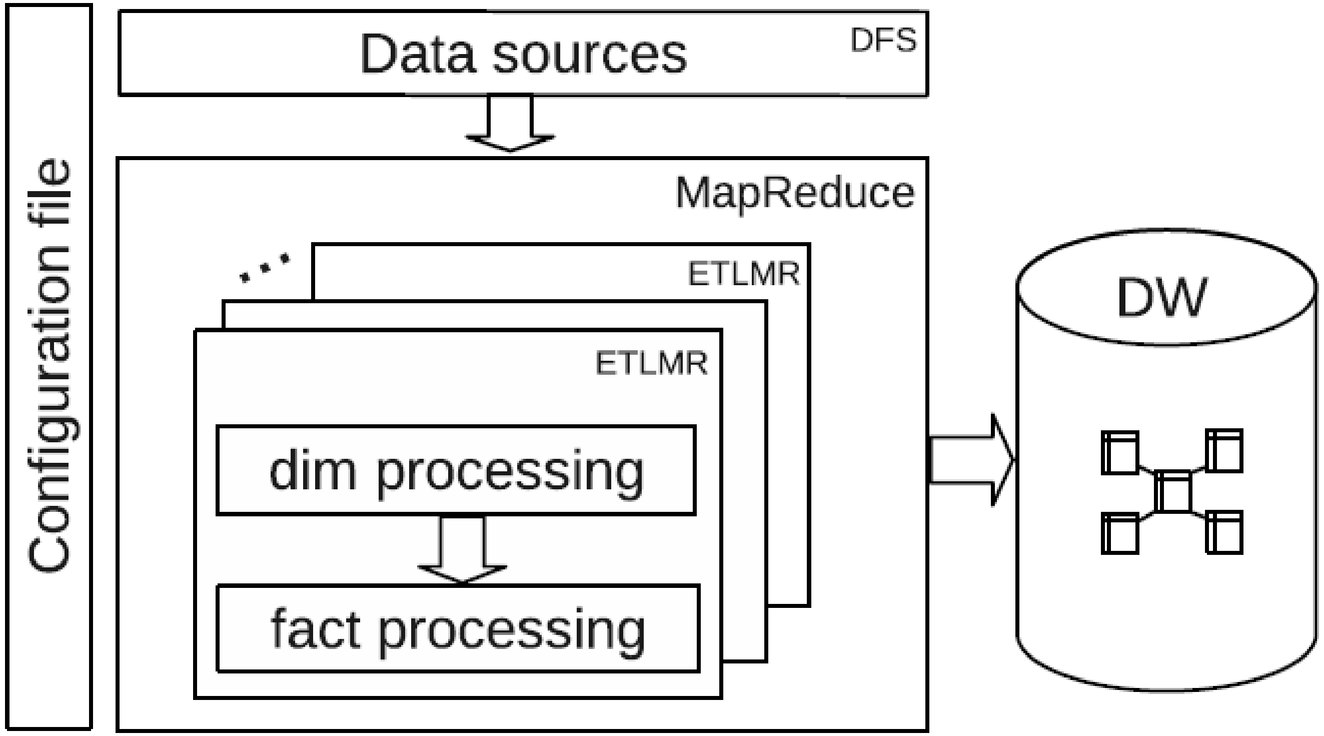
- As a remedy ***ETLMR*** is a dimensional ETL framework for MapReduce
  - Direct support for high-level ETL constructs

- ETLMR leverages the functionality of MapReduce, but hides the complexity
- The user only specifies transformations and declarations of sources and targets
  - Only few lines are needed

- Based on pygrametl but some parts extended or modified to support MapReduce

ETLMR

- An ETL flow consists of dimension processing followed by fact processing
  - Two sequential MapReduce jobs
  - In a job, a number of tasks process dimension/fact data in parallel on many nodes



How to use ETLMR

- Declare sources and targets in config.py

  from odottables import * # *Different dimension processing schemes supported*

  fileurls = [‘dfs://…/TestResults0.csv’, dfs://…/TestResults1.csv, …]

  datedim = CachedDimension(…) # *as in pygrametl*
  pagedim = SlowlyChangingDimension(…)
  pagesf = SnowflakedDimension(…)

How to use ETLMR

- Define also source attributes to use for each dimension and transformations to apply (implemented in Python)

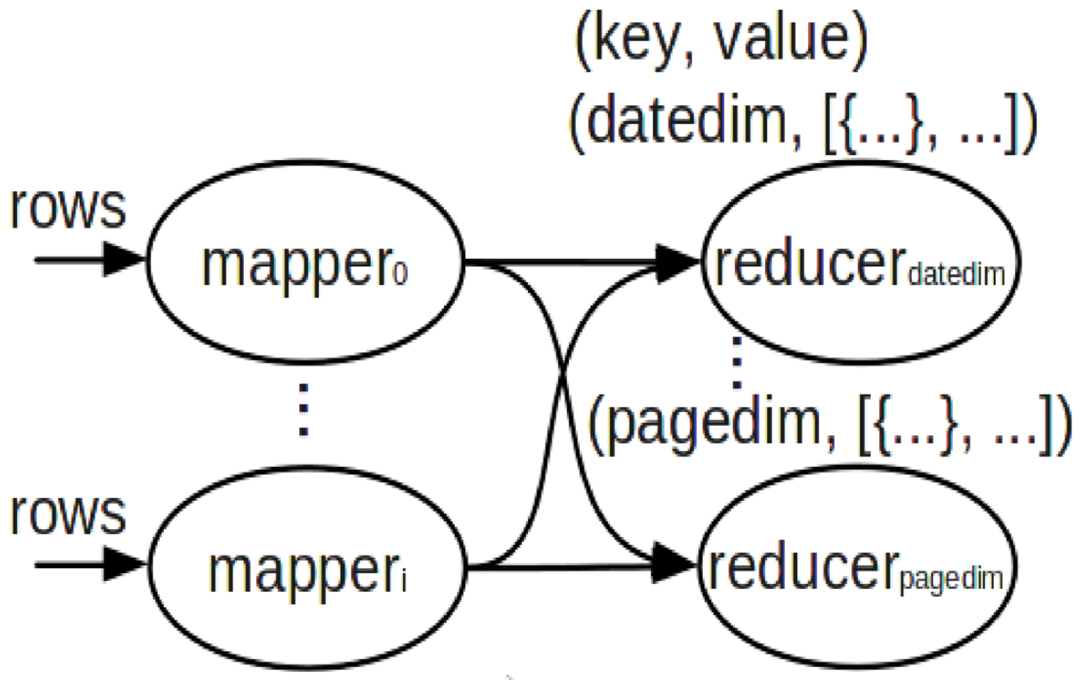
```
dims = {
      pagedim:{‘srcfields’:(‘url’, ‘serverversion’, ‘domain’, ‘size’, ‘lastmoddate’),
                    ‘rowhandlers’:(UDF_extractdomain, UDF_extractserver)},
      domaindim:{srcfields’:(‘url’), ‘rowhandlers’:(UDF_extractdomain)},
      …
}
```

Dimension processing: ODOT

- One Dimension, One Task

- Map
  - Projection of attributes
  - (dimension name, attributes)

- Reduce
  - *One* reducer processes data for *one* dimension
  - User-defined transformations
  - Key generation
  - Filling/updating the dimension table

