# Entity Fiber based Partitioning, no Loss Staging and Fast Loading of Log Data

Xiongpai QIN[1,2], Yueguo CHEN✉[1,2], Guodong JIN [1,2], Yang LIU [1,2], Yiming CONG [1,2], Xiaoyong DU [1,2]
1. Information School, Renmin University of China, Beijing, 100872
2. Data Engineering and Knowledge Engineering Lab of Ministry of Education (RUC), Beijing, 100872

***Abstract*—Real time analysis of fine granularity of log data can help people gain personalized insights on business. For example, real time analysis of e-commerce log data will help us learn recent changes of browsing and shopping behavior of specific customers, which enables us to provide personalized recommendations. To accomplish such analysis, log data should have been loaded quickly into data warehouse without loss. This paper proposes a no loss staging and fast loading solution for log data. Based on open sourced tools such as Kafka, HDFS, and Spark, we have designed and implemented an entity fiber based log data partitioning and staging method, as well as a parallel loading algorithm. Our scheme achieves a data staging performance of around 390,000 records/s, and a data loading performance of around 160,000 records/s.**

***Keywords—log data; entity fiber based partitioning; no loss staging; fast loading; Kafka; HDFS; Spark***

## I. INTRODUCTION TO ENTITY CENTRIC REALTIME ANALYSIS OF LOG DATA

Log data contains valuable information for decision making. Timely and efficiently analyzing of log data can bring significant business value. For example, by analyzing log data of servers and applications, we can infer the root causes of failures. By analyzing log data of e-commerce sites, we can learn recent changes in browsing and purchasing behaviors of specific customers. Based on that, e-commerce sites can provide more personalized recommendations [1].

In above application scenarios, people need to perform real-time analyzing work on log data *around some specific entities* (customers, products, servers, applications etc.). Entity centric analysis requires that the finest granularity of log data should be stored for later process. In the meantime, real-time analyzing requires that the log data should be loaded into data warehouse as fast as possible. In summary, the two challenges that we face when doing real-time entity centric analysis of log data include: (1) Detailed data should not be lost. (2) And the data should be loaded as fast as possible.

To tackle the two challenges, we propose an entity fiber based partitioning and staging method, as well as a parallel loading algorithm for log data. We have implemented our scheme based on open sourced tools such as Kafka, HDFS (Hadoop Distributed File System) and Spark. The experiment results show that our proposal achieves a good data staging and data loading performance.

The paper is organized as follows. Section 2 introduces the entity fiber based data partitioning and staging method, as well as the parallel loading algorithm for log data. Section 3 describes our experiment setting and experiment results with our analysis. Section 4 presents some related works, and the final section concludes the whole paper.

## II. NO LOSS STAGING AND FAST LOADING OF LOG DATA

### A. The Definition of Entity Fiber

A tuple of log data records some information of one event about some entities. For example, in log data of an e-commerce site, each tuple describes an event about some specific customer and some specific product. In this scenario, customer and product are entities. Customer is a primary entity, and product is a secondary entity. Our discussion will center around primary entities, however, the treatment of secondary entities is similar to the way we process primary entities.

We organized entities into clusters, which are called entity **fibers** (fiber in short). The mapping from entity to fiber can have some sematic meaning, or we can simply use some *Hash* or *Range* function to map entities to fibers. For example, in mobile communication applications, the *call detail records* could be partitioned according to calling intensity of different areas to which mobile phones are registered (it is called registration location of a mobile phone). For some areas with high frequency of calling such as down town area of some city, users of such area will be split into several fibers. For some rural area with rare calls, mobile users of several such areas could be combined into one fiber.

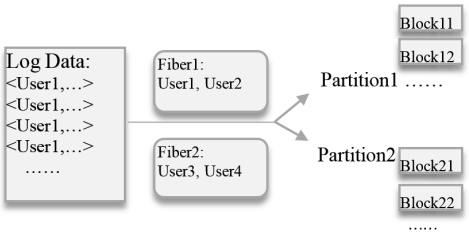


Fig. 1. Entity fibers and log data partitioning

Note: Each line of log data represent one log record.

