# Traditional ETL

A traditional data warehouse architecture consists of four layers: the data sources, the back-end, the global data warehouse, and the front-end. Typically, the data sources can be any of the following: On-Line Transaction Processing (OLTP) sys- tems, legacy systems, flat files or files under any format. Modern applications have started to use other types of sources, as well, such as web pages and various kinds of documents like spreadsheets and documents in proprietary word processor for- mats.

The set of operations taking place in the back stage of data warehouse archi- tecture is generally known as the Extraction, Transformation, and Loading (ETL) processes. ETL processes are responsible for the extraction of data from different, distributed, and often, heterogeneous data sources, their cleansing and customiza- tion in order to fit business needs and rules, their transformation in order to fit the data warehouse schema, and finally, their loading into a data warehouse.

The global data warehouse keeps a historical record of data that result from the transformation, integration, and aggregation of detailed data found in the data sources. Moreover, this layer involves datastores that contain highly aggregated data, directly derived from the global warehouse (e.g., data marts and views.) The front-end level of the data warehouse architecture consists of applications and techniques that business users use to interact with data stored in the data warehouse.

##### Operational issues and challenges

Traditionally, ETL processes deal with the following generic categories of problems:

Large volumes of data. The volumes of operational data are extremely large, and incur significant data management problems in all three phases of an ETL process.

* Data quality. The data are not always clean and have to be cleansed.
* Evolution of data stores. The evolution of the sources and the data warehouse can eventually lead even to daily maintenance operations.
* Performance issues. The whole process has to take place within a specific time window and it is necessary to optimize its execution time. In practice, the ETL process periodically refreshes the data warehouse during idle or low-load, periods of its operation; e.g., every night. Any failures of the process must also be compensated within the specified time windows.

Additionally, in each individual phase of an ETL process several issues should be taken into consideration.

Extraction. The extraction conceptually is the simplest step, aiming at the identification of the subset of source data that should be submitted to the ETL workflow for further processing. In practice, this task is not easy, basically, due to the fact that there must be minimum interference with the software configuration at the source side. This requirement is imposed by two factors: (a) the source must suffer minimum overhead during the extraction, since other administrative activities also take place during that period, and, (b) both for technical and political reasons, administrators are quite reluctant to accept major interventions to their system’s configuration.

There are four policies for the extraction of data from a data source. The na¨ıve one suggests the processing of the whole data source in each execution of the ETL process; however, this policy is usually not practical due to the volumes of data that have to be processed. Another idea is the use of triggers at the source side; typically, though, this method is not practical due to abovementioned requirement regarding the minimum overhead at the source site, the intervention to the source’s configuration and possibly, the non-applicability of this solution in case the source is of legacy technology. In practice, the two realistic policies suggest either the consideration of only the newly changed - inserted, deleted or updated - operational records (e.g., by using appropriate timestamps at the source sites) or the parsing of the log files of the system in order to find the modified source records. In any case, this phase is quite heavy, thus, it is executed periodically when the system is idle.

Transformation & Cleaning. After their extraction from the sources, the data are transported into an intermediate storage area, where they are transformed and cleansed. That area is frequently called Data Staging Area, DSA, and physically, it can be either in a separate machine or the one used for the data warehouse.

The transformation and cleaning tasks constitute the core functionality of an ETL process. Depending on the application, different problems may exist and dif- ferent kinds of transformations may be needed. The problems can be categorized as follows: (a) schema-level problems: naming and structural conflicts, includ- ing granularity differences, (b) record-level problems: duplicated or contradicting records, and consistency problems, and (c) value-level problems: several low-level technical problems such as different value representations or different interpretation of the values. To deal with such issues, the integration and transformation tasks involve a wide variety of functions, such as normalizing, denormalizing, reformat- ting, recalculating, summarizing, merging data from multiple sources, modifying key structures, adding an element of time, identifying default values, supplying decision commands to choose between multiple sources, and so forth.

Usually the transformation and cleaning operations are executed in a pipelining order. However, it is not always feasible to pipeline the data from one process to another without intermediate stops. On the contrary, several blocking operations may exist and the presence of temporary data stores is frequent. At the same time, it is possible that some records may not pass through some operations for several reasons, either for data quality problems or possible system failures. In such cases, these data are temporary quarantined and processed via special purpose workflows, often involving human intervention.