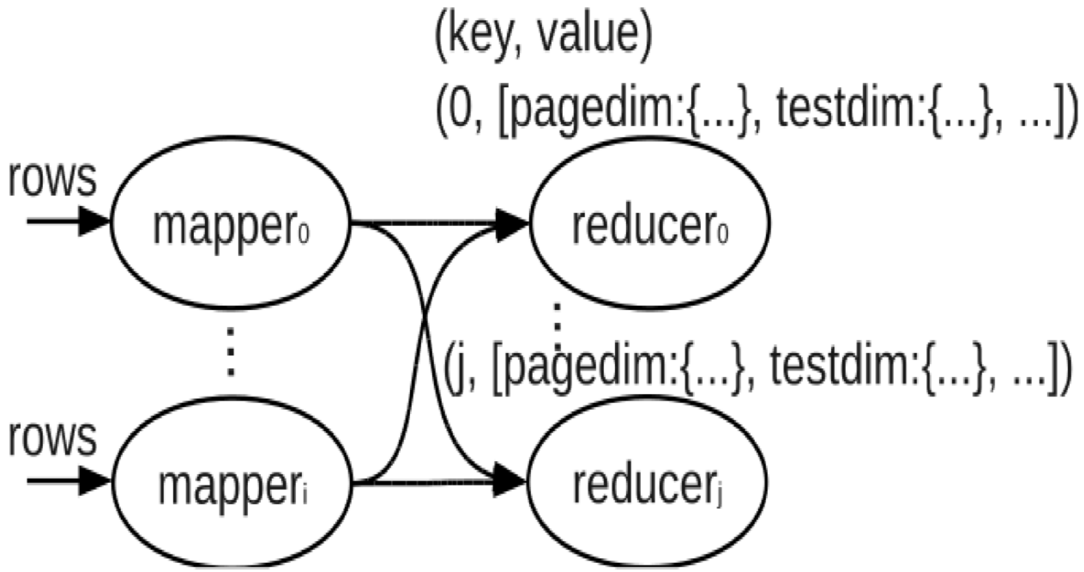


Dimension processing: ODAT

- One Dimension, All Tasks

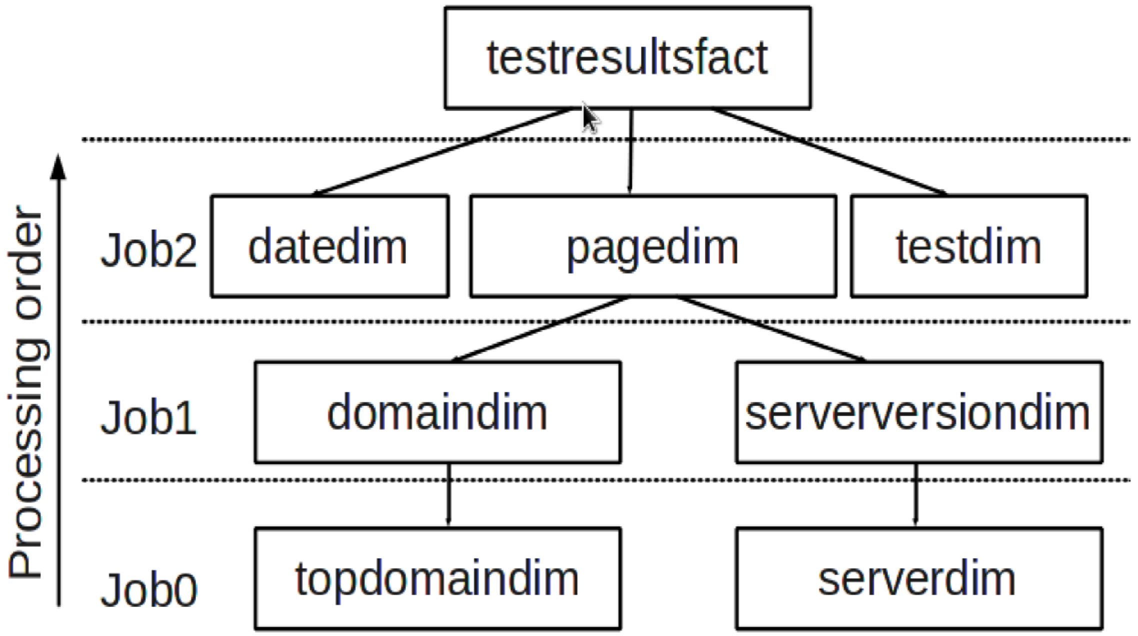
- Map
  - Projection of attributes
  - (rownumber, [dimname1:{…}, dimname2:{… }, …])

- Reduce
  - Data distributed in a round-robin fashion
  - One reducer processes data for all dimensions
  - One dimension is processed by all reducers
  - Inconsistencies and duplicated rows can occur
  - Fixed in a final step



Dimension processing: Snowflaked

- For snowflaked dimensions, an order can be given
- order = [(topdomaindim, serverdim),
  (domaindim, serverversiondim),
  (pagedim, datedim, testdim)]
- Results in three ODOT jobs



Dimension processing: Offline

- Dimension data is processed and stored locally on the nodes
- DW only updated when explicitly requested

- Processing schemes: ODOT and a combination of ODAT and ODAT (”hybrid”)

- In the hybrid, a data-intensive dimension (such as pagedim) can be partitioned based on business key (url) and processed by all tasks (ODAT-like)
- The remaining dimensions use ODOT processing

Fact processing

- In config.py we declare:
  - fact tables
  - dimensions to do lookups in
  - transformations to apply

- Fact partitions processed in parallel

Deployment

- ETLMR uses the Python-based Disco platform

- For our example (with a snowflaked page dimension), ETLMR requires 12 statements

- The most widely used MapReduce implementation is Apache Hadoop
- In later work – CloudETL – we consider ETL for Hadoop (specifically Hive)

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- Open-sourcing pygrametl
- ETLMR
- **CloudETL**
- MAIME – programmatic changes/repairs of SSIS Data Flows

Motivation

- Much attention has been given to *MapReduce* for parallel handling of massive data sets in the cloud
- *Hive* is a popular system for data warehouses (DWs) on Hadoop MapReduce
  - Instead of MapReduce programs in Java, the user uses the SQL-like HiveQL

- The ”Extract-Transform-Load” (ETL) process loads data into a data warehouse
- Pig is often used for preprocessing of data
- It is hard to do *dimensional ETL processing* in Hive/Pig
  - For batch processing, not individual look-ups or inserts
  - No UPDATEs → ”slowly changing dimensions” (SCDs) are hard to use, but they are very common in traditional (i.e., non-cloud) DWs

CloudETL

- *CloudETL* is a solution that makes dimensional ETL on Hadoop easy
- The target is Hive – we're not replacing Hive
  - CloudETL for dimensional ETL, Hive for analysis
  - We write data directly to Hive's directories
- The user defines the ETL flow by high-level constructs;
  the system handles the parallelization
- → high programmer productivity, fast performance, and good scalability
- Two sequential steps in a CloudETL workflow:
  dimension processing followed by fact processing

Dimension processing of SCDs

- For ”type 2 SCDs” (where we add row versions), the main challenge is how to handle the special SCD attributes
  - valid from, valid to, version nr.
- When doing *incremental loading*, we may need to update existing dimension members

- Collect data from incremental data and existing data
- Do transformations in mappers (incremental data only)
  - Emit <*table name, business key, change order*> as key and the rest as value
- Partition on the <table name, business key>
- Perform the updates in reducers
  - The data is already sorted by the MapReduce framework

Example

Map input

Incremental data (key, value)			
(lineno, <	url,	size,	moddate >)
(0,	<www.dom.com/p0.htm,	20,	2011-02-01>)
(1,	<www.dom.com/p1.htm,	10,	2011-01-01>)