After entities are split into entity fibers, log records are split according to entity fibers. As depicted in figure 1, user 1 and user 2 belong to fiber1, user 3 and user 4 belong to fiber2. Based on that, log records about user1 and user2 will go to the same partition - partition 1, and log records about user3 and user4 will go to another partition - partition 2. Log data of each partition is organized in blocks. For example, block 11 and block 12, … contain log records about user1 and user2 but occur at different time. In the following text, when there is no

978-1-5090-5081-9/16 $31.00 © 2016 IEEE
DOI 10.1109/PDCAT.2016.51

confusion, we also use entity fibers to refer to corresponding log partitions as well as their data blocks. Entity fiber based log data partitioning, has taken skewness of log data into account. It will mitigate the effect of receiving unbalanced amount of data by different *loaders* (described later).

In big data applications, people store the same data to several replicas (typically 3 replicas) to achieve fault tolerance. We can store primary entity based partitioning of log data in two of the replicas, and store secondary entity based partitioning of log data in the third replica. Primary entity oriented query will be routed to primary entity based partitions, while secondary entity oriented query will be routed to secondary entity based partitions.

### B. The Whole Architecure of our proposed Log Data Staging and Loading System

Figure 2 shows the whole architecture of our prototype with the shaded parts designed and implemented by ourselves. The basic flow of log data staging and loading is as follows: (1) *The log data adapter* reads log data coming from upstream data streams or log data stored in files, and hands it over to the log data partitioner. (2) *The log data partitioner* partitions the log data according to entity fibers. Log data of different fiber is written into different partitions of Kafka message queue for temporary landing. We can run more than one instance of partitioner according to throughput requirements. (3) *Log data loaders* run on Data Nodes of HDFS. They pull log data from different Kafka partitions. Each loader is responsible for pulling and loading of log data of several fibers.

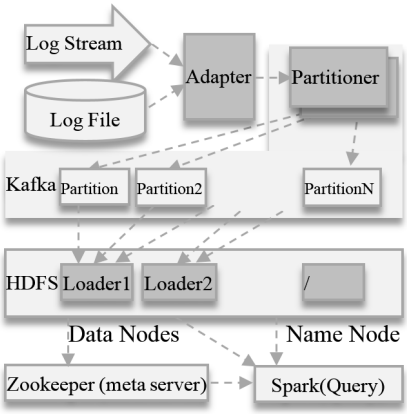


Fig. 2. No loss staging and fast loading of log data

Each loader runs in a multithread mode. Each thread pulls log data of one entity fiber. When the volume of log data of one loader reaches the threshold of one data block (256MB). The loader will organize the data in a proper format and write it into HDFS, and register some meta data in *the meta data store.*

One of meta data tables - the *Block* table records meta data about each data blocks, i.e. on which Data Node the primary replica of the block is saved, the start timestamp, end timestamp of the log data of the block, number of records of the block etc.

The goal of organizing log data of several fibers into one block, is to cope with data skewness among different fibers, and to reduce memory consumption. If we prepare different buffer for each fiber, and only when some buffer accumulates enough data for one block, we write the block into disks, then the loaders will consume much more memory when each Data Node is responsible for handling of 100s of fibers. In that case, we need to prepare 100s of buffers, before any of the buffers accumulate enough data, we cannot write any data into disks.

### C. Partitioning and Staging of Log Data

After *the log data adapter* receives log data from upstream data streams or log data stored in log files, and hands the data over to the *log data partitioner*. The *log data partitioner* dispatches the log records according to the mapping from entity to fibers, and writes log records of different entity fibers to different partitions in the Kafka message queue. The only work of the *log data partitioner* is to map and dispatch each log record, thus it can achieve a very high throughput.

Kafka is an open sourced distributed message system. It was originally developed by the LinkedIn Company, and became an Apache project later [2]. Compared to traditional message systems, Kafka has several nice properties: (1) it is a fully distributed system and it is easy to scale out to process very large volume of log data. (2) Kafka provides high throughput for both publishing and subscribing. And it supports multi-subscribers and automatically balances the consumers during failures. (3) Kafka persists messages into disks, to guarantees reliability of the message system. Data will not be lost. In our system, Kafka's task is to temporarily and reliably store log data. We have implemented a data partitioner in front of Kafka, and the partitioning will get the log data ready for later parallel loading.