Loading. After the application of the appropriate transformations and cleaning operations, the data are loaded to the respective fact or dimension table of the data warehouse. There are two broad categories of solutions for the loading of data: bulk loading through a DBMS-specific utility or inserting data as a sequence of rows.

Clear performance reasons strongly suggest the former solution, due to the overheads of the parsing of the insert statements, the maintenance of logs and rollback-segments (or, the risks of their deactivation in the case of failures.) A second issue has to do with the possibility of efficiently discriminating records that are to be inserted for the first time, from records that act as updates to previously loaded data. DBMS’s typically support some declarative way to deal with this problem (e.g., the MERGE command.) In addition, simple SQL commands are not sufficient since the ‘open-loop-fetch’ technique, where records are inserted one by one, is extremely slow for the vast volume of data to be loaded in the warehouse. A third performance issue that has to be taken into consideration by the admin- istration team has to do with the existence of indexes, materialized views or both, defined over the warehouse relations. Every update to these relations automatically incurs the overhead of maintaining the indexes and the materialized views.

Variations of the traditional ETL architecture

In the majority of cases, a conventional ETL process is designed and executed, as previously described, in three major phases: Extract, Transform, and Load. How- ever, for several business or technical reasons, other approaches are considered too. We will not elaborate on cases that involve only a subset of the ETL phases (e.g., extract and load), but we will briefly discuss a specific alternative that nowadays, gains some business interest: the case of Extract, Load, and Transform, ELT.

The crux behind the introduction of ELT solutions is twofold and based on both the feasibility of acquiring increasingly better hardware, often more powerful than needed for data warehousing purposes, and the increasing amounts of data to be handled. Hence, in the context of ELT, instead of first creating a snapshot of the operational data in the DSA and then performing the appropriate transformations, the goal is to create a snapshot of the operational data directly in the data warehouse environment, using quick batch-loading methods. Then, depending on the business needs, the administrator can decide what types of transformations to execute either on the way of data into the data marts for OLAP analysis or on a transaction-by- transaction basis in a data-mining algorithm.

The ELT approach seems beneficial in the presence of several conditions. It seems as a good solution when the operational database machines do not have enough power - while, at the same time, the data warehouse server is a more pow- erful machine - or when there is a slow network connection among the sources and the target warehouse. Moreover, when the population of the data warehouse is based on a single integrated version of the operational data, then having the trans- formation occur within the database might prove more effective, since it can take advantage of the sophisticated mechanisms that a standard RDBMS provide; e.g., the capability of issuing inserts, updates and deletes in parallel, as well as the execution of several algorithms for data mining, profiling, cleansing, and so on, from a SQL command line. ELT may be more beneficial than other conventional architectures due to the following reasons:

* it leverages RDBMS engine hardware for scalability and basically, it scales as long as the hardware and RDBMS engine can continue to scale;
* it keeps all data in the RDBMS all the time;
* it is parallelized according to the data set; and finally,
* disk I/O is usually optimized at the engine level for faster throughput.

Currently, most major players in the market provide ELT solutions as well; e.g., Informatica Pushdown ELT, Sunopsis ELT, Oracle Warehouse Builder, and Microsoft DTS.

Finally, an even newer trend suggests the use of ETLT systems. ETLT repre- sents an intermediate solution between ETL and ELT, allowing the designer to use the best solution for the current need. In that case, we can classify transformations in two groups. A first group of fast, highly selective, non-blocking transformations may be executed in advance, even in streaming data, before the fast loading of the data to the warehouse area. Then, part of the incoming data can be used for fast, near real-time reporting, while the rest can be manipulated later on in a subsequent phase. Such a vision guides us to the case of near real-time ETL, which will be the focus of the rest of this article.

