Easy and Effective Parallel Programmable ETL

Christian Thomsen
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
chr@cs.aau.dk

Torben Bach Pedersen
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
tbp@cs.aau.dk

ABSTRACT

Extract–Transform–Load (ETL) programs are used to load data into data warehouses (DWs). An ETL program must extract data from sources, apply different transformations to it, and use the DW to look up/insert the data. It is both time consuming to develop and to run an ETL program. It is, however, typically the case that the ETL program can exploit both task parallelism and data parallelism to run faster. This, on the other hand, makes the development time longer as it is complex to create a parallel ETL program. To remedy this situation, we propose efficient ways to parallelize typical ETL tasks and we implement these new constructs in an ETL framework. The constructs are easy to apply and do only require few modifications to an ETL program to parallelize it. They support both task and data parallelism and give the programmer different possibilities to choose from. An experimental evaluation shows that by using a little more CPU time, the (wall-clock) time to run an ETL program can be greatly reduced.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Data warehouse and repository*; D.3.3 [**Programming Languages**]: Language Constructs and Features —*Frameworks*

General Terms

Languages, algorithms

Keywords

Extract–Transform–Load (ETL), parallelism, Python

1. INTRODUCTION

To get data into a data warehouse, the data must first be extracted from the source system(s), then transformed to fit into the DW, and finally loaded. This is done by a process called an Extract–Transform–Load (ETL) process. It is well-known that it is a complicated and time-consuming job to make a working solution for an ETL process. Just to get the functionality right is challenging but when that is in place, much work must often go into tuning and parallelizing the solution due to the limited time available for ETL. Good parallelization is even more relevant with today's multicore CPUs which enable true parallelism but which often are not exploited well by programmers.

In this paper, we propose easy ways to enable both task parallelism and data parallelism in ETL programs. We propose constructs for parallelizing the extract part, transform part, and load part, i.e., all parts in ETL.

Previously, we have introduced the `pygrametl` framework [13] which makes it possible to create ETL programs by means of *programming* in Python [10] (possibly calling code written in other languages) instead of drawing ETL programs in GUIs as done in the leading commercial systems. This was motivated by the observation that trained specialists often work more efficiently with textual interfaces than graphical interfaces. In our experience, most ETL developers are indeed trained and skilled specialists. The ETL developers' productivity does not become high just by using a graphical tool. Code-based, i.e., textual, programming offers expressive power and is in the general case of (non-ETL) programming by far more common than graphical programming.

When `pygrametl` is used, the ETL programmer uses the `Dimension` and `FactTable` classes and their subclasses to create objects representing the dimension tables and fact tables in the DW. The programmer then calls methods on these objects. For a `Dimension`, the most important operations are `lookup` which based on the "lookup attributes" (i.e., the business key) finds the member's key value, `insert` which inserts a member, and `ensure` which calls `lookup` and continues to call `insert` if no result is found. (Other concepts from `pygrametl` will be explained when they are used in the rest of the paper.) The use of objects representing tables in the DW makes it very easy to express even complicated ETL program in few lines of code and gives the programmer a very fine-grained control of the flow. It is, however, still a complicated process to parallelize an ETL program. To remedy this situation and enable more efficient ETL programs, this paper presents novel and general functionality for performing ETL in parallel using both task parallelism and data parallelism. The functionality has been implemented as extensions to `pygrametl` and has been made to be very easy to use and only require few, simple changes of a non-parallel program to enable parallelism. The contributions include new classes for processing dimension tables and fact tables in parallel with other tasks. We also introduce novel and easy constructs to turn an ordinary function into a function running in parallel with other execution units.

The rest of paper is organized as follows. Section 2 introduces a DW schema which we use as a running example throughout the paper. Section 3 presents two new data sources added to `pygrametl`. Section 4 presents the new `Decoupled` classes that allow parallel operations on `Dimension` and `FactTable` ob-

DOLAP'11, October 28, 2011, Glasgow, Scotland, UK.

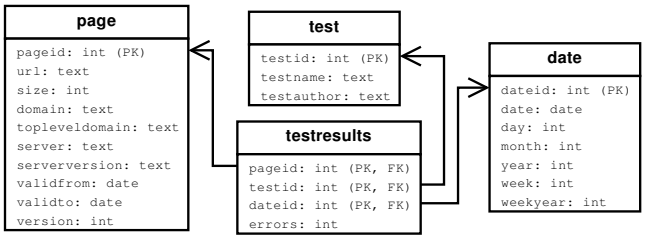

Figure 1: DW schema for the running example

jects. Section 5 presents "shared connection wrappers" that allow several concurrent ETL processes to safely share a single connection to the target DW DBMS. Section 6 presents the new "split points" and "flows" that make it easy to use functions in parallel. Section 7 presents a performance study of the new constructs. Finally, Section 8 presents related work before Section 9 concludes and points to future work.

2. EXAMPLE SCENARIO

In the following parts of the paper, we show small code examples to demonstrate how the proposed constructs are used. For space reasons, we use a very small star schema which is inspired by work we did in a previous project. It is similar to the running example in [13], but here turned into a pure star schema. An identical schema is used in [7]. In the example, we assume that data about different *tests* of web pages is to be loaded into a data warehouse with the schema shown in Figure 1. Our example schema has the fact table `TestResults` with the single measure `errors` telling how many errors a given test found on a given version of a web page on a given date. Tests, web pages, and dates are represented by the `Test`, `Page`, and `Date` dimension tables, respectively. Note that `Page` is a slowly changing dimension (SCD) where we use type-2 updates leading to a new row in the dimension table for each new version of the represented web page. The dimension table for `Page` has many more rows than the other dimension tables.