By mapping entities to entity fibers, and partitioning the log data according to entity fibers, we can speed up entity centric queries. When we query log records about some specific entities, we only need to scan data blocks containing those entity fibers. In addition, queries usually contain some time range conditions, then the scanning can confine to a few data blocks. However, when people don't designate entity condition in a query, we need to scan more data blocks.

### D. Parallel Loading of Log Data and Registration of Meta Data

*Loaders*, running on Data Nodes of HDFS in parallel, load the log data into HDFS. On each Data Node of HDFS, a *loader* is responsible for loading of log data of several fibers. There is a mapping from fiber to loader. It is stored in a table in meta data store, with a name of *Mapping*. The mapping is periodically readjusted (please refer to sub section E "discussion" of this section).

In general, when the number of entities reaches millions or billions, we can group them into around 10,000 fibers. On a typical cluster of 100 nodes, each Data Node will be responsible for handling of log data of 10s to 100s fibers. Fine granularity of partitioning of fiber is in favor of balancing loads among Data Nodes.

Each loader launches several threads according to the number of fibers that it is responsible for. Each thread will pull

data from one of partitions of the Kafka message queue. Each partition of Kafka contains log records of one fiber.

When the total volume of the data accumulated by the threads reach a threshold of one data block (256MB), the loader organizes the temporary data of each thread into a data block. Inside each fiber, the log records are sorted by timestamp; then the fibers are concatenated together, and saved into HDFS using the Parquet format (please refer to figure 3).

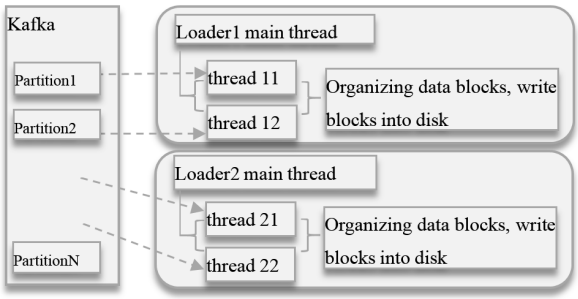


Fig. 3. Parallel processing in log data loading

Note: In Kafka, log records belonging to one fiber will be stored in one message queue partition. An elliptical frame represents a Data Node of HDFS. It is supposed that each Data Node is responsible for loading of two fibers.

Parquet is a columnar format for HDFS. Columnar format will speed up later analytic queries. Since analytic queries usually only access a few data columns, columnar layout can avoid reading of unrelated columns. The data is compressed to save disk space. Data sorting and compression will slow down loading a little bit, however it is worthy when considering the performance benefit we get later when querying the log data.

HDFS saves the primary replica of a data block into local disks on priority, and find out two other nodes in the cluster to store other two replicas. After a data block is written into HDFS, we record some information about the block into one of meta data tables - the *Block* table. Several tuples are logged into the table according to number of fibers contained in the data block. Each tuple has the following information: data block id (Block_id), fiber id (Fiber_id), minimum time stamp of the fiber (start_time), maximum timestamp of the fiber (end_time), record count of the fiber (record_count), and the logical file name of the block in HDFS (block_location).

When the information is registered, it represents that the log records of these fibers have been loaded into data warehouse. To avoid duplicate processing of these log records, we need to register some information in another meta data table - the *Offset* table, to manage the restart point of each fiber. The *Offset* table contains two columns, one is fiber id, and another is offset, which denotes that the message queue partition containing logs of corresponding fiber has been processed to the location. When the loaders fail and restart, they know where to resume the puling of log data from each fiber.

### *E. Discussion: Load Balancing, Fault Tolerance, Data Transformation and Query of the Data*

The number of Data Nodes in a HDFS cluster will increase or decrease due to expansion of the system or failure of some nodes. When the number of Data Node changes, there is a need to change the mapping from fiber to Data Node as well (change the *Mapping* table of meta data store). Meta data is managed by a meta data server, and status of Data Nodes is managed by a Zookeeper server. These two servers run on the same physical node. When the mapping has been changed, the Zookeeper server will notify related Data Nodes. Data Nodes will pull log records from Kafka according to new mapping. Having been pulled but not yet loaded data is discarded.