1. **The case for Near Real Time ETL**
   1. **Motivation**

Traditionally, ETL processes have been responsible for populating the data ware- house both for the bulk load at the initiation of the warehouse and incrementally, throughout the operation of the warehouse in an off-line mode. Still, it appears that data warehouses have fallen victims of their success: users are no more satisfied with data that are one day old and press for fresh data -if possible, with instant re- porting. This kind of request is technically challenging for various reasons. First, the source systems cannot be overloaded with the extra task of propagating data towards the warehouse. Second, it is not obvious how the active propagation of data can be implemented, especially in the presence of legacy production systems. The problem becomes worse since it is rather improbable that the software config- uration of the source systems can be significantly modified to cope with the new task, due to (a) the down-time for deployment and testing, and, (b) the cost to administrate, maintain, and monitor the execution of the new environment.

The *long term vision for near real time warehousing* is to have a self-tuning architecture, where user requirements for freshness are met to the highest possi- ble degree without disturbing the administrators’ requirements for throughput and availability of their systems. Clearly, since this vision is founded over completely controversial goals, a reconciliation has to be made:

*A more pragmatic approach involves a semi-automated environment, where user requests for freshness and completeness are balanced against the workload of all the involved sub-systems of the warehouse (sources, data staging area, ware- house, data marts) and a tunable, regulated flow of data is enabled to meet resource and workload thresholds set by the administrators of the involved systems.*

In the rest, we will translate this vision into a list of more concrete technical goals. First, we start with the goals that concern the implementation of a near real time warehouse with a view to Quality-of-Service (QoS) characteristics:

1. *Maximum freshness of data*. We envision a near real time data warehousing environment able to serve the users with as fresh data as possible in the warehouse.
2. *Minimal overhead of the source systems*. For near real time warehousing to work, it is imperative to impose the minimum possible additional workload to the sources, which -at the same time-is sustainable by the sources.
3. *Guaranteed QoS for the warehouse operation*. The near real time warehouse administrator should be equipped with tools that allow him to guarantee ser-

vice levels concerning the response time to queries posed, the throughput of all the systems involved and the freshness of data offered to the users.

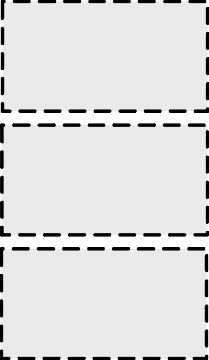
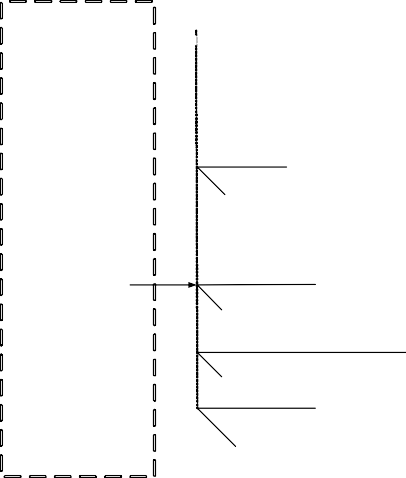
1. *Controlled environment*. Since unexpected events occur throughout the daily operation of the near real time warehouse, its administrator should be equipped with the potential to respond to these events and tune the flow of data from the sources towards the warehouse with minimal effort. Any kind of opti- mization, prediction and automatic support of the administrator in this task is highly valuable.

Practical considerations involve the following goals, too:

1. *Scalability in terms of sources involved, queries posed by the end users and volumes of data to be processed*. Naturally, it is expected that over time, requests for more fresh data from new sources, more available data, and a more energetic user community will have to be serviced by the warehouse. In this case, before other measures are taken (e.g., exploiting Moore’s law with new hardware), the degradation of performance should be smooth.
2. *Stable interface at the warehouse side*. Apart from the smooth performance degradation due to new sources, development costs should be bounded, too. It would be convenient for developers and administrators if the warehouse would export a stable interface for its refreshment to all its source sites.
3. *Smooth upgrade of the software at the sources*. We envision a transition to a system configuration where the modification of the software configuration at the source side is minimal.
   1. **General architecture**

We envision the general architecture of a near real time data warehouse consisting of the following elements: (a) Data Sources hosting the data production systems that populate the data warehouse, (b) an intermediate Data Processing Area (DPA) where the cleaning and transformation of the data takes place and (c) the Data Warehouse (DW). The architecture is illustrated in Figure 1.