Existing dimension data (key, value)						
(lineno,	<id,	url,	size,	version,	validfrom,	validto>)
(0,	<0,	www.dom.com/p0.htm,	10,	1,	2011-01-01,	null >)

Map output

(<name,	url,	scddate >,	<id,	size,	version,	validfrom,	validto>)
(<pagedim,	www.dom.com/p0.htm,	2011-02-01>,	<null,	20,	null,	null,	null >)
(<pagedim,	www.dom.com/p1.htm,	2011-01-01>,	<null,	10,	null,	null,	null >)
(<pagedim,	www.dom.com/p0.htm,	2011-01-01>,	< 1,	10,	1,	2011-01-01,	null >)

Reduce input

(<name,	url,	scddate >,	<id,	size,	version,	validfrom,	validto>)
(<pagedim,	www.dom.com/p0.htm,	2011-01-01>,	<1,	10,	1,	2011-01-01,	null >)
(<pagedim,	www.dom.com/p0.htm,	2011-02-01>,	<null,	20,	null,	null,	null >)
(<pagedim,	www.dom.com/p1.htm,	2011-01-01>,	<null ,	10,	null,	null,	null >)

Reduce output

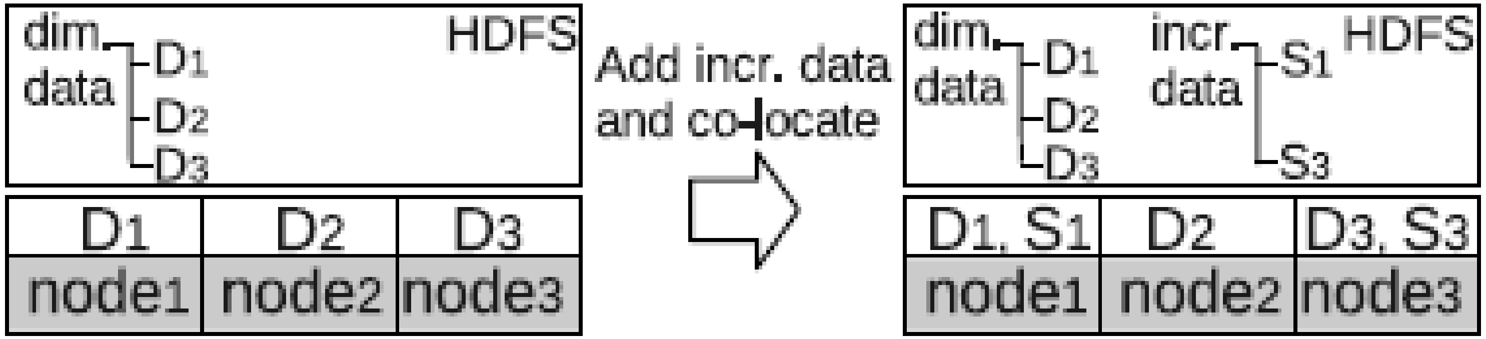
(null,	<id,	url,	size,	version,	validfrom,	validto >)
(null,	<1,	www.dom.com/p0.htm,	10,	1,	2011-01-01,	2011-02-01 >)
(null,	<2,	www.dom.com/p0.htm,	20,	2,	2011-02-01,	null >)
(null,	<3,	www.dom.com/p1.htm,	10,	1,	2011-01-01,	null >)

Dimension processing of SCDs

- For ”type 1” SCDs, we overwrite updated values
- A value may be overwritten many times
- To avoid writing unnecessary map output, we can modify the mapper to hold the current state of each seen dimension member in memory
- When the mapper is done with its split, it only outputs the current values and the reducer will do any necesary updates based on these

Processing of Big Dimensions

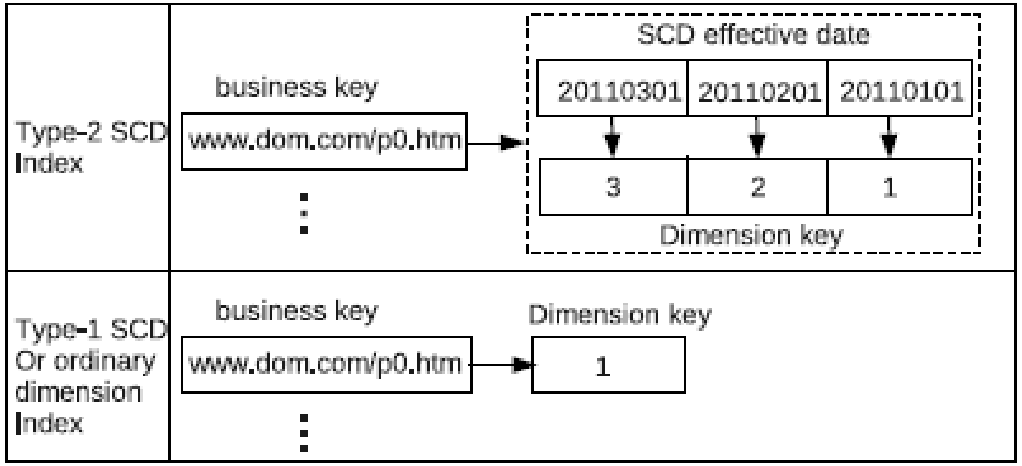
- Dimension tables are typically small compared to fact tables
- When a dimension table is big, the shuffling of data from mappers to reducers is not efficient
- In that case, we can use a *map-only* job where we exploit data locality in the distributed file system HDFS
- Co-locate existing and new data for the same parts



Fact processing

- Read and transform source data
- Retrieve surrogate key values from referenced dimensions
  - Hive does not support fast *look-ups*
  - There is usually much more fact data than dimension data

- During dimension processing, CloudETL creates *look-up indices* which map from business key values (and possibly validity dates) to surrogate key values



Fact processing, cont.

- CloudETL runs a map-only job to process fact data

Mapper

- Read relevant look-up indices into memory
- For each row in the data to load:
  - Perform transformations
  - Look-up surrogate key values in the look-up indices
  - Write out fact row

- The mappers can work in parallel on different parts of the data
- This works fine when the indices can be held in the main memory

Fact processing, cont.

- When a dimension table is too big to have its look-up index in the main memory, we suggest two alternatives
- 1) A hybrid solution where the new fact data is joined with the existing (big) dimension data by Hive. After that, the look-up indices for the smaller dimensions can be used
- 2) Partition the look-up index and require the source data to be partitioned in the same way
  - Co-locate the index partitions with the data partitions

Code for fact processing

```
/* 1) Define the fact data source */
DataReader  testResultsReader = new CSVFileReader("/user/cloudetl/input/testresults")
                              .setField("localfile", DataType.STRING)
                              .setField("url", DataType.STRING)
                              .setField("lastmoddate", DataType.DATE)
                              .setField("downloaddate", DataType.DATE)
                              .setField("test", DataType.STRING)
                              .setField("errors", DataType.INT);

/* 2) Do the necessary data transformation and look up dimension key values */
TransformingReader testresultsfactPipe = new TransformingReader(testResultsReader)
  .add(new ExcludeFields("localfile"))
  .add(new LookupTransformer("pageid", new SCDLookup(pagedim, "url", lastmoddate", -1)))
  .add(new LookupTransformer("dateid", new Lookup(datedim, "downloaddate", -1)))
  .add(new LookupTransformer("testid", new Lookup(testdim, "test", -1)));

/* 3) Define the target fact table */
DataWriter testresultsfact = new FactTableWriter("/user/cloudetl/fact", "testresultsfact")
                              .setField("pageid", DataType.INT)
                              .setField("dateid", DataType.INT)
                              .setField("testid", DataType.INT)
                              .setField("errors", DataType.INT);

/* 4) Add transformer and start ETL */
JobPlanner.addTransfer(testresultsfactPipe, testresultsfact).start();
```