For example, some node who is responsible of pulling and loading of log records of fiber11 and fiber12 fails. Then fiber 11 and fiber 12 are handed over to new nodes respectively. For fiber 11, the log records pulled by the failed node is discarded (there is no problem, because the data is not yet loaded into data warehouse), and the new Data Node will pull and load the data into data warehouse according to the offset recorded in the *Offset* table.

The mapping from fiber to Data Nodes is periodically readjusted. The objective is to guarantee that the log data is dispersed onto the cluster. From the perspective of each fiber, before readjustment of the mapping, the primary replicas of blocks (log records) of the fiber are written to some Data Nodes. After readjustment of the mapping, the primary replicas of blocks of the fiber will be written to some new Data Nodes. The readjustment of the mapping is called *Mapping Shuffle*. *Mapping Shuffle* help to balance the load of the Data Nodes, and avoid situations of some Data Nodes become too busy.

Our proposal only considers data staging and data loading. It assumes that the data coming from upstream data stream or data files is of high quality. In the scenarios that it is necessary to transform the data, we can embed data cleaning logic into the process of organizing the data blocks before writing them into HDFS.

The log records written to HDFS is organized as blocks, each block has a HDFS file name. However, logically these blocks collectively constitute a data table. Hive [3] and Spark [4] both support some *viewing* mechanism, through which multiple files (having the same schema) are seen as one logical table, on which queries could run against. For example, Spark *SQL Context* has a *unionAll* method, which combines multiple files and presents them as a logical table as follows.

```
01 var df: DataFrame = null;
02 for (file <- files)
03 {
04   val fileDf= sc.textFile(file)
05   if (df!= null)
06   { df= df.unionAll(fileDf)
07   } else
08   { df= fileDf
09   }
10 } // note: sc is a SQLContext object.
```

## III. EXPERIMENTS

### *A. Experiment Setting*

We have conducted experiments on “Renda Xing Yun” cloud platform. The Kafka cluster is comprised of 3 virtual nodes, and the Hadoop HDFS cluster is comprised of 8 virtual

nodes. Each virtual node is equipped with 8 CPU cores and 16GB of main memory. The operating system is CentOS 6.5. The versions of Kafka and Hadoop are 2.9.1-0.8.2.2 and 2.6.3 respectively. The dataset we use is the Lineitem table extracted from the TPC-H benchmark, and the volume of data is 3 GB. We have added a new column to the table to record the timestamp, at which the tuple is injected into the data staging and loading pipeline.

### B. Experiment Results

The first experiment is to study how Kafka *Batch Size* affects the performance of data staging. To guarantee the log data is sent to Kafka as fast as possible, we have loaded the data set of 3GB to main memory of a virtual node before injection of the data. The node connects to other nodes through a 1000M Ethernet.

Table I lists the times to stage the 3GB dataset with different *Batch Sizes*. From the figures we can see that, 16KB of *Batch Size* achieves a good enough performance. There is little use to increase the parameter further.

TABLE I. THE INFLUENCE OF BATCH SIZE ON THE PERFORMANCE OF DATA STAGING

| Batch Size | 4KB | 8KB | 16KB | 32KB |
|---|---|---|---|---|
| Time to Staging | 56.7s | 57.9s | 56.7s | 56.6s |

The second experiment is to study how the number of partitioners affects the performance of data staging. Table II lists the performances of staging when the number of partitioners are 1, 2, 3, and 4 respectively. Owing to highly scalability of Kafka, the performance of data staging is close to 60MB/s, it can be translated to 390,000 tuples/s.

It is unworthy to add more partitioners since the improvement of performance is limited. To further improve the performance of data staging, a feasible way is to install disks with higher I/O performance, such as SSDs.

TABLE II. THE INFLUENCE OF NUMBER OF PARTITIONERS ON THE PERFORMANCE OF DATA STAGING

| Partitioner# | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Performance of Staging | 57.7MB/s | 58.5MB/s | 59.7MB/s | 59.5MB/s |

The third experiment is to study the performance of data loading. We have run the tests for three times. The maximum loading times, minimum loading times and average loading throughputs of the 8 data nodes are listed in table III.

