ABSTRACT

Extract-Transform-Load (ETL) handles large amounts of data and manages workload through dataflows. ETL dataflows are widely regarded as complex and expensive operations in terms of time and system resources. In order to minimize the time and the resources required by ETL dataflows, this paper presents an optimization framework using partitioning and parallelization. The framework first partitions an ETL dataflow into multiple execution trees according to the characteristics of ETL constructs, then within an execution tree pipelined parallelism and shared cache are used to optimize the partitioned dataflow. Furthermore, multi-threading is used in component-based optimization. The experimental results show that the proposed framework can achieve 4.7 times faster than the ordinary ETL dataflows (without using the proposed partitioning and optimization methods), and is comparable to the similar ETL tools.

Categories and Subject Descriptors

H.2.7 [Database Administration]: Data warehouse and repository; H.2.m [Miscellaneous]

Keywords

Dataflow Partitioning; Optimization; Shared Cache; Execution Tree

1. INTRODUCTION

In data warehousing, ETL technology is a collection of tools responsible for extracting, cleaning, customizing, reformating, integration and loading data from different data sources into a central data warehouse. Dataflow is a term used in computing architectures where a number of computing units are organized logically in order to do the required computations according to user-defined rules [1]. The concept of dataflow is introduced in the ETL process to define the data movement and transformation logic from data sources to a data warehouse. For example, Figure 1 shows a typical ETL dataflow in data warehousing. The dataflow consists of four types of the components: 1) data sources, e.g., operational databases, text files and others; 2) data processing activities; 3) data Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SAC’15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695846.

Figure 1: Dataflow components of ETL

warehouse; and 4) the dependency of the activities, denoted as a dashed arrow.

Today, with the ever-growing volume of data, data warehousing is facing an increasing pressure to enhance its data processing capabilities for the evidence that ETL jobs of many enterprises may take hours or even days to complete. The delay might lead to an inaccurate business decision-making. Business intelligence (BI) developers therefore tend to improve their data warehousing system, e.g., optimize their ETL dataflow. Optimization is one of the most commonly used techniques for a legacy system, and the typical optimization techniques include dataflow re-designing, caching, parallelizing, changing job schedule policies and more. With an optimized ETL dataflow, the changes to the data in the data sources can be updated to the data warehouse faster, and users can make more accurate business decisions using the freshest data. Due to its importance, ETL dataflow optimization has received a growing attention in the past few years, e.g., [2–7]. As presented in these papers, optimizing an ETL dataflow is a non-trivial task. For example, to find an optimal dataflow design in terms of the cost of the time, the search space could grow exponentially [4]. Most of the existing ETL tools can only provide a limited capability for the optimization (such as re-ordering the activities in ETL dataflow, or using parallelism), however they do not provide more advanced features.

In an ETL dataflow, data is transferred and processed through a number of connected ETL components. In order to explain the ETL dataflow, we consider a simple scenario consisting of two components (see A and B in Figure 3). The component A processes the data and saves the results into an output cache. The results from the output cache are then transferred to an input cache of the component B using copy. Since the copying operation involves data movement and synchronization in main memory physically, frequent copying is expensive for a dataflow. Furthermore, the thread of the activity in the component B might be in an inactive state during the copying process, which may waste some CPU resources. In this paper, we propose a partitioning framework that can partition an ETL dataflow vertically and horizontally according to the component’s characteristics. Based on the partitioning, users may apply different optimization techniques to the resulting partitions.
with different levels of granularity. That is, in a dataflow partitioned by the vertical partitioning, a shared cache is used to transfer data between two connected components, thus, no copying is needed. The framework, then, does a horizontal partitioning on a subset of the vertically partitioned dataflow, and applies pipeline parallelization to process horizontal partitioned data splits. Finally, multi-threading is used to process the data within a component. We choose pipelining and multi-threading as our optimizations is for the reason that they both are the effective optimization techniques adopted by ETL many tools. Therefore, using the proposed dataflow partitioning framework, as well as the optimization techniques, we could reduce I/O, and maximize CPU usage for an ETL program.

The proposed framework is applicable to the traditional ETL dataflows, which run on a single server regularly, such as hourly, daily or weekly. In summary, in this paper, we make the following contributions. First, we classify ETL components into different categories based on the characteristics of processing data, including row-synchronized component, semi-block component, and block component. Second, we propose the partitioning algorithm to divide a dataflow based on different component categories. The resulting partitions (at subset-level or component-level within a subset) provide the foundation for users to apply optimization techniques. Third, we use the examples to illustrate how to apply the optimization technologies to the partitioned dataflows with different granularities. Fourth, we propose using shared caches within a partition, which reduces memory footprint, and I/O usage. We also present the algorithm for estimating the optimal degree of pipeline parallelization. Last, we show the empirical evidence that verifies the effectiveness of using the proposed optimization methods.