We assume that the source data comes from two files with comma-separated values (CSV files) produced by another system. One of these files holds information about downloaded web pages and the other holds information about the results of applying some tests to these downloaded files. A full implementation of a working ETL solution for this example as well as data generator and the `pygrametl` code-base can be downloaded from `pygrametl.org`.

3. PARALLEL DATA SOURCES

Typically, the data extracted from the sources must be transformed during the ETL processing. The new `TransformingSource` class can be used to apply any number of transformations to all rows coming from another data source (in `pygrametl`, a data source is something that can be iterated and yields "rows" with mappings from attribute names to values). When a `TransformingSource` is created, it is given another data source as argument as well as the transformations, i.e., functions, to apply (Python supports passing of functions as arguments). Assume that we have already defined a data source called `joineddata` which joins two other `CSVSource`s which input data from CSV files. To apply the transformations `extractdomaininfo`, `extractserverinfo`, and `convertsize` to its rows, a `TransformingSource` can be set up by the following.

```
transformeddata = TransformingSource(joineddata, \
      extractdomaininfo, extractserverinfo, convertsize)
```

It would also be possible for the programmer to call the transforming functions explicitly in her code. However, the `TransformingSource` makes it easy to have the transformations applied in parallel with other tasks as described next.

To achieve parallelism in an ETL process, a first and easy step can be to extract data in parallel with the rest of the ETL program's tasks. This is enabled by means of the `ProcessSource` class. When creating an instance of this class, an instance s of another data source is given as an argument. The `ProcessSource` will during its initialization spawn a new process[1] in which the extracting code for the given data source will execute, i.e., s executes in the new process. `ProcessSource` thus enables task parallelism.

If we use a `ProcessSource` around `transformeddata` from above, the reading, joining, and transformation of data take place in a separate process. The parent process can then spend time on other tasks.

```
transformeddata = TransformingSource( ... ) # as before
indata = ProcessSource(transformeddata)
for row in indata:
    ... # Insert the data into dimension and fact tables
```

The spawned process transfers the extracted rows (in configurably-sized batches) to the parent process such that the `ProcessSource` there can deliver rows with no or little delay. The `ProcessSource` can be used as any other source and the programmer does not have to consider that a separate process does the real work.

4. PARALLEL TABLE OPERATIONS

In an ETL program, most of the time is spent on looking up and/or inserting dimension values and inserting facts. It is thus attractive if these operations can be pushed to separate processes and performed in parallel (even if only a single database connection is used, it is often possible to perform the operations in parallel if `pygrametl` performs its own caching). To enable this, we introduce the `Decoupled` class and its subclasses `DecoupledDimension` and `DecoupledFactTable` which provide both task and data parallelism.

4.1 Decoupled Objects

The purpose of the `Decoupled` class is to be able to spawn a new process and let a given object o execute in that process (we say the object o is *decoupled*). In the parent process, a `Decoupled` instance acts as proxy for the decoupled object which is executing in the separate process. It is then the responsibility of the `Decoupled` class to enable communication between the two processes such that arguments and return values can be transferred. To decrease communication overhead, `Decoupled` internally sends *batches* of many invokations and return values between the processes.

By means of the `Decoupled` object, it is possible for the parent process to invoke a method on the decoupled object and instead of waiting for the result, the parent process can continue doing other work (enabling task parallelism). However, the return value from a method is likely to be needed eventually. Therefore, `Decoupled` can return an instance of `FutureResult` when a method on the proxied object is invoked. This `FutureResult` can later be used to get the actual return value of the method. It is also possible to invoke a method on the proxied object such that the return value is discarded.

[1] In CPython [10], the most popular Python implementation, `pygrametl` uses processes. In Jython [5], where Python is executed in the Java Virtual Machine, `pygrametl` uses threads. In the paper, we use the term "process" to denote a process or thread.

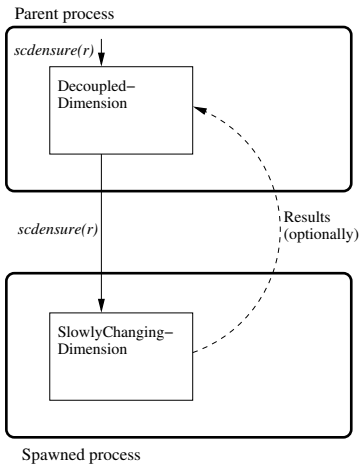

Figure 2: A decoupled `SlowlyChangingDimension`

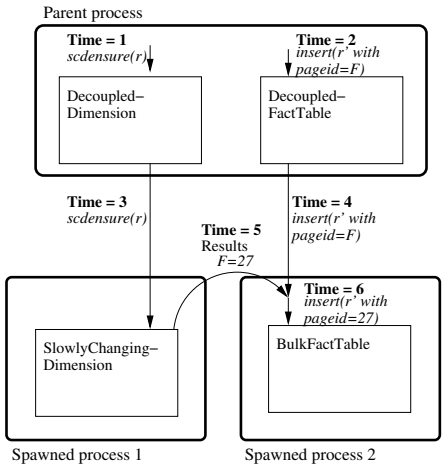

Figure 3: Two decoupled objects where one consumes the other

Consider again our running example. The `page` dimension is an SCD and large, so much time is spent on its processing. It thus makes good sense to decouple the `pygrametl` object for the `page` dimension. To do this, we use the `DecoupledDimension` class which inherits from `Decoupled` and in addition offers the same interface as the `Dimension` class and `Dimension`'s subclass `SlowlyChangingDimension` which is a specialization for SCDs.

```
pagedim = DecoupledDimension(
            SlowlyChangingDimension(name='page', ...))
```