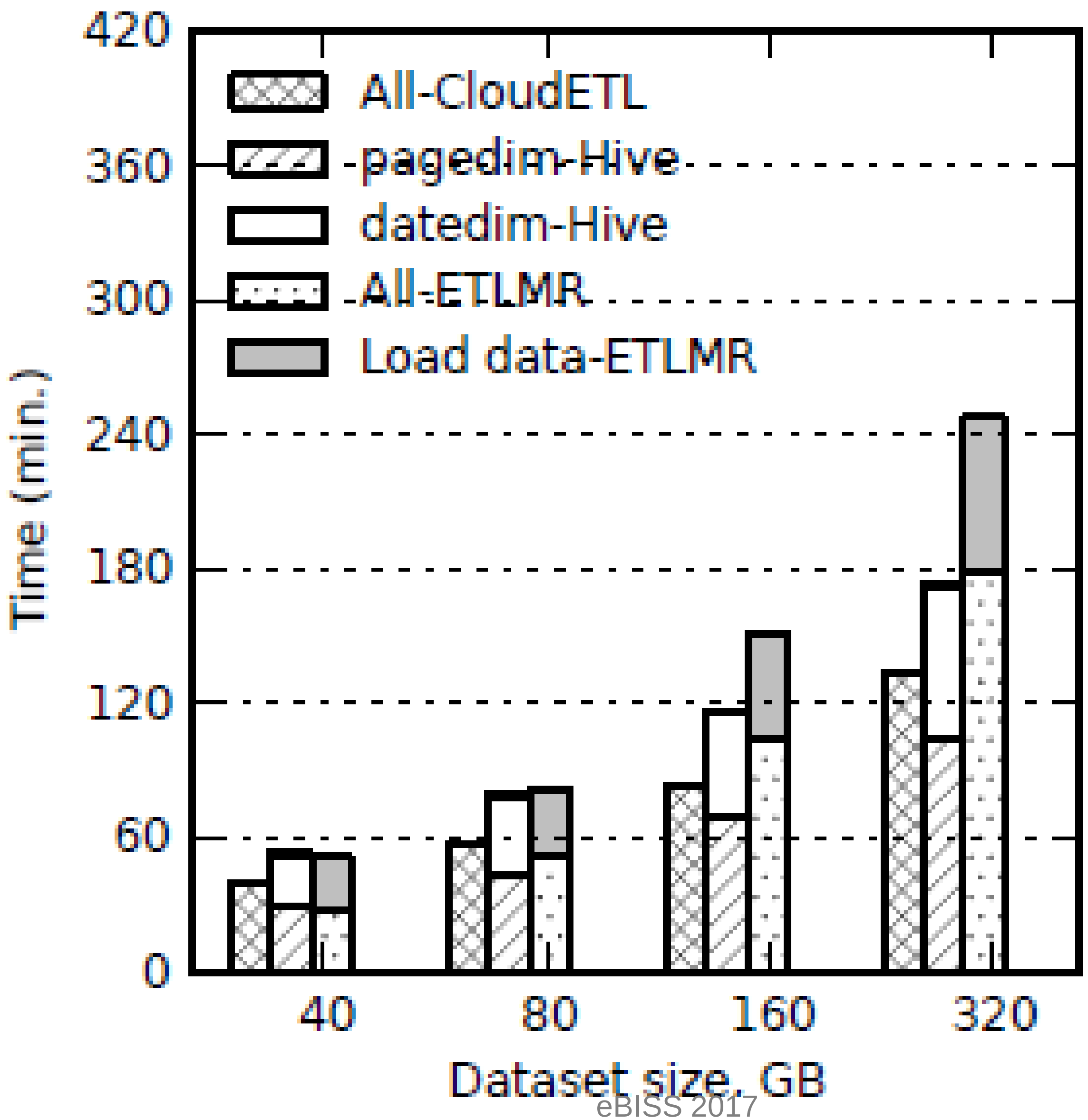
Experiments

- Tested on a private cluster
- 1 node used as NameNode and JobTracker
  - two quad-core Xeon E5606 2.13 GHz CPUs, 20GB RAM
- 8 nodes used as DataNodes and TaskTrackers
  - two dual-core Intel Q9400 2.66 GHz CPUs, 3GB RAM

- Tested with generated data set for a schema with three dimension tables and one fact table

- We compare with Hive and our previous work ETLMR

Star schema, no SCD



CloudETL summary

- While Pig and Hive are great tools, they are not ideal for ETL processing
- We have proposed CloudETL which is a tool where the user can program dimensional ETL flows to be run on Hadoop MapReduce

- CloudETL requires little programming and is efficient

- Future directions include more transformations, investigation of other backends (e.g., Spark), and making CloudETL even easier to use

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- Open-sourcing pygrametl
- ETLMR
- CloudETL
- **MAIME – programmatic changes/repairs of SSIS Data Flows**

Motivation

- A Data Warehouse (DW) contains data from a number of External Data Sources (EDSs)
- To populate a DW, an Extract-Transform-Load (ETL) process is used
- It is well-known that it is very time-consuming to construct the ETL process