The paper is organized as follows. Section 2 describes the problems and the framework. Section 3 describes cache copying scenarios and categorizes ETL components. Section 4 presents the dataflow partitioning methods, and the optimizations applied to the partitioned dataflows. Section 5 presents the implementation and the evaluation. Section 6 discusses related work. Section 7 concludes the paper and presents the future work.

2. PROBLEMS AND THE FRAMEWORK

**Definition 1 (ETL Dataflow).** An ETL dataflow is formalized as a directed acyclic graph (DAG) \([G, V, E]\), where \(V = A \cup R\), and \(E\) represents the set of logical transitions from an activity to the other activities.

In a dataflow, the row sets \(R\) go through all the components, and are processed by the activities on each of the components, i.e., does data transformation. In the dataflow, a cache is used to hold the data temporarily between two continuous neighbor components. In the entire dataflow, the caches might consume large amount of memory due to holding the intermediate data. Moreover, when the data in a cache is passed to a downstream component, the copying operation can incur additional I/O consumption. We remedy this first by doing vertical partitioning on the dataflows, then use a single shared cache to transfer data between the components within the same partition. Then, pipelining and multi-threading are applied to the partitions for the optimizations. In the next two sections, we will classify the types of components, which are the foundation to our partitioning, then we discuss how to apply the optimization techniques.

We now present the system architecture in Figure 2, which has the following five functional modules:

- **Dataflow designer**: The module is used to design an ETL dataflow with the toolkits offered by the system. The toolkits include the utilities for designing work such as update, edit, search and replace, etc., and the components for building a dataflow including different types of data sources, targets, and the components for different ETL transformation operators. The dataflow designer provides a graphic user interface in which programmers can conveniently implement a dataflow according to user requirements.

- **Dataflow partitioner**: The module is responsible for partitioning a dataflow based on the types of components. The optimization techniques are applied in each partition to improve the execution concurrency of the dataflow tasks.

- **Dataflow task planner**: The module is responsible for planning dataflow tasks automatically. When a dataflow is partitioned, the job for dataflow is thus generated into multiple tasks, and task planner will plan the execution order according to the dependency of the generated tasks.

- **Dataflow component manager**: The module is responsible for data processing and data source component management. It is in charge of component life cycle management and component specification management.

- **Metadata store**: The store is responsible for the metadata management. The metadata includes the schema information of data sources and data processing components, the specifications of ETL dataflows and the information of job and task planning. Metadata can be imported from or exported to XML files.

3. COMPONENT CLASSIFICATION

The traditional data transferring approach between two components is shown in Figure 3, i.e., from A to B, where a separate cache is used for the output and input. The data transferring is through Copy, which requires some extra memory, I/O and CPU resources. However, in some cases we could optimize this by re-using a single cache, which is called shared cache (see Figure 4). Using a separate cache or a shared cache is determined by the dataflow partitioning. Before discussing the partitioning, we will classify ETL components according to their characteristics in the following. We identify three types of components:

- **Row-synchronized Components**: The component performs row-based data processing, such as filter, lookup, splitter, data format converter, etc, which process rows one after another. In this prototype, we use lookup to replace the join, thus join is also classified into this category. In a row-synchronized component, the shared caching scheme is applied to optimize the component activities within a single thread space (see Figure 4).
• **Block Components**: The component receives rows from a single upstream component. Data processing cannot be started until all the rows are received, e.g., the component B in Figure 3. The output of component A is added into the input cache by copying operation. The components in this category mainly are the aggregation operators, such as avg, sum, min and max, etc. Each of the activities runs its own thread. A block component has to accumulate all the rows before processing, thus, it is the least efficient.

• **Semi-block Components**: The component receives the rows from multiple upstream components. Somewhat similar to the previous component, data processing cannot be started until all the rows that satisfy a certain condition have been received, e.g., the component C in Figure 5. The rows from its upstream components are added into the input cache by copying operation. The semi-block components consist of union, and the similar.

In this prototype system, we have classified most of the common used ETL components, and save as the metadata in the metadata store. However, for the flexibility purpose, users could overwrite the pre-defined settings through configuring a XML file, and also could add the a new component into a category.