All we have to do to decouple the `pagedim` object, is to surround the existing declaration with `DecoupledDimension( ... )` Figure 2 illustrates how two processes exist after this. If a method, say `scdensure`, on the `DecoupledDimension` in the parent process is called, it leads to that `scdensure` on the `SlowlyChangingDimension` in the new process is invoked

Now consider the following `pygrametl` snippet showing the core part of the ETL program for our running example (which remains unchanged even though we just made `pagedim` a `DecoupledDimension`).

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

In the parent process, the call of `pagedim.scdensure` immediately returns a `FutureResult` such that the process can continue. Before the insertion into the fact table, the correct value for `pageid` is, however, needed. One possibility is to explicitly request the result before calling `insert` on the fact table object. This would, however, block the parent process until the result was available and would thus hinder parallelism (the situation would be slightly better if we operated on `pagedim` before `testdim` and `datedim` as we then could do some work in parallel). It would also require the beforementioned batches to be of size 1. A better solution is to also decouple the `facttbl` object and let it *consume* `pagedim`. When a `Decoupled` object d_c consumes another `Decoupled` object d_p, it means that d_c's spawned process must replace `FutureResult`s created by d_p with the actual return values which the d_c process gets from d_p's process before the d_c process invokes methods on d_c's decoupled object. Simply put, this means that `FutureResult`s are replaced by actual return values before other operations take place. To decouple the object for the fact table (here of the specialized type `BulkFactTable`), the `DecoupledFactTable` class is used as in the following code

```
facttbl = DecoupledFactTable(BulkFactTable(...),
                             consumes=pagedim,
                             returnvalues=False)
```

The `consumes` argument specifies that `FutureResult`s from `pagedim` should be replaced and the `returnvalues` arguments specifies that return values from the decoupled object are not to be used later on. The parent process previously shown can now continue its operations without waiting on exchange of `FutureResult`s with actual results as the process spawned by the `DecoupledFactTable` will take care of this. This is illustrated in Figure 3 where timestamps also show in which order things happen. At time 1, `scdensure` on the `DecoupledDimension` is invoked. This immediately returns a `FutureResult` F such that the parent process can continue its work and invoke `insert` on the `DecoupledFactTable` (and, in the argument, use F for which an actual result still does not exist) at time 2. At times 3 and 4, the invocations of `scdensure` and `insert` are sent to the spawned processes, respectively. The invocation of `scdensure` on the decoupled `SlowlyChangingDimension` in Process 1 can immediately take place, but before the invocation of `insert` on the `BulkFactTable` in Process 2 can take place, the `FutureResult` F must be replaced. At time 5, the actual result for F (here `27`) is fetched by Process 2 which then replaces F with the actual result and performs the `insert` at time 6.

In summary, only the declarations of the `pagedim` and `facttbl` objects had to be updated by surrounding them with `DecoupledDimension(...)` and `DecoupledFactTable(...)`, respectively, to use parallelism. The core part of the ETL program remains unchanged.

4.2 Using Partitioned Decoupled Objects

In our running example, the `page` dimension takes much more time to process than the other two dimensions and the facts. The `page` processing remains the bottleneck even if `pagedim` is decoupled. It is therefore of interest to use data parallelism and have more than one decoupled object to process the `page` dimension. To do this, the programmer can create several `SlowlyChangingDimension` objects.

A first problem to consider is, however, the assignment of surrogate key values. In pygrametl, Dimensions use a function (called idfinder) to find the surrogate key value for a dimension member to insert if it is not already present in the data. If this function is not given explicitly by the user, pygrametl's Dimension classes simply assume that the next value should be the current maximum plus one. This is error prone when several processes operate in parallel on one physical dimension table. We therefore introduce the *shared sequence factory* which allows a single sequence to be created such that several parallel readers can get unique values from it. For efficiency reasons, a reader actually gets a range of numbers which it can use before it needs to request a new range. In our example, this can be set up as follows.

```
idfactory = getsharedsequencefactory(0) # start from 0
pagedim1 = DecoupledDimension(
              SlowlyChangingDimension(name='page',
                 ..., idfinder=idfactory() ))
pagedim2 = DecoupledDimension(
              SlowlyChangingDimension(name='page',
                 ..., idfinder=idfactory() ))
```

(Note that by using good programming practice, we could have avoided the repeated code above, e.g., by making two calls of a function that returns a DecoupledDimension object.)