Each source can be assumed to comprise a data store (legacy or conventional) and an operational data management system (e.g., an application or a DBMS, re- spectively.) Changes that take place at the source side have first to be identified as relevant to the ETL process and subsequently propagated towards the warehouse, which typically resides in a different host computer. For reasons that pertain to the



*DPFlowR*

***Data Processing Area***

*WFlowR*

*Extractor SFlowR*

*User workload*

*S1*

*Reservoir*

***Source 1***

***L***

*DIM1*

*DM1*

*Extractor SFl*

*Reservoir*

*S2*

*Reservoir*

***Source 2***

***L***

*Reservoir*

*Fact 3*

*DM2*

*Extractor SFlowR*

***L***

*S3*

*Fact2*

*Reservoir*

***Source 3***

***L***

*Fact 1*

*INDX2*

*Reservoir*

***Data Warehouse***

*owR*

*Reservoir*

*Periodic or Push -based Synchronization*

*Periodic or Pull -based Synchronization*

Figure 1: Architecture of near real time data warehouse

QoS characteristics of the near real time warehouse and will be explained later, we envision that each source hosts a *Source Flow Regulator* (SFlowR) module that is responsible for the identification of relevant changes and propagates them towards the warehouse at periodic or convenient intervals, depending on the policy chosen by the administrators. As already mentioned, this period is significantly higher that the one used in the current state-of-practice and has to be carefully calculated on the basis of the source system’s characteristics and the user requests for freshness. A *Data Processing Flow Regulator* (DPFlowR) module is responsible of de- ciding which source is ready to transmit data. Once the records have left a certain source, an ETL workflow receives them at the intermediate data processing area. The primary role of the ETL workflow is to cleanse and transform the data in the format of the data warehouse. In principle, though, apart from these necessary cleansings and transformations, the role of the data processing area is versatile: (a) it relieves the source from having to perform these tasks, (b) it acts as the regulator for the data warehouse, too (in case the warehouse cannot handle the online traffic generated by the source) and (c) it can perform various tasks such as checkpointing, summary preparation, and quality of service management. However, it is expected that a certain amount of incoming records may temporarily resort to appropriate *Reservoir* modules, so that the DPA can meet the throughput for all the workflows

that are hosted there.

Once all ETL processing is over, data are ready to be loaded at the warehouse.

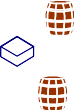
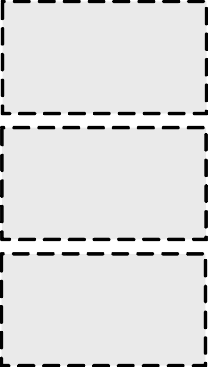
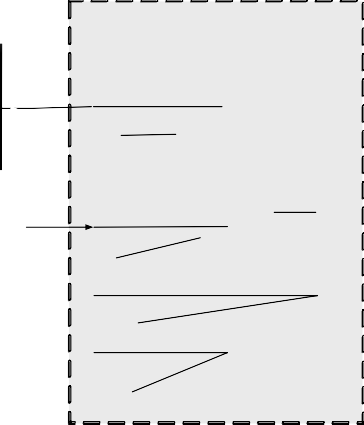
A Warehouse Flow Regulator (WFlowR) orchestrates the propagation of data from the DPA to the warehouse based on the current workload from the part of the end- users posing queries and the QoS “contracts” for data freshness, ETL throughput and query response time. Clearly, this is a load-balancing task, which we envision to be implemented over a tunable QoS software architecture.

The Data Warehouse per se, is a quite complicated data repository. There are different categories of constructs in the warehouse, which we broadly classify as follows: (a) fact tables, containing the records of real-world events or facts, that the users are mainly interested in, (b) dimension tables, which contain reference records with information that explains the different aspects of the facts, (c) indexes of various kinds (mainly, B+-trees and bitmap indexes) which are used to speed-up query processing and (d) materialized views, which contain aggregated information that is eventually presented to the users. In the rest of our deliberations, the term ‘materialized view’ will be used as a useful abstraction that allows us to abstract al kinds of summaries that are computed once, stored for the users to retrieve and query and regularly updated to reflect the current status of a one or more fact tables (e.g., data marts, reports, web pages, and any possible materialized views per se.)

Each of these constructs has a clear role in a traditional ETL setting; in the case of near real time ETL, these constructs are possibly accompanied by auxiliary structures that alleviate the burden of refreshing them very frequently.