### 4. PARTITIONING AND OPTIMIZATION

In order to optimize, we first perform the vertical partitioning by dividing a dataflow into multiple subsets, or sub-dataflows, then we use parallelization technology in each of the subsets. When we do the partitioning, the partitioning granularity should not be either too coarse or too fine. If the granularity is too coarse, the execution concurrency of the sub-dataflows will become very low, which is not able to take full advantage of the computing resources, if running on a powerful machine. On the contrary, if the granularity is too fine, it will lead to a lot of concurrent threads, and cause overall performance loss when competing for computing resources. In addition, data synchronization between two neighboring sub-dataflows could also lead to some overhead, and so does thread switching when multi-threading is used. Based on these considerations, we therefore propose the 3-level partitioning method: first vertically partition a dataflow into multiple sub-dataflows, called execution trees (the coarse-level), then horizontally partition the input of an execution tree (medium-level), and finally do a further partitioning in the components that are staggering in the execution tree (fine-level). We call this component a staggering component. For each partition with a different level of granularity, we choose an appropriate parallelization method to optimize it (if applicable).

#### 4.1 Execution Tree

**Definition 2 (Execution Tree).** An execution tree is a directed acyclic graph, defined as $T(V', E')$. It is a subgraph of the dataflow graph $G(V, E)$, where $V'$ is a nonempty subset of $V$ and $E'$ is a subset of $E$. In $T$, the vertex with indegree equal to zero is the root, and the vertex with zero outdegree is the leaf.

Suppose that we have done the coarse-level partitioning for the ETL dataflow graph $G$, which results in $n$ execution trees, $T_1, T_2, ..., T_n$. The generated execution trees also form a graph, defined as $G_r(V_r, E_r)$, where $V_r = \{T_1, T_2, ..., T_n\}$, and $E_r$ is the set of the edges of any two connected execution trees. $G_r$ is a directed acyclic graph.

We now illustrate the coarse-level partitioning using the example shown in Figure 6. The example depicts the ETL dataflow of loading data from a single data source into two target tables through a number of ETL transformation operators. As discussed earlier (Section 3) that block and semi-block components require their own caches to accumulate the data before processing, this dataflow can be partitioned into four execution trees, $T_1, T_2, T_3$, and $T_4$, each of which starts from the root, such as a data source, semi-block or block component. In an execution tree, the output of the root is horizontally partitioned into multiple splits, each of which is held in a shared cache, and passed to the downstream row-synchronized components. For any two connected execution trees, a new cache is needed, and the data are transferred to the new cache by COPY, e.g., $T_1 \rightarrow T_2$, $T_1 \rightarrow T_3$, and $T_3 \rightarrow T_4$.

Algorithm 1 shows how to partition a dataflow. It takes a dataflow graph $G$, as the input, and returns an execution tree graph $G_r$, as the output. The algorithm does the partitioning using depth-first searching (DFS), which starts from the data source components, i.e., the vertexes whose in-degree is equal to zero in $G$ (see line 6–9). An execution tree is created taking a data source component as the root (see line 7). The algorithm then does the depth-first searching the downstream components, and adds them as the children of the tree. If a component is neither block type nor semi-block type,
it will be regarded as a child added to the tree (see line 15); Otherwise, the searching for downstream components will finish. A new execution tree is created, which takes this block or semi-block component as the root (see line 17), and the tree is added into the execution tree graph, \( G_e \) (see line 18–19). The graph, \( G_e \), is returned in the end, which contains one or several partitioned execution trees (the vertexes). For each of the execution trees, we optimize it using a shared cache. All the threads for the activities use a single cache, thus data copying is not needed any more.

\[ \text{Theorem 1. For a } \frac{c}{m} \text{ sized input of an execution tree, if the degree of pipeline parallelization is set to } m, \text{ the cost is minimal.} \]

---

**Algorithm 1 Partition an ETL dataflow graph, \( G \)**

1: function \( \text{Partition}(G) \)
2: \( G_e \leftarrow \text{InitEmptyGraph}() \)
3: for all \( v \in V(G) \) do
4: \( \text{visited}[v] \leftarrow 0 \)
5: for all \( v \in V(G) \) do
6: if \( \text{InDegree}(v) = 0 \) and \( \text{visited}[v] = 0 \) then
7: \( T \leftarrow \text{CreateExecutionTree}(v) \)
8: \( V(G_e) \text{. Add}(T) \)
9: \( \text{DFS}(G, G_e, v, T) \)
10: return \( G_e \)
11: function \( \text{DFS}(G, G_e, v, T) \)
12: \( \text{visited}[v] \leftarrow 1 \)
13: for all \( u \) adjacent to \( v \) do
14: if \( \text{visited}[u] = 0 \) then
15: \( v \text{. children} \text{. Add}(u) \)
16: else
17: \( T' \leftarrow \text{CreateExecutionTree}(u) \)
18: \( V(G_e) \text{. Add}(T') \)
19: \( E(G_e) \text{. Add}(T \rightarrow T') \)
20: \( \text{visited}[u] \leftarrow 1 \)
21: function \( \text{DFS}(G, G_e, u, T) \)

---

**4.2 Inside Execution Tree Parallelization**

**Definition 3 (Inside Execution Tree Parallelization).**

The output of the root of an execution tree is horizontally partitioned, and pipeline parallelization is used within the execution tree to process the partitions.