With the two objects pagedim1 and pagedim2 operating on the page dimension, the programmer can manually partition rows between them. If one were to do this, it would, however, require changes in the core part of the ETL program when more or fewer objects were used for the page dimension. Instead, it is better to use pygrametl's new DimensionPartitioner class which handles partitioning between any number of Dimensions. The class offers an interface like that offered by the Dimension classes and thus the core part of the ETL program does not have to be changed when partitioning is introduced. When a DimensionPartitioner *dp* is created, it is given a list of Dimension objects called *parts* (which can operate on the same physical table or on different physical tables/partitions) and optionally a partitioning function. When a method on *dp* is invoked, it will invoke a method of the same name on one of the Dimension objects. The Dimension object to use is chosen by means of the partitioning function. If a partitioning function is not given by the programmer, pygrametl by default uses a function that computes the hash value of the lookup attributes of the dimension. For our example, we can partition among the previously defined pagedim1 and pagedim2 with the following code. We give an explicit partitioner (here as a lambda expression, but a reference to a named function also works) which for a given row computes the hash value of the value of the domain attribute. We could also have chosen not to specify a partitioner and just use pygrametl's default partitioner.

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

The result of this is shown in Figure 4 where the page dimension table in the DW is also partly shown. The core part of the ETL program now has its rows distributed between several objects operating on the page dimension without any changes to the rest of the code. It is also very easy to add or remove parts. Note how one of the DimensionPartioner's parts handle all dimension members with the domain value d1.com while the other handles all dimension members with domain value d2.org in Figure 4. The partitioning function should exactly be chosen such that all rows with identical values for a certain subset of the attributes (e.g., the business key as in the default partitioning function) are sent to the same part.

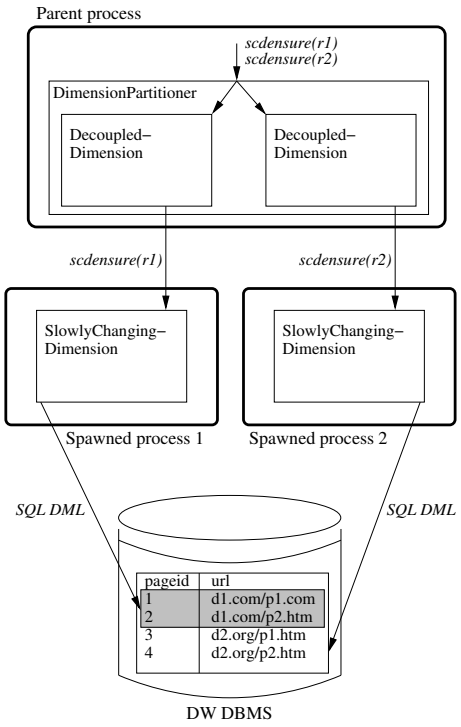

Figure 4: A DimensionPartitioner and decoupled parts

The Dimension classes, however, have the method getbyvals which from a row with values for a subset of a dimension's attributes looks up all dimension members with those values. It can be configured whether DimensionPartitioner only should send such a request to one part (this makes sense when the all parts operate on the same physical table) or to all parts and union the results (useful when different physical tables are used).

Similarly to the class DimensionPartitioner, the class FactTablePartitioner also exists. It offers an interface like that offered by FactTable and partitions between several FactTable objects. The default partitioning function in this class simply sums the values for the fact table's foreign keys to dimensions.

With these classes, it is very easy to add parallelism to a pygrametl program. One should, however, not just create several decoupled objects for each physical table and assume that performance will be better. The overhead from handling many decoupled objects and their communication can outweight the advantage they give. It is very informative to use tools like jvisualvm or top to see how the processes are working. With a tool like pygrametl, the programmer thus has a very fine-grained control of what to parallelize.

5. SHARED CONNECTIONS

In the previous section, we saw how several objects can operate on the underlying database's tables in parallel. A single database connection can, however, often not be used in parallel. One solution to this is to create a unique database connection for each Decoupled object. This is, however, not a good solution due to high resource consumption and the risk for inconsistency. Instead, pygrametl offers to *share* a ConnectionWrapper (pygrametl uses ConnectionWrappers to have a uniform view of different database connections in Python that use different parameter

formats). To share a `ConnectionWrapper`, a single function call is needed.

```
cw = ConnectionWrapper(connect(...)) # Make a connection
cw = shareconnectionwrapper(cw)      # Share it
```

The function `shareconnectionwrapper` spawns a new process (the "server process") which uses the given `ConnectionWrapper` exclusively. Further, it returns a `SharedConnectionWrapperClient` object. Everywhere else that needs access to the database (via the shared connection) uses its own `SharedConnectionWrapperClient` object (a new instance can be created by calling the `new()` method on an existing instance). This object can be used like a normal `ConnectionWrapper` object and does not require changes in the code. When an operation *o* is invoked on the `SharedConnectionWrapperClient`, it transparently adds *o* to a queue (shared by all instances) from which the server process fetches and executes tasks. In this way, only a single operation is performed on the database at a given time.