Motivation

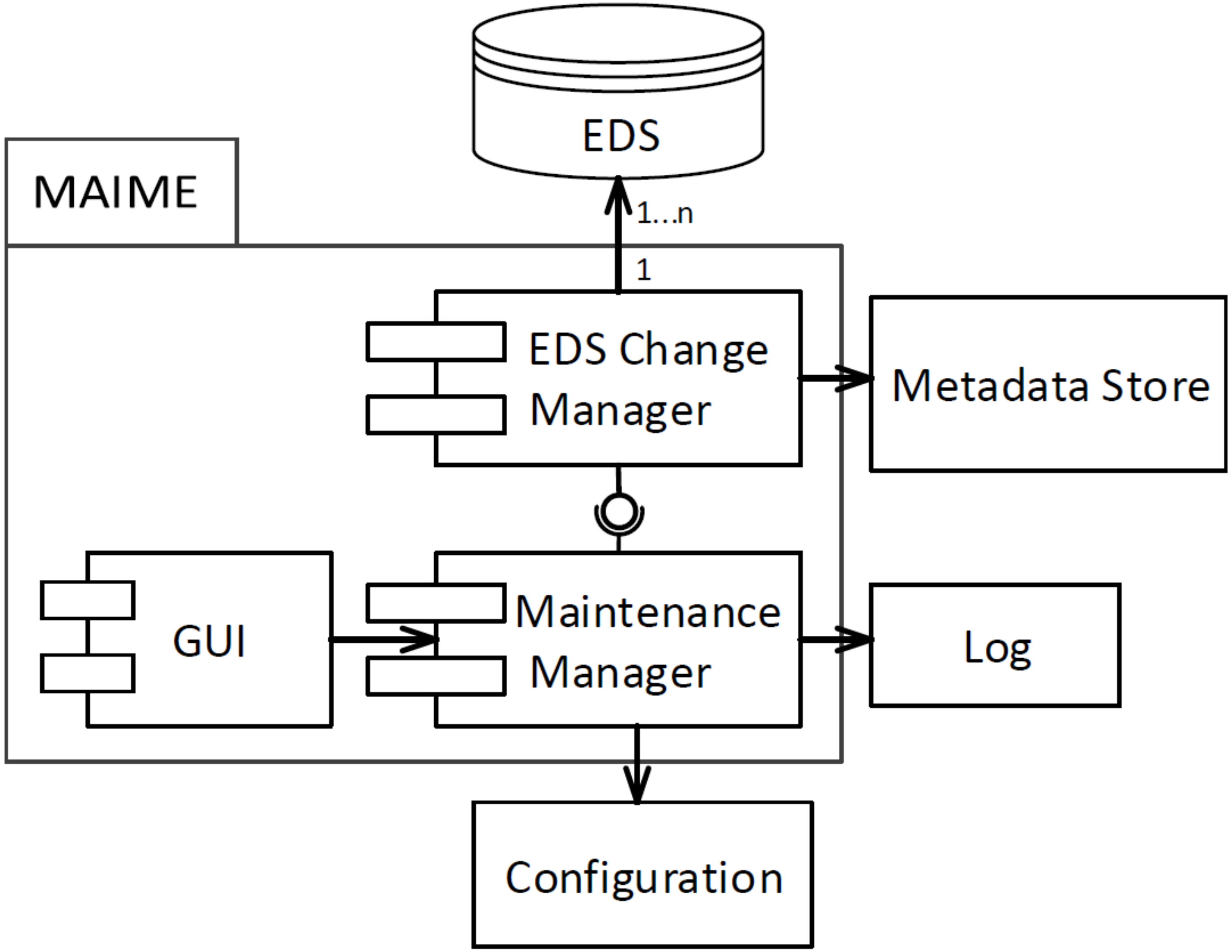
- Maintaining ETL processes *after* deployment, however, also takes much time
- Real examples
  - A pension and insurance company applies weekly changes to its software systems. The BI team then has to update the ETL processes
  - A facility management company has more than 10,000 ETL processes to execute daily. When there is a change in the source systems, the BI team has to find and fix the broken ones
  - The ETL team at an online gaming-engine vendor has to deal with daily changes in the format of data from web services
- Maintenance of ETL processes requires manual work and is time-consuming and error-prone

MAIME

- To remedy these problems, we propose the tool ***MAIME*** which can
  - detect schema changes in EDSs
  - and (semi-)automatically repair the affected ETL processes

- MAIME works with SQL Server Integration Services (SSIS) and SQL Server
  - Among the top-3 most used tools (Gartner)
  - SSIS offers an API which makes it possible to change ETL processes programmatically
  - The current prototype supports *Aggregate, Conditional Split, Data Conversion, Derived Column, Lookup, Sort,* and *Union All* as well as *OLE DB Source* and *OLE DB Destination*

Overview of MAIME



Overview of MAIME

- The **Change Manager** captures metadata from the EDSs
- The current snapshot is compared to the previous snapshot and a list of changes is produced
- The **Maintenance Manager** loads the SSIS Data Flow tasks and creates a ***graph model*** as an abstraction
  - Makes it easy to represent dependencies between columns
- Based on the identified changes in the EDSs, the graph model is updated
- When we make a change in the graph model, corresponding changes are applied to the SSIS Data Flow

The graph model

- An acyclic property graph $G = (V, E)$ where a vertex $v \in V$ represents a transformation and an edge $(v_1, v_2, columns)$ represents that *columns* are transferred from v_1 to v_2
  - The transferred columns are ”put on” the edges. This is advantageous for transformations with multiple outgoing edges where each edge can transfer a different set of columns
- Our vertices have multiple properties
- A property is a key-value pair. We use the notation *v.property*
- The specific properties depend on the represented transformation type, but all have *name*, *type*, and *dependencies*
  - except OLE DB Destination which has no *dependencies*

The graph model – *dependencies*

- *dependencies* shows how columns depend on each other
  - If an Aggregate transformation computes *c'* as the average of *c*, we have that *c'* depends on *c*

- Formally, *dependencies* is a mapping from an output column *o* to a set of input columns $\{c_1, \ldots, c_n\}$
  - We say that o is dependent on $\{c_1, \ldots, c_n\}$ and denote this $o \rightarrow \{c_1, \ldots, c_n\}$

- We also have *trivial dependencies* where *c* depends on *c*

Examples – *dependencies*

- **Aggregate:** For each output column *o* computed as AGG(*i*), *o* depends on *i*
- **Derived Column:** Each derived column *o* depends on the set of columns used in the expression defining *o.* Trivial dependencies in addition
- **Lookup:** Each output column *o* depends on the set of input columns used in the lookup (i.e., the equi-join). Trivial dependencies in addition
- **Conditional Split:** Only trivial dependencies

Other specific properties

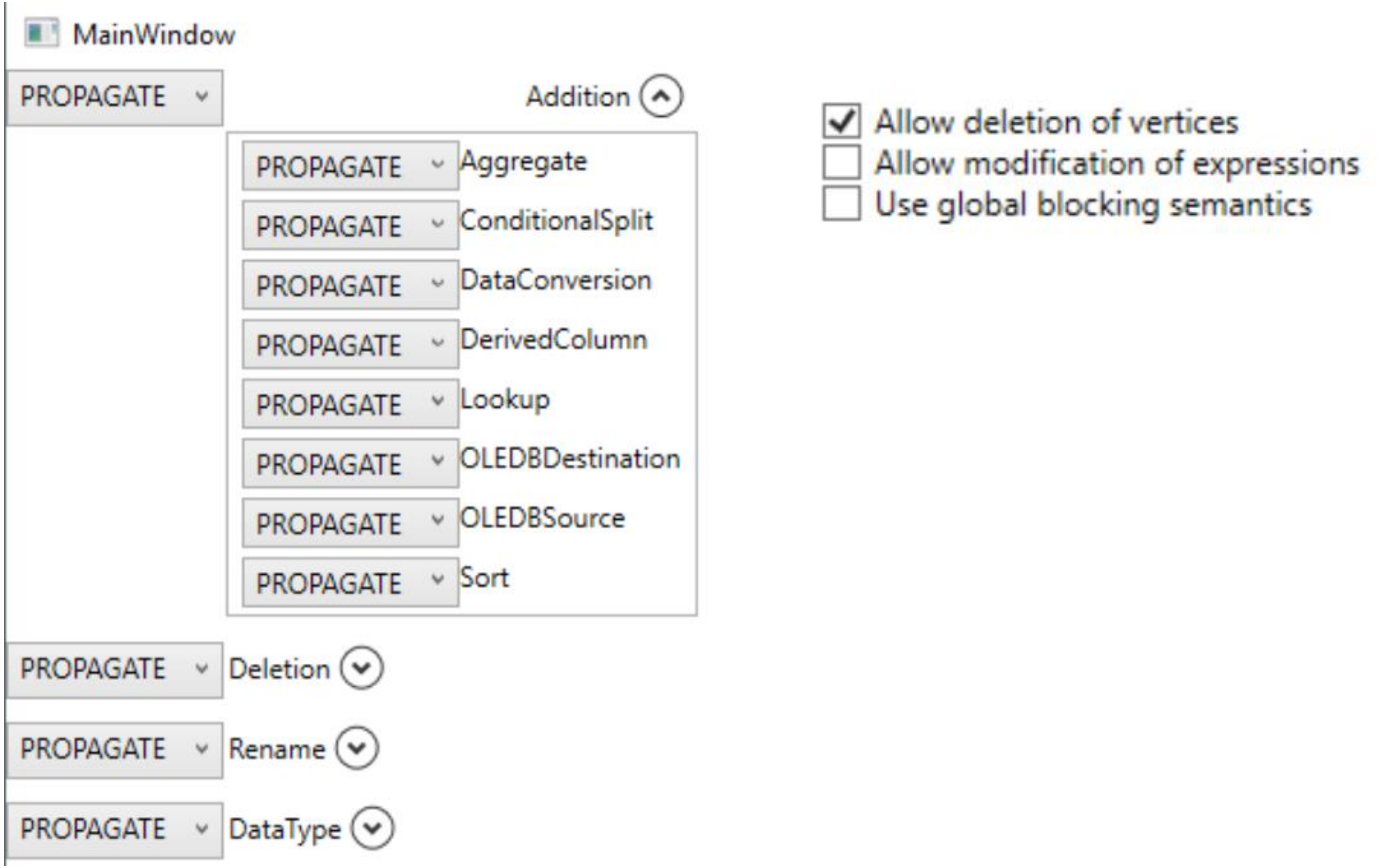
Transformation	Specific properties	In	Out
OLE DB Source	database, table, and columns	0	1
OLE DB Destin.	database, table, and columns	1	0
Aggregate	aggregations	1	many
Conditional split	conditions	1	many
Data conversion	conversions	1	1
Derived column	derivations	1	1
Lookup	database, table, joins, columns, and outputcolumns	1	2
Sort	sortings and passthrough	1	1
Union all	inputedges and unions	many	1