We now give the formalization. The ETL process of an execution tree is a 3-tuple \((I, F, O)\) where \(I\) represents the input, \(F\) represents the ETL activities of the components, and \(O\) represents the output of the execution tree (see Figure 7). We use the tuple, \((A_0, A_1, ..., A_n)\), to denote the activities of the components in an execution tree.

The function \(f_o : I \rightarrow \Sigma\) is used to denote the activity of the first component, \(A_0\), where \(f_o\) represents the input of the execution tree, and \(\Sigma\) represents the output of \(A_0\). We horizontally partition \(\Sigma\) into \(m\) even splits (the value of \(m\) is configurable), i.e., \(\Sigma = \{(I_1^{(1)}, I_1^{(2)}, ..., I_1^{(n)})\}, \) where \(I_1^{(i)}, i = 1...m\) disjoins the subsets of \(\Sigma\). Here, we use, \(I_1^{(j)}\), to denote the \(j\)th horizontal split as the input for the \(j\)th activity, and use \(|I_1^{(j)}|\) to denote the number of rows for the split. The final output of an execution tree is \(O = \{(I_1^{(n+1)}, I_2^{(n+1)}, ..., I_n^{(n+1)})\}\) where \(|I_i^{(n+1)}| > 0, i = 1, m\).

**Figure 7: Pipeline parallelization in an execution tree**

An activity, \(A_i\), can be expressed as a function, i.e., \(f_j : I_1^{(j)} \rightarrow I_1^{(j+1)}\) where \(i = 1, m\) and \(j = 1, n\). Thus, the activities \((A_0, A_1, ..., A_n)\) can be formalized as \(F(x) = f_n(f_{n-1}(...f_1(x)))\), where \(x \in \Sigma\), which is made of the activity functions recursively.

**Figure 8: Pipeline parallelization**

Figure 7 shows pipeline parallelization within an execution tree, with \(n\) activity threads running in parallel, and Algorithm 2 describes its implementation. A shared cache is created to hold each of the splits. A shared cache is a collection variable containing a set of rows, as well as the schema, which is passed through all the downstream activities to be processed. In case new attributes are added to the rows, or the existing attributes are merged or removed from the rows, the changes are reflected to the data in the shared cache variable. In the pipeline parallelization, when a component has finished processing a shared cache, it passes it to the next activity thread immediately. The degree of the pipeline parallelization is \(m\), and we set the value of \(m\) \(\leq m\) in order to limit the memory usage. We exploit a fix-sized blocking queue, and a housekeeping thread to maintain the maximal number of parallel threads, and the memory usage (see line 14–15). A shared cache is created to hold each of the splits from the first component (of block or semi-block type, see line 17–18). The pipeline consumer thread is created to process the shared cache (see line 18–19). The thread will be blocked if there is no space available in the queue. The shared cache is processed sequentially through all the activity threads (see line 1–11). For each component, it has to process \(m\) shared caches in a sequential order. Therefore, if a component is processing a shared cache, and a pipeline consumer thread passes it a new shared cache, the pipeline consumer thread will be halted by \(\text{wait}\) (see line 7). It will not be woken up until the component has finished its processing (by \(\text{notifyAll}\), see line 11).

When a shared cache is processed by the component, \(A_i\), the total processing time, denoted by \(t_i\), consists of the time for processing all rows, and miscellaneous time, denoted by \(t_0\). The miscellaneous time is consumed by any other necessary actions, such as create and clean temporary tables, and hand over a shared cache to other components, etc., typically, \((t_i - t_0) \gg t_0\). We call \((t_i - t_0)\) as net time for processing the rows in activity \(A_i\). Therefore, when a shared cache goes through all the \(n\) sequential activities, the total time is \(\sum_{i=1}^{n} t_i\) (see Figure 8). For simplicity, we now consider when the \(m\) splits can fit into the memory, and use the same number of pipelines to process the splits, i.e., \(m = m'\). Suppose that the activity \(A_j\) has the maximal time, i.e., \(t_j = \max(t_1, t_2, ..., t_j, ..., t_n, t_0)\), according to the pipeline workflow algorithm in [2], the total time is \(T_p = \sum_{i=1}^{n} t_i + mt_j + \sum_{i=j+1}^{n} t_i\).