In an ideal world, all the involved systems (sources, data processing area and warehouse) would be able to process all the data within the given time windows. Clearly, this cannot be the case in practical situation, due to many possible reasons, like the high rate of user queries, the high rate of updates, the high cost of certain parts of the transformation and cleaning stage, or even the failure of a part of the overall architecture at runtime. Practically this results to the necessity of reserving parts of the propagated data for later processing. In other words, a simple selection mechanism in the flow regulators needs to decide which data will be processed by the ETL workflow in near real time and which parts will be reserved in main memory, staged at the hard disk or simply shed in order to be processed later, during an idle period of the warehouse. Similarly, a failure in the near real time processing area or the warehouse can lead to data not propagated to the warehouse on time.

These practical considerations lead to the necessity of a compensation scheme that operates in an off-line mode (much like the traditional ETL mechanisms) and completes the missing parts of data in the warehouse and the materialized views (Figure 2.)



U

*S1*

*Reservoir*

***Source 1***

***L***

*Reservoir*

***L***

*Reservoir*

*DIM*

*1*

*DM1*

***L***

U

*S2*

*Reservoir*

***L***

***Source 2***

***L***

*Reservoir*

***L*** *Fact3*

*Reservoir*

U

***L***

*S3*

***L***

*DM2*

*Fact 2*

*Reservoir*

*Reservoir*

***Source 3***

***L***

***L***

*INDX2*

*Reservoir*

***L*** *Fact1*

***Data Processing Area***

*Reservoir*

***Data Warehouse***

Figure 2: Compensating actions for a near real time data warehouse

* 1. **Industrial approaches**

Alternative architectures for near real time ETL has been suggested in the industrial literature. In this section, we briefly describe the alternative approaches and we pinpoint their advantages and disadvantages. For further details, we refer to [18] for an excellent review of such approaches.

* + 1. **(Near) Real time partition**

The case of near real time ETL is discussed in an industrial book [18]. Although the description is informal and very abstract, the notion of real time partitions (RTP) seems that offer a solution to the problem of near real time ETL; however, for getting better performance several assumptions are considered too.

The key idea is to maintain two versions of the star schema representing a data warehouse. (One can imagine a virtual star schema defined by appropriate union view(-s) on top of the two star schemas.) One version should be static and the other real time, in terms of their population. Hence, using that approach, for each fact table of the data warehouse, a separate real time fact table is considered with the same characteristics as the static one - e.g., same grain and dimensionality. The real time fact table should contain only today’s data that are not yet loaded to the static fact table. The static fact table is populated using conventional nightly batch loads. The main difference from the conventional method is that the real time fact table periodically populates (in short periods of time) the static fact table before being emptied.

The real time fact table should have significant performance gains, thus, the

indexing in that table is minimized. In doing so, both the loading effort and the query response times are greatly benefited. However, the main advantages of this idea stem from the assumption that the whole real time fact table should fit in main memory for further fast processing. Although it may be possible to cache the fact table in memory in some cases, given that it contains data of only one day, still, this is a very ambitious assumption.

* + 1. **(Near) Real time ETL approaches**

As usual, different alternative approaches have been proposed in the market to handle the need for freshness in a data warehouse. In what follows, we briefly mention the most prominent approaches using terminology adapted from [18] and identify their limitations.

*Enterprise Application Integration, EAI*. These approaches have the ability to link transactions across multiple systems through existing applications by using software and computer systems architectural principles to integrate a set of enter- prise computer applications. An EAI system is a push system, not appropriate for batch transformations, whose functionality entails a set of adapter and broker components that move business transactions - in the form of messages - across the various systems in the integration network. An adapter creates and executes the messages, while a broker routes messages, based on publications and subscription rules.

The main benefit from an EAI system is fast extraction of relevant data that must be pushed towards the data warehouse. In general, an EAI solution offers great real time information access among systems, streamlines business processes, helps raise organizational efficiency, and maintains information integrity across multiple systems. Usually, it is considered as a good solution for applications de- manding low latency reporting and bidirectional synchronization of dimensional data between the operational sources and the data warehouse. However, as nothing comes without a cost, they constitute extremely complex software tools, with pro- hibitively high development costs, especially for small and mid-sized businesses. Also, EAI implementations are time consuming, and need a lot of resources. Of- ten, many EAI projects usually start off as point-to-point efforts, but very soon they become unmanageable as the number of applications increase.