Policies

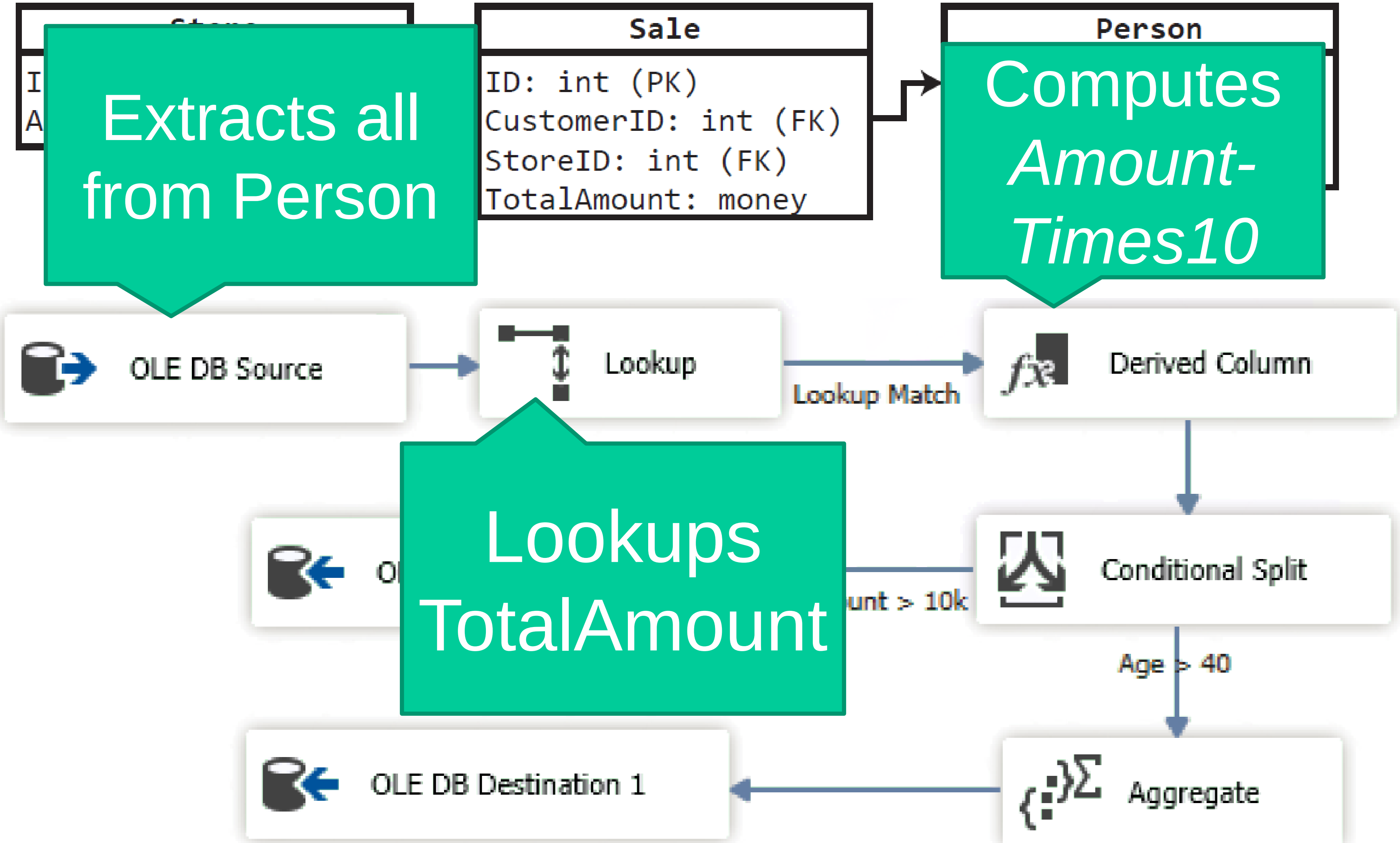
- For a change type in the EDS and a vertex type, a policy defines what to do
- For example *p(Deletion, Aggregate) = Propagate*

- **Propagate** means repair vertices of the given type if a change of the given type renders them invalid
- **Block** means that a vertex of the given type (or any of its descendants) will *not* be repaired
  - Instead, it can optionally mean ”Don’t repair *anything* if the flow contains a vertex of the given type and the given change type occurred”
- **Prompt** means ”Ask the user”