If we process all the shared caches in a sequential order, the total time, denoted by \(T_n\), will become \(m \sum_{i=1}^{n} t_i\). If we exclude the miscellaneous time of each activity, the total net time can be regarded as a constant value, i.e., \(c = m \sum_{i=1}^{n} t_i - t_0\) since the number of rows processed in each activity is a constant. Therefore, \(T_p\) can be represented as:

\[ T_p = \frac{c}{m} + (m - 1)t_j + nt_0 \]
Algorithm 2 Pipeline parallelization within an execution tree

1: class PipelineConsumerThread
2: function INIT(ActivityThreads Γ, SharedCache sc)
3: Γ′, sc′ ← Γ, sc
4: function RUN(Γ′)
5: for all a ∈ Γ′ do
6: while a.busy do
7: a.wait() ▷ Wake up when a has finished processing the
8: previous shared cache
9: a.busy ← true
10: sc′ ← a.process(sc′) ▷ Process a shared cache
11: a.busy ← false
12: a.notifyAll() ▷ Notify all the awaiting pipeline consumer threads
13: if any
14: q ← new BlockingQueue(m′) ▷ Create a blocking queue with the specified
15: new HouseKeepingThread(q).start() ▷ This thread is used to clean the
16: invalid objects in q
17: for i = 0 to m − 1 do
18: Read a split of the output of the first component into sc
19: th ← new pipelineConsumerThread(Γ′, sc)
20: q.add(th) ▷ The addition will be blocked if q is full
21: th.start()

Proof. Suppose the total number of rows processed by the staggering activity, A_j, is N (Note that N is not necessary to equal to the size of the execution tree input |Σ|). For example, the upstream component of A_j could be a filter operator for screening noisy data). The size of each horizontal split in A_j is N/m, and we assume that the time used is linear to the split size. t_j can be represented as t_j = t_0 + λN/m where λ is an coefficient of the split size. Therefore,

\[ T_p = \frac{c}{m} + (m-1)\left(t_0 + \frac{\lambda N}{m}\right) + nt_0 \]
\[ = \frac{c - \lambda N}{m} + t_0m + \lambda N + (n-1)t_0 \]
\[ \geq 2\sqrt{t_0(c - \lambda N)} + \lambda N + (n-1)t_0 \]
when \( \frac{c - \lambda N}{m} = t_0m \), i.e., \( m = \sqrt{\frac{c - \lambda N}{t_0}} \). The total time \( T_p \) has the minimal value \( 2\sqrt{t_0(c - \lambda N)} + \lambda N + (n-1)t_0 \).

From the discussion above, we can see the degree of parallelization is relevant to the size of the staggering component. But, it is obvious to know the range of the value, \( 1 \leq m \leq |\Sigma| \). If \( m = |\Sigma| \), it means that we pipeline for every single row. It has the highest degree of parallelization. However, this case will lead to frequent transferring of the shared caches between activity threads, which dominate the overall time. On the contrary, if \( m = 1 \), the ETL dataflow will degenerate to non-pipeline fashion (or sequential execution) since the input will be processed once for all through all the activities. To make an optimal optimization, we need to find staggering component, compute the degree of parallelization, then adjust all the activities. Algorithm 3 describes how to find the staggering activity, e.g., the grayed one A_j in Figure 9, and compute the optimal degree of parallelization.

Algorithm 3 Compute the optimal degree of parallelization with a given input, \( \Sigma \)

Require: The sample data set D (with \( m' \) horizontal splits), and the number of activities in an execution tree, \( n \).
1: Measure the total miscellaneous time, \( T_0 \).
2: Pipeline \( m' \) splits in an execution tree, record the time of each activity, and measure total processing time \( T_a \).
3: Find the staggering activity, \( A_j \), compute the constant \( c \leftarrow (T_a - T_0) \), and the average miscellaneous time, \( t_0 \leftarrow \frac{t_0}{m} \).
4: Run an execution tree in pipeline parallelization, and process the \( m' \) splits, and compute the coefficient \( \lambda \).
5: According the formula in theorem 1, compute the degree of parallelization, \( m' \).

Figure 10: Inside-component parallelization

4.3 Inside Component Parallelization

Definition 4 (Inside Component Parallelization). It refers to using multi-threading technology to parallelize data processing in a heavy-computation component of the dataflow.