Average loading throughput is around 24.7MB/s, it can be translated to 162,500 tuples/s, which is much superior to the loading throughput of writing sequentially to a single HDFS file, i.e. 37,000 tuple/s.

TABLE III. LOADING PERFORMANCE ON THE DATA NODES

| Experiment rounds | maximum loading time | minimum loading time | Average loading throughput |
|---|---|---|---|
| #1 | 126.2s | 122.2s | 24.9MB/s |
| #2 | 125.6s | 121.8s | 24.7MB/s |
| #3 | 127.2s | 122.5s | 24.8MB/s |

### C. Query Performance

Although our work is focusing on staging and loading of log data, however, we also present rudimentary query performance results of our prototype here.

In Hive and Spark, several files having the same schema can be loaded as a logical table, which can then be operated on or joined with other tables. Our proposal has made use of this viewing capability of Hive and Spark. By loading each data block (of a log table) as one file into HDFS, we can speed up loading by parallel processing.

After log data is staged then loaded into data warehouse, it can be queried. For entity centric queries, such as TPC-H Q1 (query 1), we can use meta data table of *Block* to filter out unrelated blocks, only blocks containing relevant data is scanned, query performance can thus be improved.

The experiment to study performance of entity centric queries is conducted on the same virtual cluster as the one used in staging and loading experiments, however we have increase the volume of data to 100GB. An entity centric query is composed based on TPC-H Q1, which calculates some aggregations on the Lineitem table with a time range condition and a designated customer key (a customer is an entity). The SQL statement is as follows.

```
SELECT SUM (quantity) AS sum_qty,
SUM (extendedprice) AS sum_price,
AVG (quantity) AS avg_qty,
AVG (extendedprice) AS avg_price,
AVG (discount) AS avg_disc,
COUNT (*) AS count_order,
MIN (orderkey) AS min_orderkey,
MAX (orderkey) AS max_orderkey
FROM Lineitem
WHERE
custkey = %s AND
messagedate > %s AND messagedate < %s
```

We have used meta data table of *Block* to filter out unrelated blocks, handed over left blocks to Spark SQL for later processing of above query, without any more optimizations. The response times are listed in table IV.

TABLE IV. RESPONSE TIMES OF ENTITY CENTRIC QUERY

| selectivity | 1% | 2% | 5% | 10% |
|---|---|---|---|---|
| Response time(s) | 18.9 | 19.7 | 23.9 | 25.7 |

We achieve less than 20 seconds of response times for queries with selectivity less than 2%. We believe that after more advanced optimization techniques such as vectorized query execution [5] are employed, the response times could be further cut down.

Worthy to mention is that above results are achieved without any data caching. We have shutdown Spark SQL and restart it for each run of the query. We also run the same query with different parameters without restarting Spark SQL, the response time of the first run of the query is consistent with above results, however subsequent runs of the query with changed parameters achives much faster response times, varying from 2 seconds to 13 seconds (with a selectivity less than 10%). It can be explained by the fact that these runs of the query with different parameters could be partially served by the data having been loaded and cached in main memory.

### D. On Data Freshness

Since the data is loaded as fast as possible, we expect that recent data will show up quickly in data warehouse for analysis. Data freshness of the data warehouse could be improved to *MINUTE* level. We expect that data coming 15 minutes before has been ready in data warehouse and could be queried when the data is continuously arriving with high throughput, which should be verified by some experiments with real life data sets.

## IV. Related Works

MapReduce [6] has established itself as the de-facto standard technology for large-scale data-intensive processing in recent years. [7] and [8] present a scalable dimensional ETL framework, ETLMR. And it makes use of MapReduce to parallelize ETL execution to achieve higher performance. ETLMR has built-in native support for operations on star schemas, snowflake schemas and slowly changing dimensions (SCDs). This enables ETL developers to compose scalable ETL flows with very few code lines. The authors proposed parallel processing methods for slowly changing dimensions, including one dimension one task, one dimension all task etc. They also pinned some dimensional tables in memory to accelerate lookup operations when processing the fact table. Their experiment results show that ETLMR has a good scalability, and it outperforms Pentaho Data Integration by three times. ETLMR is, however, built for processing data into an RDBMS-based data warehouse. The throughput of loading relies on *LOAD* performance of target RDBMS.