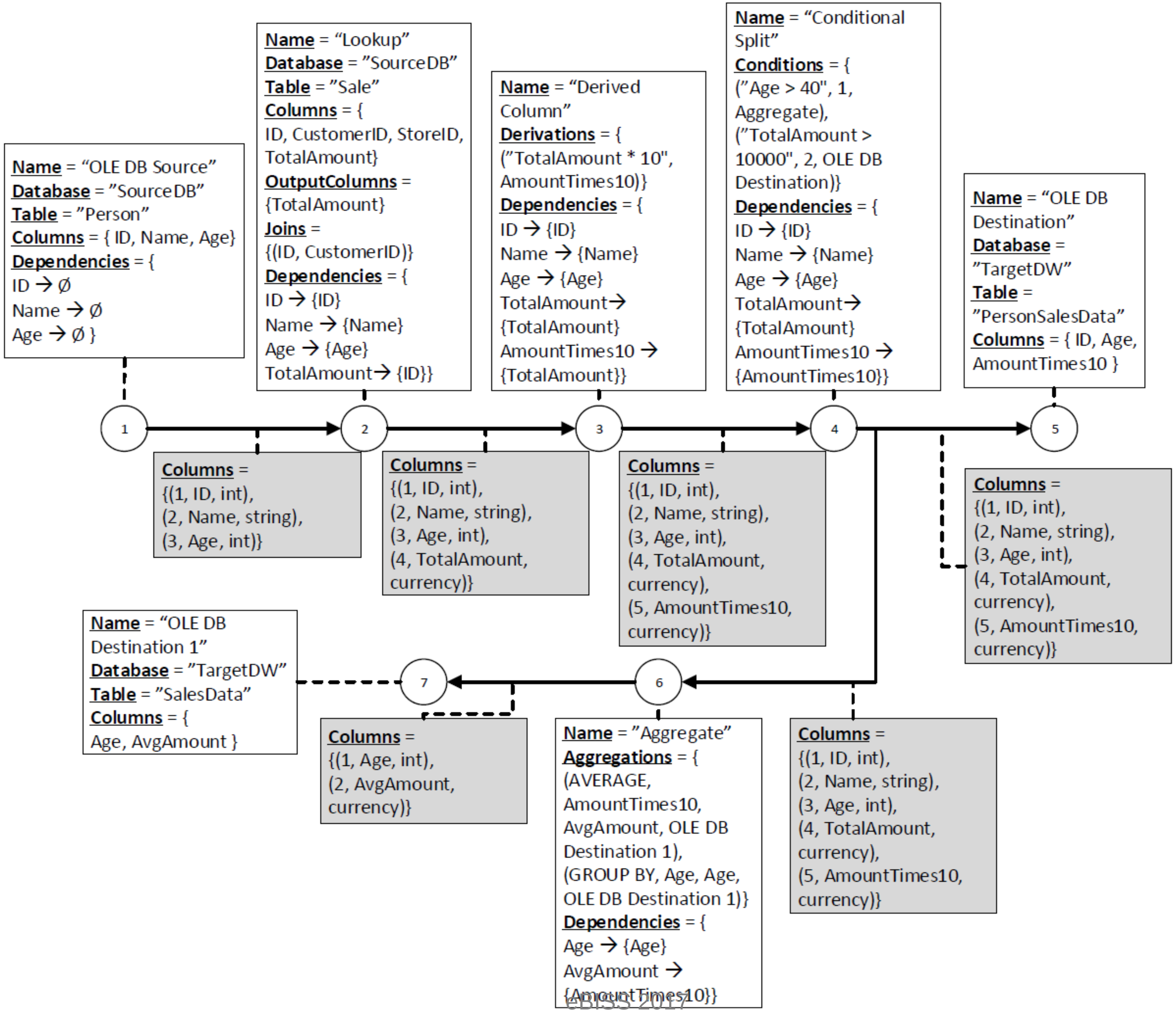
Policies


MainWindow
PROPAGATE
Addition
PROPAGATE Aggregate
PROPAGATE ConditionalSplit
PROPAGATE DataConversion
PROPAGATE DerivedColumn
PROPAGATE Lookup
PROPAGATE OLEDBDestination
PROPAGATE OLEDBSource
PROPAGATE Sort
PROPAGATE Deletion
PROPAGATE Rename
PROPAGATE DataType
Allow deletion of vertices
Allow modification of expressions
Use global blocking semantics

Example



Sale
ID: int (PK)
CustomerID: int (FK)
StoreID: int (FK)
TotalAmount: money
Person
Extracts all from Person
Computes *Amount-Times10*
OLE DB Source
Lookup
Lookup Match
Derived Column
Lookups TotalAmount
Conditional Split
unt > 10k
Age > 40
OLE DB Destination 1
Aggregate



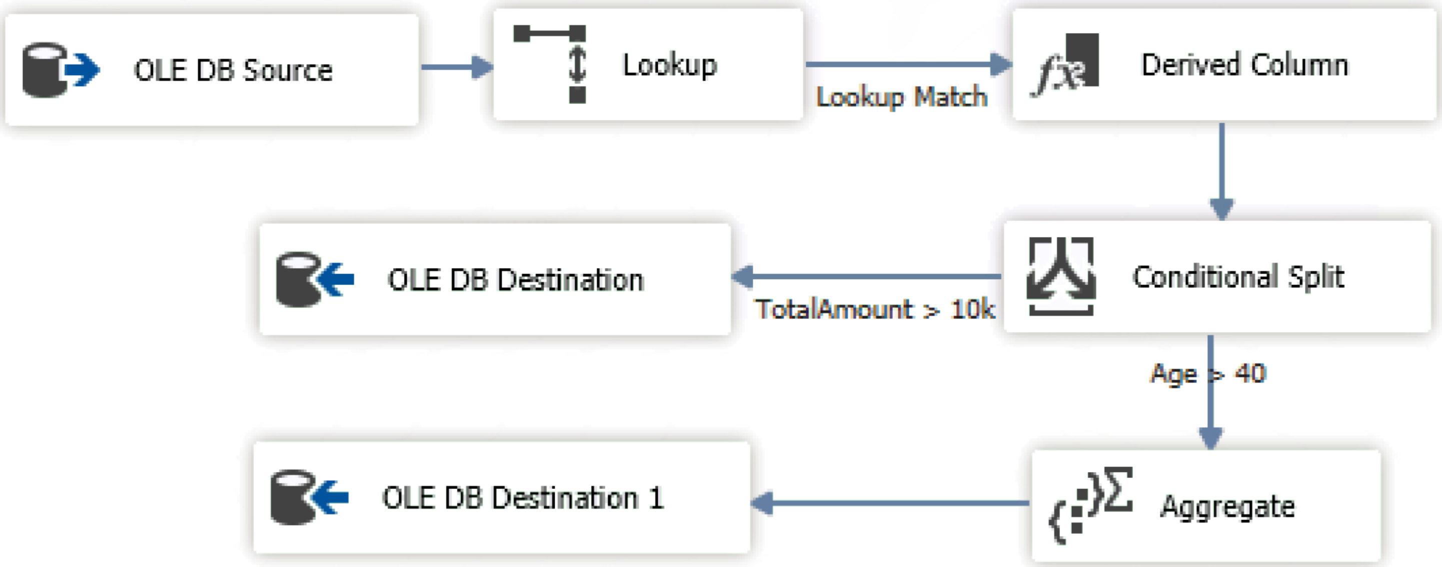
Name = “OLE DB Source”
Database = ”SourceDB”
Table = ”Person”
Columns = { ID, Name, Age}
Dependencies = {
ID → Ø
Name → Ø
Age → Ø }
Name = “Lookup”
Database = ”SourceDB”
Table = ”Sale”
Columns = {
ID, CustomerID, StoreID,
TotalAmount}
OutputColumns =
{TotalAmount}
Joins =
{(ID, CustomerID)}
Dependencies = {
ID → {ID}
Name → {Name}
Age → {Age}
TotalAmount→ {ID}}
Name = “Derived
Column”
Derivations = {
(”TotalAmount * 10",
AmountTimes10)}
Dependencies = {
ID → {ID}
Name → {Name}
Age → {Age}
TotalAmount→
{TotalAmount}
AmountTimes10 →
{TotalAmount}}
Name = “Conditional
Split”
Conditions = {
(”Age > 40", 1,
Aggregate),
(”TotalAmount >
10000", 2, OLE DB
Destination)}
Dependencies = {
ID → {ID}
Name → {Name}
Age → {Age}
TotalAmount→
{TotalAmount}
AmountTimes10 →
{AmountTimes10)}
Name = “OLE DB
Destination”
Database =
”TargetDW”
Table =
”PersonSalesData”
Columns = { ID, Age,
AmountTimes10 }
1
2
3
4
5
Columns =
{(1, ID, int),
(2, Name, string),
(3, Age, int)}
Columns =
{(1, ID, int),
(2, Name, string),
(3, Age, int),
(4, TotalAmount,
currency)}
Columns =
{(1, ID, int),
(2, Name, string),
(3, Age, int),
(4, TotalAmount,
currency),
(5, AmountTimes10,
currency)}
Columns =
{(1, ID, int),
(2, Name, string),
(3, Age, int),
(4, TotalAmount,
currency),
(5, AmountTimes10,
currency)}
Name = “OLE DB
Destination 1”
Database = ”TargetDW”
Table = ”SalesData”
Columns = {
Age, AvgAmount }
7
6
Columns =
{(1, Age, int),
(2, AvgAmount,
currency)}
Name = ”Aggregate”
Aggregations = {
(AVERAGE,
AmountTimes10,
AvgAmount, OLE DB
Destination 1),
(GROUP BY, Age, Age,
OLE DB Destination 1)}
Dependencies = {
Age → {Age}
AvgAmount →
{AmountTimes10)}
Columns =
{(1, ID, int),
(2, Name, string),
(3, Age, int),
(4, TotalAmount,
currency),
(5, AmountTimes10,
currency)}