In an execution tree, the computational load of some components is much heavier than others, and forms the bottleneck. We make the further optimization of using multi-threading to parallelize data processing in a component. Multi-threading is applied to the non-block type components, e.g., the Filter row component in T_j of Figure 6. Figure 10 shows the scenario. For a shared cache, \( C \), delivered to it, the system firstly divides the rows in \( C \) evenly into multiple splits, i.e., \( C = \{c_1, c_2, ..., c_n\} \). Then, the component spawns a number of threads to process the \( n \) splits in parallel (the value of \( n \) can be set in the system configuration file), each of which results in an output \( d_i, i = 1, \ldots, m \). Finally, the outputs \( d_1, d_2, ..., d_n \) are merged by the row order synchronizer, which maintains the row order of the output to be the same of the input (In some cases such as the activities sort-filter-merge, the input and output row order of filter should not be changed since the downstream activity is merge). The merged rows are saved into another shared cache \( C' \), and delivered to the downstream component.

5. IMPLEMENTATION AND EVALUATION

We implement this framework based on the open source ETL tool, Talend [21], also known as Talend Open Studio for Data Integration with an intuitive, graphical, drag and drop design environment. We add the extension, dataflow partitioner, to the transformation engine, and make it support the shared cache and pipeline parallelization. The degree of pipeline parallelization of execution trees can be configured in the configuration file. For the non-block
type components including the row filter, splitter, explorer, expression, lookup, etc., they are extended to support the inside-component parallelization using multi-threading. The number of threads is also configurable. If the number is not set, the system uses one as the default value, which means that the inside-component parallelization is disabled. In addition, XML is used as the metadata repository (DBMS can also be configured to use), and the repository manager is extended to support managing the partitioning information of a dataflow.

The host of the experiments is a HP ProLiant DL380 G5 Server with two Intel Xeon quad-core processors (2.4GHz), 8GB RAM and a SATA hard disk (350 GB, 3 GB/s, 16 MB Cache and 7200 RPM), running Ubuntu 12.04 LTS with 64bit Linux 2.6.32 kernel. JVM 1.6.0.29, and the option “-Xms2500m -Xmx6000m” are used to run the program. PostgreSQL 8.4 is used as the data warehouse DBMS with the settings “shared_buffers=512MB, temp_buffers=128MB, work_mem=56MB, checkpoint_segments=20” and default values for other configuration parameters.

We use the star schema benchmark (SSB) based on TPC-H [9] for the evaluation. The star schema consists of one fact table, lineorder, and four dimension tables including customer, part, supplier and date (see [9] for the detailed schema information). In the experiments, we use the fix-sized data sets for the dimension tables: customer (150,000 rows, 725M), part (24,000 rows, 323M), supplier (231,000 rows, 150M) and date (10,000 rows, 134K), but vary the size of fact data. In all tests, we first load the data sets into the data warehouse, then execute the dataflow corresponding to a SSB query, and finally write the query results into a text file. The evaluation divides into two parts: the evaluation of the proposed partitioning methods and the applied optimization techniques; and comparing with other ETL tools.

5.1 Evaluate the Framework

We first use query Q4.1 (see the script below) to evaluate the optimization framework itself.

```
SELECT d_year, c_nation, SUM(lo_revenue - lo_supplycost) AS profit
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey=c_custkey AND lo_suppkey=s_suppkey AND lo_partkey=p_partkey AND lo_orderdate=d_datekey AND c_region = 'AMERICA' AND s_region = 'AMERICA' AND (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
GROUP BY d_year, c_nation
ORDER BY d_year, c_nation
```

Figure 11 shows the corresponding dataflow, which is partitioned into three execution trees using Algorithm 1. The dataflow reads the rows from the fact table lineorder, then joined with the four dimension tables respectively by lookup. The join conditions are lo_custkey=c_custkey AND lo_suppkey=s_suppkey AND lo_partkey=p_partkey AND lo_orderdate=d_datekey AND c_region = 'AMERICA' AND s_region = 'AMERICA' AND (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2'). Component 6 filters the rows that fail to join with any of the dimensions (whose returned key values are equal to the default value, 1). Component 7 selects the necessary field values for the query by projection. Component 8 computes the value of the expression (lo_revenue - lo_supplycost) of each row. Component 9 groups the computed values by sum operator, then component 10 sorts the values in each group, and component 11 writes the final results into a text file. The partitioned dataflow comprises three execution trees, T₁, T₂ and T₃. T₁ consists of eight components, whilst T₂ and T₃ contain one and two components, respectively. T₁ has four lookups, which are the relatively expensive operators in terms of the computing resource consumptions.