*Fast transformations via Capture - Transform - Flow (CTF) processes*. This solution resembles a traditional ETL process too. CTF approaches simplify the real time transportation of data across different heterogeneous databases. CTF solutions move operational data from the sources, apply light-weight transformations, and then, stage the data in a staging area. After that, more complex transformations are applied (triggered by the insertions of data in the staging area) by microbatch ETL

and the data are moved to a real time partition and from there, to static data stores in the data warehouse. CTF is a good choice for near real time reporting, with light integration needs and for those cases where core operations may share periods of low activity and due to that, they allow the realization of data synchronization with a minimal impact to the system.

*Fast loading via microbatch ETL*. This approach uses the idea of real time partitioning (described in section 3.3.1) and resembles traditional ETL processes, as the whole process is executed in batches. The substantial difference is that the frequency of batches is increased, and sometimes it gets as frequent as hourly. Several methods can be used for the extraction of data - e.g., timestamps, ETL log tables, DBMS scrapers, network sniffers, and so on. After their extraction the data are propagated to the real time partition in small batches and this process continuously runs. When the system is idle or once a day, the real time partitions populate the static parts of the data warehouse. The microbatch ETL approach is a simple approach for real-time ETL and it is appropriate for moderate volumes of data and for data warehouse systems tolerant of hourly latency. The main message it conveys, though, is mainly that dealing with new data on a record-by-record basis is not too practical and the realistic solution resolves to finding the right granule for the batch of records that must be processed each time.

*On-demand reporting via Enterprise Information Integration (EII)*. EII is a technique for on-demand reporting. The user collects the data he needs on-demand via a virtual integration system that dispatches the appropriate queries to the un- derlying data provider systems and integrates the results. EII approaches use data abstraction methods to provide a single interface for viewing all the data within an organization, and a single set of structures and naming conventions to represent this data. In other words, EII applications represent a large set of heterogenous data sources as a single homogenous data source. Specifically, they offer a virtual real time data warehouse as a logical view of the current status in the OLTP systems. This virtual warehouse is delivered on-the-fly through inline transformations and it is appropriate for analysis purposes. It generates a series of (SQL) queries at the time requested, and then it applies all specified transformations to the resulting data and presents the result to the end user. EII applications are useful for near-zero la- tency in real time reporting, but mostly for systems and databases containing little or no historical data.

* 1. **The infrastructure of near real time ETL**

The software architecture described in section 3.2 is constructed in such a way that the goals of section 3.1 are achieved with QoS guarantees. In this subsection, we discuss possible alternatives for the infrastructure that hosts this software architec-

S

R

S

t1

A

B

t2

...

tn

Original state

R

t3

...

tn

A B

t2 t1

(b) pipelining

R

S

A

B

...

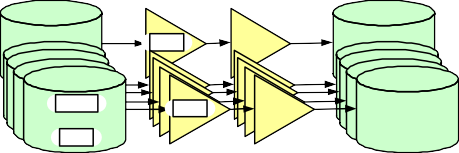
t1

tn

(a) Sequential execution

t3

t2



R1

S1

A1

t1

B1

Rk

Sk

t2k

...

Ak

tk

Bk

tn

(c) partitioning

Figure 3: (a) Sequential, (b) pipelining, and (c) partitioning execution of ETL processes [37]

ture and facilitates the near real time refreshment of the data warehouse.

A traditional approach towards the topology of the near real time warehouse would structure it as a linear combination of tiers - in fact, as a 2-tier or 3-tier configuration. Apart from the traditional architectural configurations, it is quite natural for an endeavor of the magnitude of a near real time warehouse to opt for configurations where the architecture exploits some forms of parallelism. In this subsection, we discuss how different parallelism techniques affect the execution of ETL processes and suggest improvements in the presence of certain requirements. In general, there exist two broad categories of parallel processing with respect to the flow and volume of data: *pipelining* and *partitioning*.