Sometimes user-defined functions (UDFs) also need synchronized access to the database connection. A typical example appears with bulk loading where `pygrametl`'s `BulkFactTable` uses call-back of a UDF that invokes the actual bulk loading in a DBMS-specific way. It is then necessary that the UDF also uses the database connection from the server process. However, the UDF may use other functionality than that exposed by the `ConnectionWrapper` and `SharedConnectionWrapperClient` classes. Therefore, `shareconnectionwrapper` can also be given a list of functions (called `userfuncs`) that it should be able to invoke *in* the server process *via* any of the `SharedConnectionWrapperClient`s. For example, the UDF `myloader` for bulk loading can be made available in this way by the following.

```
def myloader(*args):
    # DBMS specific bulk loading code here...

cw = ConnectionWrapper(connect(...))
cw = shareconnectionwrapper(cw, userfuncs=[myloader])
```

After this, `shareconnectionwrapper` has dynamically created a method named `myloader` on the `SharedConnectionWrapperClient` objects. When this method is invoked, the invocation will be queued like all other database operations until it is executed in the server process. The only thing to update in the declaration of the `BulkFactTable` object is the `bulkloader` argument telling which UDF to call. Now the dynamically created method `myloader` on (a unique instance returned by `new` on) the `SharedConnectionWrapperClient` should be used.

```
facttbl = BulkFactTable(..., bulkloader=cw.new().myloader)
```

6. MAKING FUNCTIONS PARALLEL

In many ETL programs, complex and time-consuming transformations take place before the data is inserted into the database. With a code-based solution like `pygrametl`, the ETL programmer can obviously already choose to use several processes to do the transformations. However, this can be complex to set up and influences the program's organization heavily. We wish to make it possible to focus on the logic and write a non-parallel ETL program which then easily can be turned into a program doing different things in parallel. To achieve this, we introduce the `@splitpoint` annotation the so-called *flows*.

6.1 Split Points

The `@splitpoint` annotation (which itself is a function) in some sense does for functions what the `Decoupled` class does for classes. When the function f is annotated with `@splitpoint`, a new process s is spawned such that each function call of f in the parent process returns (with no return value) nearly immediately and instead leads to that f is executed in the process s. We call f a *split point* as it *splits* execution between the parent process and the separate process s. Note that the process s is only set up once – not on each call of f. The `@splitpoint` can also be set up such that return values from f eventually become available in a queue in the parent process or another process. This is done by passing the queue object as an argument to the `@splitpoint` annotation. The `@splitpoint` can also be made to start several processes to execute the annotated function. For example, the function `myfunc` can be made to run in two separate processes by the annotation in the line right before the function definition shown below. Calls of `myfunc` in the parent process are automatically being executed by the first idle instance. This is shown in Figure 5.

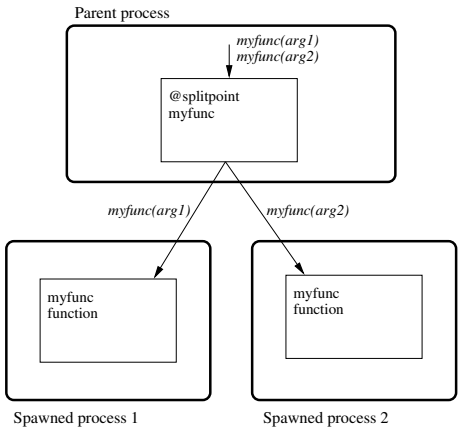

Figure 5: The split point `myfunc` and its spawned processes

```
@splitpoint(instances=2)
def myfunc(*args):
    # Do the work here...
```

When a function f is annotated with `@splitpoint`, all its work, including other function calls, is done in the dedicated process. It is, however, possible to "merge" function calls again if only a single function instance of g should exist, e.g., to insert into a fact table. That can be done by annotating g with `@splitpoint` as well and only create one instance. All calls of g from the different instances of f are then executed in the single process for g. The following shows an example.

```
@splitpoint(instances=4)
def dotransformations(row)
    # Do some (expensive) transformations
    ...
    insertfact(row)

@splitpoint   # the instances argument defaults to 1
def insertfact(row):
    facttbl.insert(row)
```

The `@splitpoint` annotation enables both task and data parallelism as many different split points can execute at the same time and one split point can execute in many processes. Because the annotated function executes in a separate process, the process calling the split point may not know when the separate process has finished. Therefore, `pygrametl` provides the function `endsplits` which blocks a calling process until all the separate processes have finished executing all calls of split points made before the call of `endsplits`.

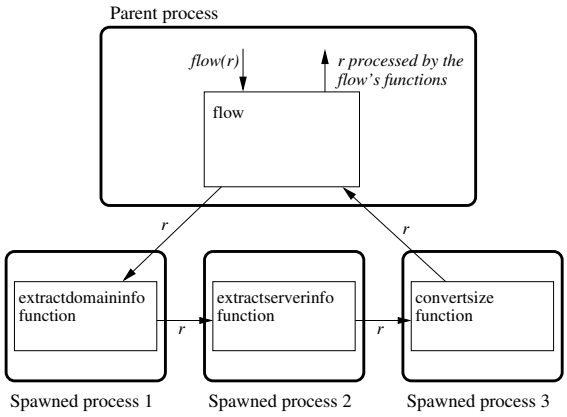

Figure 6: A flow with three functions in three processes

6.2 Flows

Another way to use different functions in parallel (i.e., task parallelism) in `pygrametl` is to use *flows*. A flow F consists of a sequence of functions, $f_1, \ldots, f_n$ running in parallel in different processes. F is callable such that $F(x)$ corresponds to first calling $f_1(x)$, then $f_2(x)$, etc. The return values from the f_i's are ignored, but their side-effects on rows are preserved such that they can update, add, and delete values in rows. The f_i functions process different data parts at a given time. Unlike `@splitpoint` functions, a flow internally groups the flow arguments into (configurably-sized) batches such that less locking and synchronization is needed to transfer the arguments between the different processes.