We first evaluate the speedup when the pipeline parallelization is applied to T₁. We execute the dataflow when the fact table is loaded with 2, 4, and 8 GB data sets, respectively. We use this relatively small workload in order to make the job complete within a certain time limit, i.e., one hour, and this workload is in line with the load of most of the traditional ETL jobs running on a single server regularly. Figure 12 shows the experimental results. The speedup value is computed by dividing the sequential execution time (no pipeline) by the time of using pipelines. As shown, for each data set the speedup scales nearly linear when the number of the pipelines is less than eight, but the speedup decreases dramatically when more pipelines are added. It is because more threads have been generated when the degree of pipelines is increased. This causes by CPU bound (we show this in the next experiment). From Figure 12, we also see that the speedup is more significant to a bigger sized data set. The optimal number of pipelines is about eight, where the speedups are 4.7x, 3.9x and 3.7x for the fact table loaded 2, 4, and 8 GB data, respectively.

Figure 13 shows the CPU usage when the fact table is loaded 8 GB data. We vary the number of cores by booting the system with the kernel option maxcpus = n where n = 2, 4, 6 and 8. As shown, when the degree of pipelines increases, the CPU usage grows as well, due to the increased number of threads.

We now evaluate the inside-component parallelization using multi-threading. For testing purpose, we remove the index on the attribute s_nation of the supplier table, and also disable the pipeline parallelization. This leads to the lookup operation on the supplier table be the bottleneck of the whole dataflow. We execute the dataflow when the fact table is loaded with 4 GB data, and the system is configured with 2, 4, 6 and 8 cores. Fig 14 shows the speedup lines when the number of the threads used by the lookup component of supplier is scaled from 1 to 16. The speedup of each test grows at the beginning, but decreases when the number of threads is increased. The performance deterioration is also due to the computing resource competition for over threading. The results also indicate that the speedup is better if multi-core is used.

We now measure the performance improvement using the proposed optimization methods. We first test the sequential execution, i.e., no parallelization, when shared caching is used and not used, respectively. The size of fact data is scaled from 1 to 8 GB. Figure 15 shows the results. The sequential execution of using shared caches has made ~ 10% performance improvement over not using shared cache, due to the removal of data copying operation within execution trees. We now enable the pipeline parallelization and the share caching, and measure the time. The number of pipelines is set to eight, which showed the best performance in the previous experiment. The execution is 4.7x and 3.9x faster than the sequential executions without and with using shared caches, respectively.
5.2 Compared with other ETL tools

We now compare this framework (abbreviated as Part frm.) with the open source ETL tool, Kettle [8], and a commercial ETL tool E (due to the license issue, we omit its real name). The reason that we select Kettle is that its architecture is most similar to ours, also implemented in Java, and runs on Java virtual machine (JVM). We select the first query in each of the four SSB query categories (note that the queries in each category only differentiate in the query conditions), denoted by Q1, Q2, Q3 and Q4. We run the test with the fact table loaded 8 GB data set. The commercial tool E supports both component based and pipeline parallelizations. Kettle only supports component-based multi-threading. To make the comparison fair, we use the same number of threads and the same step for all the tools (eight threads for each query). We first compare ours with the two others when the multi-threading parallelization is enabled, and shared cache is used. The results are shown in Figure 16. We now compare when the pipeline parallelization is enabled in our framework and E. Since Kettle does not provide the built-in supports for pipeline parallelization, we make the workaround, i.e., for the connected row-synchronized components we horizontally split the dataflow into multiple parallel sub-flows by a splitter, then merge the results from all the sub-flows. The number of sub-flows is set to the same size of the pipelines of our framework (i.e., eight). The results are shown in Figure 17. In both of the test cases, our framework outperforms Kettle in all the four queries, and outperforms E in Q1 and Q2. The comparison with Kettle verifies the effectiveness of our proposed partitioning-based optimization by considering the high similarity of our framework to Kettle, e.g., regardless of the performance difference caused by the implementation language. Our current version uses lookup to replace join operator, which can be partitioned into the same execution tree with the other row-synchronized components. Since Q1 and Q2 both have one join operation with the smallest date dimension table. Only one execution tree is needed. Furthermore, a shared cache can be used for each parallel thread, thus, both queries show the best efficiency. While, Q3 and Q4, they have three and four join operations with larger dimension tables, respectively. Our framework uses more time than E. But, in overall our proposed solution is still comparable to E.