Example

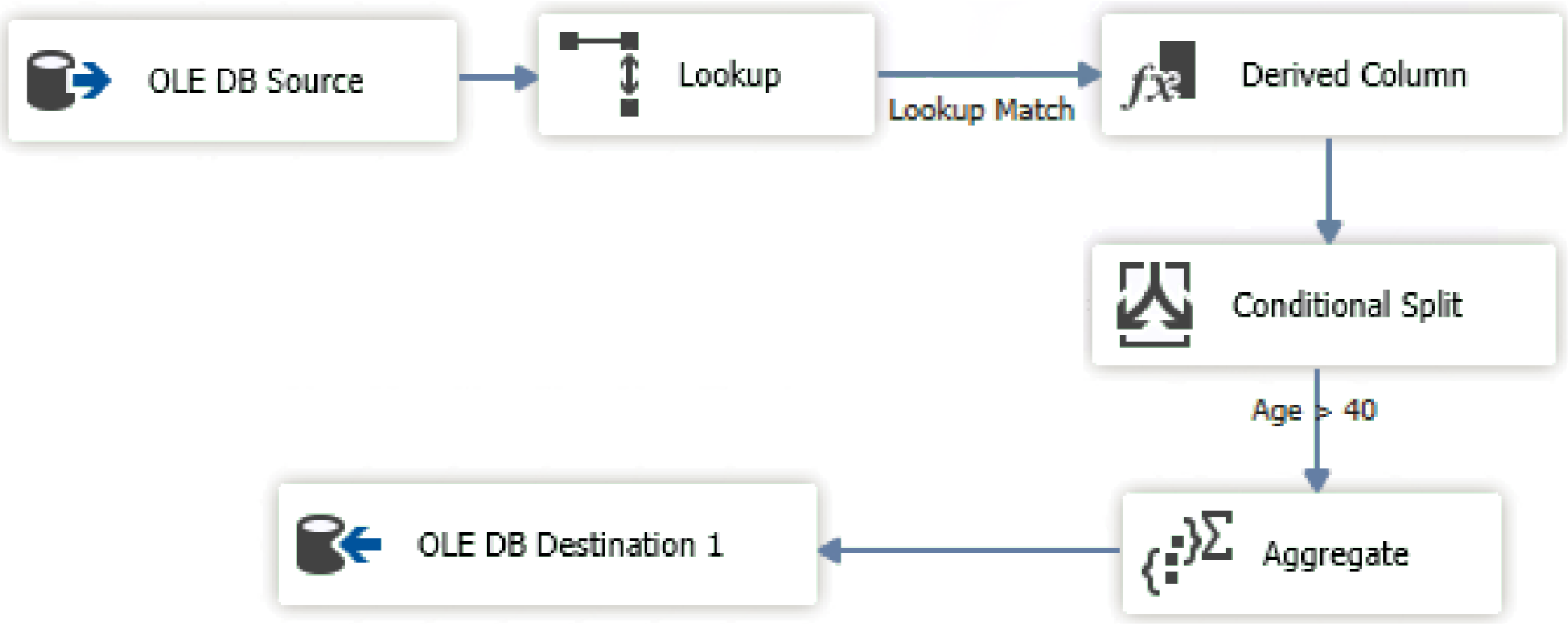
- Now assume the following changes:
  - Age is renamed to RenamedAge in the Person table
  - TotalAmount is deleted from the Sale table
- MAIME will traverse the graph to detect problems and apply fixes (i.e., propagate changes)
  - Renames are easily applied everywhere
  - For deletions, *dependencies* are updated for each vertex
- From the *dependencies*, MAIME sees that AmountTimes10 in Derived Column depends on something that does not exist anymore
- ➔ The derivation is removed (but the transformation stays)

Example

- It is also detected that one of the edges from the Conditional Split no longer can be taken
  - The edge is removed
  - Its destination is also removed since it has no in-coming edges anymore



Result


OLE DB Source
Lookup
Lookup Match
Derived Column
Conditional Split
Age > 40
Aggregate
OLE DB Destination 1

Comparison to manual approach

	1st attempt		2nd attempt		3rd attempt	
	Manual	**MAIME**	**Manual**	**MAIME**	**Manual**	**MAIME**
Time (seconds)	187	4	159	4	**59**	**4**
Keystrokes	23	0	15	0	**12**	**0**
Mouse clicks	88	4	85	4	**38**	**4**

Conclusion

- Maintenance of ETL processes *after* deployment is time-consuming
- We presented MAIME which detects schema changes and then identifies affected places in the ETL processes
- The ETL processes can be repaired automatically – sometimes by removing transformations and edges

- Positive feedback from BI consultancy companies

- In the future, the destination database could be modified, e.g, when a column has been added to the source or changed its type

Related work

- ***Hecataeus*** by G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Yannis Vassiliou
  - Abstracts ETL processes as SQL queries, represented by graphs with subgraphs
  - Detects evolution events and proposes changes to the ETL processes based on policies
  - Propagate (readjust graph), Block (keep old semantics), Prompt
  - Policies can be specified for each vertex/edge
- ***E-ETL*** by A. Wojciechowski
  - Model ETL processes through SQL queries
  - Policies: Propagate, Block, Prompt
  - Different ways to handle changes: Stanadard Rules, Defined Rules, Alternative Scenarios

Agenda

- Introduction to pygrametl – a framework for programmatic ETL
- Explicit parallelism in pygrametl
- A case-study
- Open-sourcing pygrametl
- ETLMR
- CloudETL
- MAIME – programmatic changes/repairs of SSIS Data Flows

References

- C. Thomsen and T. B. Pedersen: ”pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers”. In *Proc. of DOLAP*, 2009
- C. Thomsen and T. B. Pedersen: ”Easy and Effective Parallal Programmable ETL”. In *Proc. of DOLAP*, 2011
- O. Andersen, B. B. Krogh, C. Thomsen, and K. Torp: ”An Advanced Data Warehouse for Integrating Large Sets of GPS Data”. In *Proc. of DOLAP*, 2014
- X. Liu, C. Thomsen, and T. B. Pedersen: ”MapReduce-based Dimensional ETL Made Easy”. *PVLDB 5(12)*, 2012
- X. Liu, C. Thomsen, and T. B. Pedersen: ”ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce”. *TLDKS VIII*, 2013
- X. Liu, C. Thomsen, T. B. Pedersen: ”CloudETL: Scalable Dimensional ETL for Hive”. In *Proc. of IDEAS*, 2014
- D. Butkevičius, P. D. Freiberger, F. M. Halberg, J. B. Hansen, S. Jensen, M. Tarp, H. X. Huang, C. Thomsen: ”MAIME: A Maintenance Manager for ETL Processes”. In *Proc. of DOLAP*, 2017