A flow is created by calling the `createflow` function. Consider, for example, the functions `extractdomaininfo`, `extractserverinfo`, and `convertsize` which we in Section 3 handled in a `TransformingSource`. Instead, we could create a flow consisting of these functions.

```
flow = createflow(extractdomaininfo, exctractserverinfo,
                  convertsize)
```

This results in the situation shown in Figure 6. A row to be processed can now be given to the flow simply by calling it as in `flow(row)` or alternatively `flow.process(row)` which does the same. Arguments that enter a flow need to be consumed again later (but the flow contains rows in a queue until they are consumed). A single processed row can be fetched by calling the `get` method on the flow object. It is also possible to iterate a flow such that each processed row is fetched in turn by using

```
for row in flow:  # do something with the row
```

In the flow created above, each of the three given functions runs in its own process. It is also possible to group functions such that two or more functions run in the same process. The example below shows the creation of a flow where `extractdomaininfo` and `extractserverinfo` run in the same process.

```
flow = createflow((extractdomaininfo, exctractserverinfo),
                  convertsize)
```

It is possible to feed data into a flow from one process while another process concurrently consumes the processed data. To do this, we can use the `@splitpoint` annotation to spawn a separate producer process which runs concurrently with the parent process that consumes the data. This is shown in the following example. Note that the call of `producedata` starts the execution in the spawned process such that the parent process can continue to call `consumedata` and thus do work in parallel.

```
flow = createflow(...)

@splitpoint
def producedata():
    global flow
    for row in somesource:
        flow(row) # Give the row to the flow
    flow.close() # Signal end of data

def consumedata()
    global flow
    for row in flow: # Now transformed by the flow
        # Do something with the data
        ...

producedata() # Happens in a separate process ...
consumedata() # ... so we concurrently can proceed here
print 'Done'
```

7. PERFORMANCE

In this section, we consider the performance of `pygrametl`. To do this, we use the schema shown in Figure 1 and use `pygrametl` to create ETL programs to load it (the programs are available at `pygrametl.org`). We also compare the performance of the `pygrametl` programs to ETL programs created in the leading open source ETL tool Pentaho Data Integration [6] (PDI).

All experiments are performed on a server with two quad-core 1.86GHz Xeon CPUs, 16GB of RAM, and 10,000 rpm harddisks. The server runs Linux 2.6, PostgreSQL 8.4, and we use Jython 2.5.2 for `pygrametl` and Java 6 SE. We use PDI version 3.2 (version 3.2 is not the newest, but gives better results than 4.0 and 4.1 in this experiment) which is implemented in Java. The Java Virtual Machine is allowed to allocate up to 10GB RAM which is more than enough for all considered programs. The data sets are generated in the same way as in [13]. There are five tests being applied to all pages from 2,000 domains once per simulated month. Each domain has 100 pages. For each month, there are thus 1 million facts. A page becomes updated from one month to another with probability 0.5. That means that 100,000 new page versions are inserted into `page` for each month (200,000 for the first month). The reported performance numbers are averages based on five experiments.

Figures 7 and 8 show the processing times for data sets of different sizes (the reported performance numbers are averages based on five experiments). Figure 7 shows elapsed time while Figure 8 shows CPU time.

The figures show the processing times for different ETL implementations. The "pygrametl1" implementation is a non-parallel program using `pygrametl`. It does not use the new support for parallel work in `pygrametl`. The "pygrametl2" implementation does use the new parallelism functionality in `pygrametl`. Concretely, pygrametl2 uses a `DimensionPartitioner` which in turn uses two `DecoupledDimensions` for the `page` dimension which is an SCD and much larger than the other dimensions. In this example, the performance does not improve by using more `DecoupledDimension` instances. Further, pygrametl2 uses a `DecoupledFactTable` for the fact table to load. Both pygrametl1 and pygrametl2 are transactionally safe such that each of them only has a single database connection (which is "shared" in pygrametl2's case). Figures 7 and 8 also show the processing times when PDI is used. PDI is used in two modes. In the "PDI1" implementation, it only has one database connection like pygrametl1 and pygrametl2 do. In the "PDI2" implementation, it uses bulk loading of facts which requires a separate connection to the database such

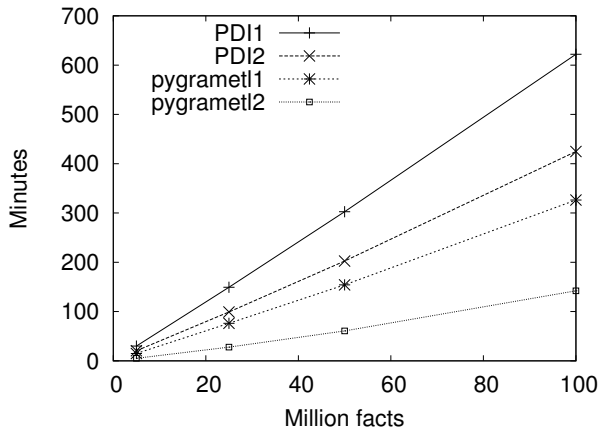

Figure 7: Elapsed time

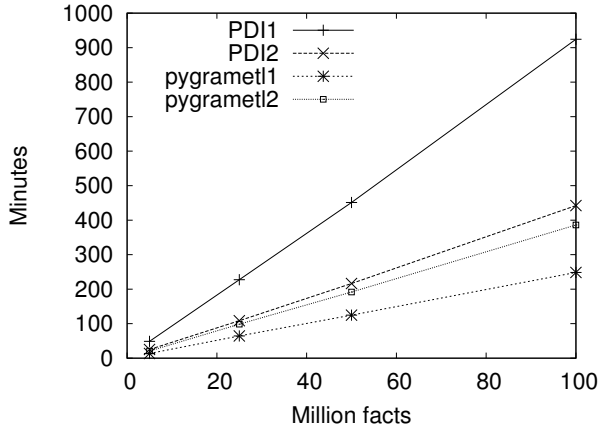

Figure 8: CPU time

that two different connections are used. In other words, the PDI2 solution is not transactionally safe.