In Figure 3, the execution of an abstract ETL process is pictorially depicted. In Figure 3(a), the execution is performed sequentially. In this case, only one instance of the ETL process exists. Figures 3(b) and 3(c) show the parallel execution of the process in a pipelining and a partitioning fashion, respectively. In the latter case, larger volumes of data may be handled efficiently by more than one instance of the ETL process; in fact, there are as many instances as the partitions used.

**Pipelining methods**. The execution of an ETL process can be coarsely di- vided in three sub-processes: extraction, transformation, and loading. In pipeline parallelism, the various activities of these three sub-processes are operating simul- taneously in a system with more than one processor. While the ETL process lasts, the extraction module reads data from the sources and keeps feeding a pipeline with the data it had already read. In the meantime, the transformation module runs in another processor and it is continuously sending data to another pipeline. Similarly, the loading module, which runs in a third processor, is writing data to the target recordset. This scenario performs well for ETL processes that handle a relative small volume of data.

**Partitioning methods**. For large volumes of data, a different parallelism policy should be devised: the partitioning of the dataset into smaller sets. The idea is to use different instances of the ETL process for handling each partition of data. In other words, the same activity of an ETL process would run simultaneously by several processors, each processing a different partition of data. At the end of the process, the data partitions should be merged and loaded to the target recordset(s). For partitioning, many implementations have been proposed with the common goal to provide equal size partitions to facilitate the load of data to a single target. The most frequently used methods are the following. (Here, we use a terminology adopted from the DataStage tool [2], but the categorization is typical of a broader group of commercial ETL tools.)

*Round robin partitioning*. The records are distributed among the different processing nodes in a round robin fashion: the first record goes to the first node, the second record to the second node, and so forth. This method is appropriate for resizing uneven partitions of an input data set and it is often used as the default method for the initial partitioning.

•

*Random partitioning*. The records are randomly distributed across all pro- cessing nodes. This method can be used for resizing partitions, as well, and it can guarantee that each processing unit handles (near) equal-sized parti- tions. Moreover, this method produces results that are similar to the ones of the round robin method, but with a higher overhead than the latter due to ex- tra processing required for the estimation of a random value for each record; this value is used as a criterion for the partitioning.

•

*Hash by field partitioning*. Tuples with the same values for all hash key at- tributes are assigned to the same processor. Hence, related tuples are placed in the same partition. This property may be a prerequisite for a certain ac- tivity (e.g., duplicate elimination.) A similar method that requires simpler computation is the modulus partitioning, which is based on a key column modulo the number of partitions. Range partitioning is another method that places related tuples (having their keys within a specified range) in the same partition. This can be useful for preparing a dataset for a total sort.

•

*Follow-the-database partitioning*. This method suggests to partition data in the same way a DBMS would partition it. Therefore, the tuples processed by the ETL process and the respective tuples of a database table would be handled by the same parallel operator (this may significantly reduce the I/O cost.) Such a method is useful for update operations, and works very well when the warehouse is partitioned too.

•

Two additional methods are frequently used in ETL processes to facilitate the partitioning. Although they do not intend the direct partitioning of the input data, still they are useful in an environment consisting of subsequent activities. The goal of both methods is to efficiently pass partitioned data through a sequence of ETL activities.

*Same partitioning*. This method does not perform a repartitioning, but it takes as inputs the partition outputs of the preceding stage. Essentially, it does not allow redistribution of data, which remains in the same processor, but, it is useful for passing partitioned data between activities.

•

*Entire partitioning*. This method is appropriate when a parallel execution is desired and at the same time, all instances of an activity in all the processors should have access to the complete dataset if needed. Example application is the creation of lookup tables.

•

The reverse of the partitioning operation is to gather all the data together again. Typical approaches include: round robin, ordered, and sorted merge gathering. In general, this procedure is non-deterministic. Nevertheless, if order matters, a sorted merge should be favored.

**Combination**. In practice, a combination of pipelining and partitioning can be used to achieve maximum performance. Hence, while an activity is processing partitions of data and feeding pipelines, a subsequent activity may start operating on a certain partition before the previous activity had finished.

Several commercial ETL tools (e.g., DataStage) provide the functionality of repartitioning data between the activities of the process. This can be useful in many cases, such as when alteration of data groups is needed; e.g., in the case where data are grouped by month and a new grouping per state is needed.