[9] has extended the work of [7] and [8], and proposed a cloud based ETL framework, CloudETL. Hive is an RDBMS-like system for data warehouse application on Hadoop with scalable analytical features. CloudETL has used Hadoop (open sourced implementation of MapReduce) to parallelize the ETL execution to process data into Hive. Users define the ETL process by means of high level operators, and do not have to worry about the details of MapReduce computing model. CloudETL has built-in support for star schemas and SCDs. CloudETL also provides the support of data co-location to speed up later data queries. Their experiment results show that CloudETL scales very well and greatly outperforms the dimensional ETL capabilities of Hive both in terms of performance and programmer productivity. In general, CloudETL outperforms Hive loading capability by 3.9 times.

A two-level data staging ETL for handling transaction data is proposed in [10]. Their method detects the changes of the data from transactional processing systems, and uses two staging databases for temporarily storing of data, and to facilitate the data processing in an ETL workflow. The first staging database employs the same schema as the one the OLTP system uses, and finishes the work of identifying data changes and data transformation. The second staging database employs the same schema as the one the back end data warehouse uses, its work is to validating primary/foreign key relationship, and loading of the data.

Different from above works, our work is to accelerating data loading of log data for later entity centric analysis, we don't consider handling of dimensional tables (we will handle dimensional table data loading and consider join optimization in future work). Secondly, we have use Kafka's persistence capability to temporarily store the log data as fast as possible without data loss. Entity fiber based data partitioning is performed when injecting data into Kafka, which get data ready for later parallel loading. Thirdly, we have designed multi-thread loaders to pull data from Kafka and load data directly into HDFS in parallel. Finally, the data residing in HDFS can by queried by Hive or Spark.

## V. Conclusion

Valuable information is buried in log data, entity centric real-time analysis of fine granularity of log data help us to learn recent behaviors of related entities. Such analysis requires that detailed log data is loaded into data warehouse as fast as possible without loss. We propose a fast staging and loading method for log data. We have implemented an entity fiber based data partitioning and staging method, and a parallel data loading algorithm, which is based on open sourced tools including Kafka, HDFS, and Spark. Our experiment results show that our proposal achieves a good performance, meets the need of fast staging and loading of log data.


## Acknowledgment

This work is funded by NSF China under grant #61170013 and #61432006, and Science and Technology Department of Guangdong Province under grant # 2015B010131015 "Industrialization of High Throughput and Real-time Business Intelligence System on Big Data".


## Reference


[1] Haoqiong Bian, Yueguo Chen, Xiongpai Qin, Xiaoyong Du. A Fast Data Ingestion and Indexing Scheme for Real-Time Log Analytics. APWeb 2015, pp. 841-852.

[2] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, Joe Stein. Building a Replicated Logging System with Apache Kafka. PVLDB, 2015, 8(12): 1654-1655.

[3] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2009, 2(1): 1626-1629.

[4] AMP Lab. Spark SQL. http://spark.apache.org/sql/, 2016.

[5] Andrei Costea, Adrian Ionescu, Bogdan Raducanu, Michal Switakowski, Cristian Bârca, Juliusz Sompolski, Alicja Luszczak, Michal Szafranski, Giel de Nijs, Peter A. Boncz. VectorH: Taking SQL-on-Hadoop to the Next Level. SIGMOD 2016, pp.1105-1117.

[6] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, Bongki Moon. Parallel data processing with MapReduce: A survey. SIGMOD Record 2011, 40(4): 11-20.

[7] Xiufeng Liu, Christian Thomsen, Torben Bach Pedersen. ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. DaWak 2011, pp. 96-11.

[8] Xiufeng Liu, Christian Thomsen, Torben Bach Pedersen. MapReduce based Dimensional ETL Made Easy. PVLDB 2012, 5(12): 1882-1885.

[9] Liu Xiufeng, Christian Thomsen, Torben Bach Pedersen. CloudETL: Scalable Dimensional ETL for Hive. 18th ACM International Symposium on Database Engineering & Applications 2014, pp. 195-206.

[10] Xiufeng Liu. Two-level Data Staging ETL for Transaction Data. http://arxiv.org/pdf/1409.1636v1.pdf, 2014.