As can be seen in Figure 7, the parallel `pygrametl`-based program pygrametl2 is the fastest followed by the non-parallel `pygrametl`-based program pygrametl1. The CPU time grows linearly in the size of the data set for all programs while the elapsed time grows nearly linearly as the system becomes I/O bound. To illustrate this better, Table 1 shows the CPU usage (calculated as CPU time / elapsed time) for the programs. It can be seen that for larger data sets, the CPUs are idle more of the time. It can also be seen that pygrametl2 utilizes the available CPUs better than the other programs. While it is running, pygrametl2 utilizes roughly the double amount of CPU power than PDI1 which has the second highest CPU usage.

When we consider the differences between pygrametl1 and pygrametl2, it is clear that the new support for parallelism pays of. A small increase in used CPU time gives a large increase in performance. The available CPUs are used more intensively, but for a shorter amount of (wall-clock) time and thus the total amount of CPU time spent is not much bigger. Table 2 shows how much extra CPU time is used and how much the performance is improved when using pygrametl2 instead of pygrametl1.

Million facts	5	25	50	100
PDI1	162.5%	152.3%	149.1%	148.6%
PDI2	123.4%	109.4%	107.0%	104.1%
pygrametl1	90.4%	84.6%	80.9%	76.0%
pygrametl2	392.2%	352.6%	315.1%	271.6%

Table 1: Average CPU usage while running

Million facts	5	25	50	100
Extra CPU usage	53%	53%	54%	55%
Performance improvement	183%	172%	153%	130%

Table 2: pygrametl2 vs. pygrametl1

8. RELATED WORK

Many ETL and data integration tools are on the market. The Magic Quadrant from Gartner [3] presents an overview of the market. A survey paper [16] presents an overview of academic efforts. No work focussing on parallelization of ETL processes is found.

As ETL tools often operate on very large amounts of data, parallelization is widely applied in dominating ETL tools. Parallel execution can sometimes be achieved without the developer doing anything to enable it. In PDI [6], for example, each "step" (shown as a box in the graphical representation of the ETL flow) executes in its own thread. It is also possible to define that several instances of a step should run in parallel and have the rows distributed among them in round robin fashion. In Microsoft's popular ETL tool SQL Server Integration Services [8] (SSIS) many threads are also used to execute a data flow. SSIS uses a more advanced policy than PDI and considers "execution trees" of transformations that can be processed by the same thread. An execution tree can be assigned one or more threads depending on its CPU utilization. Further, parameters can be used to limit the number of parallel threads. However, it is very different how well different (graphical) designs for a given job exploits parallelism. Therefore, parallelism and the way the tool applies it must also be considered carefully when a graphical ETL tool is used.

In contrast, the developer has complete and direct control over the parallelization in `pygrametl` as she alone decides which parts should execute in parallel. The support for `TransformingSources` and flows makes it possible to group certain transformations together to utilize processes efficiently. Also the support for `Decoupled` objects and `DimensionPartitioner` and `FactTablePartitioner` makes it easy to process (some or all) dimensions and fact tables in separate processes.

For massive amounts of data, a solution running on a single server may not scale well enough. Recently, the MapReduce framework [2] has become very popular for parallel processing of huge amounts of data on clusters of commodity machines. It has been argued that a common use case for MapReduce is to perform ETL-like tasks [12]. MapReduce is, however, a generic programming model and it does not have any built-in support for DW specific concepts such as dimensions and fact tables. Hive [14] is a DW system built on top of MapReduce and has an SQL-like query language for data analysis. Another system for data analysis built on top of MapReduce is Pig [9]. Hive and Pig do not have built-in support for dimensional ETL functionality such as filling an SCD. A recent paper [7] presents the tool ETLMR which modifies `pygrametl` (without the new functionality for parallel execution) to be used on MapReduce. ETLMR scales well and is simple to scale up and down when needed. With ETLMR it is thus easy to cre-

ate scalable dimensional ETL solutions on MapReduce with few lines of code. The constructs in this paper are, on the other hand, for the still more common scenario where the ETL program is run on a single, powerful server with multicore CPUs. This does not scale as much as a MapReduce-based solution, but does not incur overhead from the MapReduce-framework and is thus efficient in doing actual ETL work with the consumed CPU resources. Further, the programmer has more flexibility and fine-grained control with `pygrametl` and the extensions described in this paper. The programmer can thus explicitly decide which parts parallel processing should be applied to and, e.g., start more instances of a split point or a `Decoupled` object for computationally heavy parts. With `pygrametl`, it is also possible to run through the source data only once and update both dimension tables and fact tables. With ETLMR, it is necessary to run through the source data twice since the dimension and fact processing phases cannot be combined. As the constructs in this paper either use threads or "fork" processes, it is also possible to share states and resources between the different parallel parts. For example, the shared connection wrappers make it possible to safely share a single database connection.