6. RELATED WORK

The optimization of ETL process has particular importance since ETL processes have to complete their tasks within specific time frames [10]. Some research works are found to optimize the ETL process. Simitsis et al. propose the theoretical framework [4,5] that formalizes an ETL scenario as a directed acyclic graph (DAG), and handles ETL optimization as a state-space problem. When given an ETL scenario, the proposed framework can find an equivalent logical scenario that has the best execution time by searching the state space. This formalized DAG provides the theoretical foundation to the dataflow partitioning of our work. Tziovara et al. propose the approach [7] of producing the physical scenario with a logical ETL template as the input. But, unfortunately the above research works do not have any open source implementations available to our comparison. Li and Zhan determine the critical path of ETL processes have to complete their tasks within specific time frames [10]. Behrend and Jörg use rule-based optimization for the incremental ETL flows [12]. The rule-based optimization rewrites the rules with regard to the algebraic equivalences, however the rules need to be hard-coded during initial deployment [13]. The QoX-driven approach [6] is proposed to deal with the quality and the optimization objectives of ETL design. In comparison to the works above, we use the partitioning-based method (in addition to parallelization) to optimize an ETL dataflow. The traditional “brute-force” method by re-organizing the ETL constructs can cause an explosive searching space. In contrast, our framework takes the component characteristics into account for partitioning and optimization. The optimization cost is much smaller. To the best of our knowledge, this is the first work to optimize ETL using the partitioning method according to the characteristics of the component in an ETL dataflow. A plethora of commercial ETL tools exist, such as IBM’s Data Warehouse Manager [14], Informatica’s PowerCenter [15], Microsoft’s Data Transformation Services [16] and Oracle’s Warehouse Builder [17]. The commercial tools provide advanced GUI to ease ETL design, however, most of them use the black-box optimization at the logical level, e.g., re-ordering of the activities in the ETL dataflow. Some ETL engines such as PowerCenter [15] support
“push down” optimization, which pushes the operators that can be expressed in SQL from ETL flows down to the source or target DBMS engine [5]. The remaining transformations are executed in the ETL data integration server. In contrast, our framework optimizes an ETL dataflow by partitioning, and optimize the partitions at three levels of granularity using shared caching, pipelining and multi-threading, which can take full advantage of the computing resources of the server and reduce I/O usage.

The latest trend of data warehousing is to support big data, and offer real-time/right-time capability, e.g., [3, 24]. The emergence of the cloud computing technologies, such as MapReduce, makes it feasible for ETL to process large-scale datasets. As the evidence, two open source MapReduce-based systems, Pig [18] and Hive [25], become increasingly popular in data warehousing. But, they both are designed for the generic purpose for big data analytics, with limited ETL capabilities, somewhat like DBMSs other than full-fledged ETL tools. To complement this, Liu et al. propose a scalable dimensional ETL programming framework, ETLMR [23, 26] using MapReduce, but maintains its simplicity in implementing a parallel dimensional ETL program. Besides, the framework [27] is proposed to improve the dimensional ETL capability in Hive. We believe that the cloud-based ETL technologies could also be benefit from our framework, i.e., making use of power of CPU, reduce I/O using the proposed optimization techniques, e.g., dataflow partitioning, shared caches, and the parallelization of execution trees.

7. CONCLUSION AND FUTURE WORK

To improve the efficiency of an ETL dataflow, optimization is one of commonly used techniques, however, it is a non-trivial task due to the complexity of ETL dataflows. In this paper, we have proposed a partitioning-based framework to optimize ETL dataflow. Unlike the other optimization methods, the proposed framework takes into account the characteristics of ETL components, and based on the characteristics to do the partitioning and optimizations. In this paper, we have proposed the use of shared caches of transferring data, which can not only minimize the memory footprint but also reduce I/O usage. Furthermore, we have also described how to apply the optimization techniques to the resulting partitions of different granularity. Pipelining and multi-threading both are employed as well for our optimization. In addition, we have proposed the algorithm to estimate the optimal degree of pipeline parallelization. Finally, we have evaluated the proposed framework comprehensively, and compare the similar ETL tools. The results show that the proposed framework is 4.7 times faster than the ordinary ETL dataflows (without using the proposed optimization techniques), and comparable to the similar tools.

In the future, this framework could be extended in several different directions. First, the self-adapted configuration is desirable such that the system can automatically configure the number of pipelines needed, based on the statistical data of the previous runs. Second, it is interesting to add a cost model to the system, based on the cost model the system generates the optimal execution plan for ETL dataflow. Third, we will optimize the join operator, instead of using lookup. Fourth, due to the growing interest of moving ETL to the cloud platform today, it is also interesting to improve the system so that it could be deployed in a clustering environment, whereas, the proposed optimization methods are used to take full advantage of the computing resources of each node.

8. REFERENCES