We note that according to Gartner [3], commercial ETL tool providers are also beginning to explore use of MapReduce. The most recent versions of PDI [6] also has some support for MapReduce.

The `FutureResult` class used by the `Decoupled` classes in `pygrametl` is inspired by the use of "Futures" in programming languages such as Java [4] and the newest Python 3.2 [11] (but `pygrametl` is still compatible with Python/Jython 2.5). The `FutureResult` is, however, specialized for the typical ETL scenario where, e.g., an `insert` operation is never cancelled. Thus, it does not need possibilities for cancelling an operation. The `pygrametl` implementation does, however, allow the programmer to transfer a `FutureResult` object to another process and let that process fetch the actual result. This is exactly what happens when a `Decoupled` object consumes another `Decoupled` object.

Annotations are also used to make functions execute in separate processes in the PyCSP framework [1]. PyCSP, however, requires in and out channels to be set up explicitly and code to start the execution. `pygrametl`'s approach is less general, but simpler to use as only the annotation is needed. The purpose of `pygrametl` is exactly to make it easy to write (possibly parallel) ETL programs, not to be a general parallel framework (like PyCSP is).

9. CONCLUSION AND FUTURE WORK

In this paper, we have introduced different constructs to enable an ETL programmer to explictly parallelize different parts in all three phases of an ETL program. Some parts in an ETL program may be blocking and prevent parallelization, but for the many parts that are not, the constructs can provide significant performance improvements. The constructs have been designed to be very easy to apply without requiring many or complicated modifications of a non-parallel ETL program. This makes it easy to experiment with different setups and fine-tune an ETL program.

Both data parallelism and task parallelism can be exploited with the new constructs. Data parallelism can be achieved by means of the `@splitpoint` annotation (starting more than one instance) as well as the `Partitioner` classes in combination with the `Decoupled` classes. The `@splitpoint` annotation and the `Decoupled` classes can also be used to enable task parallelism. So can the `ProcessSource` class and the flows created by `createflow`. The programmer thus gets a variety of possibilities to choose from when parallelizing an ETL program. With the possibility for sharing a `ConnectionWrapper`, the parallelized ETL program can still be transactionally safe and only use a single connection to the DBMS.

In the presented performance experiments, it can be seen that the suggested constructs give good improvements. A small increase in used CPU time gives a large increase in performance. The available CPUs are used more intensively, but for a shorter amount of (wall-clock) time.

We plan to continue to improve the performance and usability of the new support for parallelism as we get more experience about how users apply them. As pointed out in [15], commercial tools and academic research have not provided full-blown optimization techniques for ETL processes. While we do not intend to aim for complete, automatic optimization, it would be beneficial if it was possible to to give the programmer hints about how to improve performance. It should thus be possible to enable a monitoring system that identifies, e.g., bottlenecks and gives suggestions about what to parallelize. Future work includes development of such a monitoring system and heuristics. As previously argued, we believe code-based ETL is relevant in many cases. An interesting future solution would also include a GUI from which conceptual modeling can be translated to `pygrametl` code with (semi-)automatic idenfication of parts to parallelize with the methods in this paper. Further, we intend to mature the framework and add support for more built-in standard ETL operations, recovery, etc.

10. REFERENCES

[1] J. M. Bjørndalen, B. Vinter, and O. Anshus: "PyCSP – Communicating Sequential Processes for Python". In *Communicating Process Architectures*, pp. 229–248, 2007.

[2] J. Dean and S. Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters". In *Proc. OSDI*, pp. 137–150, 2004.

[3] T. Friedman, M. A. Beyer, and E. Thoo: "Gartner Magic Quadrant for Data Integration Tools", 2010

[4] `download.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html` as of 2011-08-12.

[5] `jython.org` as of 2011-08-12.

[6] `kettle.pentaho.com` as of 2011-08-12.

[7] X. Liu, C. Thomsen, and T. B. Pedersen: "ETLMR: A Highly Scalable Dimensional ETL Framework based on MapReduce". In *Proc. DaWaK*, pp. 96–111, 2011.

[8] `microsoft.com/sqlserver/2008/en/us/integration.aspx` as of 2011-08-12.

[9] C. Olston et al. "Pig latin: a not-so-foreign language for data processing". In *Proc. SIGMOD*, pp. 1099–1110, 2008.

[10] `python.org` as of 2011-07-02.

[11] `python.org/dev/peps/pep-3148/` as of 2011-08-12.

[12] M. Stonebreaker et al. "MapReduce and Parallel DBMSs: Frineds or Foes?". *CACM* 53(1):64–71 2010.

[13] C. Thomsen and T. B. Pedersen: "pygrametl: A Powerful Programming Framework for Extract–Transform–Load Programmers". In *Proc. DOLAP*, pp. 49–56, 2009.

[14] A. Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". In *Proc. ICDE*, pp. 996–1005, 2010.

[15] V. Tziovara, P. Vassiliadis, A. Simitsis: "Deciding the Physical Implementation of ETL Workflows". In *Proc. DOLAP*, pp. 49–56, 2007.

[16] P. Vassiliadis: A Survey of Extract–Transform–Load Technology. *IJDWM* 5(3):1–27, 2009.